Graphics on the HP 48G/GX

By R. Ray Depew

GRAPHICS ON THE HP 48G/GX

by

R. Ray Depew

Grapevine Publications, Inc.
P.O. Box 2449
Corvallis, Oregon 97339-2449 U.S.A.

Acknowledgments

Thanks goes once again to Hewlett-Packard for their top-quality products and documentation. Thanks also to the gang on `comp.sys.hp48` (you know who you are), to Charlie Patton for his assistance with 3-D graphics, and to Jim Donnelly for his assistance with menu grobs.

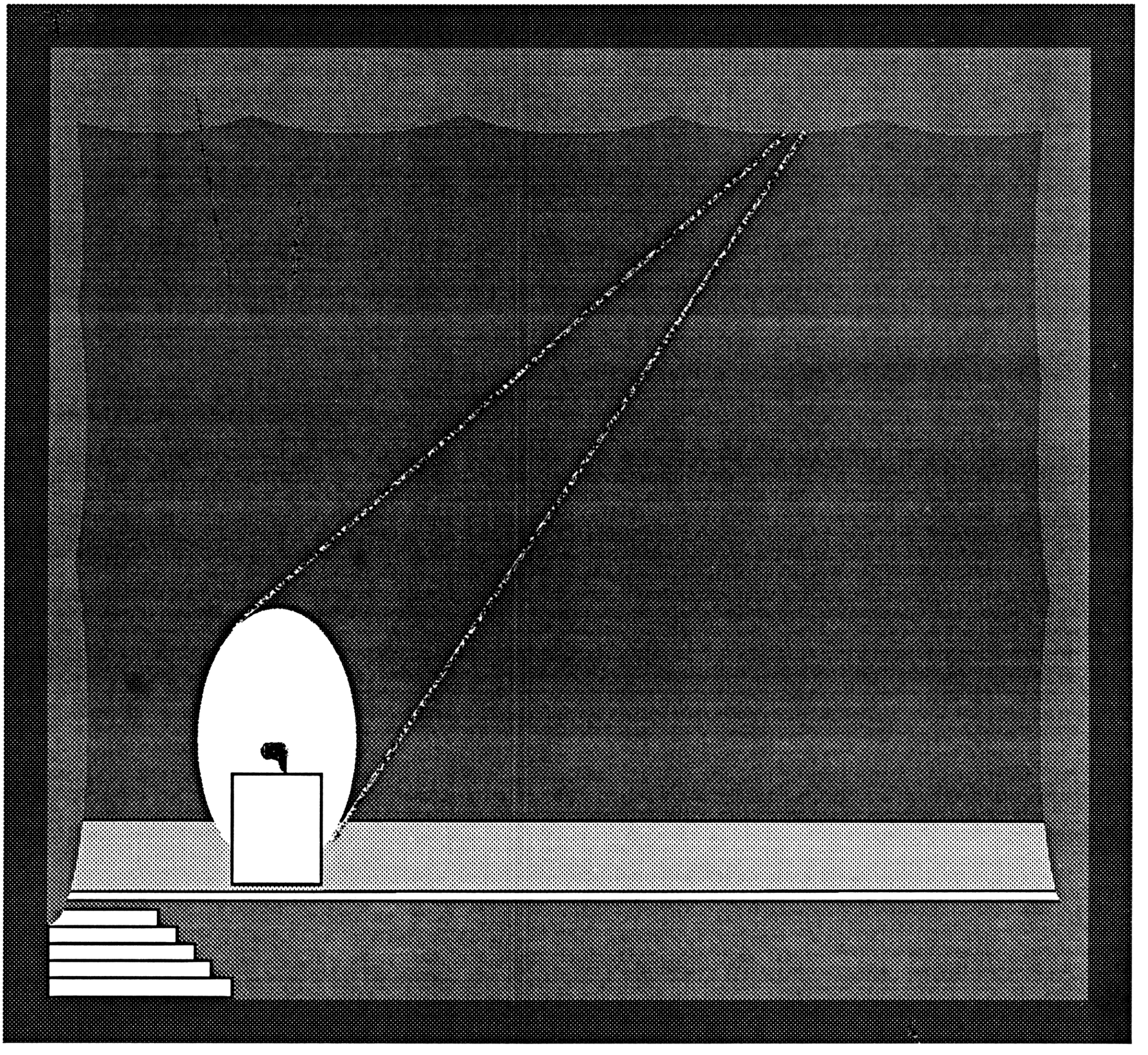
Pen-and-ink illustrations by Robert L. Bloch.

Printed in the United States of America
ISBN 0-931011-42-6

First Printing – October, 1993

To my sweet wife, Valerie, who encouraged and indulged me in this effort from the start, and whose love and cookies helped me to finish it.

Contents



1. Introduction

What This Book Is About

The HP 48G/GX calculator ("48" for short) is the latest in a long line of great handheld calculators from Hewlett Packard Company. It combines nearly all of HP's most popular features into one package.

The 48 makes handheld problem-solving and/or data manipulation easier than ever before. Among other new capabilities, it offers you the **EquationWriter**, the **Solver**, and the **Plotter**.

- With the EquationWriter, you can enter an equation in *textbook notation*—just the way you normally see it on paper (as opposed to *algebraic notation*, which forced you to count parentheses and put all your terms on one line).

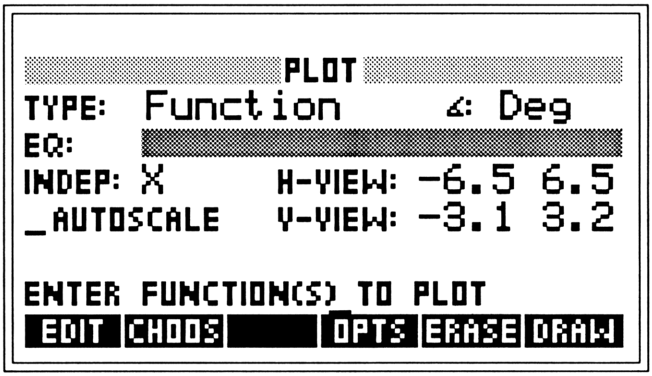
- With the most powerful version of HP Solve to date, you may never have to write another program again: The 48 Solver lets you *solve your equation directly from the equation form,* rather than having to translate it into a program.

- One of the greatest—but most neglected—features of the 48 is its Plotter, and more generally, its graphics capability. You can manipulate the entire 64×131-pixel display, with many powerful built-in functions. And you needn't stop at 64×131 pixels. This book will show you how that display is only a small window into a much larger world of graphics power.

First, take just a moment to see these three capabilities in action. This is just a "warmer-upper" to pique your interest—so don't worry—you'll get more explanation on all of this in the chapters to come....

Plotting a Simple Function

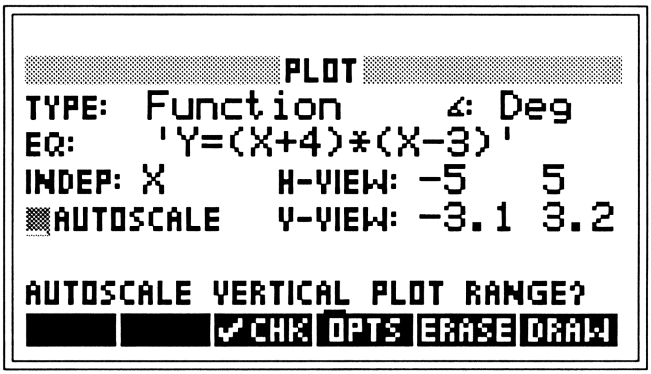
Set your display mode to FIX 2 ([2][SPC][α][α][F][I][X][ENTER]). Then begin with this simple quadratic function: $y = (x + 4)(x - 3)$. Start the PLOT application by pressing [→][PLOT]. The plot input form will appear:



Press [←][EQUATION] (the [←][ENTER] key) to enter the EquationWriter. Then press these keys (if you make a mistake, backspace it out with [◀]): [α][Y][←][=][←][()][α][X][+][4][▶][←][()][α][X][−][3][▶]. Your equation should look like this:

Y=(X+4)·(X-3)▯

Press [ENTER] to store your equation and return to PLOT. Next, enter the x-domain, say, -5 to 5: Press [▼][▶][5][+/−][ENTER][5][ENTER]. You should see:

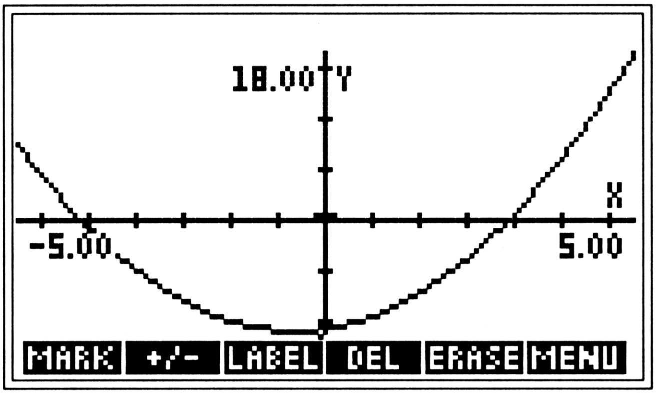


Press **✓CHK** to let the 48 calculate the y-range automatically.

Now plot the function: Simply press **ERASE** **DRAW**.... The display will blank out, then fill with a parabola as the 48 calculates and plots each point.

Now press **EDIT** (NXT) **LABEL** to label the axes.

Your display should then look like this:



Adjusting Your Plot

Of course, you can change your *y*-range—it doesn't have to be the one that the machine automatically calculated.

Press CANCEL twice. Now, to choose a *y*-range of –20 to 30, type in the coordinates of the lower left and upper right hand corners of the plot: `(-5,-20) (5,30)`, and press PRG **PICT** **PDIM**.

Now press ↱ PLOT **ERASE** **DRAW**....Your previous parabola is erased, and a new parabola is drawn in its place. Press **EDIT** NXT **LABEL** to label the axes.

But notice this: Press ↱ ◀, then press and hold down ▲. The display scrolls down as the cursor travels up the *y*-axis to *y* = 30.... Now where's your parabola? Press and hold ▼ to bring it back into sight. The point here is that you can make your plots *larger than the display.*

So keep in mind that you can either check the `_AUTOSCALE` field in the PLOT input form to tell the 48 to calculate the y-range for you—sufficient to fit the display; or you can specify your own y-range manually, by modifying the `Y-VIEW` field in that input form.

Both scaling options are useful: For example, use `_AUTOSCALE` to give you a "feel" for where your function plot will lie. Then use **PDIM** to stretch or shrink your plotting range, in a way similar to the **ZOOM** functions provided in the graphics environment. (You'll read more about ZOOM later in this book; see also your User's Guide for details on the 14 different ZOOM functions.)

Solving Within the Plotter

You can do more with your parabola than just look at it and marvel: Hidden in that display is a graphics cursor, shaped like a crosshair. Press ▼ and ◀ a couple of times to find it.

Now, find out what the two roots of this function are: Press and hold ◀ until the crosshair is close to the left side of the plot, where the function crosses the x-axis. Now press NXT PICT FCN ROOT ....

The crosshair zeroes in on the root and the bottom line of the display tells you that the root is at -4.00!

Press − or NXT to get the menu back, and then SLOPE to find the slope of the function at this root point ($x = -4$).... The slope is -7.00. Now ▼ and ▶▶ to find the cursor, then press and hold ▶ to get to the right side of the screen. Now use ROOT and SLOPE again to find that the slope at the positive root is 7.00, as it should be.

Press − EXTR to find the extremum, or lowest point on the function. It's at (-0.50,-12.25). Press − or NXT to bring back the menu, then ◀◀◀ NXT F(X) to find the function value at the current location.

As you can see, you can utilize most of the capabilities of the Solver *without ever leaving the Plotter application.* And while this quadratic function was admittedly simple, you can do these same things with much more complicated functions—you'll see how in later chapters.

Now press CANCEL twice to return to the Stack display. See? The roots, that you just calculated from inside the Plotter have also been placed on the Stack—for your subsequent use (and calculating enjoyment)!

Freehand Graphics

Using the built-in capabilities of the Plotter and Solver are perfect for many needs. But when you want to create custom graphics of your own, that's a job for the PICTURE EDIT menu.

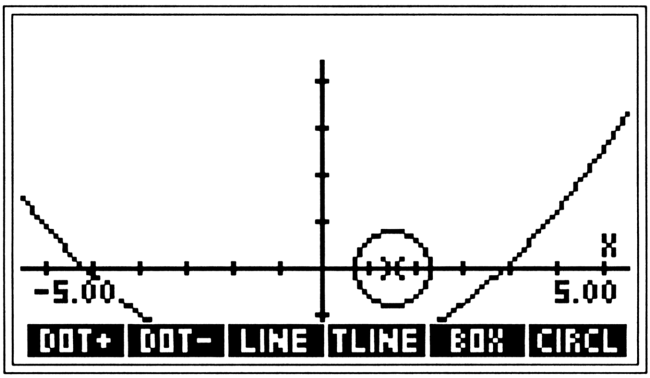
Often the 48 gives you more than one way to do things. For example, the PICTURE menu comes up automatically when a plot is completed or (in a program) when the PICTURE command is executed, or when you (manually) press ← PICTURE. Do that now—press ← PICTURE. The menu looks like this:

ZOOM | (X,Y) | TRACE | FCN | EDIT | CANCL

And now press EDIT, to see the PICTURE EDIT menu:

DOT+ | DOT- | LINE | TLINE | BOX | CIRCL

Using the ▲, ▼, ▶ and ◀ keys, put the cursor half an inch to the right of the origin. Now press × (multiply), then ▶ a few times. You'll see an × where the cursor appeared originally—but now the cursor is sliding to the right. Now press CIRCL.... You'll eventually see this:



You're doing *freehand drawing* on a plot drawn by the 48!

Next look at the menu items labeled **DOT+** and **DOT-**.

DOT+ turns pixels on (makes them black), while **DOT-** turns pixels off (makes them white). The ■ annunciator appears in the **DOT+** or **DOT-** menu key label to indicate which one is active.

Experiment with **DOT+** and **DOT-** by pressing each once...then twice...while moving the cursor around....

See? If **DOT+** is activated, to deactivate it, press the **DOT+** menu key once more. The annunciator will turn off—so you can move the cursor about freely, without trailing a black line behind you. In the same way, if **DOT-** is activated, press **DOT-** a second time to move around without erasing whatever images you've just finished making.

Grobbing Around

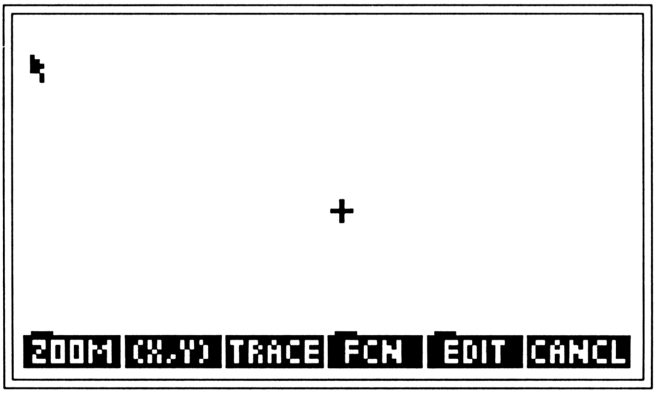
For the next exercise, press CANCEL until you return to the Stack. Now, carefully type (without quotation marks):

GROB 3 6 103070304040 ENTER

You should see Graphic 3 x 6 on Level 1 of the Stack. Now press the following keys:

PRG PICT PICT STO ← PICTURE

You should see a small arrow in the upper left corner of the display, like this:



You've done freehand drawing without even using the GRAPHICS menu. (Actually, you have created a *grob*—more on that soon.)

Is It Real—Or Is It...?

Now, just for fun, press CANCEL to return to the Stack display. Then fill the lowest four levels of the Stack with any objects you want, and press the following keys:

PRG **GROB** NXT **LCD→** PRG **PICT** **PICT** STO ← PICTURE

Look at the menu. That's the first page of the PICTURE menu.... What's it doing in the Stack display?

Press **EDIT**. If the **DOT+** annunciator isn't on, press **DOT+** once to turn it on. Then use the arrow keys to move the cursor around the display.... You're drawing all over your Stack display!

The secret? You're not really drawing on the Stack display (and you can confirm this by pressing CANCEL to return to the real Stack display). Rather, you've created a *grob image* of the Stack display—and stored it in the graphics display. The advantages of this feature for documenting your programs and creating friendly output should be obvious—and you'll see other uses for this later on, too!

What Next?

By this time, hopefully, you've gotten a taste—and whetted your appetite—for what the 48 can do. Of course, it would take *several* books to tell you all the great things it can do, but this book is to show you how to use the new graphical features in the 48.

To do that, this book is divided into three parts:

1. Beyond-the-Manual Basics

To give credit where credit is due, HP has carefully documented just about every feature they built into the machine. But face it—it's hard to *show* you everything a new application can do in a manual of any reasonable size. So that's what the first part of the book will do with the graphical features:

Chapter 2 should help you be more comfortable—and more effective—with the **EquationWriter**.

Chapter 3 shows you how to unlock the *real* power of the **Solver**. You have already seen how it looks in its "Sunday best"—running inside the PLOT application—but wait until you see it "getting down and dirty," in its work clothes!

Chapter 4 teaches you the basics—the "care and feeding"—of **grobs**, the *graphics objects* in the 48. You'll learn how to conjure them up and manipulate them as easily as any other object.

2. Advanced Use—the Graphics "Power Tools:"

Chapters 5-8 go beyond the basics. To help you to effectively use graphics, you'll build a toolkit of convenient and useful routines for storing and recalling grobs, combining text and graphics, etc.

Next, you'll see how to use those tools: You'll tip your head sideways and learn how to do "sideways plotting"—strip charts, waveforms and the like. And you'll see how to create and use freehand graphics in the display.

You'll explore the three-dimensional plotting capabilities built into the HP 48G/GX—and you'll see how to use them to visualize abstract functions and data more easily. You'll even see how to make all your graphics come alive with the 48's animation tools.

3. Full-Blown Applications:

Chapters 9 and **10** present several self-contained applications that use programmable Plotter and Solver commands.

Some of these applications are useful as is, while others are offered in hopes that you'll then alter them for your own purposes ("Oh wow—if I change that one subroutine I can ...").

Keep in mind, however, that this book is not necessarily meant to be read from cover to cover. Here are a few suggestions....

Notes on Using this Book

Of course, read this book with your 48 by your side. You needn't do every example or program here, but it's a lot easier to try things—or clarify them—right away, rather than waiting until later, when you've forgotten what was so mystifying and/or exciting. Also, if this is your own personal copy of this book, then by all means, write in the margins, inside the covers, etc. Make the book useful to *you*. Keep a highlighter and a notepad handy—and use them.

First Note: As you can tell from those opening "warmer-upper" keystrokes, *this book assumes that you already know a few things* about your 48. You should know how to:

- **Name** objects, **edit** them, **store/recall** them—and how to **manipulate** them on the Stack (e.g. [SWAP] or [DROP] them, etc.);
- Use **menus** and **menu keys**—and the [NXT] and [⬅][PREV] keys;
- Use the MODES menu and input form to set display and calculations modes;
- Use **directories** and "move" through a **directory structure**;
- Build **strings**, **algebraic expressions/equations**, **binary integers**, and **programs**.

This book may occasionally offer reminders on some of these basics, but that's about it. For a good tutorial on all these sorts of topics, read

<u>**An Easy Course in Programming the HP 48G/GX**</u>

This book is available from your HP dealer or from the publisher.

Or, if you simply need some "brushing-up" as you go, here's how to use your 48 User's Guide ("UG") alongside this book:

- First, carefully reread the UG's chapter 2, called "Objects."

- Work through the examples in chapter 7 of the UG. The EW is something new—far ahead of other machines—and it takes a little practice to get used to. (For best results, keep a stack of homemade oatmeal-chocolate-chip cookies nearby, to pass the time while the 48 redraws the display.)

- Before you start on Chapter 3 here, skim once more through chapter 18 in the UG (just work through the examples they provide). The basic Solver is easy to learn, and once you understand it, Chapter 3 in this book will be much more useful.

- When you've reached the end of Chapter 3 here, you're ready for a serious intermission. Watch some mental junk food on network TV. Eat some real junk food. Eat some real food. Take a nap.

- When you come back, reread chapters 9 and 22-24 in the UG. Then work through Chapter 4 here, to learn the fundamentals of grobs—and some "good habits" you should consider adopting.

- After that, you can pick and choose among the remaining chapters in this book. If you don't understand something, come back to Chapters 2-4—or to the index of the UG—for help.* If something here is still unclear, write to the publisher.

*Note: Certain advanced topics, such as input forms and pop-up windows, are described not in the UG but rather in the <u>HP 48G Series Advanced User's Reference Manual</u>, available separately from Hewlett-Packard or from your HP dealer.

Second Note: As in any computer, there are 4 kinds of "features" in the 48:

- **Documented Features**. Designed features that are described or at least mentioned in the HP manual(s).

- **Undocumented features**. Designed features which work predictably—and sometimes usefully—but nevertheless don't make it into the manual(s), for various reasons.

- **Unsupported Features**. Features or operations that HP "accidentally" left accessible to users but were never intended for use by the general buying public. These features can greatly enhance your calculator's capabilities, but their misuses often carry drastic consequences (e.g. `Memory Clear`). So these features are neither encouraged nor documented by HP.

- **Bugs**. A bug is simply a design mistake in program code. A bug's behavior may be predictable or erratic, but its consequences are undesirable. If you find a bug in your 48's operation, report it at once to HP. If you find a bug in any code in this book, please write to the publisher.

This book will use primarily **Documented Features**, so that all its examples and programs will work on all 48's. You'll also encounter a small handful of **Undocumented Features** that HP publicized after the manuals were written. You may even find a few **Unsupported Features**.

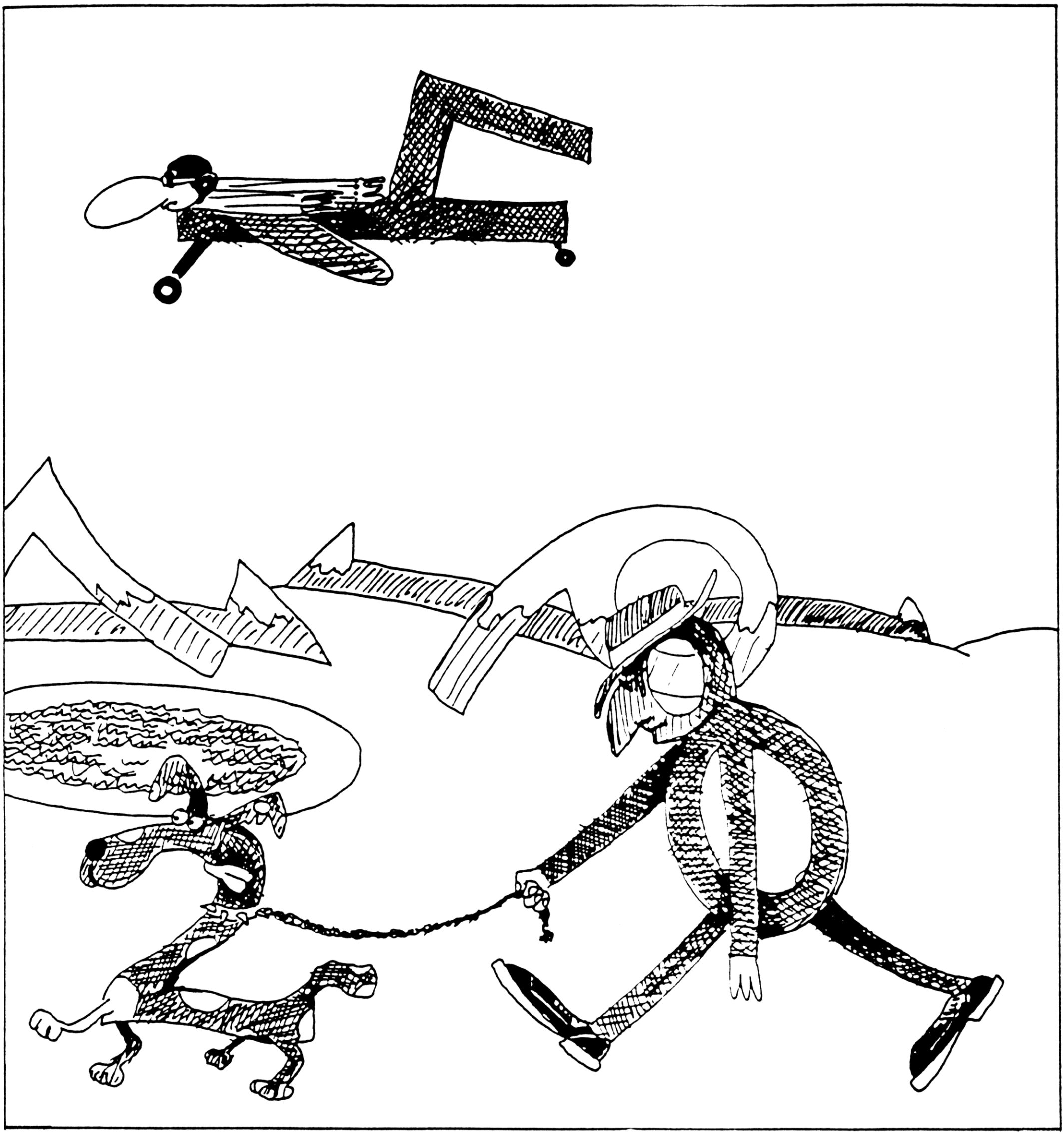
Third Note: The procedures, examples and programs in this book won't hurt your 48. None of the ideas and procedures described should give you the dreaded `Memory Clear` (if you get such a message, retrace your steps very carefully, to see where *you* went wrong). In general, if you fear memory loss—for whatever reason—it's a good idea to back up your valuable files frequently.

All the examples in this book worked on HP 48G/GX ROM version K. If you use them exactly as they appear in this book (forgiving typos), they should work fine on your HP 48G or HP 48GX, as well, if your ROM version is K or later.* But feel free to experiment, too: try some things differently from the way the book does it, and see if you can improve on the ways you see them done here.

Note: Because of the enhancements made to the HP 48 operating system in the HP 48G and HP 48GX, these examples may or may not work with older HP 48S and HP 48SX. If you want to study graphics on the older machines, there is a book very similar to this one, written exclusively for the HP 48S/SX. For more information, contact the publisher.

Fourth Note: Go!

*To identify the ROM version in your machine, type [α] [α] [V] [E] [R] [S] [I] [O] [N] [ENTER].



2. The EquationWriter

Preparations

First, you need to create a directory for this chapter—so you don't clobber anything you may already have going:

Press [→][HOME], then type 'G.CH2' [←][MEMORY] DIR CRDIR [VAR] G.CH2 to get into this brand-new G.CH2 directory. The menu items should now all be blank, and the Status Area at the top of the display should show { HOME G.CH2 }

Opening Remarks

The EquationWriter (EW) is one of the 48's most exciting features—perhaps setting it apart from all other handheld machines. In a world that turns on legal questions of "look and feel," the EW display may *look* like some brand-x displays you've seen, but it *feels* quite different.

The EW version in the G series of HP 48's is much faster than the original version introduced with the S series (the HP 48S and HP 48SX), but it is still no speed demon—you may at first be put off by that. At least work through this chapter before deciding.

Indeed, you may find that the speed doesn't matter; the very existence of the EW is one of the most revolutionary advances in calculator technology to-date. Ever since the first FORTRAN compiler or BASIC interpreter let you enter equations on a digital computer, you've had to cram the normal, two-dimensional, *textbook notation* equations into the single line of display characters—*algebraic notation*—in order to be understood by the software. There had to be a better way....

There *is* a better way: Even with the EW's not-so-blinding speed, it will usually take you far less time to enter an equation correctly into the EquationWriter than with the "algebraic" form.

As you discover this, you'll probably go through these three typical stages with the EW:

- **Excitement & Delight:** "Wow—look at what this can do!" Typically, this lasts about twice as long as it takes you to work through the EW chapter in the Owner's Manual.*

- **Frustration & Discouragement:** Fed up with its slowness—or not yet completely understanding it—many are tempted to abandon the EW in favor of the Command Line editor. These people may have as much trouble trying to debug their algebraics on the Command Line, but they don't realize it, having accepted line editors and their attendant frustrations as the cost of machine algebraics.

- For those who survive, there's the third stage, characterized by your high school band teacher's pet motto: "Proficiency comes through practice" (translation: **"Use It Or Lose It"**).

Actually, the EW and the Command Line Editor (CLE) are *both* useful in certain situations: If the EW's slowness bothers you, then use it strictly as an equation <u>writer</u>, or <u>viewer</u>, but not as an <u>editor</u>.

*By the way, have you worked through that chapter yet? If not, put a bookmark—not a cookie—here, and go do all the examples in that chapter.

So What Does It Do?

When you write an equation or an expression on paper...

$$\int_{b^3-4.32}^{a^3+1} \frac{\sqrt{\dfrac{x^3-22x+1}{\ln x + x}}}{3\ln x + e^{x-4.2}}\,dx$$

...you use this *textbook notation,* an easy way for your brain to understand the problem: It detects *visual patterns* (position, size, enclosure, etc.) to give you an immediate grasp of what's being said.

Compare that with the computerized *algebraic notation* for the above expression:

```
∫(b^3-4.32,a^3+1,√((x^3-22*x+1)
/(LN(x)+x))/(3*LN(x)+EXP(x-4.2)),x)
```

It's not so clear at one glance, is it? So the EW lets you enter and view the expression in whichever notation you prefer (inside the 48 it's always represented the same way, no matter which way you enter it).

Then, after you've entered the equation, the EW also provides several tools for manipulating and modifying it. It can even recognize *parts* of the equation to modify, using the properties of algebra and calculus!

Examples

Like the Command Line, you can use the EW to write *algebraic expressions*, *equations* and *unit objects*. An algebraic expression is half an equation; an equation is two algebraic expressions joined by an equal sign (=). For example, the positive root of a quadratic equation is this algebraic expression:

$$\frac{-B+\sqrt{B^2-4AC}}{2A}$$

How would you enter this, using the EW?

To Do This	**Press This**
Enter the EW and start a numerator.	[⮠] [EQUATION] [▲]
Use [y^x] instead of [x^2]—it looks better.	[−] [α] [B] [+] [√x] [α] [B] [y^x] [2]
Close the exponent.	[▶]
Forgetting to close subexpressions with [▶] *is a common EW error!*	
Imply a [×] between a number and the letter following it. The letter is taken as the start of a variable or function name.	[−] [4] [α] [A] [×] [α] [C]
Close the subexpression opened by [√x].	[▶]
Close the numerator/start the denominator.	[▶]
Again, imply the [×].	[2] [α] [A]
Close the denominator.	[▶]
Place the expression onto the Stack.	[ENTER]

Complex *unit* objects are also easy to assemble with the EW. Look, for example, at:

The universal gas constant, R:

$$R = 8.315\frac{\mathrm{J}}{\mathrm{mol}\cdot\mathrm{K}}$$

The gravitational constant, G_c:

$$G_C = 9.8\frac{\mathrm{kg}\cdot\mathrm{m}}{\mathrm{s}^2\cdot\mathrm{N}}$$

To enter R using the EW:

[←][EQUATION][8][•][3][1][5][→][_] (_ denotes a unit object)
[→][UNITS][NXT] ENRG J [/]
[→][UNITS] MASS [←][PREV] MOL [×]
[→][UNITS][NXT] TEMP K [▶]

Then press [ENTER] to put this constant onto the Stack.

To enter G_c:

[←][EQUATION][9][•][8][→][_]
[→][UNITS] MASS KG [×][→][UNITS] LENG M [/]
[→][UNITS] TIME S [y^x][2][▶][×]
[→][UNITS][NXT] FORCE N [▶]

Then press [ENTER] to put this constant onto the Stack.

Using the EquationWriter

This would be a good place to insert a table of all the keystrokes used in the EW. But your HP User's Guide already has a complete table.

To be really proficient with the EW, just remember these...

Rules of Thumb:

- ▶, ▲ and ← (not ◀) are the most frequently used keys in the EW.
- Use ▲ to start a numerator, then ▶ to finish it and start the denominator (incidentally, ▼ acts identically to ▶).
- ▶ finishes all subexpressions ("it slices...it dices"):

  It finishes powers, as in y^x;

  It finishes numerators and starts denominators

  It finishes denominators and exits the fraction

  It finishes square roots and other roots: $\sqrt[x]{y}$

  It finishes mathematical functions, such as $\sin(x)$

  It jumps to the next parameter when constructing a derivative, an integral or a sum

  It exits a parenthesized subexpression, such as $a + (b + c)$

  It finishes *any* pending subexpression (and ↱ ▶ finishes all pending subexpressions).

- [←] is the only real editing key you have. Each time you press [←], it "undoes" the last keystroke in the equation. Press it repeatedly to go as far back in the equation as you want (the pause is always longest after the first press).

- If you notice an error deep inside your equation, your options are limited. *Do not* press [◀], trying to move the cursor to the error ([◀] takes you to the *Selection Environment*—an upcoming topic).

- Most analytical functions, such as those in the MaTH menu and the powerful IFTE function, work inside the EW. If a function requires parameters, you enter the function, then the parameters, separated by [SPC], and finally [▶] to close the parameter list. For example, to enter the function IFTE(A, B, C), you would press [PRG] BRCH [NXT] IFTE [α][A][SPC][α][B][SPC] [α][C][▶].

- All the UNITS menus work inside the EW.

There are 4 ways to exit the EW:

- [EVAL] evaluates the expression and puts the result onto the Stack.

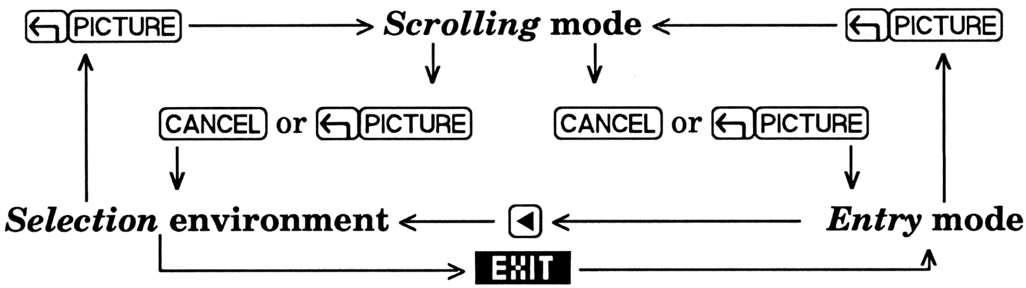
- [ENTER] puts the equation on the Stack as an algebraic, then exits gracefully.

- [↰][EDIT] gives up in disgust and slams the (usually) unfinished equation into the Command Line for further editing. After editing, you can press [ENTER] to return to the EW, and [ENTER] again to place the equation onto the Stack.

- [CANCEL] is the "panic button." It dumps the whole thing into the waste basket and escapes to the safety of the Stack display.

The Selection Environment

The EquationWriter actually consists of three separate environments (also called *modes*). Here's how to switch between the three modes:



If you accidentally pressed ◀ while practicing with the EW, you may have noticed that you had to wait a terribly long time for the display to do anything. Go ahead—try it now (then go get a cookie).... When the smoke finally clears, you can use the arrow keys to move quickly around the equation, highlighting terms and operators as you go. You'll also see this menu: RULES EDIT EXPR SUB REPL EXIT

This is the Selection Environment, where you can easily select various parts of the equation you're building, to edit or rearrange them. The last menu item, EXIT, simply sends you back to the normal EW display—but look at what the other menu items do for you:

RULES is a compilation of rules for algebraic manipulation—to let you massage the form of your equation or expression. EDIT and EXPR generally work together to let you select the highlighted portion of the equation for individual editing on the Command Line. You can then press ENTER to put this edited expression back into your equation, or CANCEL to abort the edit and return to the EW.

Try one—key in the Ideal Gas Law:

$$pV = nT\left(8.315\frac{\text{J}}{\text{mol}\cdot\text{K}}\right)$$

Now press ◀ and use the arrow keys to move the highlight around, pressing **EXPR** occasionally. Notice these things:

If the first · is highlighted, **EXPR** includes P · V.

If the = is highlighted, then **EXPR** includes the whole equation.

If the _ is highlighted, **EXPR** includes the unit object.

If the ——— is highlighted, then **EXPR** includes just the units.

If the · between mol and K is highlighted, **EXPR** includes only the denominator of the units.

Pressing **EXPR** a second time highlights only the operator (but pressing **EXPR** when a term is highlighted doesn't do anything).

SUB extracts a copy of the highlighted operator, term or expression and puts it on the Stack. **REPL** replaces the highlighted term or expression (but not operator) with the object on Stack Level 1.* These are useful when you have an often-repeated sub-expression, or when you want to modify only a small part of the equation.

*Warning: **REPL** copies, then drops the object on Level 1—it's gone. ⮫ UNDO can get it back for you, but it will also undo your last equation-editing session.

A Fourier Series Example

Here's a fun equation for playing with the PLOT functions, so key it in now as EW practice. This is the Fourier Series representation for a full-wave rectified sine wave:

$$f(t)=\frac{2A}{\pi}-\frac{4A}{\pi}\sum_{n=1}^{N_{max}}\frac{\cos n\omega t}{4n^2-1}$$

where A is the amplitude of the wave, ω is its frequency, and N_{max} is the highest harmonic you want to include (see MULTIPLOT in Chapter 9 for an application which uses N_{max}).

You should be able to enter that equation into the EW without much trouble, but here are a few reminders to help:

- Enter f(*t*) as just plain [→|] ([α][⟵][F]).
- π is [⟵][SPC].
- Use ω ([α][⟶][W])—not w—for ω (omega).
- Enter the summation as [⟶][TAN][α][⟵][N] [▶][1][▶][α][α][N][⟵][M][⟵][A] [⟵][X][α][▶][COS] [α][⟵][N][×] [α][⟶][W][×][α][⟵][T][▶][÷][4][×][α][⟵][N][y^x] [2][▶][−][1]
- Don't use [4][x²] for the $4n^2$ term. Instead, use [4][α][⟵][N][y^x][2][▶].

Work at this until you get it. Then press [ENTER] to put the completed equation onto the Stack, and name it FOYAY: [′][α][α][F][O][Y][A][Y][α][STO].

Test Your Skill

At this point, you should have worked through the EW examples in the Owner's Manual. If not, do it—now. Then here's a simple self-test:

The classical expression for the behavior of a series RLC circuit is

$$v = L\frac{dI}{dt} + IR + \frac{1}{C}\int_0^t I\,dt$$

1. Enter this equation with the EW and store it as RLC.

2. Rewrite the equation as

$$v = L\frac{d}{dt}\left(I_0 e^{st}\right) + I_0 e^{st} R + \frac{1}{C}\int_0^t I_0 e^{st}\,dt$$

   and save it as RLCEXP (for RLC EXPonential).

3. Rewrite the equation as

$$v = L\frac{d}{dt}\left(A_0 \sin \omega t\right) + A_0 \sin \omega t R + \frac{1}{C}\int_0^t A_0 \sin \omega t\,dt$$

   and save it as RLCPER (for RLC PERodic).*

Turn the page to see the EW solutions....

*There. That takes care of about 25% of your undergraduate electronics textbook. The 48 can now solve symbolically for any one of the variables, via ISOL. It can simplify the equations by solving the integral and the first derivative, and differentiate or integrate, too. But that's for another book.

Solutions

1. Press [←][EQUATION] to enter the EquationWriter, then:

[α][←][V][←][=][α][L][→][∂][α][←][T][▶][α][I][▶][+][α][I][×][α][R][+][1][÷][α][C][▶][→][∫][0][▶][α][←][T][▶][α][I][▶][α][←][T][ENTER]

You should then see `'v=L*∂t(I)+I*R+1/C*∫(0,t,I,t)'` at Stack Level 1.

Press ['][α][α][R][L][C][α][STO] to store this.

2. [←][EQUATION] enters the EW. Then press [α][I][α][←][O][←][eˣ][α][←][S][×][α][←][T][ENTER], to put the expression `'Io*EXP(s*t)'` onto Stack Level 1.

Now press [ENTER][ENTER][VAR] **RLC** [▼][CANCEL], then [◀] to the first I, and press **REPL**. Next, [▶] to the second I, and press **REPL**; then [▶] to the last I, and press **REPL** [ENTER].

On Level 1, you should now see

`'v=L*∂t(Io*EXP(s*t))+Io*EXP(s*t)*R+1/C*∫(0,t,`
`Io*EXP(s*t),t)'`

(The line breaks will be different than those shown here.)

Press ['][α][α][R][L][C][E][X][P][α][STO] to store this.

3. [←][EQUATION] enters the EW (alternatively, you could do the entire problem at the Command Line—always keep this in mind).

Then press [α][A][α][←][O][SIN][α][→][W][×][α][←][T][ENTER], to put

'Ao*SIN(ω*t)'

onto Stack Level 1.

Now press [ENTER][ENTER][VAR] RLC [▼][CANCEL], then [◄] to the first I, and press REPL. Next, [►] to the second I, and press REPL; then [►] to the last I, and press REPL [ENTER].

On Level 1, you should now see

```
'v=L*∂t(Ao*SIN(ω*t))
+Ao*SIN(ω*t)*R+1/C*∫(0,t,Ao*SIN(ω*t),t)'
```

(The line breaks will be different than those shown here.)

Press ['][α][α][R][L][C][P][E][R][α][STO] to store this.

How did you do on this little self-test?

If you need more practice, do it now, on your own—or go back over the examples in the HP User's Guide.

Other Things

Here are a few other EW tidbits to know:

Printing: If you press [ON]-[I/O] (that's [ON] and [1] simultaneously), you can print out the current EW equation on the HP 82240B printer. However, if you print the equation—in some form or another—from the Stack, you will get a better-looking printout. Here are your options:

From the EW,

- pressing [STO] saves a grob image of the equation on the Stack;
- pressing [→][""] saves a string (ASCII) version to the Stack;
- pressing [ENTER] saves the equation as an algebraic and then exits the EW.

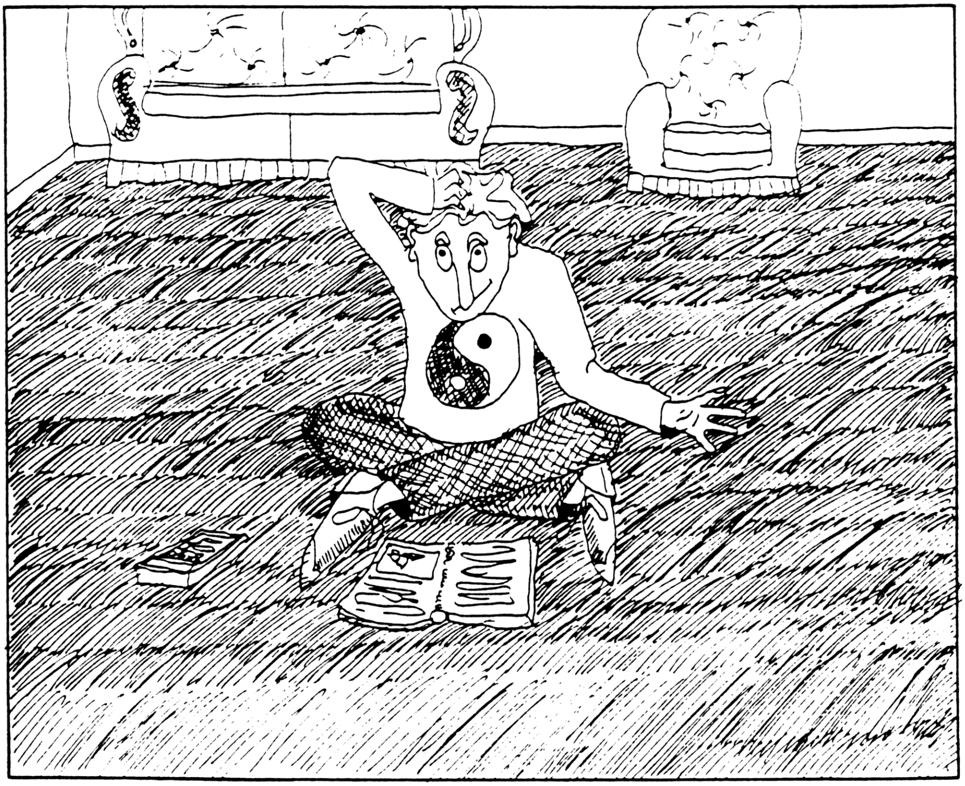
Put whichever version you want printed onto Stack Level 1, then press [←][I/O] PR1. The HP 82240B infrared printer even provides cutting lines for splicing together printouts of large grobs.*

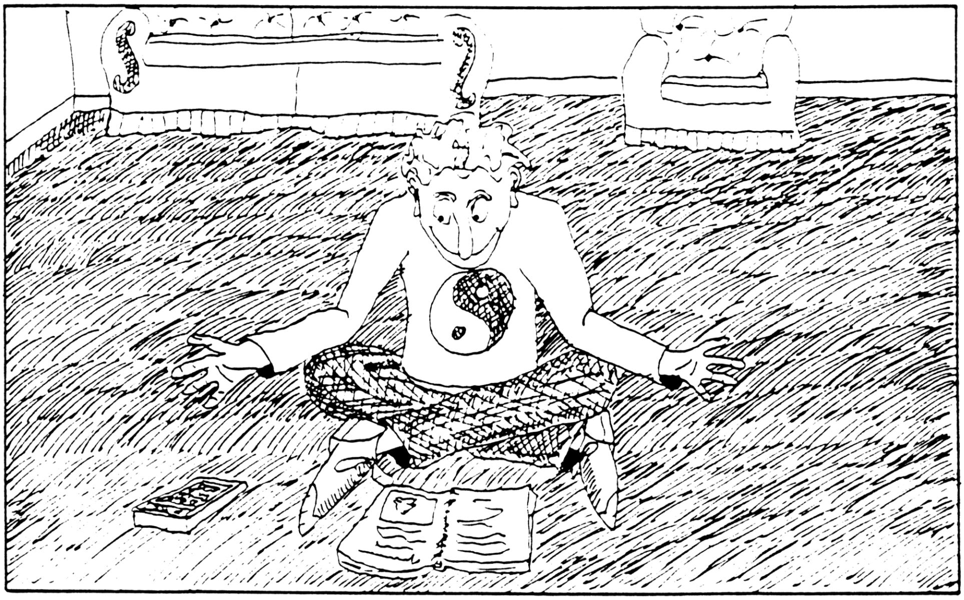
*The PCL and Epson printer drivers print large grobs and strings without the need for cutting lines. See Chapter 10 for more details on these printer drivers.

Closing Remarks

One of the best uses for the EW is to build—and later, to view—your own libraries of equations, constants and units. That way, you won't have to decipher the algebraic notation used on the 48 Command Line and in the rest of the world. A single glance in the EW will tell you everything you need to know about the equation.

Don't give up on it too easily. The entire EW concept is a new one for handheld computing, and you'll surely see it more in the future. In the meantime, remember the words of Mr. Whetstone, your high school band teacher: "Proficiency comes through practice."





3: The Solver

Opening Remarks

This is the most sophisticated Solver HP has yet produced. The more you use it, the more valuable you'll find it to be. In many cases, the problems you used to solve by writing programs can be handled more easily and quickly with the Solver.

The Solver is indeed like another programming language. In the past, you had to translate the equation(s) into a program—a list of data and operations to perform on it. But compared to this Solver, those ingenious and sophisticated programs now appear clumsy, slow, and incredibly complicated. Of course, you can still write step-by-step programs for the 48, but after reading this chapter, you may decide to save your programming skills for more worthy challenges then equations.

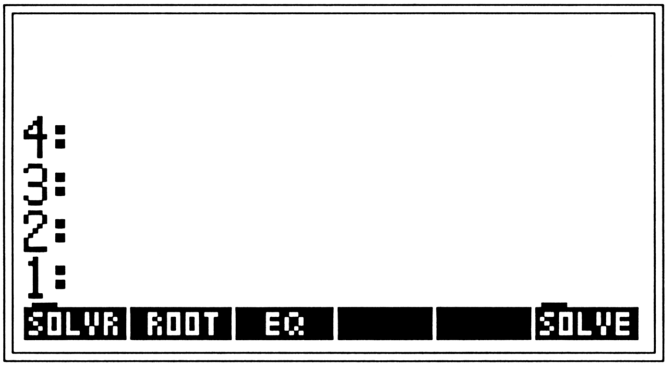
The HP 48G/GX offers 4 ways to use the Solver: programmable commands; the PLOT application; a menu-based interface (as in the HP 48S/SX); and the SOLVE EQUATION input form. All methods use the same internal routines; none is more accurate than any other. The examples here will show solutions for the menu-based interface (plain background) and the newer input-form interface (shaded background).

Preparations

First, you must create a directory for this chapter—so you don't clobber anything you may already have: Press ⮣ HOME, then type 'G.CH3' and ⮠ MEMORY DIR CRDIR VAR G.CH3 to move to this new G.CH3 directory. The menu items should now all be blank, and the Status Area at the top of the display should show: { HOME G.CH3 }

Apples and Oranges

If apples cost $.29 each and oranges cost $.89, and you have $20 to spend, how many of each can you buy? There are many possibilities, and the Solver is ideal, because it lets you play What-If: *"If I buy 3 apples, how many oranges can I get?"* So type the following equation onto the Stack, and name it 'Fruit': TOTAL=CSTA*APPLES+CSTO*ORANGES. Now press [⬅] [SOLVE] ROOT (or press [➡] [SOLVE] [ENTER] and go to page 43):



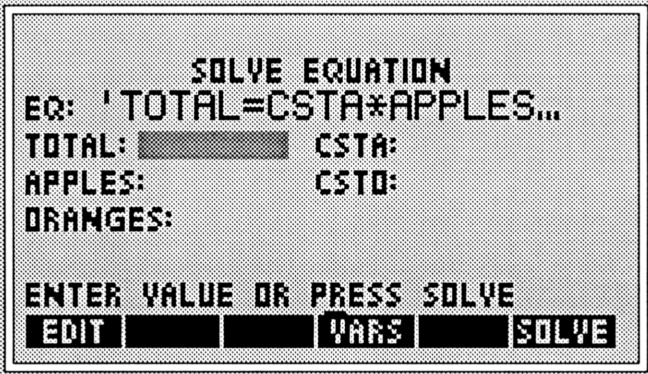
To use Fruit as the *current equation*, type 'Fruit' and press [⬅] EQ. This stores 'Fruit' into EQ, a name reserved for the *current equation*. Then press SOLVR to get into the Solver itself. Now things are simple:

- Pressing a menu key *stores* a value into a variable name;
- Pressing [➡] prior to the menu key *recalls* the value to the Stack;
- Pressing [⬅] prior to the menu key *solves* for that variable.

Press [2] [0] TOTA [.] [2] [9] CSTA [.] [8] [9] CSTO, then [➡] TOTA to recall your $20.00 to Stack Level 1. Now you're ready to solve.

If you buy 8 apples, how many oranges can you buy? Press [8] APPL [⬅] ORAN... <u>Result</u>: ORANGES: 19.87 Or, if you buy just 5 of each, how much will that cost? Press [5] APPL [5] ORAN [⬅] TOTA...
<u>Result</u>: TOTAL: 5.90 (Skip now to page 45)

With the EQ: field highlighted, press CHOOS, then press [▼] until Fruit is highlighted; press OK or [ENTER]. This will store the contents of 'Fruit' into EQ. You should see this:



Now you're all ready:

- Use the arrow keys ([▲], [▼], [◄], [►]) to *move* to a variable's field;
- [ENTER]ing data *stores* a value into that variable name;
- EDIT lets you *edit* the variable's contents;
- SOLVE *solves* for that variable;

  (Use [NXT] to see the rest of the menu.)
- RESET OK *blanks* the variable's field;
- CALC lets you *work in the Stack* before OK ing an input value (and CALC [ENTER] OK recalls the variable value to the Stack);
- TYPES is a *help* command that tells you the allowed object type(s) for that particular field.

Input the known values first: With the TOTAL: field highlighted, press [2][0][ENTER][.][2][9][ENTER][►][.][8][9][ENTER]. Press [▲][▲][NXT] CALC [ENTER] OK to recall your $20 to Stack Level 1.

Now you can play What-If:

If you buy 8 apples, how many oranges can you buy?

Use the arrow keys to move to the **APPLES:** field. Then press [8][ENTER][▶] **SOLVE**, then **INFO** to see the solution's full precision and the reason Solver settled upon that value:

```
ORANGES:
19.8651685393
Zero
```

Or, if you buy just 5 of each, how much will that cost?
Press **OK** to close the INFO box, then [▲] to **APPLES:**, press [5][ENTER] [▶][5][ENTER][▼] **SOLVE**, then **INFO** to see the solution's full precision and the reason Solver settled upon that value:

```
TOTAL: 5.9
Zero
```

You may find that this is one area where the menu interface has the advantage over the input form interface: The input form has a maximum of three lines for fields. When the number of variables exceeds three, the form shows fields in columns—up to four columns per screen—making the field names and contents hard to read. Only when the fields exceed twelve in number does the input form offer a second page (the **VARS** item changes to **MORE**).

Notice the last item in the Solver menu: **EXPR=**

If your equation is a bona fide, "grammatically correct" equation (two algebraic expressions linked by a =), **EXPR=** will solve for each side of the equation and display the results in Stack Levels 1 and 2. This is useful in cases where an exact solution may be impossible—or unbelievable—and you want to see if the left-hand side really does equal the right-hand side.

If your "equation" is really just an expression, then **EXPR=** will calculate its current value and put this at Stack Level 1.

If you see a special on oranges, say, 6 for $8.00, you can quickly see how "special" the special really is. Just set the number of apples equal to zero and solve the equation for the corresponding cost of one orange:

[0] [APPL] [6] [ORAN] [8] [TOTA] [⬅] [TOTA]

(or use the arrow keys to highlight the **TOTAL:** field and press [8] [ENTER] [▶] [0] [ENTER] [▶] [6] [ENTER] [◀] [◀] **SOLVE**)....

Some bargain—$1.33 each! Better to buy them singly at $.89!

The next two examples mimic two built-in features of the HP 48G/GX —the Ideal Gas Law (found in the Equation Library) and the Time Value of Money (part of the SOLVE application). These "non-built-in" versions illustrate the extended uses of the Solver.

The Ideal Gas Law

For the next example, take something from chemistry and physics—the Ideal Gas law: P*V=n*R*T

- P is the pressure of the gas
- V is the gas volume
- n is the number of moles of the gas
- R is the ideal gas constant, 8.315 J/mol•K
- T is the absolute temperature of the gas.

Enter this equation, using either the Command Line or the EquationWriter, so that you have 'P*V=n*R*T' on Level 1 of the Stack. Then store it into a variable: 'IdealGas' [STO]. Next, retrieve the value for *R* from the Constants Library and store it in the variable 'R'. To do this, type 'R' CONST [→][ARG][STO]. 'R' should now contain the value 8.31451_J/(gmol*K).

Press ['][VAR] IDEAL [ENTER] to put the name 'IdealGas' onto the Stack, then [←][SOLVE] ROOT to enter the SOLVE menu. Press [←] EQ to store the IdealGas equation into EQ.

(Or, press [→][SOLVE][ENTER] to get into the SOLVE EQUATION input form. Press CHOOS, then [▼] until IdealGas is highlighted. Press OK or [ENTER] to make it the current equation.)

Now use this equation to calculate the number of moles of air in a typical bicycle tire: For a 27"×1.25" tube, the volume is about 33.13 cubic inches. Use T = 70°F, and P = 80 psi (but to account for atmospheric pressure, 14.7 psi, you must use 80+14.7, or 94.7 psi).

In the menu-based Solver: [9][4][.][7][→][UNITS][NXT] PRESS PSI
[→][MENU] [P] [3][3][.][1][3] [→][UNITS] VOL IN^3
[→][MENU] [V] [7][0] [→][UNITS] [NXT] TEMP °F
[→][MENU] [T]

Solve for n: [←][N].... Result: Bad Guess(es)—an error message.

(Or, in the input-form Solver: press [▶] until the highlight is in the P: field. Then press [9][4][.][7] [→][_] [→][UNITS][NXT] PRESS PSI [ENTER]
[3][3][.][1][3] [→][_] [→][UNITS] VOL IN^3 [ENTER]
[▼][7][0][→][_][→][UNITS][NXT] TEMP °F [ENTER]

Solve: [▲][▲] SOLVE ... Result: Bad Guess(es)—an error message.)

When working with unit objects, you must store an initial guess for the variable you're solving for.* So, press [1][→][UNITS] MASS [←][PREV] MOL [→][MENU] [N]. Now try n again: [←][N]. Result: n: 1.10_mol

(Or, in the input form, press OK to clear the error message, then [1] [→][_] [→][UNITS] MASS [NXT][NXT] MOL [ENTER]. Now try solving for n again: [◀] SOLVE ... INFO Result: n: 1.09629984511_mol)

Well, you got a result. Too bad it's wrong. *"What?"* Yep, it's wrong....

*If you get this Bad Guess(es) error while solving for a unit object, press [←][REVIEW] to get a summary of the contents of each variable and of the current equation. Often, you'll have forgotten to press [←] when *solving* for the unknown, thus inadvertently *storing* some (incorrect) object there instead. **Remember:** Press the unshifted key only to *store* a value in a named variable! [←] *solves* for the variable, [→] *recalls* the variable contents to the Stack.

For the Solver to ignore units entirely, you must PURGE all variables named in the equation and re-enter the Solver. That means you're likely to clobber the Gas Constant, R, which is a variable to the machine. Later in this chapter you'll see how to keep the Gas Constant safe from harm.

This isn't the fault of the Solver, but stems from a quirk in the way ***temperature units*** are converted. You can read more about this quirk on page 10-10 of the User's Guide. The Solver makes no errors converting other types of units, but it is often suckered into making *relative* instead of *absolute* temperature conversions. And it doesn't tell you it's doing this—it just gives you the wrong answer. To be safe, always convert temperatures to Kelvins before using them with the Solver.

So, in the menu-based Solver, recall the temperature ([→][T]) and convert it to Kelvins ([←][UNITS][UBASE]) and then recalculate n:

[→][LAST MENU][T][←][N] Result: n: 0.14_mol

(In the input-form Solver, press [▼][NXT][CALC][←][UNITS][UBASE][←][CONT][OK][▲][NXT][SOLVE].... Result: n: 0.144884530289_mol)

Now then, for subsequent calculations, if you know that the previous value of the variable has the correct units, then you can just store a numeric value on top of it, and it will assume those same units.

Example: Find out how many cubic inches of air at atmospheric pressure are compressed into that bicycle tire.

Atmospheric pressure is 14.7 psi, so in the menu-based Solver, press [1][4][•][7][P] to store the value in P—using the psi units from last time (the correct units appear on the Status Line). Now press [←][V] to find the volume of uncompressed gas.... Result: V: 213.43_in^3

The input form isn't smart enough to preserve units. You must [▲] to the P: field and [1][4][•][7][→][_][→][UNITS][NXT][PRESS][PSI][ENTER][SOLVE].

The Time Value of Money

Next up—for all you finance wizards—is the Time Value of Money equation.

$$0 = PV + PMT\left[\frac{1-(1+I)^{-N}}{I}\right] + FV(1+I)^{-N}$$

where

PV	is the Present Value of the loan or investment.
PMT	is the periodic (monthly, annual, ...) PayMenT.
FV	is the Future Value of the loan or investment.
N	is the Number of periodic payments or compounding periods.
I	is the Interest rate per compounding period.

Build this equation using the EW or the Command Line (the EW is easier) and put it onto Stack Level 1. Then name it—type 'TVoM' STO. This TVoM equation is a mainstay of all business calculators, but it comes in handy even for engineers trying to buy houses, figure out their IRA's, or calculate the balances on their student loans.

Example: You want to buy a \$95,000 home. You have \$10,000 for the down payment, and you want to finance the rest at 8.0% for 30 years.

For the menu-based Solver, press ' VAR TVOM ENTER, to put 'TVoM' onto the Stack, then ⬅ SOLVE ROOT to enter the SOLVE menu. Press ⬅ EQ to store the TVoM equation into EQ, then press SOLVR.

For the input-form Solver, press ➡ SOLVE ENTER, then CHOOS and use the arrow keys to highlight the TVoM equation. Press ENTER or OK to select it.

Now, the PresentValue is the money you're going to receive right now, $85,000 (OK, you may never see it, but the bank is technically giving it to you to give to the seller). The Future Value is what you'll owe at the end of the mortgage term—i.e. nothing (hopefully). So press [8][5][0][0][0] [PV] [0] [FV] (or, with the input form: [8][5][0][0][0][ENTER][▶][▶][▶][0][ENTER]).

Since this is a 30-year loan with monthly payments, the term, N, is 30×12, or 360. The monthly interest rate is 0.08÷12, or .0066667. To enter the term and the monthly interest, press [3][6][0] [N] [·][0][0][6][6][6][6][7] [I] (or [▼][▼] to the I: field, then [·][0][0][6][6][6][6][7][ENTER][3][6][0][ENTER]).

Now just press [⬅][PMT] (or [▲][▲][▶] [SOLVE]) to find that your monthly payment is $623.70. The minus sign means that it's money subtracted from your pocket.

Notice that both the Ideal Gas and the Time Value of Money equations use variables named N or n.* So after you've used each equation, you'll see not one but two [N] labels in your VAR menu. You can press [➡][VIEW] from either the VAR menu or the menu-based Solver variable menu to see which is which (or—if it really bothers you—store the two equations in separate sub-directories inside the G.CH3 directory).

Anyway, since you've used a capital N for one and a small n for the other, the Solver can tell them apart, and that's the main thing. But if you use the identical variable N in two separate equations in the same directory, ***beware***—especially if either uses a unit object: You'll get all sorts of nasty messages until you purge the unit-object N.

*Note also that the built-in TVM application ([➡][SOLVE][▲][ENTER]) is much faster than either of the 2 Solvers with 'TVoM'. But don't erase TVoM yet; it will prove useful in the next few pages.

Customizing the Solver

Keeping the Gas Constant a Constant

In your IdealGas equation, you just *know* that sooner or later, someone will try to check the value of the Gas Constant by pressing [R] instead of (→)[R] (or by pressing SOLVE with the R: field highlighted). So why not take it off the Solver menu (or input form) altogether, preventing access to it there? Yes—you can do that: You can design your own variable menu for use with your equations, omitting variables that don't vary—like the Gas Constant.

To do so with the menu-based Solver, just put your equation into a *list*:* { 'p*V=n*R*T' { p V T n } }. In this list, the equation comes first, followed by a list of the variables you *do* want on the menu. You can put those variables in any order—say, with the most frequently used variables first (handy if there are more than six variables).

Put this list onto Stack Level 1, and then type 'IdealGas2' (STO) and 'IdealGas2' 'EQ' (STO) to name it and make it the current equation. Now press (→)(SOLVE) ROOT SOLVR to see your customized menu that hides [R]:

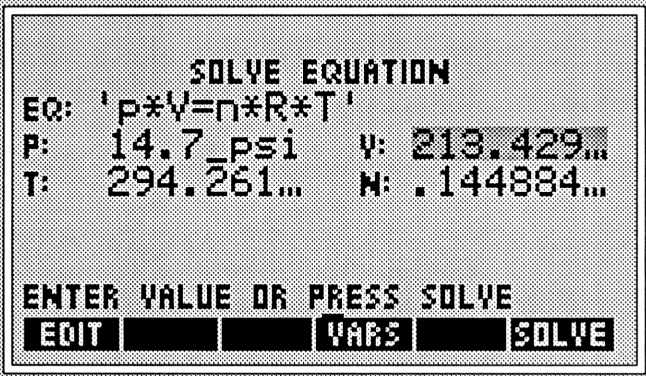
[P] [V] [T] [N] EXPR= [ ]

*The menu-based Solver accepts any of the following as an "equation:"

- an algebraic equation or expression (example: 'p*V=n*R*T');
- a real-number constant (example: 8.315);
- a program that uses only global (no local) variables to return exactly one result to Stack Level 1 (example: « p V * n R * T * - »);
- a list of one or more of the above (example: { 'p*V=n*R*T' 8.315 'T=To+ΔT' });
- a list of one or more of the above, plus an extra item—a list usable as a CST menu (example: { 'p*V=n*R*T' { p V T n } }).

With the input-form Solver, you can customize the menu by pressing VARS EDIT and editing the list to look like this: { p V T n } (Note that during your editing, you can use your own global VAR menu to save yourself some typing.)

When you finish, press ENTER ENTER or ENTER OK to return to the input form. The R: field has now disappeared from the input form:



Running Programs from Inside the Solver

The variable Solver menu list structure can also include *executable programs* (note: the input-form Solver cannot do this). In the Ideal Gas law, for example, suppose you're using your 48 to monitor the amount of gas in a pressurized reactor. Volume and temperature are constant, and you can calculate the quantity of gas from the measured pressure. Hypothetically you'd have a program, `READP`, to read a pressure sensor and put the value onto the Stack. To simulate that here, just use `READP` (Checksum: `# 45658d` Bytes: `37.5`), a constant: `« 5_atm »`

So replace `P` in your variable list with a list of this form: `{ "`*menu label*`"` `{ «`*prg1*`» «`*prg2*`» «`*prg3*`» } }`. The `"`*menu label*`"` is the label that will appear on the menu; `«` *prg1* `»` is the program that its *unshifted* selection will execute. `«`*prg2*`»` and `«`*prg3*`»` are the programs that the ⬅- and ➡-shifted selections of this item will execute, respectively (but these are optional; you can ignore the shift keys and simplify your list to `{ "`*menu label*`" «`*prg1*`» }`).

Let the unshifted menu key be the call to `READP`. Therefore `«`*prg1*`»` will be `« READP DUP 'P' STO 1 DISP 1 FREEZE »`. This reads the pressure, stores it into the variable name `'P'`, and displays it in the Status Area—just as the Solver would do for a value that you keyed in. Then `«`*prg2*`»` will be an empty ("do-nothing") program, `« »`, since you don't plan to *calculate* the pressure. And `«`*prg3*`»` will be `« P »`, to recall the value in `P` to Stack Level 1—just as any other ➡-ed variable key would do in the Solver.

Thus, the list to replace `P` becomes `{ "P" { « READP DUP 'P' STO 1 DISP 1 FREEZE » « » « P » } }`.

Now type VAR ↱ IDEAL ▼ to edit a copy of `IdealGas2`. When you finish, your list should look like this:

```
{ 'p*V=n*R*T' { { "p" { « READP DUP 'p' STO 1 DISP
  1 FREEZE » « » « p » } } V T n } }
```

Press ENTER to put it onto the Stack. Store it as `'REACTOR'`. Now specify `REACTOR` as the current equation (' REACT ENTER ↰ SOLVE ROOT ↰ EQ), and start the Solver (SOLVR). The display looks a little different, as shown here:

```
REACTOR: { 'p*V=n*R* …
────────────────────────
4:
3:
2:
1:
  P  │  V  │  T  │  N  │EXPR=│
```

If you ↱ VIEW the variables, you'll see only `V`, `T` and `n`, since `p` is no longer a Solver variable (notice that the P item is white- on-blue, instead of the blue-on-white). This is how the 48 helps you differentiate between variables and programs in the menu. Try the unshifted and shifted P key to see how it works....

The unshifted key displays `'5_atm'` in the status line (and notice that with a slightly more elaborate program in the variable list, you could make it display `p:  5_atm`).

The ↰ key does nothing (as you intended), and the ↱ key puts the value of `'p'` onto the Stack.

A More Versatile TVoM Equation

The next thing to change is your TVoM equation a little bit (look back on page 49 to see the original). You customize with the Solver to make the equation easier to use (note: the input-form Solver cannot do this):

- First, include a factor to account for when the payments are made (i.e. the beginning or end of the month). This factor is a multiplier to the PMT:

  $$0 = PV + (1 + I * Begin?)PMT\left[\frac{1-(1+I)^{-N}}{I}\right] + FV(1+I)^{-N}$$

  Begin? will be a true/false variable, with a value of 1 if payments are made at the beginning of the month, or 0 (the default) if payments are made at the end of the month.

- Next, change all occurrences of I to I/100. This way, you can enter 5% interest as [5] [I], instead of [.] [0] [5] [I].

- Finally, to accommodate interest compounded quarterly or monthly, introduce a variable called Per (periods per year)—the number of compounding periods in a year (12 for monthly payments, 4 for quarterly, 1 for annual, etc.).

  Thus, since N is the number of years, N*Per will be the total number of periods—and payments. And I/(100*Per) will be the interest per compounding period.

By now, the TVoM equation is a monster. In textbook notation, it is:

$$0 = PV + \left(1 + \frac{I * Begin?}{100Per}\right) PMT \left[\frac{1 - \left(1 + \frac{I}{100Per}\right)^{-NPer}}{\frac{I}{100Per}} \right] + FV\left(1 + \frac{I}{100Per}\right)^{-NPer}$$

Or, in algebraic notation, it is:

```
0=PV+(1+I*Begin?/(100*Per))*PMT*((1-(1+I/(100*Per))^
-(N*Per))/(I/(100*Per)))+FV*(1+I/(100*Per))^-(N*Per)
```

Yep, that's right: *You* get to build this, using whichever method you wish—EW or Command Line—to edit the current version of TVoM (quiz: which method would *you* rather use?). Go...

Finished? OK, now if you were to store this equation (don't do it yet), the Solver would give you seven variables to juggle, plus the **EXPR=** item besides. But you can make the equation a bit more friendly, by attaching this variable list to it:

```
{ N I PV PMT FV { "SETUP" { « VIEWP » « MQA »
« BEGEND » } } Per Begin? }
```

No—you don't need to re-enter the equation. Using your list-building process, just put the current monster TVoM equation on Stack Level 2, the variable list on Level 1, then press [2] [PRG] **LIST** **→LIST**...and save the whole thing in 'TVM2'.

You now have a full-fledged Solver "program." Type 'TVM2' STEQ (STEQ is the same as 'EQ' [STO]). Then start the Solver.

You should get a display like the one below.

```
TVM2: { '0=PV+(1+I*Be…
─────────────────────
4:
3:
2:
1:
 N | I | PV | PMT | FV | SETUP
```

This version of TVoM is more "friendly" than the first one: On the first page of its two-page menu are the commonly-used variables, plus a SETUP menu key. SETUP serves three functions.

Unshifted SETUP will run a program called VIEWP (for "view parameters"), which displays the current settings of the variables Per and Begin?: If Per has a value of 1, 4 or 12, the first status line will show ANNUAL, QUARTERLY or MONTHLY, respectively; if Per has any other value, say 5, the first status line will show 5 PERIODS/YEAR. And, if Begin? contains zero, the second status line will show PMTS AT END; otherwise it will show PMTS AT BEGINNING.

⬅ SETUP will run a program called MQA to rotate the Solver through monthly, quarterly or annual payments. And ➡ SETUP will run the program BEGEND, which toggles the value of Begin? between 1 and 0. Both MQA and BEGEND call VIEWP to update the display.

The second page of the variables menu gives you direct access to Per and Begin?, so you can set bimonthly payments or calculate interest compounded daily—when Per must have a value other than 1, 4 or 12.

Here are the three programs, VIEWP, MQA and BEGEND:

VIEWP

Checksum: # 14516d

Bytes: 415.5

```
« IFERR 'Per' RCL
  THEN DROP MQA 'Per' RCL
  END
  → per
  « IF 'per==4'
    THEN "QUARTERLY"
    ELSE
      IF 'per==12'
      THEN "MONTHLY"
      ELSE
        IF 'per==1'
        THEN "ANNUAL"
        ELSE
          Per IP →STR
          " PERIODS/YEAR"
          +
        END
      END
    END
  »
  1 DISP
  IFERR 'Begin?' RCL
  THEN DROP BEGEND
    'Begin?' RCL
  END
  → begin
  « IF 'begin'
    THEN "PMTS AT BEGINNING"
    ELSE "PMTS AT END"
    END
    2 DISP 1 FREEZE
  »
»
```

MQA

Checksum: # 17323d

Bytes: 164

```
« IFERR 'Per' RCL
  THEN DROP 1
  END
  → per
  « IF 'per==1'
    THEN 4
    ELSE
      IF 'per==4'
      THEN 12
      ELSE 1
      END
    END
  »
  'Per' STO VIEWP
»
```

BEGEND

Checksum: # 34006d

Bytes: 90.5

```
« IFERR
    'Begin?' RCL
  THEN DROP 0
  ELSE NOT
  END
  'Begin?' STO VIEWP
»
```

Linking Equations: Solving Several at Once

For this next topic (note: the input-form Solver cannot do this), go back to your "Apples and Oranges" equation. Suppose you've borrowed your nephew's little red wagon—which can hold only 50 pounds—to haul your groceries home. How many apples and oranges can you afford—and still be able to get them home?

Hmm... to avoid exceeding either your budget or your wagon's capacity, you now have two problems. The first is already taken care of by your existing `Fruit` equation:

```
TOTAL=CSTA*APPLES+CSTO*ORANGES
```

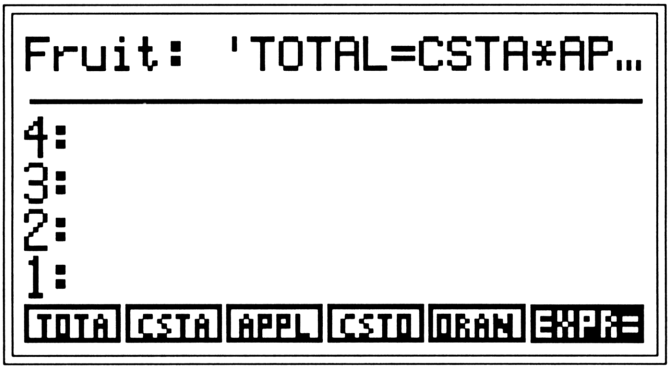
But now there's this new equation (key it in and store it as `'Wagon'`):

```
LOAD=WT.A*APPLES+WT.O*ORANGES
```

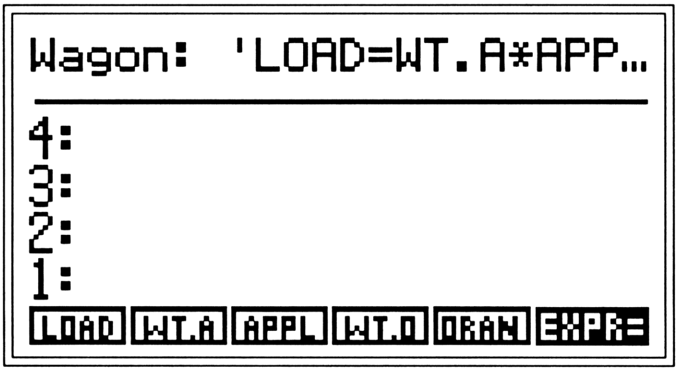
The Solver lets you *link* equations in order to solve several at once. To use this feature, you combine the equation names in a list and give the list a name.

So create the list `{ Fruit Wagon }`. ENTER it and store it as `'Shopping'`. Then type `'Shopping' STEQ` to make this list the current equation.

Now press ⬅ SOLVE ROOT SOLVR to start the Solver. Your display will look like the one below.



Notice that the Solver is ready to work on the first equation in the list, 'Fruit'. But press NXT and notice the new menu label: NXEQ. Press NXEQ now to see what it does.



Get the idea? If you have several equations in your list, such as { EQ1 EQ2 EQ3 EQ4 }, NXEQ bumps EQ1 to the last place in line, moves all the other equations up one place, { EQ2 EQ3 EQ4 EQ1 }, and sets up the Solver to work on EQ2.

Now press NXT NXEQ a few times until the Solver returns to 'Fruit'. It's time to test all this!…

Press [→][VIEW] to see that each variable in 'Fruit' has an assigned value (the values in the examples at the beginning of this chapter should still be there: CSTA should contain 0.29, and CSTO should contain 0.89).

Now press [NXT] [NXEQ] to go to the 'Wagon' equation. Apples are about three to a pound, so press [.][3][5] [WT.A] to enter an apple's weight. Now imagine some big, juicy oranges—about a pint each: Enter [.][5] [WT.O]. Solve for the total weight by pressing [←][LOAD]....

For another variation on the problem (and to further demonstrate the "What-If?" nature of the Solver), how much would it cost to fill your wagon with an equal weight of apples and oranges?

Press [2][5] [LOAD] [0] [ORAN] [←][APPL].... <u>Result</u>: APPLES: 71.

Then press [0] [APPL] [←][ORAN].... <u>Result</u>: ORANGES: 50.

Then [7][1] [APPL] [5][0] [ORAN], then [NXT] [NXEQ] to get back to the costing equation, and [←][TOTA].... <u>Result</u>: TOTAL: 65.09

That's the cost of a wagonful of equal weights of apples and oranges.

Another good example of a set of linked equations is this set for linear motion:

$$v = v_0 + at$$

$$x = x_0 + \frac{1}{2}(v_0 + v)t$$

$$x = x_0 + vt + \frac{1}{2}at$$

$$v^2 = v_0^2 + 2a(x - x_0)$$

Enter these four equations and store them into `'M1'`, `'M2'`, `'M3'`, and `'M4'`, respectively.

Then store the list `{ M1 M2 M3 M4 }` into `'MOTION'`.

Now you can solve for x, x_0, v, v_0, a and t, if you know any three of them: You store the three (or more) known values and then use **NXEQ** and ⬅ REVIEW to cycle through the equations, solving each one in turn, until there are no more undefined variables.

Solving with linked equations does have some limitations:

- The Solver won't search for undefined variables nor define or solve for them automatically. For example, if you define everything but the variable ORANGES in the Fruit equation—so that its value is implied—then solve for LOAD in the Wagon equation, you'll still get the error message: Undefined Variable(s).

- In some iterative methods using more than one equation, the order of solving the equations determines whether the solutions converge or diverge. The Solver cannot help you avoid diverging solutions.

Fortunately, there are two workarounds for these limitations:

- Since the Solver is programmable, you can automate much of the process for use in analysis and design of iterative solutions.

- The Multiple Equation Solver application (⬅ EQ LIB MES) can solve for all the unknowns in a system of equations, given the necessary minimum number of independent variables.

For most of your needs, the normal interactive Solver is sufficient, but if you need more, stay tuned for more information on programmability—and on the MES!

Using the Solver on Ill-Mannered Functions

Earlier versions of the Solver accepted only “well-mannered” functions; you couldn’t use it with square waves, step functions, or other real-world functions. For those, you had to resort to programming.

Well, no more. The 48’s Solver can handle it all. The key to making it work is to *think ahead*. Plan out exactly how you’ll approach your problem from the start. With planning and practice, you can now make the Solver do what used to require a lot more programming.

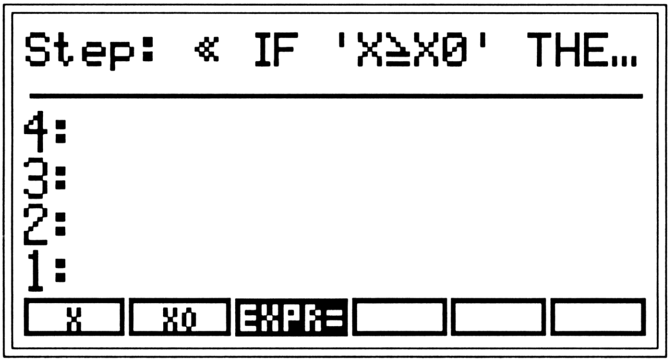
Try it: For the step function $y = \begin{cases} 1 \text{ where } x \geq x_0 \\ 0 \text{ where } x < x_0 \end{cases}$, write a simple

program:

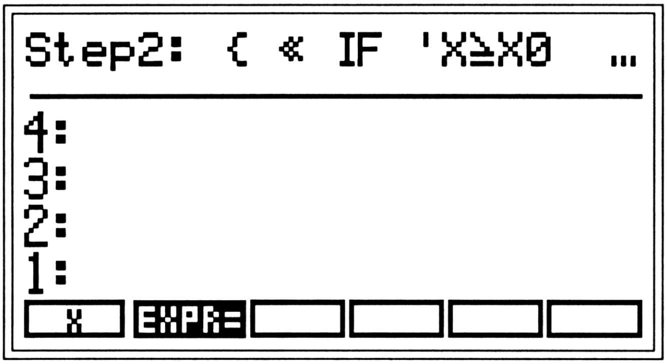
```
« IF 'X≥X0'
  THEN 1
  ELSE 0
  END
»
```

Next, name the program, say, 'Step'
(Checksum: # 29349d Bytes: 51).

Press 'Step' STEQ (ENTER) (⬅)(SOLVE) ROOT SOLVR, and see:



Just as with an algebraic equation, the Solver examines the program, extracts variable names and builds a variable menu from those names. And you can "lock in" values by specifying a variable list and omitting the fixed values. For example, for the menu-based Solver, change `Step` to `Step2: { « IF 'X≥X0' THEN 1 ELSE 0 END » { X } }`
Now x_0 is omitted from the menu, so the menu-based Solver appears as



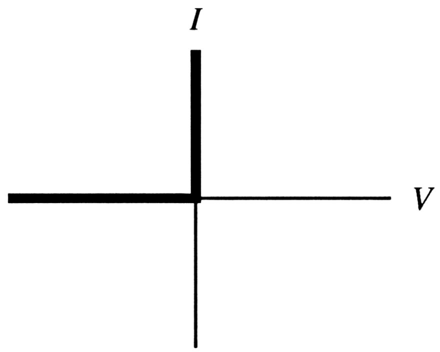
In the input-form Solver, press [→][SOLVE] [ENTER] CHOOS, then [▼] until `Step` is highlighted, then [ENTER] or OK. To omit `X0:` from the form, press VARS EDIT [▶][▶][▶][DEL][DEL][DEL][ENTER][ENTER]

Of course, this function is ill-mannered; it can't be differentiated: Trying to do so onto the Stack with [→][∂] gives a `Bad Argument Type` error; trying it in the Plotter via FCN F' gives `Invalid EQ`. Even rewriting the program as a user-defined function doesn't help:

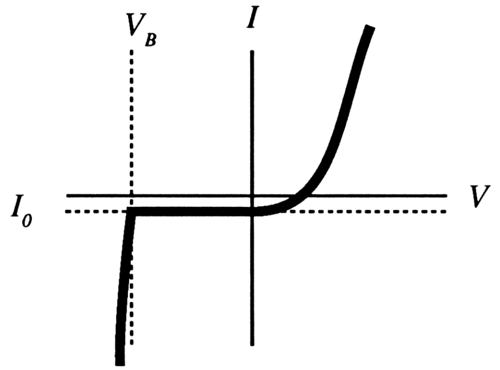
```
« → x x0 « IF 'x<x0' THEN 0 ELSE 1 END » »
```

This still isn't written as an algebraic, and the 48 can differentiate only algebraics. But also in the [PRG]-BRCH menu—on the very last page—is IFTE, which *can* be used in algebraics. For example, the above step function can be rewritten simply as `IFTE(X<X0, 0, 1)`. And IFTE can be differentiated and integrated—like a constant coefficient that passes transparently through the differentiation or integration.

One problem that has vexed engineers for years—and led to many ingenious programs—is how to model a real diode. A diode is a kind of electronic "One Way" sign, ideally allowing infinite current flow in one direction (called *forward bias*) and zero current flow in the other direction (called *reverse bias*). Here's a plot of voltage vs. current for an ideal diode:



Well, a real, solid-state diode isn't quite that good:



Typically, the transition from forward to reverse bias takes place at *about* $V = 0$ volts. Under reverse bias ($V < 0$) the current is fairly constant at $I_0 = 1$ picoampere to 1 microampere. Under forward bias ($V > 0$), the diode current follows this relation:*

$$I = I_0\left(e^{\frac{V}{.0259\text{ volts}}} - 1\right)$$

*This assumes a constant temperature of 300 K. A good electronics text will give you temperature-dependencies for both I and I_0. Also, the Equation Library offers a more rigorous equation.

If the reverse bias voltage exceeds a given value V_B, or *breakdown voltage*, then the diode loses all effectiveness and becomes essentially a short circuit—current is very high.

So a good diode equation should model all three areas of the *V-I* curve, and it should be continuous. It can be done using two nested IF...THEN...ELSE commands in a program—or two nested IFTE functions in a single equation:

```
I=IFTE(V<Vb,1E99*V,IFTE(V>0,Io*(EXP(V/.0259)-1),-Io))
```

Type this in and call it DIODE (Checksum: # 44495d Bytes: 127). This matches the diode model very well and maintains a continuous function through the three regions of forward bias, reverse bias and breakdown.

For example, a typical diode has these characteristics:

$$I_0 = 10^{-6}\ \mathrm{A}$$
$$V_B = -10\ \mathrm{V}$$

Storing these two values completely defines your diode—and since the variables are naturally arranged in the variable menu, you don't even need to create a variables list!

The Care and Feeding of derFN

It may seem strange to have a section on functions in the middle of the Solver chapter, but such considerations of ill-behaved functions are important for using the Solver inside the Plotter—coming up next.

In many cases you will find it easier to differentiate an equation and solve for the variables in the resulting first-derivative equation. But if your original equation contains several functions for which the 48 cannot find a derivative, it will indicate this by creating a dummy derivative and listing the variables available to solve the problem.

Press ['][MTH] **REAL** [NXT] **ABS** [α][X][ENTER], then ['][α][X] [ENTER] [↱][∂]. You'll get the algebraic function 'SIGN(X)'. Now press ['][α][X][ENTER] [↱][∂] again, to get the function 'derSIGN(-3,1)'.

"Where did this come from?" you may well ask.

To answer your question, repeat the calculation, but this time create the algebraic '∂X(SIGN(X))' and press [EVAL]. This time you get: 'derSIGN(X,∂X(X))'. Now you can see what happened in the first case: instead of stopping at a symbolic representation of the differential, the 48 went on and completely evaluated the variables, replacing X with -3 (currently stored in X) and calculating the derivative of a constant (1). Press [EVAL] again to see this substitution.

Moral: If you want to completely evaluate a derivative in one step, use the Stack method. For symbolic representation of the derivative or for *stepwise differentiation*, include the derivative into your algebraic and evaluate to the level you need. See your HP UG (pages 20-9 through 20-10) for more details.

Now, next question: What is this derSIGN all about?

This is the 48's way of saying "I don't know how to differentiate the function SIGN(X), but I'll use these placeholders for X and dX until you show me how the derivative should be defined."

You'll probably face the same problem with many of your own user-defined functions. When you use FCN F' on one of these functions, if the 48 can't find a numerical approximation to the derivative, it will give you a nasty message and give up.

You can avoid this by trying all your derivatives beforehand. If you find a derFN somewhere in your differentiated expression, then you should consider how the function should be differentiated.

For example, with SIGN(X), it's obvious that 'derSIGN(X,dX(X))' is zero everywhere but at $x = 0$, where it is infinitely large. So you could create the function « → x dx 'IFTE('x==0',1E499,0)' » and store this as 'derSIGN'. When you evaluate derSIGN after defining it, you'll get a result of 0 (assuming -3 is still stored in X).

SIGN is a *unary* function; it acts on only one argument. By contrast, percent is a *binary* function—two arguments. For example, the derivative of '%(X,Y)' with respect to Z is: 'der%(X,Y,∂Z(X),∂Z(Y))'

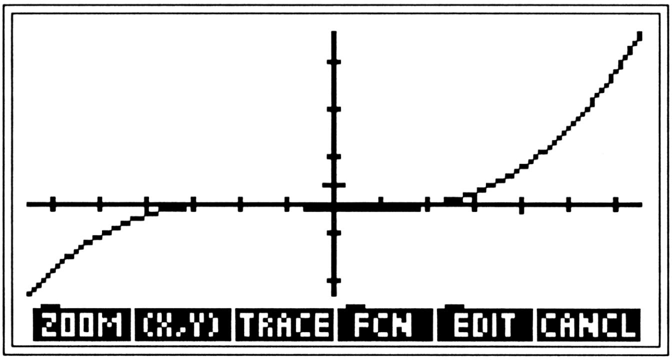
Page 20-11 in the User's Guide gives a solution for 'der%'. Work out other *user-defined derivatives* in the same manner.

Note: This also works with the DIFFERENTIATE input form (→ SYMBOLIC ▼ ENTER), but → ∂ is faster and takes fewer keystrokes.

Using the Solver Inside the Plotter

The 48 Solver really shines inside the Plotter application, where it's even more versatile than in its stand-alone form. For example, create a 3rd-degree polynomial: `'X^3+2*X^2-5*X-6'`. Store this into EQ (and press ['][α][X][←][PURG] to ensure that `X` won't get in the way).

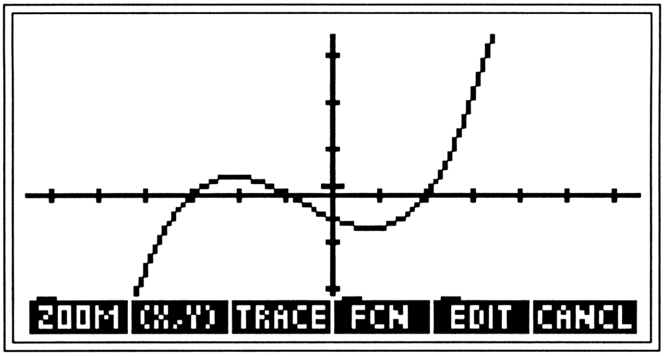
Now, a good mathematician would be able to tell by inspection that it's a cubic curve from lower left to upper right, with a "dipsy-doodle" around the x-axis that crosses the axis 3 times (you can tell that it has 3 real roots, right? ... right?...).* Prove it: Press [→][SOLVE] to get the **PLOT** input form. Press [▼][α][X][ENTER] to put `X` into the **INDEP:** field. Press [▼][◀] **✓CHK** to enable the **AUTOSCALE** option, then **ERASE** **DRAW**.



No big deal, right? And you can use the **ZOOM** commands to get to the interesting part of the curve. The menu in the display is the PICTURE menu (you saw this briefly in Chapter 1). Press [◀][◀] to find the graphics cursor. Then press and hold [◀] until the cursor is above and to the left of the leftmost root. Press **ZOOM** **BOXZ** to mark the point. Now press and hold [▶] to move the cursor past the rightmost root, then press and hold [▼] until the cursor is about four pixels below the *x*-axis.

*A good mathematician could also work out these roots in his/her head—or maybe use the polynomial root-finder (via [→][SOLVE][▼][▼][ENTER]), but that's a topic for another book....

Now press ZOOM. The Plotter will redraw the function:



Reminder: Press [CANCEL][↰][◄][CANCEL][↰][◄] to toggle between the PLOT input form and the current plot. If you're not using the input form, press [CANCEL][◄][CANCEL][◄] to toggle between the Stack display and the current plot. Pressing [◄] sends you from an idle Stack display (i.e. no Command Line or interactive Stack) to the graphics display. Pressing [CANCEL] returns you to the Stack display. Also, pressing [↰][◄] will go to the graphics display from *almost anywhere*; the [◄] shortcut is worth remembering.

Press FCN to see the Solver and other *function analysis tools*. The Solver is built into the first two of these menu items: ROOT and ISECT.

With ROOT (as described in Chapter 1), you use the [▲][▼][◄] and [►] keys to position the graphics cursor near where the curve crosses the x-axis, then press ROOT.

Try finding the three roots of the polynomial: –3, –1 and 2....

(When the menu disappears, press [NXT] or [–] to get it back.)

There are some significant differences between the way that the Solver application works in its stand-alone form and the way it works within the **ROOT** operation:

- The stand-alone Solver solves for any variable you want, but the **ROOT** version solves for the value of the independent variable which makes the dependent variable go to zero. To solve for a different variable using **ROOT**, you must change independent variables from the PLOT input form or by typing '*varname*' [←][SOLVE] **PPAR** **INDEP** from the command line.

- Another difference is that the Solver will display *intermediate results* for you if you press any button except [CANCEL] while it's thinking ([ENTER] is probably the easiest key to find while you're watching the display). The Solver tells you, with a short message, how it arrived at the answer, and it puts the numeric result onto the Stack with the variable name for a tag.

  ROOT, by contrast, doesn't give you intermediate results or a message, but it does position the cursor exactly on the intersection (useful for subsequent operations like **SLOPE**). Also it puts the result onto the Stack as a real number—with the tag Root: —and displays the numeric result on the graphics display until the next keystroke.

- If the function does not have a real root, such as with 'Y=X^2+2', the Solver finds a local *extremum* (minimum or maximum). It then puts that *x*-value onto the Stack and the Extremum value in the Status line.

  ROOT puts the closest approximation onto the Stack and flashes EXTREMUM on the graphics display, positioning the cursor at the extremum of the function and displaying the numeric result.

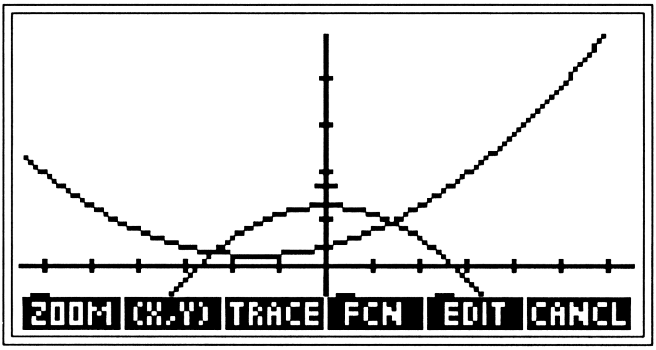
- Note that in some cases (as in the 'Y=X^2+2' example cited here), the Solver and ROOT will return slightly different values of X for the extremum.

- ROOT can return results that are difficult or impossible to coax out of the Solver. If the Solver's answers don't make sense, enter the Plotter, declare your unknown as the independent variable, and solve for it graphically. And note that if EQ contains a *list* of two or more equations, then the Plotter will plot all the functions, but ROOT will find the roots of the *first* equation, and ISECT will find the points of intersection between the *first two* equations in the list.

The majority of equations you'll plot have an isolated variable on the left of the equals sign—or no equals sign at all. But you may occasionally have an equation such as this:

$$15 - 2x^2 = x^2 + 3x + 5$$

The Plotter treats this equation as two separate algebraics, separated by an equals sign; it plots them both.

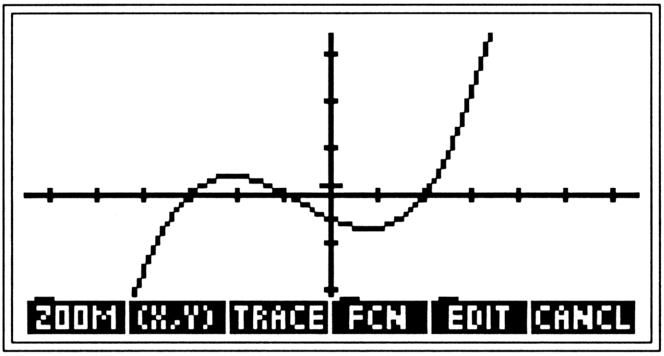


ROOT finds only the point where the right hand side of the equation equals zero. In order to find the roots of the equation, you must use **ISECT** to find the point(s) where the two function plots intersect.

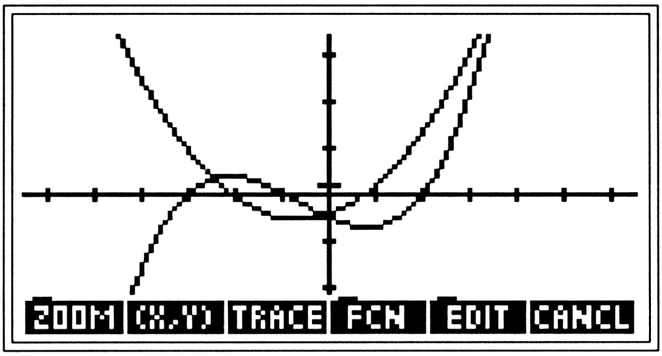
Of course, you can get around this by subtracting the left side from the right side to get an equation of the form `'0=fn(X)'`, but sometimes you do want to see both sides of the equation separately.

Look at some other items on the FCN menu. At first glance, you might think that **SLOPE** and **F'** do the same thing, but not quite: **SLOPE** computes the slope of the function *at the cursor location* (though the cursor need not be right on the curve; it will "home in" on the curve once the result is computed and displayed).

F' computes *and plots* the derivative of the equation *at every x-value* in the plot range. It also adds the equation for the first derivative to the list in EQ (or, if EQ contains a single equation, then **F'** creates a list with the new equation inserted at the start of the list). To see this, use the PLOT input form and **ZOOM** **BOXZ** **ZOOM**, as on pages 70-71:



Now, pressing **FCN** (NXT) **F'** adds a parabola to the display, since the first derivative of a cubic function is a quadratic:

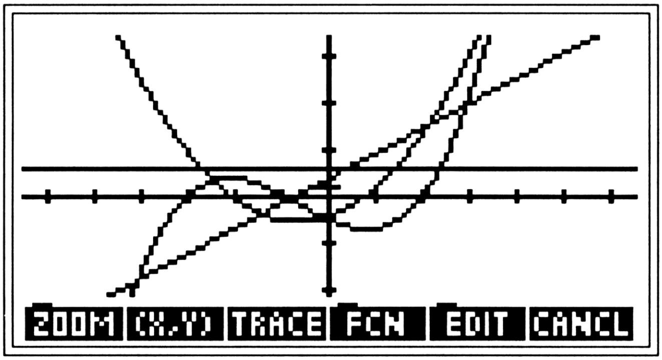


And now EQ is: `{ '3*X^2+2*(2*X)-5' 'X^3+2*X^2-5*X-6' }`

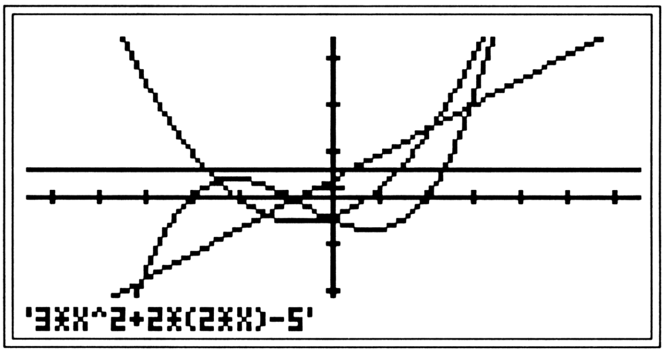
Press **FCN** [NXT] **F'** twice more (give each press time to draw).... The list in EQ becomes

```
{ 6.00 '3*(2*X)+4'
          '3*X^2+2*(2*X)-5' 'X^3+2*X^2-5*X-6' }
```

And the next two derivatives—a slanted line and a horizontal line—appear on the display:



The menu item **FCN** [NXT] **NXEQ** simply makes the next equation in the EQ list the current ("first") equation. For example, after you have pressed **NXEQ** twice, your display should look like this:



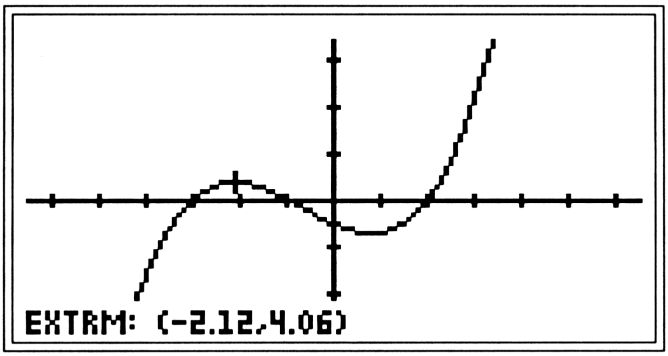
The "first equation" is now the parabola.

For unruly equations, such as $15 - 2x^2 = x^2 + 3x + 5$, **NXEQ** will swap the left- side and right-side expressions, and all **FCN** operations will then act upon the new right-hand side.

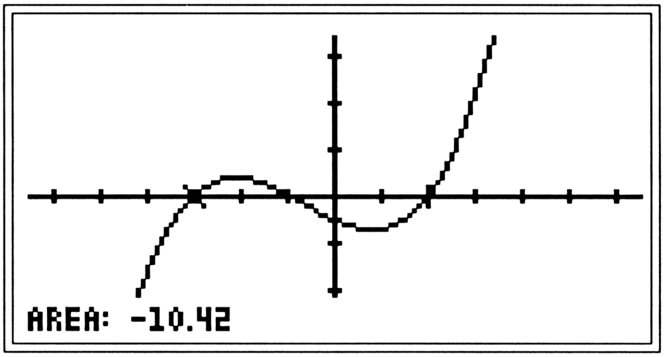
Keep in mind that you can switch back and forth between the Plotter and Solver at any time—and use **NXEQ** in either application. And keep in mind also that if you alter any other variables used in the equations, you must redraw the graphics display (by pressing **ERASE** **DRAW**).

FCN (NXT) **F(X)** simply returns the function value at the current cursor location. For unruly equations, **F(X)** returns the value of the right-hand side; the Plotter's **F(X)** is the graphical analog of the Solver's **EXPR=**.

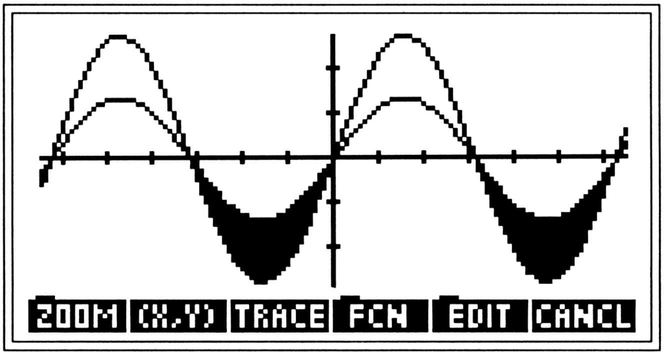
FCN **EXTR** returns the coordinates of an extremum of a curve—but it won't tell you if it's a maximum or minimum. With the third-degree poly-nomial, pressing **EXTR** with the cursor just to the left of the origin re-turns this display:



AREA does a numeric integration on the "first equation" in EQ, with respect to the x-axis. You just put the cursor near the starting point, and press **AREA** or ⊠ to mark one limit. Then put the cursor near the other limit and press **AREA**.... It takes awhile, and you get only the labeled integral, but it's easy—try it: Find the area under the curve between the greatest and least roots of the third-degree polynomial. Move the cursor near the least root and press **FCN** **ROOT** ⊟ **AREA**. Move it near the greatest root and press **ROOT** ⊟ **AREA**. You'll see:



SHADE helps show the area used in integration. With the x-limit still at the least root, and the + cursor still at the greatest root, press NXT or ⊟ (if necessary) to get the menu. Then press **SHADE**.... ***Note*** that **SHADE** colors only the area below the curve and above the x-axis. However, when you store a list of multiple expressions in EQ, shade colors the area *above the first expression* and *below the second expression*. For example, here's the shaded plot of `{ '2*SIN(X)' 'SIN(X)' }`:



The Multiple Equation Solver (MES)

The menu-based Solver allows you to solve a set of linked equations, provided you solve them one at a time, cycling through each via **NXEQ** until you find the answer you were seeking. (An example is on page 59.) You enter all your known values, and then find an equation with only one unknown. You solve for that one, then continue switching equations until you solve for the unknown you really want (or all of them).

HP built the Multiple Equation Solver (MES) to automate this manual process. Try it now: Go to the `G.CH3` directory and type `'Shopping'` `STEQ` to make `Shopping` the current equation. Press ⬅ EQ LIB **MES** **MINIT** to create the reserved variable `Mpar`.* Then press **MSOL** to enter the MES Solver menu. You should see 3 pages of menu items:

TOTA	CSTA	CSTO	LOAD	WT.A	APPL
WT.O	DRAN				ALL
MUSE	MCAL				

In practical terms, the MES feels much like the menu-based Solver: To enter a known value into a variable, you simply key in the value and press the appropriate menu key. To solve for an unknown, you press ⬅, then the menu key; to recall a value, you press ➡ first.

*Press VAR **MPAR** to have a look at it.... All you see is `Library Data`. The MINIT command takes all the equations in EQ, extracts variable names from them, and builds a list of the equations, variable names and other important information. Unfortunately, you cannot directly access this list like other reserved variables, but the MES provides tools to modify it indirectly. Note that MES will run only in a directory containing an `Mpar`, but that different directories can have different `Mpars`, so you can switch from directory to directory with the MES Solver menu active. (If you do this and accidentally change to a directory without an `Mpar`, the 48 will default to the MTH menu.)

Now, re-work the example on pages 59-61: You have \$20. Apples cost \$.29 and weigh .35 lbs.; oranges cost \$.89 and weigh .5 lbs. You want 8 apples and as many oranges as you can afford, taking them home in the wagon, which can carry up to 50 lbs. Will the wagon hold up?

Press 2 0 TOTA · 2 9 CSTA · 8 9 CSTO · 3 5 WT.A 8 APPL NXT · 5 WT.O NXT NXT. Note how the menu labels change from inverse to normal as you enter a value, indicating that the variable's value is now *user-defined*—"sacred." The MES will not change the value of a variable with a normal menu label *except when you're solving for it*. By contrast, inverse labels indicate variables whose value is *calculated*, or unknown at the start of the problem, and the MES may calculate or change the value of this variable in future calculations.

Now press ← LOAD to solve for the total weight of the purchase. Watch the status line of the display. It says `Searching` as it decides what equations it needs to solve to determine the LOAD, then looks for the first one of those with only one unknown. Next it says

```
Solving for ORANGES
ORANGES: 19.87
Zero
```

then

```
Solving for LOAD
LOAD:   12.73
Zero
```

and finally puts the tagged result, `LOAD: 12.73`, onto Stack Level 1.

Notice how ■ indicators have been added to all the menu labels. A ■ in a *user-defined* variable's label, such as CSTO■, shows that the MES *used* this variable in the most recent calculation; a ■ in a *calculated* variable's label, such as LOA■ indicates that the MES *calculated* a new value for this variable in the most recent calculation. In this particular case, all the variables were used or calculated (all the menu labels have ■ in them), but there will be cases when only some of the variables are involved in a calculation, and only those menu labels will have the ■. All ■'ed variables are related—as "participants" in the *most recent solution*—but the values of the un■'ed variables may or may not be consistent with them. Press [·][2][9] [CSTA■] to store a new value into CSTA. The ■'s disappear; the new value you store invalidates the last solution—all calculated variables are "unknown" again.

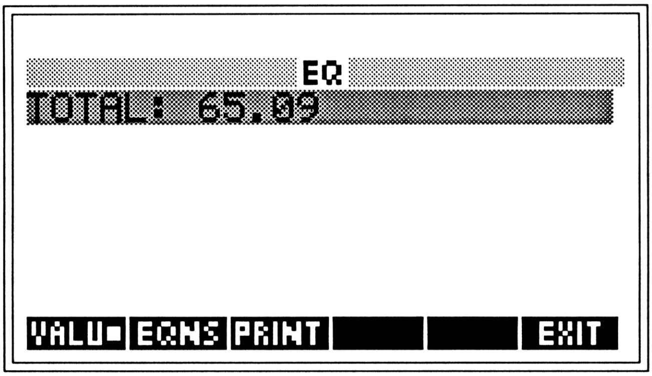
Next, how much will it cost to fill your wagon with an equal weight of apples and oranges? First, press [→][VIEW] to be sure that CSTA is still `0.29`, and CSTO is still `0.89`. Now TOTAL, APPLES, and ORANGES must become *calculated* variables. You use the MCALC command to do this: First, put the list `{ TOTAL APPLES ORANGES }` onto the Stack. Now, press [NXT] until the [MCAL] menu key appears. Press [MCAL] [NXT]. The variables you specified are now in inverse labels; they are no longer sacred—the MES can change them. Press [2][5] [LOAD] [0] [APPL] [NXT] [←][ORAN] and watch the status line. The MES will now solve only for ORANGES (and since TOTAL, CSTA and CSTO were not used in the calculation, no ■ boxes appear in their labels).

Now, to preserve the value of ORANGES while solving for APPLES, use the MUSER command: ['][ORA■] [NXT] [MUSE], then [←][PREV] to confirm that the [ORA■] label has changed to [ORAN]. Now you can press [←][PREV] [5][0] [LOAD] [←][APPLE] to solve for APPLES alone. When you tell MES to solve for the sacred variable APPLES, it sees it as no longer sacred, calculates its value and changes its menu label from [APPLE] to [APP■].

Of course, 71.43 is not a realistic amount of apples to buy, but 71 is OK. So press [7][1] [APP■] (the label will change back to [APPLE]), and press [←][TOTA] to solve the problem completely. Result: `TOTAL: 65.09`

The ALL *Key*

Press NXT until you see the ALL menu label. ALL has 3 important uses in the MES. First, ⇒ ALL pulls up the *progress catalog*, a kind of "show-your-work" notepad for the most recent calculation. The progress catalog for the previous example looks like this:



The first line shows the name of the equation set used in the solution (it defaults to EQ if no name is supplied). Subsequent lines show the values for all variables calculated in the last solution (i.e. all variables with labels like TOT■). EQNS shows the equations used to solve for each variable; VALUE re-displays the values. PRINT sends all of this information to the printer; EXIT returns to the MES Solver menu.

⇐ ALL is the "solve for all" command. Just as you use the ⇐ prefix to solve for a specific variable, ⇐ ALL solves for all calculated variables. Try it—press ' ORA■ NXT MCAL ⇐ PREV ⇐ ALL and watch as the MES solves for both ORANGES and TOTAL.

Unshifted ALL is the "undefine all" command. It turns all variables into calculated variables, with inverse labels. This is the most drastic use of ALL—and the most useful: It wipes the slate clean, so you can enter a completely new solution.

Other Tips on MES

- On page 51, you wanted to keep the gas constant a constant. Similarly here, suppose you want to keep the fruit prices from being overwritten with garbage. The MES Solver is not quite as helpful as the 48's other Solvers, but you can make the prices user-defined variables, via `{ CSTA CSTO } MUSER`.* Or, at the very least, you can move them to the end of the menu, via the MITM command. MITM takes two arguments: a title string on Level 2, and a variable list on Level 1. The list must contain *all* the variables in the equation set, but you can reorder them and insert null strings (`""`) to serve as blank menu keys. Try it: Type `"SHOPPING"` [ENTER], then press [←][{ }] and use the MES Solver menu and [→][""] to create the list `{ ORANGES APPLES TOTAL LOAD WT.O WT.A "" "" CSTO CSTA }`. Now press [←][EQ LIB] [MES] [MITM] to reorder the menu, and [MSOL] to view it.*

- Because of the way the MES works, some sets of equations cannot be solved. MES looks through all the equations for an equation containing only one unknown and solves that equation first. Thus it is possible to have equations arranged in such a way that the MES cannot solve them. The UG (p. 25-9) shows 2 equations in 2 unknowns which you can solve by hand, but which the MES still chokes on, because it can solve for only one unknown at once:

  `'x1=v0+a*t1'` and `'x2=v0+a*t2'`

  To solve this, subtract one equation from the other to eliminate `v0`: `'(x1-x2)=a*(t1-t2)'`. Then put the *three* equations into a MES list.

*But this step can be negated merely by pressing the [ALL] key, so *be careful*.

- Be aware also of other "gotchas" with equation sets—things which you take for granted, such as positive, negative and complex square roots; absolute vs. relative temperatures (see page 48); unit objects; multiple trig solutions (e.g. tan 45°= tan 225°); and bad guesses. See pp. 25-8 to 25-11 in the UG for more ideas.

- You can help the MES with initial guesses. As with the other HP48 Solvers, a guess may be one value, or a list of 2 or 3 values. When you enter a guess, the variable label will change to "user-defined." (See Chapter 18 in the UG for more information.)

In a nutshell, that's the Multiple Equation Solver. You can read more about it in the UG, pages 25-6 to 25-11, and in the AUR if you have it. To summarize the essentials:

- Create your list of linked equations, just as you would create for the regular HP48 Solver. Store the list in EQ.
- Execute MINIT (⬅ EQ LIB MES MINIT) to initialize Mpar.
- Execute MSOLVR (⬅ EQ LIB MES MSOL) to enter the MES Solver.
- Store known values by entering them and pressing the unshifted menu keys. Solve for individual variables by pressing the ⬅ed menu keys, or solve for all unknowns by pressing ⬅ ALL.
- All values marked with ■ or ▣ were used or calculated, respectively, in the most recent solution; they are internally consistent.
- You can protect variable values from being overwritten by specifying a list of their names and executing MUSER (MUSE). Likewise, you can unprotect selected variables by listing their names and executing MCALC (MCAL), or, for all variables, via ALL.

Programmable Use of the Solver (and MES)

Sometimes you need to use the Solver *in the middle of a program.* STEQ and RCEQ are programmable, and you can store or solve for variables *interactively* during the program. For example, to store the equation into EQ and invoke the menu-based Solver:

```
« ... 'eqname' STEQ 30 MENU HALT ... »
```

When the 48 encounters this, it stores '*eqname*' into EQ, activates the SOLVR menu (number 30) *and halts program execution.* You can then use the Solver to store values or run other programs from its variable menu, then press ⬅ CONT when ready to resume the program. Or, to *avoid halting* the program during the Solver, you can instead use ROOT, after setting up the Stack so that ROOT finds its arguments:

Inputs:	→ Outputs:
3: symbolic or program (the equation)	
2: global variable name	
1: real, cmplx., list or unit (1st guess)	1: real, complex or unit (ans.)

Or, to store the equation into EQ and invoke the input-form Solver:*

```
« ... 'eqname' STEQ # B4001h LIBEVAL ... »
```

When the 48 encounters this, it stores '*eqname*' into EQ, activates the SOLVE EQUATION input form *and suspends program execution.* You can then use the input form as you normally would. When you press OK or CANCL, program execution resumes automatically.

*Be ***sure*** you enter the LIBEVAL number ***exactly*** as shown. You can corrupt your 48's memory if you enter the wrong value. Note that # B4001h is # 737281d, if you'd rather use decimal integers.

Here's an example of using ROOT. This program calculates payments for a 5-year, $15,000 loan at various interest rates. The program (AMRT: Checksum: # 28425d Bytes: 226) uses the original TVoM equation (p. 49) and invokes ROOT to print a table of rates and payments:

```
« 15000 'PV' STO 0 'FV' STO 60 'N' STO
  .05 .15
  FOR int
    int DUP 12 / 'I' STO 3 FIX →STR "→ " +
    'TVoM' 'PMT' -100 ROOT 2 FIX →STR +
    PR1 DROP .01
  STEP
»
```

A more polished version would give prettier output, but you get the idea. Another example: To solve partial pressures, you can combine

```
« ...'IdealGas' STEQ 30 MENU HALT... »
```

or

```
« ... 'IdealGas' STEQ # B4001h LIBEVAL ... »
```

and

```
« ...xxx.xx_mol 'n' STO 'IdealGas' 'p' 1_atm ROOT... »
```

The MES is also programmable. If, for example, you want to solve the equations of motion within a program, you could include the sequence: « ...'MOTION' STEQ MINIT MSOLVR... » simply to set up the equation and invoke the MES. Or you could use this sequence:*

« ...'MOTION' STEQ MINIT *value varname* STO... (repeat as needed) ...{ *sacred varnames* } MUSER { *non-sacred varnames* } MCALC *desired varname* MROOT... »

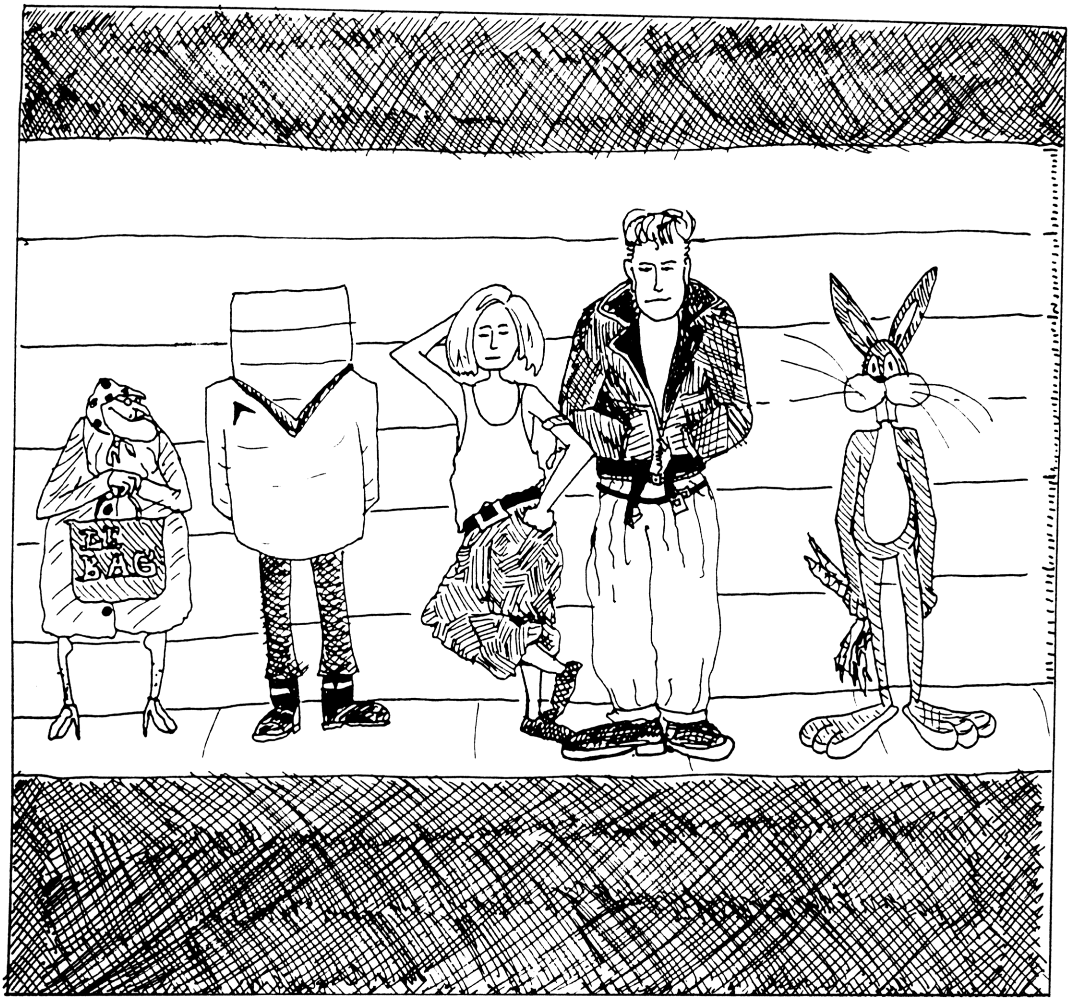
*Note that substituting "ALL" for *desired varname* will instruct MROOT to solve for all unknowns. And MITM is programmable, too: « ..."Title String" { *all varnames* } MITM... »

Review

Okay, set down your calculator, grab a handful of cookies, and think for a moment about the 48 Solver application. You heard it suggested at the start of this chapter that it's really another programming language —even another programming environment. And you've seen the acrobatics the Solver can do:

- You learned about two of the Solver environments—menu-based and the input-form—and the strengths and weaknesses of each.
- You learned how to customize the Solver menus, how to protect variables and perform "outside" tasks from inside the Solver.
- You saw how the Solver is integrated with the Plotter application, and you learned about differences between the graphical Solver and the stand-alone Solver.
- You learned how to solve multiple equations at once—with or without the MES (Multiple Equation Solver).
- You were introduced to using the Solver within a program.

As you can see, if your work relies on math to any degree, the 48 Solver can greatly reduce the amount of « *...programming...* » you do. The Solve Equation Library contains 300 prewritten equations covering dozens of different topics—and new equation libraries are being compiled constantly. Of course, « *...programming...* » isn't dead; there will always be needs for it. But now the Solver can do many of the things that formerly *had* to be done in a « *program* ». So get comfortable with the Solver—using a handheld computer has never been so easy!



4: What's a Grob?

Opening Remarks

With its ability to manipulate complex information in the forms of objects, the 48 makes it easy for anyone to do serious graphics on a handheld machine—something not possible before. Other handhelds have "large" screens or dot-matrix displays but nothing as accessible or versatile as the 48 *grob* (its proper name is "***gr**aphics **ob**ject*," but the 48 shortens this to *grob*).

A Clean Slate

Before you start, set up your machine for some good, hard graphics work:

- First, in your **HOME** directory, create a directory called **TOOLS**, to store your programs.
- Then, in that **TOOLS** directory, create another directory called **PICS**, where you'll store your grobs and do your graphics work.

This will prevent you from clobbering other object names and prevent both your **HOME** directory and working directory (**PICS**) from becoming too cluttered. So from now on (unless specifically directed otherwise), store all programs in **TOOLS** and all grobs in **PICS**. And when actually *using* (executing/evaluating) any program or grob, do so from **PICS**.

Now it's time to talk about grobs....

What *Is* a Grob?

A grob is simply another way for the 48 to store data. You're already familiar with matrix objects, program objects, character string objects, complex number objects, etc.

A grob is just another kind of object—a pixel-by-pixel description of an image that can be displayed on the 48 display, or passed to another 48 or PC, or "dumped" to a printer. A grob can also be manipulated or combined with other grobs—just as other objects can be manipulated and combined in various ways.

Create a simple grob to experiment with—plot a sine wave:

If you're not in RADians mode, press [⬅][RAD]. Then press [➡][PLOT][SIN] [α][X][ENTER][▼][◀] ✓CHK ERASE DRAW....

The graphics display should fill with a sine wave—big deal.

Press [CANCEL][CANCEL] to exit graphics mode.

Move into your new PICS directory, and then press [PRG] PICT PICT [→][RCL] 'SINE' [STO].

PICT is the reserved name in which the 48 stores the current graphics display (much as EQ is the reserved name in which the 48 stores the current equation). Therefore, PICT can be [STO]'ed and [RCL]'ed, but it cannot be deleted (yes, you can [PURGE] it, but a new PICT will be automatically created if you then plot a function or press [←][GRAPH]). So make a mental note: Don't use PICT as an object name, because the 48 has reserved that name for its own use.

In the above exercise, PICT [→][RCL] placed the grob representing the current graphics display onto Stack Level 1. Then 'SINE' [STO] stored it under that name in your PICS directory.

Now take a closer look at this grob. Press [VAR] SINE [▼], and you'll see GROB 131 64, followed by a mass of characters.

What do all those characters mean? To get a better idea, compare them with an "empty" grob: Press [ENTER][←][PLOT] ERASE to clear the graphics display, and then [PRG] PICT PICT [→][RCL] 'EMPTY' [STO] to store the blank display as an object called 'EMPTY'. Now press [VAR] EMPTY [▼], to see GROB 131 64, followed by a mass of zeros.

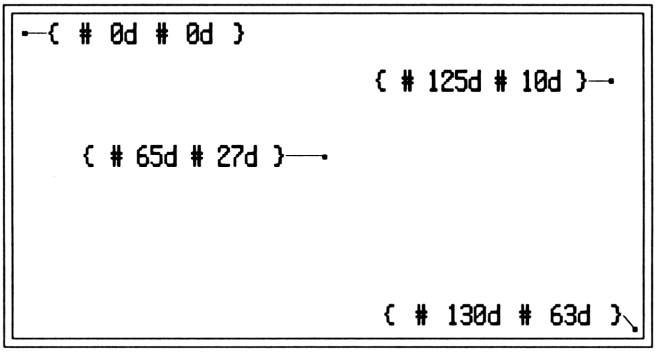
This is the Stack's representation of a grob. The word GROB simply tells you that the object is a grob. The second "word", 131, is the number of *columns* of pixels (dots) in the grob. The third "word", 64, is the number of *rows* of pixels in the grob. And then the huge "word" after that is a hexadecimal bitmap of all the pixels themselves, where every digit represents 4 pixels.

Pixel Numbers vs. User Units

A grob's size is normally expressed as "*m* pixels wide by *n* pixels high." For example, the display grob PICT has a normal default size of 131 pixels wide by 64 pixels high. But you can also express such dimensions in *user units.* User units allow you to define the scale and limits of PICT in more convenient units—to save you a conversion between Cartesian coordinates and pixel locations every time you want to modify PICT.

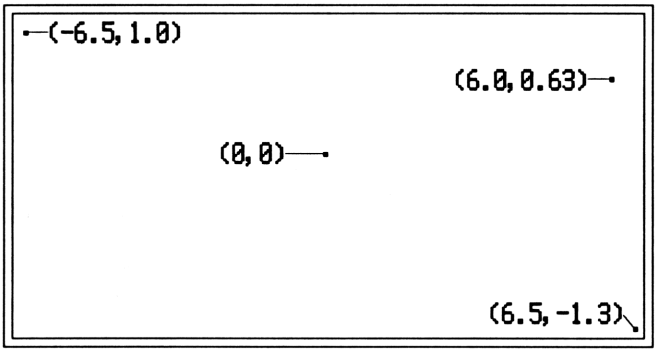
To illustrate this, return the SINE grob to the graphics display and view it, by pressing VAR SINE PRG PICT PICT STO ◄.

Each pixel in this 131×64 grob is defined by a list of two binary integers, of the form { # col # row }. These are "pixel coordinates." Here are a few pixel locations expressed in their pixel coordinates:



However, recall that when you plotted the sine wave, the 48 used the default *x*-axis range of –6.5 to 6.5, and it assigned the *y*-axis range to be –1.3 to 1.0. These ranges were in user units.

A graphical location in user units is expressed in the form of a complex number, `(x, y)`. Here are the same four locations as on the previous page, but expressed in user units rather than in pixel coordinates:



Comparing the two diagrams, notice that their scales behave differently: The pixel coordinate scale always starts at `{ # 0d # 0d }` in the upper left-hand corner, and the numbers increase as you proceed downward and to the right. But the user-units scale starts at whatever values you (or, by default, the 48) have defined, and these numbers increase as you move *up*ward and to the right.

So, which scale should you use? Obviously, user units are much more convenient in many respects. You do your computations, you plug in the numbers, you plot them—just as on graph paper.

Anyhow, HP has made the plotting commands versatile enough to accommodate both scales. And the PRG **PICT** functions **PX→C** and **C→PX** allow you to quickly convert from one scale to the other if you want to see both sets of the numbers.

But performing grob manipulations with user units does have a couple of disadvantages. First of all, it's slower. The 48 doesn't "think" in user units. When you give it a graphics command with real or complex arguments, it has to find out what the current graphics scale is, then convert the arguments to binary integers (pixel coordinate values) and then execute the command. This can increase your program execution time by as much as 50 percent.

Secondly, user units don't always remain the same. They can differ from directory to directory and program to program, as you redefine them. So always check the graphics scale before manipulating grobs, if you're going to do so in user units.

With those considerations in mind, you can see that if your application involves a good deal of plotting and mathematical modeling, then user units are for you. On the other hand, if your application involves placing text in grobs, extensive fiddling with bitmaps, or mixing grobs of unknown user units, then you should stay with pixel coordinates. As a good rule of thumb, if you're doing too many conversions from one scale to the other, it's a sure sign that you need to switch to the other scale.

“Roll Your Own” Grobs

You have several ways to create a grob (i.e. put one onto the Stack):

- [←][PLOT] ERASE [PRG] PICT PICT [→][RCL] creates an empty 131×64 grob.

- To create an empty grob of a *specified size,* use the BLAN (BLANK) command. You put the number of columns (as a decimal integer) at Stack Level 2, and the number of rows (as a decimal integer) at Stack Level 1, then press [PRG] GROB BLAN. The empty grob will be placed at Level 1.

- To turn any object into a grob, put the object at Level 2 and a real number on Level 1. Then press [PRG] GROB →GRO (that's →GROB). If the real number is 1, 2 or 3, the 48 will use the small, medium or large font, respectively, to create the grob. If that argument is 0 and the object is an algebraic or unit object, its grob will be created in textbook format—as in the EquationWriter.

- [PRG] GROB [NXT] LCD→ copies the current display to a grob.

- [←][PLOT] DRAW and [→][PLOT] DRAW will create a grob named PICT with a function or statistical data plotted on it. To then put this grob onto the Stack, you type PICT [→][RCL] (from the Stack display), or [STO] (from within the Graphics display).

- [STO] converts to a grob directly from the EquationWriter.

- You can also create a grob on the Command Line. For example (do this now), type GROB 8 2 83FF [ENTER].... See?

The Hexadecimal Bitmap

That grob you just created is 2 rows (of pixels) tall and 8 columns (of pixels) wide. An 8×2 grob therefore has 16 pixels ("picture elements").

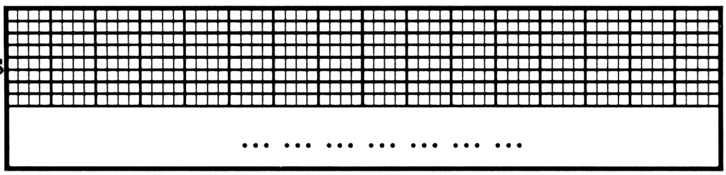
A hexadecimal digit*, expressed in binary form, can hold information for 4 pixels. For example, the hex number B (which has a decimal value of eleven), is expressed in binary as 1011. So the hex number B can describe a row of 4 pixels, where all but the second pixel are "on" (dark); the second pixel is "off" (light). Similarly, a hex 0 (binary 0000) would be all pixels "off", and a hex F (binary 1111) would be all pixels "on".

The 48 always uses an *even number* of hex digits for each row. So if your grob is between 1 and 8 pixels wide, you'll need 2 hex digits to describe that row—even if you use only a few of those pixels.

Since each hexadecimal digit represents 4 pixels in a row, it's easy to think of a grob as a collection of 1-row, 4-column bitmaps:

m columns

n rows



*If you don't understand hexadecimal numbers, keep your place here while you read Appendix A.

In the grob you just created (via `GROB 8 2 83FF`), for example, the digits `83` described the first row of pixels; the digits `FF` described the second row.

Unfortunately, HP decided that the bitmaps should read backward from the conventional ordering of the digits in a binary number. That is, you might naturally *think* that `83` would describe this bitmap:

hex digit value		8				3		
binary place value	8	4	2	1	8	4	2	1
pixel value	1	0	0	0	0	0	1	1

But no—it doesn't. Rather, the `83` describes this bitmap:

hex digit value		8				3		
binary place value	1	2	4	8	1	2	4	8
pixel value	0	0	0	1	1	1	0	0

Perplexed? It's understandable. This takes some getting used to—and to help that process along, take a look at your grob....

The SEE Program

The 48 doesn't have a quick command to let you "see" the graphics representation of a grob on the Stack, so you need to write one now.*

Notice that **PICT** STO takes a grob from Stack Level 1 and puts it into the reserved variable PICT, and that the ⬅ PICTURE command lets you view and manipulate PICT.** Your Mission: incorporate your observations into a program, 'SEE' (Checksum: # 9380d Bytes: 25).

Solution:
```
« PICT STO PICTURE
»
```

In your **TOOLS** directory, type this on the Command Line and press ENTER. Then type 'SEE' STO.

Now, with any grob in Stack Level 1, SEE will let you see it immediately—try it! Use SINE, EMPTY, or your GROB 8 2 83FF—whatever.

Create other grobs using the Command Line, and view them using SEE. Remember: If you use too few digits, the 48 will simply "pad" the grob with zeros, but if you use too *many* digits, it will give you an *error message.*

*If you don't know how to write programs on the 48, place a bookmark here, skim over the chapter on "Programming the HP 48" in the Owner's Manual, then return here.

** Yes, you could use the PVIEW command in place of ⬅ PICTURE, but PVIEW requires an argument in Level 1, and it doesn't allow access to the graphics editing menus—not so handy.

What Does a Grob Eat?

A grob eats memory. Lots of it.

Even a 0×0 grob uses 10 bytes of memory. And how would you make a 0×0 grob to see this? A couple of different ways, actually:

GROB 0 0 (ENTER)

or

0 # 0 (PRG) GROB BLAN

What's more, if you were to convert that 0×0 grob to a string, "GROB 0 0", it would use *14* bytes.

As you can see, memory use is of primary consideration when you're working with grobs. So here are two quick utilities to help you measure grob size:

GSIZE

Checksum: # 52100d

Bytes: 78

```
« → w h
   '10+h*(1+IP((w-1)/8))'
»
```

GSIZE takes the row and column arguments from the Stack and gives you the size of the graphics object itself.

$SIZE

Checksum: # 4548d
Bytes: 130

```
« DUP2 SWAP →STR SIZE
  SWAP →STR SIZE SWAP
  → w h lw lh
  '12+lw+lh+2*h*(1+IP((w-1)/8))'
»
```

$SIZE takes the row and column arguments and gives you the size of the *string* representation of the grob. This is very important to know if you're uploading grobs in ASCII format to another computer; the 48 must have enough memory to hold both the binary and the ASCII representations.

Keep these two utilities in your TOOLS directory. They'll help you budget your memory resources as you develop graphics applications. For example, they'll tell you that a screen-sized, 131×64 grob uses 1098 bytes, and its corresponding string uses 2193 bytes. And a 200×200 grob needs 5010 bytes in binary and 10018 in ASCII.

As you can see, grobs eat memory in big bytes.

The Grob as Icon

Grobs that are 21×8 have a special application in the 48—as *menu icons*. To create an icon via freehand drawing:

- In the PICTURE environment, press [←][CLEAR][→][◀][→][▲], then [×], then [▼] *seven* times, then [▶] *twenty* times, then **EDIT** **BOX**.

- Use freehand drawing (see Chapters 5 and 8) to draw your icon. Then erase the outline, if you wish.

- Press [→][◀][→][▲], then [×], then [▼] *seven* times, then [▶] *twenty* times, and then ([NXT]) **SUB**, to copy your icon to the Stack.

- Put your unshifted/shifted key actions* on Stack Levels below the icon, specify the icon Level, and press [PRG] **LIST** **→LIST**.

Repeat the above steps as needed to create more icon lists. Then give the number of menu items and press **→LIST** [→][MEMORY] **MENU**.

Or, here's a "pre-fab" example: Key in this custom menu list (it's all 1 object—don't hit [ENTER] until the very end—and ignore line breaks):

```
{ { GROB 21 8
0000006081009042080124040218030C0010000000000000 "SINE" }
{ GROB 21 8
0000004801006C81105A4908492504C81302480100000000 "SAW" }
{ GROB 21 8
0000008F1E70801240801240801240801240E0F3C1000000 "SQUARE" }
{ GROB 21 8
0000006775D11555501553D11555506775D1000000000000 "YEAH!" }
} MENU
```

[ENTER].... A very interesting custom menu—4 grobs as labels!

*See chapter 30 of the User's Guide ("Customizing the HP 48") for more information on creating custom menus. And you may want to make a note there that 21×8 grobs can act as menu labels.

Not just interesting—useful: You can fit only 4-5 characters of text into a menu label, but an icon—even of that size—is a picture worth a thousand words. In fact, you can even create menu labels with the little box that appears/disappears to indicate status—like this: ON■ OFF. To do this, you use the SYSEVAL command.* The table below shows 4 SYSEVAL codes and the results they would give via this sequence:

"LABEL" *syseval-code* SYSEVAL PICT STO PICTURE

SYSEVAL code	Description	Result of above sequence
# 3A328h	normal label	LABEL
# 3A3ECh	directory	LABEL
# 3A44Eh	inverse	LABE
# 3A38Ah	status "on"	LABE■
(see below)	MES "unknown solved"	LAB ■

Use the 21×8 grobs created by these codes just as you would use the 21×8 grob icons on the previous pages. To create the MES menu grob, use one of these two routines:

(Checksum: # 12967d Bytes: 100.5)
```
« 1 3 SUB #3A38Ah SYSEVAL { # 0h # 0h } GROB 21
  8 FFFFF1EFFFF0EFFFF0EFFFF0EFFFF0EFFFF0EFFFF0 GXOR »
```

(Checksum: # 49860d Bytes: 81.5)
```
« 1 4 SUB #3A44Eh SYSEVAL { # Fh # 2h }
  GROB 5 5 00C1C1C100 REPL »
```

The first routine uses the "indicator on" SYSEVAL, cropping the label string to make it fit. The second routine uses the "inverse" SYSEVAL—slightly smaller and faster—also cropping the string to make it fit.

**Warning:* SYSEVAL can be very dangerous. If you enter an incorrect SYSEVAL code, you can cause a Memory Clear. Enter SYSEVAL codes *very carefully*—and back up your memory first!

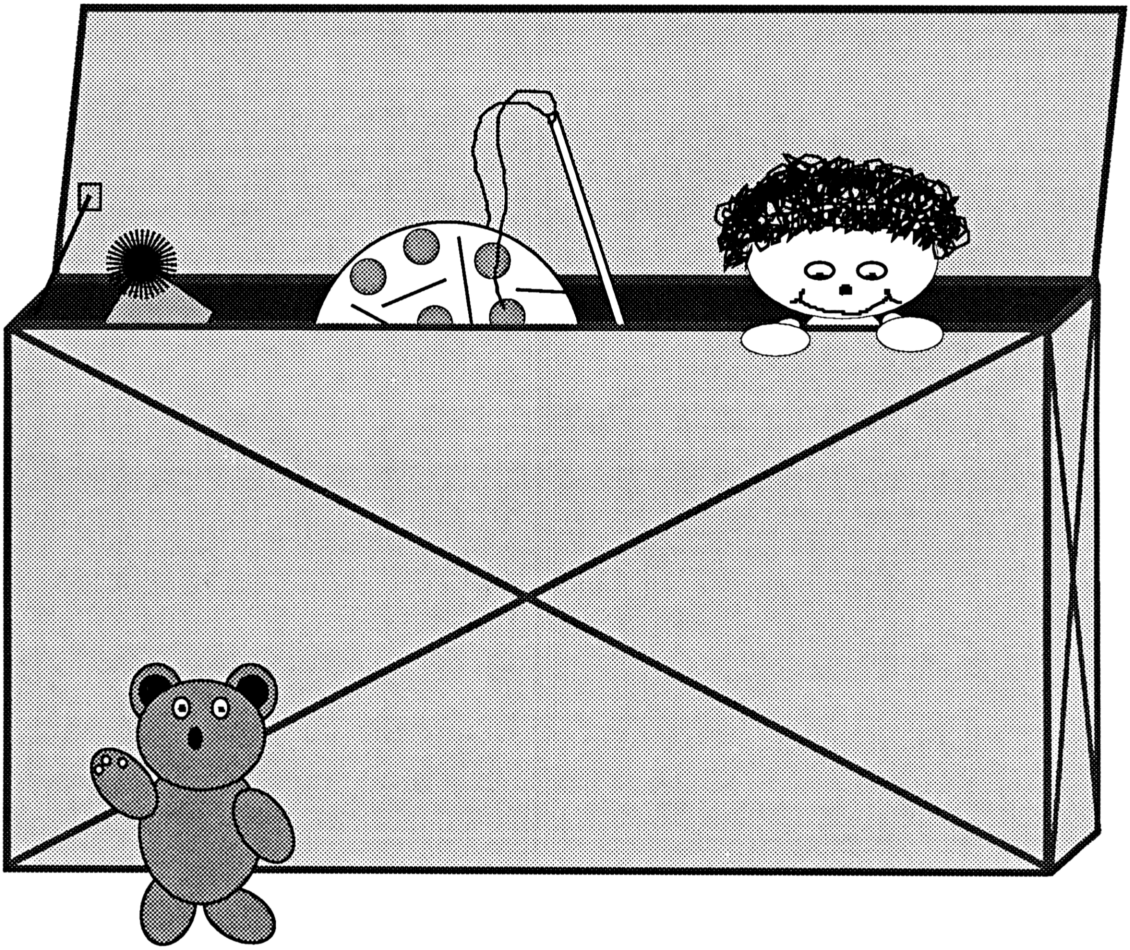
Three other useful menu SYSEVAL codes are: # 3A1FCh (DispMenu1) which causes the 48 to update a custom menu display *immediately*; and # 4E2CFh (TurnMenuOff) and # 4E347h (TurnMenuOn), which cause the menu line to "turn off" or "turn on" during a program. Here is a short demo of these codes (Checksum: # 52019d Bytes: 238):

`« { "HELLO" "THERE"`	*Create a*
`{ } { } "BYE" } TMENU`	*temporary menu.*
`# 3A1FCh SYSEVAL`	*Display this menu,*
`1 100 FOR n n 1 DISP NEXT`	*and count to 100.*
`# 4E2CFh SYSEVAL`	*Turn the menu off,*
`"MENU OFF" 3 DISP`	*give message,*
`1 100 FOR n n 1 DISP NEXT`	*and count to 100.*
`# 4E347h SYSEVAL`	*Turn the menu on,*
`"MENU ON" 3 DISP`	*give message,*
`1 100 FOR n n 1 DISP NEXT`	*and count to 100.*
`0 MENU`	*Restore previous menu.*
`»`	

Review

In this chapter, you created the **TOOLS** and **PICS** subdirectories to hold your grobs and your programs—and to help you organize your thoughts. You also learned:

- how a grob is represented graphically and numerically—and how much memory it eats;
- how to use the GROB *row col nn...* notation, so that you can read or write a grob from the Command Line;
- how to create grobs—both empty or with pre-plotted patterns in them—and how to use them in custom menus.



5: Graphics Basics

The Graphics Functions

Now that you understand what a grob is and how it is built, return to the built-in graphics functions and run through them briefly. They are all programmable to some degree, and you're going to see that programmability at work now, too.

HP chose to scatter the graphics commands among several different menus (a custom menu might be very handy—food for thought). Some are in the ⬅PLOT and ⬅PICTURE menus, some under PRG-GROB, and some under PRG-PICT, PRG-IN and PRG-OUT. For a reference listing of the graphics commands, see Appendix B.

Now, as you know, you can get to the PICTURE display by pressing ◀ from the normal Stack display. However, the more general form of the command is ⬅PICTURE—and in a program listing, ⬅PICTURE gives you the PICTURE command, which causes the program to halt in the PICTURE display with the PICTURE menu active. Then CANCEL returns you to the Stack display and continues program execution (note: in a program, the TEXT command also returns you to the Stack display).

(Incidentally, on the HP 48S/SX, the graphics display command was called GRAPH. Try typing this on your HP 48G/GX: « GRAPH » ENTER)

To view a grob in the Stack display, put the grob onto Stack Level 1 and use the →LCD command (PRG GROB NXT →LCD).

The grob will fill the display with its upper-left pixel in the upper-left corner of the display, overwriting everything except the menu line (and the menu remains active). →LCD does not halt program execution.

To activate the graphics display without the menu line—and still without halting program execution—use the PVIEW command.

PVIEW requires an argument in Stack Level 1—the location of the pixel to be in the upper-left corner of the display. Normally, this would be the row 0, column 0 pixel, so you would put { # 0d # 0d } in Level 1 and press PVIEW. Remember that the first number in this list is the column number; the second is the row number. Remember also that, if you wish, you may give the coordinates of the upper-left corner in user units instead, with a complex number (x, y), where you choose the coordinates x and y.

Within the PICTURE environment, pressing ← PICTURE a second time removes the menu and puts you in a "scrolling mode." In this scrolling mode, you can use the arrow keys and →'ed arrow keys to scan around a large grob, with the display acting as a "window" into the grob. In fact, PVIEW is the programmable equivalent of this scanning capability.

Press ← PICTURE a third time to return to the PICTURE display, or press CANCEL to return to the Stack display.

The Secrets of PPAR

As you read in Chapter 4, every grob has associated with it a height and a width, measured in pixels. The height (rows) and width (columns) appear in the Stack display as Graphic *ccc* x *rrr*
or in the Command Line as GROB *ccc* *rrr* *dddd....*

If you ever need to test a grob within a program, the programmable command SIZE returns the number of columns to Level 2 and the number of rows to Level 1.

With that in mind, consider this: Associated with the plotting and graphics routines is a reserved variable named PPAR (for Plot PARameters). Like the reserved variables IOPAR and PRTPAR, PPAR is created (if it doesn't already exist) only when a routine invokes it. (Note also that there is another reserved variable, VPAR, associated with the 3-D plotting routines—discussed in Chapter 6.)

That is, PPAR is invoked or created *anytime you activate the graphics environment, even if you don't see the graphics display.* Specifically, PPAR is invoked by:

- [←][PICTURE] or [◄];
- [←][PLOT] or [→][PLOT];
- Any drawing function (most of these are in [PRG]-PICT);
- PVIEW with user units (a complex number)—but not with a list of binary integers or an empty list.

And of course, PPAR can be STOed, RCLed and PURGed, like any other variable. The contents of PPAR, however, must follow this pattern:

{ (x_{min}, y_{min}) (x_{max}, y_{max}) *indep res axes ptype depend* }

You set these 7 parameters from the PLOT menu, or by using the PLOT menu commands inside a program. The default values are:

```
{ (-6.5,-3.1) (6.5,3.2) X 0 (0,0) FUNCTION Y }
```

Alternate forms for the last five parameters in PPAR allow you to take advantage of certain "advanced" plotting options. Watch PPAR change as these options are invoked:

X plotting range restricted to -5, +5:

```
{
  (-6.5,-3.1) (6.5,3.2) { X -5 5 }
  0 (0,0) FUNCTION Y
}
```

Here, the *indep* field has changed to show the minimum and maximum values to be plotted for the independent variable.

Resolution (step size) changed to 5 pixels:

```
{
  (-6.5,-3.1) (6.5,3.2) { X -5 5 }
  # 5d (0,0) FUNCTION Y
}
```

If the *res* parameter has a value of 0.0 (user units) or #0d (pixels), then the default (calculate and plot at every pixel column) is used. Otherwise, function values are calculated and plotted at the interval specified by *res*, in user units (a real number) or pixels (a binary integer).

Tic spacing on axes changed to 3.14 (x), .5 (y) user units:

```
{
  (-6.5,-3.1) (6.5,3.2) { X -5 5 }
  # 5d { (0,0) { 3.14 0.50 } } FUNCTION Y
}
```

The *axes* parameter changes from a complex number denoting the intersection of the *x* and *y* axes, to a list containing that number and another list denoting the space between tic marks on the *x* and *y* axes.

Tic spacing changed to 2 (x), 5 (y) pixels:

```
{
  (-6.5,-3.1) (6.5,3.2) { X -5 5 }
  # 5d { (0,0) { # 2d # 5d } } FUNCTION Y
}
```

As this example shows, the tick spacing can be in either user units or pixels. If the sublist is `{ 0.0 0.0 }` or `{ # 0d # 0d }`, then the default tic spacing of 10 pixels is used.

Axis labels changed from `X` and `Y` to something else:

```
{
  (-6.5,-3.1) (6.5,3.2) { X -5 5 }
  # 5d { (0,0) { # 2d # 5d "TIME" "RATIO" } }
  FUNCTION Y
}
```

Here, the *axes* parameter has expanded yet again. The list now includes two strings, which replace the default axis labels (usually `"X"` and `"Y"`) used by the **LABEL** command.

The next section will show how to create this expanded version of PPAR (it is also automatically created by **OPTS** inside the PLOT input form).

This short program:

```
« PICT SIZE PPAR 28 FS? 29 FS? 31 FS? »
```

will tell you everything you need to know about the graphics display —if you can read it.

All of the PPAR information is also available from the PLOT input form and its **OPTS** input form. This same information is also displayed, in a little different format, when the [←][PLOT] menu and [←][PLOT] **PPAR** menus are active (if the information is not displayed, press **INFO** or [→][VIEW]). In a program, the best way to get at the PPAR data is to recall the contents to the Stack and either OBJ→ or SUB to extract the parts that you need.

Bear in mind that each directory in the 48 can have its own PPAR, which can cause you trouble if you work in user units and switch directories a lot.

For example, if you're working in DIR1 where PPAR contains x_{min}=-10 and x_{max}=10, and then you switch to DIR2 where PPAR contains x_{min}=0 and x_{max}=6.28, you'll get undesirable results if you use DRAW or any user-unit commands without first adjusting x_{min} and x_{max}.

Generally speaking, you'll need to get only the plotting limits at the start of PPAR. In the next section, you'll see how to get out more information.

The PLOT Menu

The PLOT menu consists of 2 pages of commands and submenus:

PTYPE PPAR EQ ERASE DRAX DRAW
3D STAT FLAG LABEL AUTO INFO

These selections give you access to the same tools as in the PLOT input form. For example:

- PTYPE and 3D-PTYPE correspond to the TYPE: field;
- EQ and its shifted versions correspond to the EQ: field;
- ERASE corresponds to ERASE in the input form;
- DRAX and DRAW together work like DRAW in the input form;
- LABEL is identical to LABEL in the ⬅PICTURE-EDIT menu;
- the AUTO key corresponds to the _AUTOSCALE checkbox;
- the FLAG menu gives you access to the 3 system flags that control other aspects of plotting;
- the PPAR menu is equivalent to most of the other fields, and to the fields in the OPTS input form. The 3-D viewing parameters that aren't covered in PPAR can be found in 3D VPAR (these are covered in more detail in Chapter 6);

To create a graph with these tools takes a little more work than the input form and you must often press ◀ or ⬅PICTURE to see your graph. On the other hand, *these commands are programmable* (and you will see them hard at work in Chapter 9). Look at them now in detail:

The **EQ** key is the only key with ⬅ and ➡ features. Pressing **EQ** recalls the contents of EQ to the Stack (same thing as RCEQ), unless EQ doesn't exist yet, in which case it puts 'EQ' on the Stack. ⬅ **EQ** performs the command STEQ, which stores the item in Level 1 into EQ. ➡ **EQ** performs the command RCEQ.

ERASE erases the contents of PICT. It's identical to the **ERASE** found in the PICTURE **EDIT** menu and the PLOT input form menu, and it's programmable. (By contrast, the more drastic **RESET** resets PPAR to its default values, resizes PICT to its default 131×64 size, and erases the contents of PICT. **RESET** is not programmable—nor is it recoverable; there's no LAST GRAPHICS command. So use **RESET** with care!)

DRAX is a command for drawing axes inside the PICT grob. It is useful inside a program, used in conjunction with DRAW. For example, the MULTIPLOT program in Chapter 9 uses **DRAW** and **DRAX** together to make iterative plots on the same axes.

DRAW is the programmable plotting command. **DRAW** turns on the graphics display, plots the contents of EQ, then turns off the graphics display. *It does not draw axes or labels*, and the graphics display remains active only while plotting EQ. So ⬅ **EQ** **ERASE** **DRAX** **DRAW**, (or the program « STEQ ERASE DRAX DRAW ») is an easy way to plot a function or equation from Stack Level 1.

LABEL reads the contents of PPAR and adds axis labels to PICT. It doesn't check for the presence of axes; **ERASE** **LABEL** is a valid command sequence. **LABEL** uses the current numeric display format (thus STD format often produces too many digits in your plot). **LABEL** is the programmable version of the **LABEL** in the PICTURE-**EDIT** menu.

AUTO calculates the y-values of the dependent variable in EQ, for every value of the independent variable from x_{min} to x_{max}. It sets y_{max} equal to the maximum calculated value, and it sets y_{min} eight pixels (in user units) lower than the minimum calculated value. **AUTO** is a programmable substitute for the **_AUTOSCALE** setting in the **PLOT** input form. **AUTO** does not draw anything; **AUTO** **DRAW** does.

The **INFO** key displays information about some current plot settings. This display disappears when you press [CANCEL] or do something that affects the Stack. It reappears when you press **INFO**, [NXT] or [→][VIEW]. **INFO** is not programmable.

The **3D** menu is described more in the next chapter.

The **STAT** menu contains statistical plotting tools, which will not be discussed in this book.

The PTYPE and PPAR menus give you direct (and programmable) control over PPAR. The PTYPE (and 3D -PTYPE) menu keys set the *ptype* parameter as indicated by the keys. The PPAR menu keys are:

INDEP	DEPN	XRNG	YRNG	RES	RESET
CENT	SCALE	*W	*H	AXES	ATICK
PPAR	INFO				PLOT

The last three keys make life easier. You already know about INFO.

PLOT returns you to the main plotting menu.

PPAR is a typing aid, recalling the contents of PPAR to the stack or adding the word `PPAR` to the command line (if PPAR doesn't exist, then `'PPAR' RCL` creates one on the spot). Unfortunately, PPAR does not have the ⬅➡ capabilities of EQ. However, you can fake it if PPAR is in your VAR or CST menu: Pressing ⬅NAME or ➡NAME not only stores/recalls the contents of the variable, but inserts '*name*' STO or '*name*' RCL into a program.

INDEP and DEPN (INDEP and DEPND) specify the independent and dependent variables by name. Defaults are `X` and `Y`—but those won't work in equations such as `'Impact=(Mass*Speed^2)/2'`.

Note that you can use a list, instead of just a name, to specify the range over which the function may be plotted. For example, to plot just the first two revolutions (720°) of a spiral, you'd type `{ 'θ' 0 720 }` INDEP. Then you could use small programs to recall those parameters:

`« PPAR 3 GET »` (independent variable)

`« PPAR 7 GET »` (dependent variable)

The 48 gives you three different ways to independently specify values for x_{min}, y_{min}, x_{max}, and y_{max}: **CENT SCALE**, **XRNG YRNG**, and PMIN PMAX

The **CENT SCALE** (CENTR and SCALE) combination is most useful for specifying a certain point as the center of the plot, then scaling the x- and y-axes relative to each other—as for a polar or conic plot.

CENT accepts a real number argument to center the plot along the x-axis; or a complex argument to center the plot in both x and y. The inverse of CENTR would be a program that finds the center of PICT:

```
« PPAR OBJ→ 6 DROPN DUP2 - 2 / DUP RE - PICT
  SIZE SWAP DROP B→R 1 - / ROT ROT + 2 / + »
```

SCALE takes two real-number arguments: the x-axis scale and the y-axis scale — both in units *per ten pixels*. Thus, if (0,0) is the center of your 131x64 grob, and your x-axis scale is, say, 5, then your grob's x_{min} will be (-130/2)*(5/10) or -32.5, and its x_{max} will be 32.5. The inverse of SCALE—to find the x and y scales—would be this program:

```
« PPAR OBJ→ 6 DROPN SWAP - 10 * C→R PICT SIZE
  1 - B→R ROT SWAP / ROT ROT B→R 1 - / SWAP »
```

The more rectangular **XRNG YRNG** combination is the most intuitive for **FUNCTION** type plots and general drawing. **XRNG** and **YRNG** are identical in function, each taking 2 real number arguments: the minimum range value, x_{min} or y_{min}; then the maximum range value, x_{max} or y_{max}. XRNG and YRNG are both programmable. Their inverse functions are:

```
    « PPAR 1 2 SUB RE EVAL »    for xmin, xmax
and « PPAR 1 2 SUB IM EVAL »    for ymin, ymax.
```

PMIN and PMAX are mentioned in the User's Guide only in the Operation Index after the Appendices. To use these two commands, you must key them in (or assign them to a custom menu or key). They were used to set the display limits on the HP 28S, and are included in the 48 for compatibility. The 48 stores the display limits in PPAR as the complex numbers (x_{min}, y_{min}) and (x_{max}, y_{max}). Their inverse functions are:

« PPAR 1 GET » for PMIN, and
« PPAR 2 GET » for PMAX.

AXES and **ATICK** are related functions. **AXES** defines the coordinates where the drawn axes will intersect, and optionally sets alternate axis labels and spacing between tick marks. It takes one argument, which may have any one of the following formats:

(*x*, *y*)
{ (*x*, *y*) "*x-label*" "*y-label*" }
{ (*x*, *y*) { *xtick* *ytick* } }
{ (*x*, *y*) { *xtick* *ytick* } "*x-label*" "*y-label*" }
{ (*x*, *y*) *ticks* }
{ (*x*, *y*) *ticks* "*x-label*" "*y-label*" }

ATICK sets the spacing between tick marks. It takes a single argument of the form { *xtick* *ytick* } or *ticks*. The complex number (*x*, *y*) is the point where the axes intersect. "*x-label*" and "*y-label*" are strings that replace the default axis labels when **DRAX** is executed. *xtick* and *ytick* can be real numbers (to describe the tick spacing in user units) or binary integers (to describe the tick spacing in pixels). One number, *ticks*, can be used instead of { *xtick* *ytick* } if *xtick* and *ytick* are identical.

The inverse function for both AXES and ATICK is: « PPAR 5 GET »

*H and *W are the only programmable "ZOOM" commands in the 48. Both *H and *W *leave PICT unchanged*, but they multiply the height or width by a real-number argument. An argument greater than 1 "zooms out," showing more range with less detail; an argument less than 1 "zooms in," showing less range but more detail.

Be careful with *H and *W! Because PICT remains unchanged, it's possible to get plots with different scales superimposed on each other. For example, here's what happens when *W is used carelessly:

```
« ERASE
  'Y=SIN(X)' STEQ
  RAD DRAW 2 *W DRAW
  2 *W DRAW
»
```

So to avoid serious trouble, it's a good idea to always follow a *H or *W command with ERASE.

RES sets the *resolution* of the plot, according to the real or binary number given as an argument. If the argument is real, DRAW will calculate and plot a function value at intervals of that many *user units*. If the argument is a binary integer, DRAW will calculate and plot the function value at intervals of that many *pixel columns* in a FUNCTION plot. An argument of 0 or # 0d resets the resolution to the default—every single pixel column.

The inverse of RES would be: « PPAR 4 GET »

Finally, there's the **FLAG** menu:

AXES	CNCT	SIMU			PLOT

This menu controls the value of 3 system flags related to plotting. The menu keys toggle the flag values (default states for the flags are *clear*).

Flag -28 is the "sequential plot" flag. When it's clear (**SIMU**), multiple functions in EQ are plotted sequentially, one after the other. When it's set (**SIMU■**), the multiple functions are plotted simultaneously. This is more a matter of aesthetics than processor speed, but you may have memory problems trying to plot too many functions simultaneously .

Flag -29 is the "draw axes" flag. When it's clear (**AXES■** on the menu), axes are added to a plot made from the PLOT input form. When it's set (**AXES**), axes are not added. This flag doesn't affect plots made from the [←][PLOT] menu; you still have to use **DRAX** for these.

Flag -31 is the "connect-the-dots" flag. When it's set (**CNC■**), the 48 will connect each consecutive pair of plotted points with a line. When it's clear (**CNCT**), the 48 only plots the points it calculates. Using **RES** and flag -31 together can save you a lot of computation time.

Here is a program to imitate these keys:

« *flag* DUP IF FS? THEN CF ELSE SF END »

You can also set/clear/check these flags in the [→][MODES]-**FLAG** Browser or [→][PLOT]-**OPT** **PLOT OPTIONS** input form.*

*To make your life easier, the **AXES**, **CNCT** and **SIMU** keys are typing aids: Pressing [←] and one of these keys *sets* that flag; pressing [→] and the key *clears* the flag—works even in program entry.

The PRG-GROB Menu

Behind the PRG key are four menus useful for doing graphics work: the GROB, PICT, and OUT menus. The PRG-GROB menu contains programmable functions for manipulating grobs on the Stack:

→GRO BLAN GOR GXOR SUB REPL
→LCD LCD→ SIZE ANIM

All of the grob-building methods mentioned earlier (page 95) are programmable. Three of these live in the PRG-GROB menu:

→GRO takes the object in Stack Level 2 and turns it into a grob, using the font size specified in Level 1. The font size specifier is a real number between 0 and 3 and is interpreted as follows:

font size	grob's character height (in pixels)	
3	10	
2	8	
1	6	(characters are all uppercase)
0	10	(for text and numbers), or
	EW	(for algebraics and unit objects)

Try one: Retrieve the TVoM algebraic from your G.CH3 directory, then press 0 PRG GROB →GRO. You'll briefly see the EquationWriter view of TVoM before a long grob is returned to Stack Level 1.

BLAN creates a blank grob from width and height arguments in Levels 2 and 1.

LCD→ takes a "snapshot" of the current display and stores it as a grob on the Stack (STO does this for the EW and the graphics environment).

Four extremely useful commands allow you to store part of an image as a grob, and to superimpose a small grob on a larger one:

SUB lets you extract part of a grob (just as you extract part of a list or string object). When used with a grob, **SUB** takes the grob or PICT from Level 3, and the upper-left and lower-right corners of the area to be SUB'bed from Stack Levels 2 and 1, respectively.

Try extracting part of the SINE grob: Move to the PICS directory. Press (VAR) **SINE** { # 50d # 18d } (ENTER) { # 85d # 40d } (PRG) **GROB** **SUB**. You get a 36×23 grob. Press (⬅) (UP) (VAR) **SEE** to view it.

The commands **GOR** ("Grob OR"), **GXOR** ("Grob XOR") and **REPL** ("REPLace") let you *superimpose* one grob upon another. These commands all take the same arguments—the target grob (or PICT), the location, and the grob to be added. The location (Level 2) specifies the spot on the *target grob* (Level 3) where the upper-left corner of the *grob to be added* (Level 1) will go.

Both GOR and GXOR give a kind of transparency effect thanks to the Boolean logic. GOR will superimpose the pixels of the two grobs in such a way that if *at least* one of the pair of corresponding pixels is "on" then the pixel in the resulting grob is "on." GXOR, on the other hand, will superimpose the pixels so that *exactly* one of the corresponding pair must be "on" in order to turn "on" the pixel in the resulting grob. GXOR, in particular, is useful for manipulating cursors and other kinds of objects that need to always be visible within the background—whether it be dark on light or light on dark.

SUB and REPL work here much as they work within the PICTURE EDIT environment. Recall that the interactive menu also includes a DEL command, to delete or blank out part of a grob, but this isn't in the PRG-GROB menu. The best you can do is to create a grob of the right size, using BLANK, then REPL it onto PICT or the grob.

→LCD is more of a PRG-OUT command than a PRG-GROB command. →LCD replaces the stack display with a grob taken from Level 1. You played with this while "Grobbing Around" in Chapter 1.

SIZE takes a grob from Level 1 and returns two binary integers representing the width (or number of pixel columns) and height (or number of pixel rows) of the grob.

ANIM (ANIMATE) is a fun one. For arguments, it takes a real number on Level 1, and that number of grobs in the Levels immediately above it. ANIMATE cycles through the series of grobs, starting at the highest one and rolling the stack to display the next one, etc., pasting them in the upper-left corner of PICT and displaying the result (an endless loop of `PICT { # 0d # 0d }` *grob* REPL).

Try it. Put this onto the Stack:

```
GROB 4 1 1 GROB 4 1 2 GROB 4 1 4 GROB 4 1 8 4
```

Now press ANIM.... You'll see whatever was in PICT before (probably a piece of `SINE`), plus a little scrolling light in the upper-left corner. Press CANCEL to stop the show.

Notice: the Stack is unchanged (grobs in their original order); you can restart just by pressing ANIM again.

Instead of a real number, ANIMATE can use a list argument of the form:

{ *ngrobs* { # *x-pixel* # *y-pixel* } *duration* *cycles* }

ngrobs and *cycles* are the number of grobs to be used and the number of times the animation should run. If *cycles* is zero, the show will cycle until CANCEL is pressed. # *x-pixel* and # *y-pixel* are binary integers specifying the location inside PICT where the upper-left corner of the grobs should be pasted. *Duration* is the interval (in seconds) for each frame.

The single real-number argument you used earlier was equivalent to { *ngrobs* { # 0d # 0d } .1 0 }. Now try a list argument: Instead of a 4 in Level 1, use the list { 4 { # 40d # 20d } .2 10 }....

On machines with black LCD pixels, *duration* values faster than the default 0.1 (1/10 second) may cause the grobs to cycle too fast to be seen (the black crystals are too slow to keep up). If this is the case, then adjust the *duration* parameter to slow down the animation. Machines with blue LCD pixels (Version K) shouldn't have the problem.

ANIMATE can produce some very entertaining effects, but it's also very useful in showing 4-dimensional functions—and in viewing 3-D plots from various viewpoints without having to re-draw them every time. Subsequent chapters will show how to use it effectively. ANIMATE (and the other 3-D tools suite) is based on the work of some real giants in the HP 48 programming world.

*Some early editions of the User's Guide contain an erroneous description of the list. If you follow the directions in the UG, page 9-10, you may get an ANIMATE ERROR: Wrong Argument Count message. If so, after reading this explanation, take a permanent ink pen and enter the correct version of the list in the UG.

The PRG- PICT Menu

The PRG- PICT menu contains programmable graphics functions for modifying PICT:

PICT | PDIM | LINE | TLINE | BOX | ARC
PIXON | PIXOF | PIX? | PVIEW | PX→C | C→PX

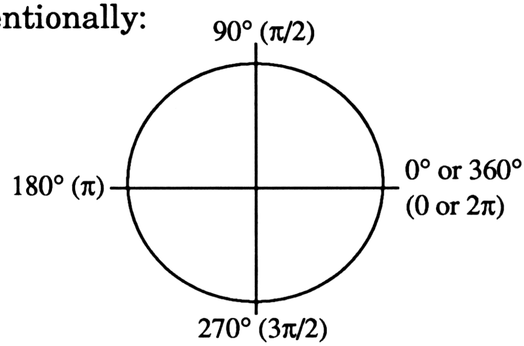
PICT is a typing aid (but unfortunately, you cannot use ← or → to easily store or recall the contents of PICT).

PDIM is a powerful command that allows you to re-dimension PICT. It can affect PICT and PPAR in different ways—best explained on pages 24-3 to 24-6 of the User's Guide.

The commands BOX, LINE and TLINE require two arguments for end-points or diagonal corners. Results are identical to those achieved with the BOX, LINE and TLINE in the interactive graphics environment. You can express the points either in user units—via complex numbers: (-1.35, 20.6)—as a CAD system does; or as decimal integers representing the pixel column and row: { # 31d # 55d }. In either case, the first term represents the *x*-axis and the second term the *y*-axis. The top left pixel of a grob is always { # 0d # 0d }.

The commands C→PX and PX→C allow you to convert between the two, *according to the current values of PPAR.* Remember that each directory will have its own PPAR and its own unique user units.

The interactive graphics environment has a **CIRCL** operation but no **ARC**; here you have an **ARC** but no **CIRCL** (a circle is a 360° arc). **ARC** takes four arguments. The first two are the center of rotation (in Stack Level 4) and the radius of the arc (Level 3). The units (user vs. pixel) for these arguments must match (a radius' user units are *x*-axis units only; you can't get an ellipse). The last two arguments are the starting angle (Level 2) and the ending angle (Level 1). Angles are measured conventionally:



In the interactive graphics environment, **DOT+** and **DOT-** determine whether a pixel will be turned on or off as the cursor lands on it. Pressing one key cancels the other; pressing the same key twice leaves the pixels untouched as the cursor moves around. In programs, use PIXON and PIXOFF to do this. They operate on the pixel located at the coordinates given in Level 1. The pixel may be expressed as a complex number in user units, or as a list of two binary integers. To test individual pixels, use the **PIX?** command (returns 1 if the pixel is on; 0 if it's off).

And this tool, TPIX (Checksum: # 29273d Bytes: 38.5), toggles any given pixel:

```
« DUP IF PIX? THEN PIXOFF
  ELSE PIXON END »
```

You already know about **PVIEW**. It appears in both the (PRG)-**PICT** menu and the (PRG)-**OUT** menu. Speaking of which,...

The PRG- OUT Menu

PVIEW TEXT CLLCD DISP FREEZ MSGB

The first page of this menu contains commands that control the display. You already know about PVIEW (recall page 106).

TEXT simply restores the normal stack display.

CLLCD simply clears the display. Usually the 48 does it automatically, but sometimes—as with DISP—you must do it yourself.

Use DISP to build a *text* display other than the normal Stack display. The display is divided into 7 lines. DISP takes the object from Level 2 and displays it in size-2 font (8 pixels high), on the line specified in Level 1. The uppermost line is numbered 1; the lowest, 7. DISP also honors NEWLINE's (α →↵); grobs can have more than one line of text.

FREEZ prevents parts of the display from updating until some key is pressed. The Level-1 integer indicates which part(s) to freeze:

1	Status area	5	Menu & Status area
2	Stack & Command Line	6	Menu & Stack/ Command Line
3	Status & Stack/Command Line	7	Entire display
4	Menu		

MSGB (MSGBOX) takes a string from Level 1 and displays it in a *message box*—like the kind you get when you press →PLOT →Σ ENTER, except for the little "alert" sign that the built-in applications use. MSGBOX will try to parse your string (breaking it only at spaces, if possible) and will display only the first 75 characters.

Other Graphics Commands

You can also add grobs with the [+] key and invert them with the [+/−] key or via the NEG command. Use the NEG function to create inverse video effects in your applications. Use addition to combine small grobs quickly or "stamp" frames and legends onto common-sized grobs.

For two grobs of *exactly* the same size, addition goes pixel-by-pixel, equivalent to: « grob1 { # 0d # 0d } grob2 GOR »

Inverting a grob inverts all the pixels, turning the black ones white and the white ones black. Just for fun, put the SINE grob onto the Stack. Then [+/−] PICT [STO] and press [◄] to see your creation....

Grobs with row sizes that aren't multiples of 8 are inverted only insofar as their bits actually represent pixels. Thus, GROB 2 2 0000 inverted becomes GROB 2 2 3030. The 3's represent the displayed pixel pairs, but the 0's are placeholders—bits that don't represent pixels.

And NEG and [+] together do a GAND ("Grob AND"), a function HP seems to have omitted. Here's GAND (Checksum: # 61392d Bytes: 31):*

« NEG SWAP NEG + NEG »

Store this into your TOOLS directory. Then try it out, using GROB 2 2 3000 and GROB 2 2 1010. Result: GROB 2 2 1000

*If the grobs are not of the same size, use this version of GAND (Checksum: # 60472d Bytes: 36), which takes the same arguments as GOR, GXOR and REPL:

« NEG ROT NEG ROT ROT GOR NEG »

Building a Toolbox

With all of its capabilities, the 48 is still missing some useful commands. Such commands are called utilities, and now you're going to create them yourself—along with some "standard" grobs for use in testing/troubleshooting programs. You've already created the SEE utility (in your **TOOLS** directory), to "view" a grob on the Stack. Also, you have TPIX to toggle pixels, GAND for Boolean addition, and GSIZE and $SIZE for memory management.

How about a pair of utilities to store/recall grobs from/to the graphics display? Suppose you create a gorgeous picture—how *do* you save it? Exit to the Stack display, put the name 'GORGEOUS' on Level 1, and use a program, named STOPIC (Checksum: # 49324d Bytes: 30.5):

```
« PICT RCL SWAP STO
»
```

The grob goes onto the Stack and is then SWAP'ped to bring the name to Level 1. Then the grob is stored and the Stack is left as before. Put STOPIC into your **TOOLS** directory.

RCLPIC does the opposite, taking an object name from Stack Level 1 and (only if it's a grob) storing it into the graphics display. As RCLPIC avoids using GRAPH and PVIEW, it's very general and programmable:

```
« DUP
  IF VTYPE 11 SAME
  THEN RCL PICT STO
  ELSE →STR
    " not a GROB!"
    + DOERR
  END
»
```

RCLPIC (Checksum: # 12051d Bytes: 90.5) chastises you if the named object isn't a grob. Store it alongside STOPIC, in your **TOOLS** directory.

Now you need to create three empty grobs (change to the PICS directory now, to store them there). Create a 200×200 grob called BIG; a 131×64 grob called NORMAL; and a 2×2 grob called TINY, as follows:*

For each grob, put the number of columns (# 200d, # 131d or # 2d) onto Stack Level 2; the number of rows (# 200d, # 64d or # 2d) onto Level 1, and select BLANK from the PRG-GROB menu. Then type the name ('BIG', 'NORMAL' or 'TINY') into the Command Line and press STO.

Next, create two non-empty grobs: First, load the Stack with any four objects, then store the Stack display as a grob, by pressing PRG GROB NXT LCD→ NXT 'DISPLAY' STO.

Second, type GROB 5 8 4040E0E0F1F14040 ENTER 'ARROW' STO, to build and store an "arrowhead" grob.

With these 5 good grobs to work with, switch to the TOOLS directory to create a custom menu. This custom menu is defined in a list inside a program (feel free to modify the list to serve your own needs):

```
«
  { PICS PICT BLANK ERASE →LCD LCD→ →GROB SEE
    STOPIC RCLPIC
  }
  MENU
»
```

Store this menu-building program called GRAFX (Checksum: # 41596d Bytes: 75) in your TOOLS directory.

*If you're working on an HP 48G (not GX), your machine's memory is undoubtedly getting crowded. Now is a good time to back up the directories on your 48, and then delete anything you won't need immediately, like the G.CH2 and G.CH3 directories. You may also wish to omit the 200×200 grobs in these lessons, if they won't fit into your machine.

Sines and Big Sines

In Chapter 4, you used a sine wave to illustrate some of the graphics capabilities of the 48. Go back now and repeat the exercise on page 90 (don't forget to use RADians mode).... Then store this plot in a grob called SINE (type CANCEL CANCEL PICS 'SINE' ENTER STOPIC).

Now create a sine wave plot using the BIG grob: Make sure you're in the PICS directory. Put the name 'BIG' on Level 1 and execute RCLPIC. Press ↱ PLOT, and be sure the current equation is 'Y=SIN(X)'. Then set H-VIEW to -10 and 10 and V-VIEW to -1.1 and 1.1 (do *not* select AUTOSCALE—that would reset XRNG and YRNG). Now press ERASE DRAW to draw the plot... (cookie time).

When the plot finishes, press EDIT NXT LABEL to add the finishing touches, and then have a look at this monster. With the PICTURE menu displayed, the arrow keys have the following functions:

1. Unshifted arrow keys move the cursor within the display "window." At the edge of the window, they scroll the display across the grob—to its actual edge.

2. ↱'ed arrow keys jump the cursor to the edge of the window. At the edge of the window, ↱'ed arrow keys jump the cursor and display to the edge of the grob.

3. ↰ ◀ puts you in scrolling mode. Think of scrolling as viewing a large picture through a small window or frame: You don't move the picture, you move the window.

Press ⬅◀ now, to get into scrolling mode. In scrolling mode, no cursor is visible, and the arrow keys have the following functions:

1. Unshifted arrow keys scroll the display across the grob.
2. ➡ed arrow keys jump the display to the edge of the grob.
3. ⬅◀ returns you to the interactive graphics environment.

Press CANCEL twice to return to the Stack display. Then, in the PICS directory, enter the name 'BIGSINE' onto Level 1 and execute STOPIC.

Now you can review both SINE and BIGSINE any time you want—and you can also practice with other graphics functions on these grobs.

Review

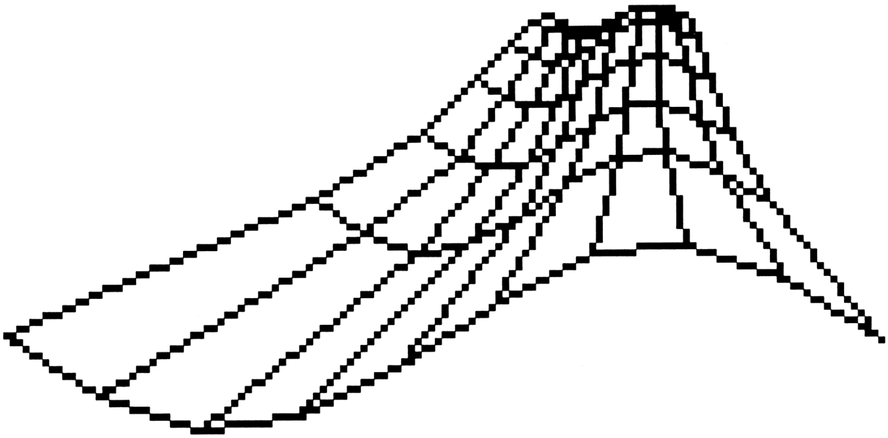
In this chapter, you explored the graphics commands in several of the 48's built-in menus. Then you began to augment those commands with your own graphics "toolbox"—a collection of programs and sample grobs useful in your own graphics development work.

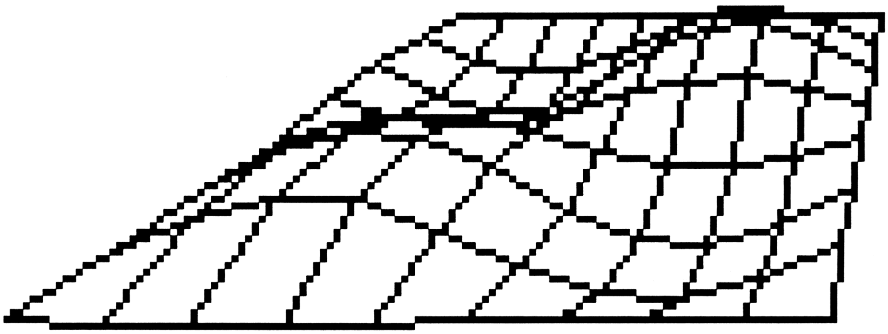
At this point, then, you should have these programs in **TOOLS**:

GRAFX	builds a custom menu to make graphics work easier.
RCLPIC	recalls a grob to the graphics display.
STOPIC	stores the graphics display in a grob.
GAND	does a pixel-by-pixel "AND" of two grobs.
TPIX	toggles individual pixels on and off.
$SIZE	finds the byte-size of a grob's string representation.
GSIZE	finds the size of a grob, in bytes.
SEE	graphically displays the contents of a grob.

And you should have these grobs in **PICS**:

BIGSINE	a 200×200 sine-wave plot, with axes
ARROW	a 5×8 arrowhead
DISPLAY	a 64×131 "snapshot" of the Stack display
TINY	a blank 2×2 grob
NORMAL	a blank 64×131 grob
BIG	a blank 200×200 grob
EMPTY	a blank 64×131 grob
SINE	a 64×131 sine-wave plot, with axes





6: Three-Dimensional Graphics

The Basics

"See severed heads that almost fall right in your lap! See that bloody hatchet coming right at you!"

— Weird Al Yankovic, commenting on 3D as an entertainment medium.

Unbeknownst to Weird Al, some more constructive uses for three-dimensional graphics were presented at HP user's groups over the past several years. They were marvelous application examples—and great algorithms for the HP 48S/SX. Then a math professor developed and placed into the public domain a set of 3-D plotting utilities he called "SUITE3D," which received such a positive response from the HP48 user's community that HP adapted it for inclusion in the HP48G/GX.

Although the 3-D tools in the HP 48G/GX don't pretend to be as good as those in expensive CAD packages, they are indeed useful at least for "visualizing functions of two variables," if not for analyzing them (and HP included some rudimentary 3-D analysis tools anyway.)

The best introduction to the 3-D plotting tools is in the section called "Plotting Functions in Three Dimensions" in the Quick Start Guide (QSG) that came with your machine. More detail is given in chapter 23 of the User's Guide (UG), "Plot Types," starting with the section called "Plotting Functions of Two Variables," on page 23-22.

If you haven't yet read those sections, then now is a good time to set this book down, get a handful of cookies and work through those sections of the QSG and the UG....

On page 6-7 of the QSG are two diagrams explaining the concepts of *view volume*, *view plane*, and *eyepoint*. The more clearly you understand these concepts, the better you can use the 3-D tools on the 48.

Imagine looking though the window of a pet store at some puppies in a playpen inside the store. They can't escape the playpen; it limits the area in which you can view them. The *view volume* in 3-D plotting is like the playpen: the display of the function is confined within it.

The diagrams in the QSG shows the orientation of the *x*-, *y*- and *z*- axes as they relate to these concepts:

- The "floor of the pet shop" is the *x-y* plane; the *z*-axis is vertical.
- The shop window—between you and the puppies—is parallel to the *x-z* plane. This is the *view plane*.
- You are standing along the *negative y-axis*, some distance from the window. That vantage point is your *eyepoint*.

The 48 imposes two restraints on the plotting tools:

- Your eyepoint must stay at least one unit away from the view plane—on the outside only. You can't mash your nose against the glass to get a better view, nor can you go into the store to get a better look, nor can the puppies' playpen be wheeled outside.*
- The view plane must stay parallel to the *x-z* plane. You can't twist or bend or lever open the store window.

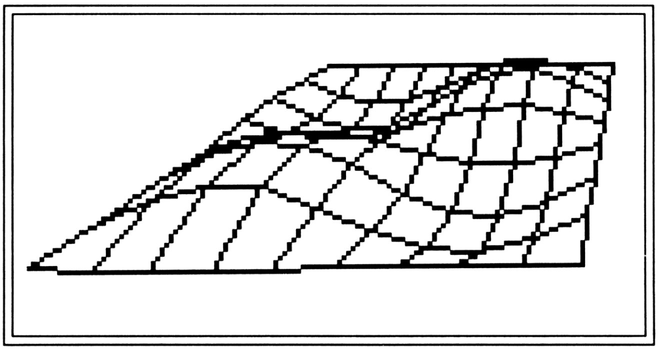
*You can get around these restraints by use of some mathematical sleight-of-hand, such as scaling and rotating functions. We'll talk about rotating a plot in the following pages.

The 48 gives you six different tools to use in 3-D analysis, in the [⬅][PLOT] [NXT] **3D** **PTYPE** menu: **SLOPE** **WIREF** **YSLIC** **PCON** **GRID** **PARSU**. To compare these, you need a function that can be displayed well in each of the six tools. Use the [➡][PLOT] application or the [⬅][PLOT][NXT] **3D** menus to set up these parameters, which are stored in VPAR (to be discussed next):

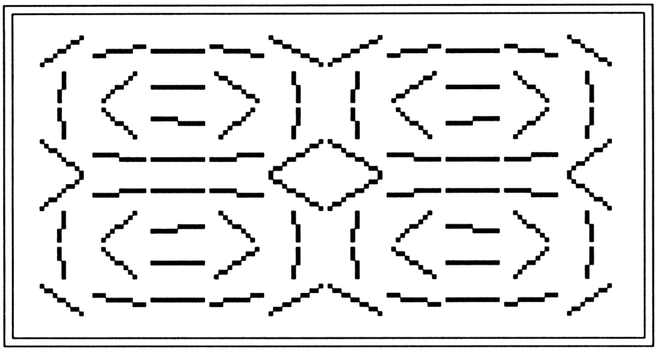
```
EQ: 'SIN(X)*SIN(Y)'         ∡: RAD
INDEP: X    STEPS: 10
DEPND: Y    STEPS: 8
XVOL: -3.2 3.2    YVOL: -3.2 3.2    ZVOL: -1 1
XRNG: -3.2 3.2    YRNG: -3.2 3.2
XE: 4    YE: -10    ZE: 8
```

(To set this from the PLOT input form: PICT [⬅][PURG][➡][PLOT][SIN][α][X] [▶][X][SIN][α][Y][ENTER][⬅][RAD][▲][▲] **CHOOSE** [α][W][ENTER][▼][▼][▶][1][0][ENTER][▶] [8][ENTER] **OPTS** [3][•][2][+/-][ENTER] [3][•][2][ENTER] [3][•][2][+/-] [ENTER] [3][•][2] [ENTER][1][+/-][ENTER][1][ENTER][4][ENTER][1][0][+/-][ENTER][8][ENTER] **OK**.)

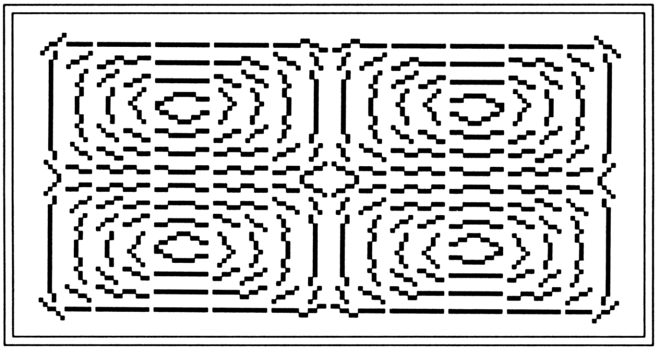
WIREF, or WIREFRAME, is the most commonly recognized form of 3-D plotting. As the name implies, a *wireframe* plot is an array of points in space, connected by line segments parallel to the *x-z* and *y-z* planes. Here is a wireframe plot of 'SIN(X)*SIN(Y)' (press **ERASE** **DRAW**):



PCON (PCONTOUR) creates a *pseudo-contour* plot: an array of points on the *x-y* plane, with a short line segment drawn through each point, showing the direction a *contour line* (a curve of constant *z*-value, or "altitude") would have at that point. Here's the same 'SIN(X)*SIN(Y)' in a pseudo-contour plot (press CANCL, ▲ to TYPE: field, then CHOOS and ▼ to Ps-Contour, ENTER ERASE DRAW):

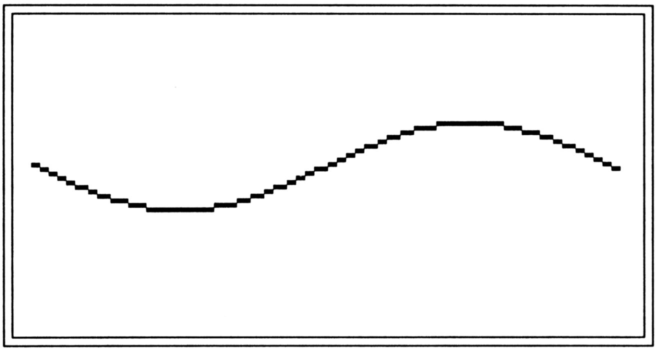


This plot is not very easy to decipher; the point spacing is too coarse. Try increasing the X steps from 10 to 20, and the Y steps from 8 to 16 (press CANCL ▼ ▼ ▶ 2 0 ENTER ▶ 1 6 ENTER ERASE DRAW):

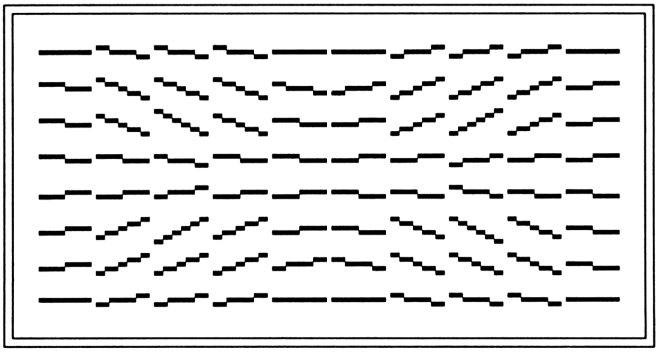


In both plots, you can see the maxima (peaks) in the upper-right and lower-left quadrants, and the minima (valleys) in the upper left and lower-right quadrants. Often, you can take a printout of a pseudo-contour plot and use a pen to connect the lines to generate the contours (or, use the CONTOUR program in Chapter 9 to draw real contours).

YSLIC (YSLICE) is yet another view of the function. A *y*-slice plot is a series of two-dimensional plots, generated as a function of X, with Y held constant for each slice—as if you took the wire-frame plot and cut it into slices. YSLICE uses the ANIMATE routine to demonstrate dynamically how the function varies with Y. It leaves a counter and a pile of grobs on the stack. Press CANCL, ▲ to TYPE: field, CHOOS α Y ENTER ▼ ▼ ▶ 1 0 ENTER ▶ 8 ENTER ERASE DRAW ... CANCEL to stop. (Note: the MULTIPLOT program in Chapter 9 offers an alternative to YSLICE.)



At first, a SLOPE (SLOPEFIELD) plot resembles a pseudo-contour plot (choose the Slopefield type plot, then ERASE and DRAW):



Like PCONTOUR, SLOPEFIELD produces an array of points on the *x*-*y* plane, with a line segment through each point. But here the slope of the line segment indicates the value ("altitude") of the function at that point. Compare the two plot types: The high points on the wireframe correspond to the steep lines on the slopefield; the middle, zero-value points on the wireframe correspond to the level lines on the slopefield.

The last two plot types, **GRID** (GRIDMAP) and **PARSU** (PARSURFACE) are better left to people who understand the math behind them.
Now, about VPAR: VPAR (short for View PARameters) is the reserved name of the list containing all the information necessary for 3-D plotting. (Depending on the plot type and the parameter settings in VPAR, the 48 may also adjust some PPAR parameters such as Xrange and Yrange.) VPAR is a list of 15 real numbers:

$$\{\ x_{left}\ \ x_{right}\ \ y_{near}\ \ y_{far}\ \ z_{low}\ \ z_{high}\ \ xx_{left}\ \ xx_{right}\ \ yy_{left}\ \ yy_{right}\ \ x_e\ \ y_e\ \ z_e\ \ n_x\ \ n_y\ \}$$

All of the VPAR parameters may be set manually from the (⬅)(PLOT)(NXT) **3D** **VPAR** menu commands (even within a program) or the **PLOT** and **PLOT OPTIONS** input forms. Any given parameter may also be set via the command sequence « ... VPAR *n* ROT PUT ... » (where *n* is the position in the VPAR list of that parameter). Similarly, you can retrieve any parameter or parameters via « ... VPAR *n* GET ... » or « ... VPAR n_1 n_2 SUB EVAL ... ».

The first three pairs of numbers in VPAR define the view volume (and in the VPAR menu or in a program, you do enter them as pairs of real numbers: -1 1 YVOL, for example). Note that $y_{near} < y_{far}$, always. Note also that only WIREFRAME, YSLICE and PARSURFACE use z_{low} and z_{high}; PCONTOUR, SLOPEFIELD and GRIDMAP ignore them.

Here's how to enter the values from a program or the VPAR menu.

x_{left} x_{right} XVOL y_{near} y_{far} YVOL z_{low} z_{high} ZVOL

To retrieve their values: VPAR 1 2 SUB EVAL for x_{left} and x_{right}
VPAR 3 4 SUB EVAL for y_{near} and y_{far}
VPAR 5 6 SUB EVAL for z_{low} and z_{high}

The next two pairs of numbers define the range for the *input sampling grid* used by GRIDMAP and PARSURFACE (the other plot types ignore these or set them to the corresponding values in the view volume).

To enter the values from a program or the VPAR menu:

xx_{left} xx_{right} XXRNG yy_{left} yy_{right} YYRNG

To retrieve their values: VPAR 7 8 SUB EVAL for xx_{left} and xx_{right}
VPAR 9 10 SUB EVAL for yy_{left} and yy_{right}

The next three values, x_e, y_e and z_e, define the eyepoint. Only the WIREFRAME and PARSURFACE plots care about the eyepoint, which you enter as three real numbers, like this: x_e y_e z_e EYEPT

To retrieve the eyepoint coordinates: VPAR 11 13 SUB EVAL

Remember that y_e must always be at least one "unit" less than y_{near}, the lesser y-coordinate of the view volume. If, for example, you try to set y_e to -4.0 and y_{near} is -3.2, the 48 will reset y_e to -4.2.

The final two parameters, n_x and n_y, specify how many points will be calculated in each direction. YSLICE appears to ignore the n_y parameter (the other plot types use it), but all plot types need n_x.

To enter the values from a program or the VPAR menu:

n_x NUMX n_y NUMY

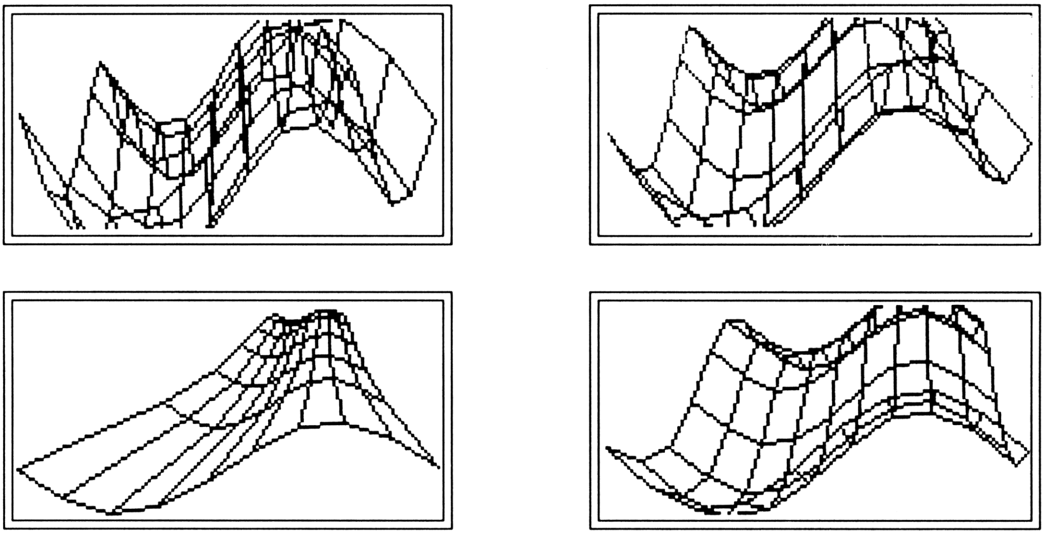
To retrieve their values: VPAR 14 GET for n_x
VPAR 15 GET for n_y

Getting the Most Out of Wireframe Plots

Every kind of 3-D plot is useful, but most persons use a WIREFRAME plot most often. This section will introduce some tools that help you utilize the WIREFRAME plotting tool more effectively.

Choosing an Eyepoint

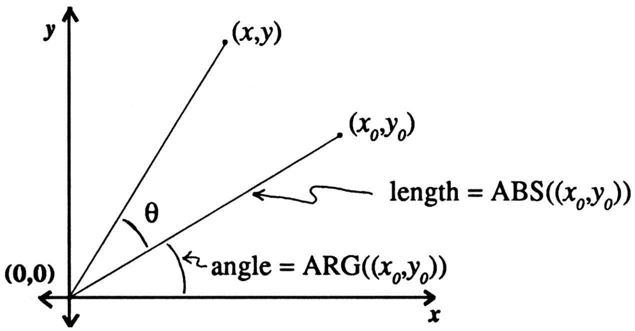
The 48 automatically adjusts the plotting limits so that, no matter what eyepoint is selected, the view volume is centered in the display. The tricky part is selecting an eyepoint that gives an informative view of the plot. For example, here are 4 different views of SIN(X)+SIN(Y):



For most functions, the optimum eyepoint is at least one view-volume away from the function (that's y_e). Height and vertical placement are more subjective, but it's usually good to place the eyepoint one view-volume above the function (that's z_e), and slightly shifted to one side or the other (that's x_e)—to visually disrupt any symmetry in the function.

Rotating the View

It would be nice if you could see the function you're plotting from back angles as well as from the front. Unfortunately, you can't get past the view window (and neither can the view volume). To get around the problem, *rotate* the function itself—recast x and y into something useful in the new coordinate system. A complete rotation involves a lot of vector arithmetic and can easily double the time to generate a single plot (to say nothing of a series of them), but you can rotate around the z-axis, so that only x and y need to be modified. Examine this figure:



Suppose you have a vector, x_0+y_0i, and you want to rotate it θ° in the x-y plane, around the point (0, 0). Expressed as a complex number in polar form, the vector is (r, *Angle*), where r is the *magnitude* of the vector (ABS((X0, Y0)) on the 48). *Angle* is the *polar angle* from the positive x-axis to the vector (ARG((X0, Y0)) on the 48). To rotate the vector, you multiply the complex number by the unit vector (1, θ). The result:*

in polar coordinates: (ABS((X0, Y0)), ARG((X0, Y0))+θ)

in rectangular coordinates: (ABS((X0, Y0))*COS(ARG((X0, Y0))+θ),
ABS((X0, Y0))*SIN(ARG((X0, Y0))+θ))

*Note that when θ=0, this general form reduces to (X0, Y0).

To make this all convenient, you can create a program called ROXY ("ROtate in X and Y") that will convert any algebraic expression in x and y into one that can be rotated as described. The program uses a global variable, θ, so that it will work inside the 3-D plotter. It takes a symbolic object (a function of x and y) from Level 1 and returns the transformed version. (This formula uses x and y instead of X0 and Y0.)

ROXY (Checksum: # 42966d Bytes: 195.5)

```
«
   { X x } ↑MATCH DROP
   { Y y } ↑MATCH DROP
   { x 'ABS((X,Y))*COS(ARG((X,Y))+θ)' } ↑MATCH DROP
   { y 'ABS((X,Y))*SIN(ARG((X,Y))+θ)' } ↑MATCH DROP
»
```

Try it: Create the hyperboloid 'X*Y' and store it in your **TOOLS** directory as 'HYP'. Then press 'HYP' (→)(RCL) to put 'X*Y' onto the Stack. Now press (VAR) **ROXY** to get:

```
'ABS((X,Y))*COS(ARG((X,Y))+θ)
                    *(ABS((X,Y))*SIN(ARG((X,Y))+θ))'
```

You can then store this ROXY'ed form of HYP into EQ and use it with a program like this (be sure to set DEGrees mode and WIREFRAME plot type before you start):

TRYIT (Checksum: # 46869d Bytes: 228.5)

```
« { # 0d # 0d } PVIEW
   0 330
   FOR t t
      'θ' STO ERASE DRAW PICT RCL
      { # 1d # 1d } t 1 →GROB REPL 30
   STEP
   { 12 { # 0d # 0d } .2 100 } ANIMATE
»
```

Approximate running time: 12:23 to create 12 frames of HYP.

You can use ROXY with any function $f(x,y)$, but it's a good idea first to name the *non*-transformed version of your function and use it to set up VPAR. When you've positioned VPAR correctly, then you can use ROXY to put a transformed version of the function on the Stack, store it into EQ, and run TRYIT.

If you want to try your hand at rotations in other planes or around other axes, you'll need to do some reading on coordinate transformations. You can find some good work on coordinate transformations in HP48 Insights, by William C. Wickes, or in the HP42S Owner's Manual from Hewlett-Packard. Both books are available from EduCalc Mail Store (1-800-677-7001).

Translating

On a related subject, here's how to *translate* a function. The routine TRXY uses the global variables ΔX and ΔY to define the transformed function.

TRXY (Checksum: # 43460d Bytes: 82.5)

```
«
   { X 'X-ΔX'} ↑MATCH DROP
   { Y 'Y-ΔY' } ↑MATCH DROP
»
```

Try it: Since HYP isn't a good sample function for this program, create 'SIN(X)+SIN(Y)' and store it (in **TOOLS**) as 'EGGS'. Then recall 'EGGS' to the Stack and run TRXY on it. Store the resulting equation into EQ (not into 'EGGS'), and turn on **RAD**ians mode. Then store 0.7854 (that's π/4) into ΔX; and 1.571 (that's π/2) into ΔY.

Now enter and execute the following program (Checksum: # 1324d Bytes: 234.5). Be sure you set **RAD**ians mode before you start.

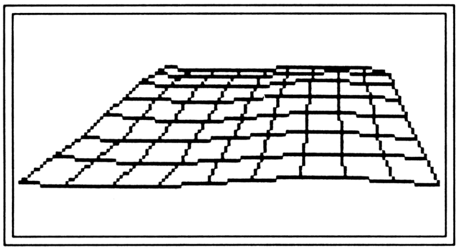
```
«
   { # 0d # 0d } PVIEW 0 7
   FOR dx dx
      'π/4' →NUM * DUP 'ΔX' STO 2 / 'ΔY' STO
      ERASE DRAW PICT RCL { # 1d # 1d }
      dx 1 →GROB REPL
   NEXT
   { 8 { # 0d # 0d } .2 100 } ANIMATE
»
```

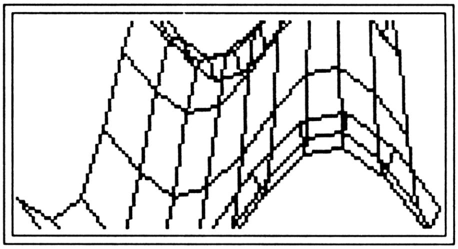
Approximate running time: 5:08 to create 8 frames of EGGS.

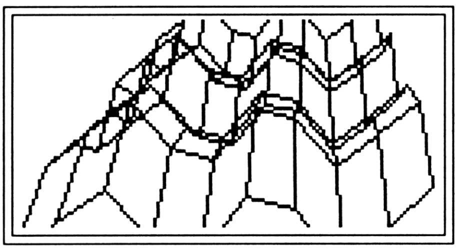
Zooming and Panning

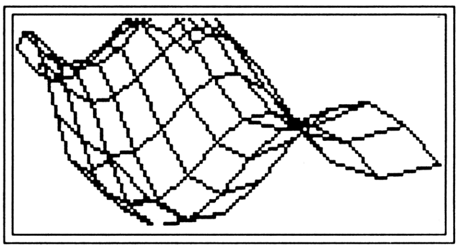
The 3-D plotting routines built into the HP48G/GX are written to take fullest advantage of the 48's small display. This means that the routines will distort the view as required to fill the display, even if the eyepoint is miles away from the view volume—as if there were a huge telescopic lens at the eyepoint, always trained on the view volume.

You can see this by plotting the EGGS function repeatedly, varying the XVOL, YVOL and ZVOL parameters. You'll notice that the plot always extends from the left edge of the display to the right edge. Strangely enough, the *z*-axis is *not* automatically scaled; it's possible to adjust ZVOL so that your plot either becomes very flat, or extends beyond the top and bottom edges of the display:









Even with this distortion, it is useful to look at a function from several different angles and distances. Here are three programs that create series of plots while varying the three components of the eyepoint. The running times indicated are for the following plot setup:

TYPE: Wireframe
EQ: '0' (this is a flat plane)
INDEP: X STEPS: 3
DEPND: Y STEPS: 3
XVOL: −1 1 YVOL: −1 1 ZVOL: −1 1
XE: 1 YE: −3 ZE: 2

This first program varies x_e, the x-component of the eyepoint. In movie parlance, this is called "panning," so the program is called XPAN. The program takes three arguments from the Stack: beginning x_e, ending x_e, and x_e increment. It leaves a stack of grobs and an ANIMATE counter.

XPAN (Checksum: # 28913d Bytes: 168)

```
«
   → xinc
   «
      FOR x
         'VPAR' 11 x PUT ERASE DRAW
         PICT RCL { # 1d # 1d }
         VPAR 11 13 SUB 1 →GROB REPL
      xinc STEP
   »
   DEPTH ANIMATE
»
```

Try it now: type 1 -3 2 EYEPT -8 8 1 XPAN.... It will take a little over 2 minutes to create 17 grobs.

YPAN does the same thing with y_e, the y-component of the eyepoint. It takes three arguments: beginning y_e, ending y_e and y_e increment.

YPAN (Checksum: # 24445d Bytes: 168)

```
«
   → yinc
   «
     FOR y
        'VPAR' 12 y PUT ERASE DRAW
        PICT RCL { # 1d # 1d }
        VPAR 11 13 SUB 1 →GROB REPL
     yinc STEP
   »
   DEPTH ANIMATE
»
```

Try it now: type 1 -3 2 EYEPT -3 -19 -1 YPAN....

This third program, ZPAN, varies z_e, the vertical component of the eyepoint, using three z_e arguments in the same manner as XPAN and YPAN.

ZPAN (Checksum: # 13701d Bytes: 135.5)

```
«
   → zinc
   «
     FOR z
        'VPAR' 13 y PUT ERASE DRAW
        PICT RCL { # 1d # 1d }
        VPAR 11 13 SUB 1 →GROB REPL
     zinc STEP
   »
   DEPTH ANIMATE
»
```

Try it now: type 1 -3 2 EYEPT -8 8 2 ZPAN....

Since the three programs are so similar, you may be able to combine them into one all-purpose PAN program. Can you?

Plotting in Four Dimensions

If you could make the function vary with time as well as with x and y, then you could create 4-dimensional plots. Good news: You can use ANIMATE to do just that. Since the 3D plotting tools already work on functions of X and Y, it is easy to create a function of X, Y and T. You simply make T a global variable, create several plots of the function for different values of T, and use ANIMATE to review them.

In fact, you can write a program to do the plotting for you automatically. This four-dimensional plotting program, called PL4D, takes three arguments from the stack: the starting time, ending time and time increment. Like X/Y/ZPAN, it uses ANIMATE to display the plots and leaves them on the stack with the ANIMATE counter.

PL4D (Checksum: # 37239d Bytes: 128):

```
«
   → tinc
   «
      FOR t
         t 'T' STO ERASE DRAW
         PICT RCL { # 1d # 1d }
         T 1 →GROB REPL
      tinc STEP
   »
   DEPTH ANIMATE
»
```

As an example, try turning a paraboloid inside out. One expression for a paraboloid is: $z = ax^2 + by^2$. So create the expression 'A*X^2+B*Y^2' and store it as BOLOID. Now store the expression '0.2*T' into 'A'; and the expression '0.3*T' into 'B'. A and B are now functions of T.

Set up your plotting parameters as follows:

```
INDEP: X    STEPS: 7
DEPND: Y    STEPS: 7
XVOL: -1 1   YVOL: -1 1   ZVOL: -1 1
XE: .8   YE: -3   ZE: 1.5
```

Now type `-4 4 2 PL4D` and see what happens....

Here's another example: Store the expression `'T/(X^2+Y^2)'` into EQ. Set up your plotting parameters as follows:

```
INDEP: X    STEPS: 10
DEPND: Y    STEPS: 8
XVOL: -1 1   YVOL: -1 1   ZVOL: 0 4
XE: 1    YE: -3   ZE: 3
```

Now type: `0 0.8 0.1 PL4D`....

If you have already written a consolidated version of `X/Y/ZPAN`, you may want to consider adding `PL4D`'s capabilities to it. On the other hand, that may introduce so much programming overhead as to weigh down the program, making it too big and too slow. That's your decision.

Extensions and Alternatives to ANIMATE

ANIMATE is one of those "why didn't I think of that?" routines that was just begging to be written. The core of the routine could be written as:

```
«
   { # 0d # 0d } PVIEW
   1 SWAP
   FOR n
      n ROLL PICT
      { # 0d # 0d } ROT REPL
   NEXT
»
```

(Of course, this quick version ignores the optional list argument and omits all type-checking, stack size checking and so on. All that extra code would naturally be built around the core shown above.)

The input for ANIMATE is simple in its beauty: a stack of grobs and a counter. This arrangement is important in the several programs you can create to enhance ANIMATE: PRANIM, SSTEP, BSTEP and COMBINE.

PRANIM allows you to print out the sequence of grobs on an 82240B infrared printer, or on a PCL- or Epson-compatible printer (if you have the PCL or Epson graphics print driver installed). It leaves the Stack unchanged, as long as it's not interrupted while it's running.

PRANIM (Checksum: # 63294d Bytes: 49)

```
«
   1 SWAP FOR n
   n ROLL PR1 NEXT
   n
»
```

SSTEP allows you to view one grob at a time, at your own pace. BSTEP does the same thing, only backwards, by using the ROLL command instead of ROLLD. The Stack is unchanged, except that the grobs may be out of order from the stepping.

SSTEP (<u>Checksum</u>: # 515d <u>Bytes</u>: 122.5)

```
«
   → n
   «
     n ROLL DUP PICT
     { # 0d # 0d } ROT REPL n
     { # 0d # 0d } PVIEW 7 FREEZE
   »
»
```

BSTEP (<u>Checksum</u>: # 52273d <u>Bytes</u>: 122.5)

```
«
   → n
   «
     n ROLLD DUP PICT
     { # 0d # 0d } ROT REPL n
     { # 0d # 0d } PVIEW 7 FREEZE
   »
»
```

COMBINE is useful with YSLICE (and marginally so with WIREFRAME) for creating composite plots by superimposing all the grobs on one another. COMBINE removes the counter and grobs from the Stack, leaving a single grob on Level 1. MULTIPLOT in Ch. 9 uses the same principle.

COMBINE (<u>Checksum</u>: # 8942d <u>Bytes</u>: 39)

```
«
   1 - 1 SWAP START + NEXT
»
```

→LIST turns the arguments for ANIMATE into a list for storage or transfer; OBJ→ converts the list back into arguments for ANIMATE.

Although ANIMATE is an elegant routine, there are alternatives. One of these is to combine all frames of the animation into one larger grob, and use PVIEW to scan to different locations in the grob.

Consider that a full-size grob (131×64) requires 1,098 bytes. If you have a series of ten grobs, therefore, you will need 10,980 bytes of memory. If, instead, you paste all ten frames into a tall, skinny 131×640 grob, you will need 10,890 bytes—not too much different. But if you paste all ten frames into a short, wide 1310×64 grob, you will need just 10,506 bytes. If your available memory is getting short, that 476-byte difference is significant.

To experiment with such an alternative to the ANIMATE tool, here is a small (119.5-byte) program that will cycle through a short, wide grob of any size. It doesn't require a counter like ANIMATE, and it doesn't take any input. It assumes that you've already stored the grob into PICT, and it uses PVIEW to move around and display different parts of PICT. It moves even faster than top-speed ANIMATE:

```
«
   0 PICT SIZE DROP B→R
   FOR n
      IF
         'n≥524'
      THEN
         0 'n' STO
      END
      n R→B # 0d 2 →LIST PVIEW
   131 STEP
»
```

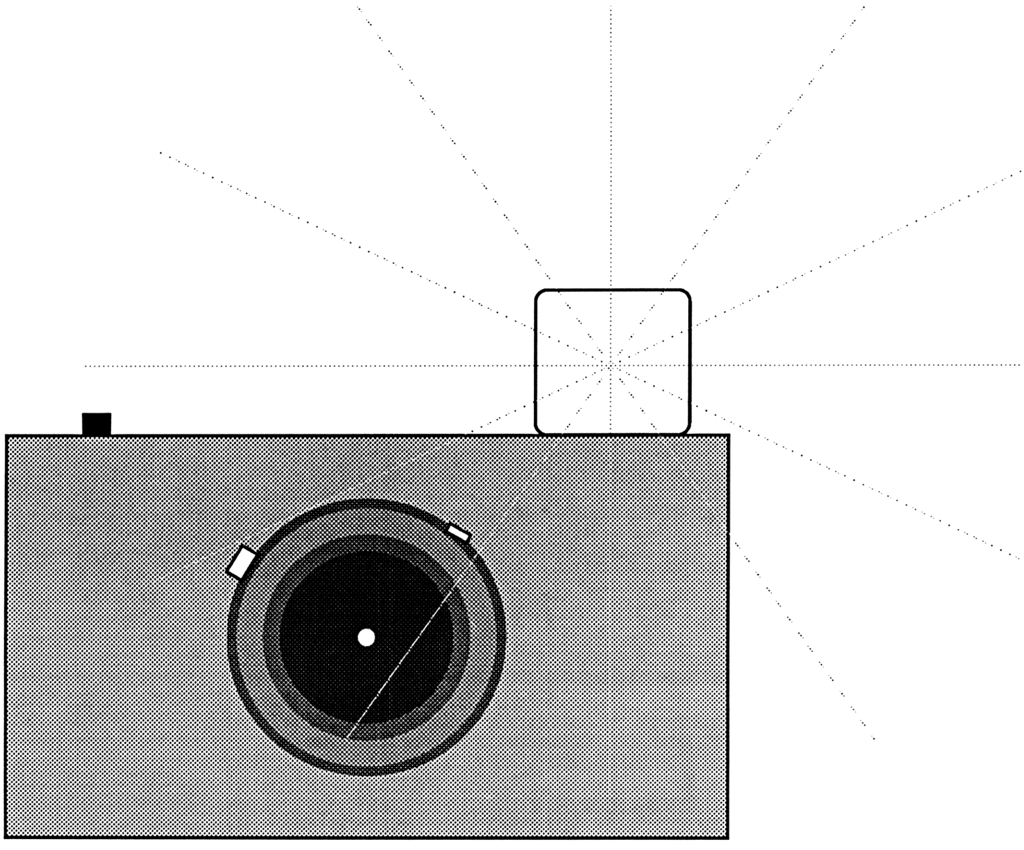
*CAUTION: If you try to run ANIMATE after you have turned PICT into a huge, misshapen grob like the ones used here, it may appear to "hang" with the first frame of the animation displayed and the "busy" annunciator lit. This has something to do with the oversized grob. To fix the problem, press [CANCEL] 'PICT' [↰][PURG] and try ANIMATE again.

Review

By now you should understand better the concepts of *view volume*, *eyepoint* and *view plane*. You should know how to manipulate the eyepoint, the plotted function and VPAR to get the best view of your 3D plots. By combining this knowledge with your knowledge of ANIMATE and its alternatives, you'll be able to use the 3-D tools to their fullest.

Although this chapter has concentrated on WIREFRAME plots, the principles you've learned can be applied to other 3D plot applications as well. Keep in mind that the HP48 never promised to be a handheld CAD tool or a 3D analytical tool; the applications are meant to help you "visualize" the relationships between three variables.

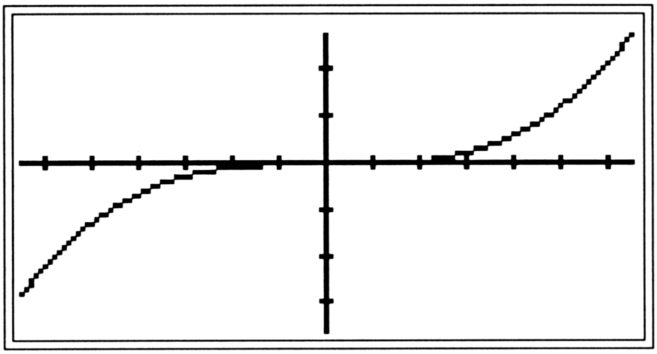
Don't be afraid to experiment. The 3-D plotting tools are perhaps the most complex tools on the 48, from a user's point of view, and they will take some practice before you become adept at using them. Remember what your band leader said.



7: Graphics Improvements

Opening Remarks

The PLOT routines give accurate graphical representations of your functions or statistical data. Still, a plot like the one below doesn't tell you much except the shape of the function. For example, you can't tell what the 3 roots of the function are—and you may not even recognize the function.

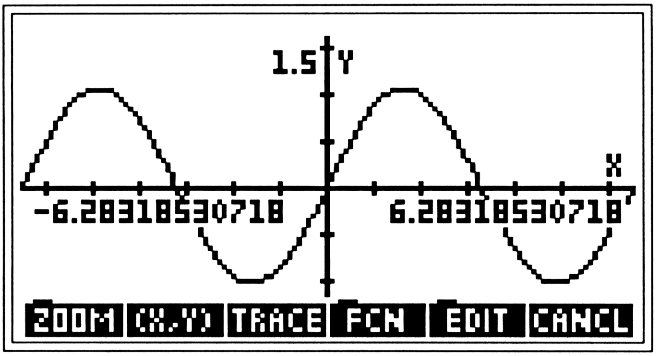


But the 48 does have a command to give the plot some scale—and then you can write a program to add text onto the plot anywhere you wish. You're going to do that here.

Also, you'll be learning how to use the BOX, LINE, TLINE and CIRCLE commands to make your plots more informative.

Labelling the Axes

If you've already tried axis labels, you probably got results like these:



The axis label format uses the current numeric display format. So an x-axis label of 2π might be plotted in the following ways, depending on your current numeric display format:

STD	6.28318530718
FIX 4	6.2832
SCI 1	6.3E0

Here's a simple exercise to try the different label formats.

1. Type 'SINE' RCLPIC to put your SINE grob into PICT.
2. Press ⬅◀ (or ⬅PICTURE).
3. Press EDIT NXT LABEL. You should see a picture like the one above.
4. Press CANCEL, then ➡MODES. Change the number format to, say, Fix 4 or Sci 1. Then repeat steps 1-3 to see how the labels change.

This technique also works with BIGSINE and other oversized plots.

Adding Text to Graphics

Suppose you have a 200×200 grob with a multifunction plot on it and you want to include the names of the three functions being plotted. There isn't a built- in function for adding that text.

You can use the cursor control keys with **DOT+** and **DOT-** to draw the individual letters, but that's tedious—and there's a better way.

Create a new command (call it GLABEL) that places text into the graphics display (or into PICT), with the upper left corner of the text at the coordinates specified. Like most 48 graphics functions, GLABEL should allow you to specify the coordinates either in user units or in pixels. Also, you should be able to specify a font size for the text: 1, 2 or 3 will select small, medium or large text; 0 will select either large text or special formatting (textbook or matrix format), whichever is applicable. Here's a Stack diagram for GLABEL:

	Stack Inputs	Stack Outputs
3:	Location { # *col* # *row* } or (*x*, *y*)	
2:	text string to be placed	(None)
1:	Text size (0, 1, 2 or 3)	

And here is GLABEL (Checksum: # 65476d Bytes: 33):

```
« →GROB PICT
  ROT ROT GOR
»
```

Store a copy of GLABEL in your **TOOLS** directory.

Now make two variations of GLABEL.

Name the first variation GL↓ (<u>Checksum</u>: # 60923d <u>Bytes</u>: 115.5):

```
« →GROB DUP2 PICT
  ROT ROT GOR SWAP
  DUP TYPE SWAP
  IFERR C→PX
  THEN
  END
  OBJ→ DROP 4 ROLL
  SIZE # 2d + SWAP
  DROP + 2 →LIST
  IF SWAP 1 SAME
  THEN PX→C
  END
»
```

GL↓ puts a label into the graphics display and then returns the location two pixels below the lower left corner of the grob. This will help when you want to create blocks of left-justified text of varying sizes in your graphics display.

Store GL↓ into the **TOOLS** directory.

Name the second variation GL→ (<u>Checksum</u>: # 57747d <u>Bytes</u>: 172):

```
« →GROB SWAP DUP
  TYPE SWAP
  IFERR C→PX
  THEN
  END
  ROT DUP2 SIZE NEG
  # 10d + # 0d SWAP
  2 →LIST ROT ADDB
  PICT SWAP 4 ROLL
  GOR # 2d + # 0d 2
  →LIST ADDB
  IF SWAP 1 SAME
  THEN PX→C
  END
»
```

GL→ puts a label into the graphics display, and then returns a location two pixels to the right of the upper right corner of the grob. This will help when you want to create a line of various-sized text in the graphics display.

Store GL→ into the **TOOLS** directory.

Note that before you can use GL→ you must write the small utility it uses: ADDB adds two pixel locations as binary integers.

Here are the Stack diagram and program listing for ADDB:

Stack Inputs	Stack Outputs
2: *location* { # col_2 # row_2 }	
1: *location* { # col_1 # row_1 }	1: *new location* { # col_1+# col_2 # row_1+# row_2 }

And here is ADDB (Checksum: # 18393d Bytes: 51)—store it into your TOOLS directory:

```
« OBJ→ DROP ROT
  OBJ→ DROP ROT
  + ROT ROT +
  SWAP 2 →LIST
»
```

Now look at GL→ once again.

Note that it aligns the *bottom* edges of the text in the graphics display. Since **GOR**, **GXOR** and **REPL** align to the top left corner of the grob, GL→ must compute the location of the bottom edge as if your text were a 10-pixel high grob. That is, since your text will end up as a grob of height 6, 8 or 10 pixels, depending on the font you use, to align the text correctly, GL→ must account for those differences in height.

As an illustration, first use GLABEL alone to create a line of text in the graphics display, using all three fonts. To better see what happens, incorporate all the commands into a program and [EVAL] it from the Stack.

```
« { # 0d # 0d } PVIEW
  { # 0d # 0d } "TEXT1"
  1 GLABEL                          (for the first line)
  { # 22d # 0d } "TEXT2"
  2 GLABEL                          (for the second line)
  { # 54d # 0d } "TEXT3"
  3 GLABEL                          (for the third line)
  PICTURE
»
```

You'll see three different sizes of text, aligned at the top edges, like this:

TEXT1TEXT2TEXT3

But NOBODY WRITES like THIS. It's TOO hard to READ.

The largest text font on the 48 (not counting equations and unit objects) creates grobs that are 10 pixels high. The command sequence

```
« ... SIZE NEG
   # 10d + ...
»
```

adjusts the placement of text grobs of any size such that all the text ends up aligned at the bottom edges.

Now, erase the display, and then use GL→ to create a line of text like the one you created above, and see the difference. Again, to see it happen, put all the commands in a program and EVAL it from the Stack.

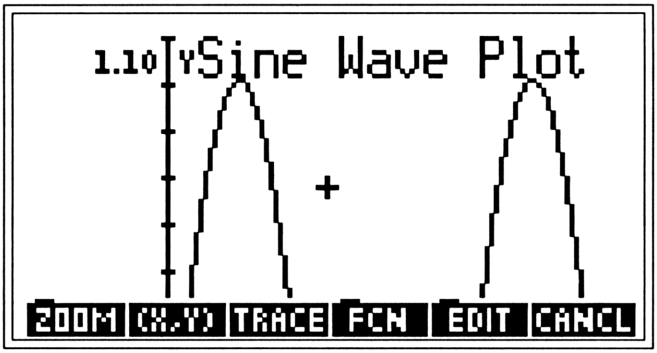
```
« { # 0d # 0d } PVIEW
  { # 0d # 0d } "TEXT1" 1 GL→   (for the first line)
  "TEXT2" 2 GL→                 (for the second line)
  "TEXT3" 3 GL→                 (for the third line)
  PICTURE
»
```

You'll get the following effect. Notice how the text is aligned on the bottom edge:

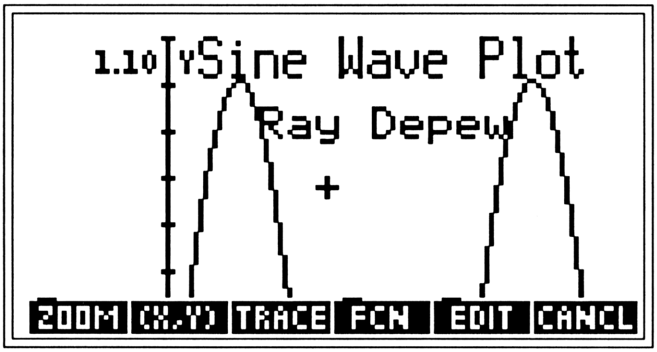
TEXT1TEXT2TEXT3

Now test GLABEL itself:

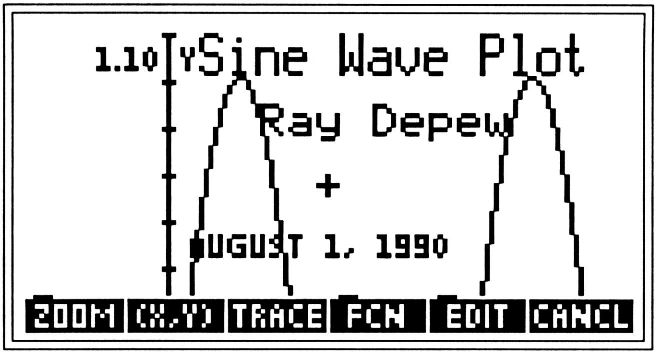
Move back to PICS. Put BIGSINE into the graphics display (type 'BIGSINE' RCLPIC). Then type [←][PLOT] PPAR [6][•][5][+/−][SPC] [6][•][5] XRNG and [1][•][3][+/−][SPC][1] YRNG to set the correct ranges. Now type (.5, 1) "Sine Wave Plot" [3][ENTER] GLABEL [ENTER], and [◄] to see your creation (use the arrow keys to scan around until you see this):



Now put { # 120d # 15d } onto Stack Level 3, your name in quotes onto Level 2 and the number 2 onto Level 1. Execute GLABEL, then [◄]. You should see something similar to this:



Now put (0.35, 0.5) onto Level 3, "August 1, 1990" onto Level 2, and the number 1 onto Level 1. Execute GLABEL, then press [◄].... You should see the date in 6-pixel text below your name, like this:*



Save this as BIGSINE (in PICS) again (remember how—page 130?).

Now try this: [⬅][PLOT] PPAR RESET creates a blank 131×64 grob. Then type [⬅][UP] { # 1d # 2d } "Welcome" [3] [VAR] GL↓. You should see { # 1d # 14d }. Now press [◄] to see Welcome in the graphics display. Next, type [CANCEL] "to the new" [2] GL↓ "HP 48GX" [3] GL↓ "Graphical Expandable calculator" [1] GLAB—and press [◄] to see your creation—a startup screen (more on this in Chapter 8)!

Best of all, GLABEL, GL→ and GL↓ can be used as subprograms in your own programs, and they can be easily rewritten as functions—or into functions. They don't halt program execution, and they're not interactive; they take their arguments from the Stack. They're also fairly tidy: they clean up the Stack after themselves. However, they do alter PICT irreversibly, and they don't include error checking—they assume you have given them correct inputs.

*Warning: If you execute GLABEL from your TOOLS directory, you may get different results from those pictured here. GOR and other graphics commands compute user units as specified by PPAR *in the current directory*. If your directories have PPAR's with differing user units, your results will be unpredictable. Therefore, it may be advisable to avoid user units in cases like this.

Here's one more handy routine, called CTR, that centers text around a given point in a grob. The text is drawn in font size 1:

CTR

Checksum: # 63567d

Bytes: 60

Stack Inputs	Stack Outputs
3: *target* GROB *(may even be PICT)*	
2: *location* { # *row no.* # *column no.* }	
1: "*text*"	1: *modified* GROB

```
« 1 →GROB DUP SIZE
  DROP 2 / ROT EVAL
  SWAP ROT - SWAP 2
  →LIST SWAP GOR
»
```

Store CTR into your **TOOLS** directory. Then test it and experiment with it as you wish.

Adding Graphics to Enhance Plots

Purge PICT and pull out BIGSINE again. Now suppose you want to label the origin. How do you do this?

Press ◀ EDIT to get to the PICTURE EDIT menu. Then use the arrow keys to position the cursor on the origin and press ×. Press any arrow key four times, then CIRCL. Now the origin is circled. Next, press the arrow keys to get the cursor at the 4 o'clock position on the circle. Press × again. Press ▶ *fifteen times*, then ▼ *eight times*, then TLINE.

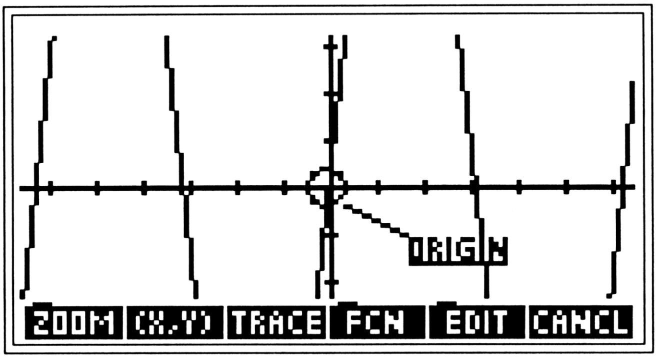
You've now drawn a line from the circle to some arbitrary point. The Toggle LINE function draws a line that turns black pixels white and white ones black. Now press ENTER to save the pixel position to the Stack. Then press CANCEL to return to the Stack for a moment.

Back in the Stack display, you see the digitized cursor position on Level 1. You want to label the origin as either ORIGIN or 0.0000 (your choice). With the cursor position on Level 3, put either "ORIGIN" or 0 onto Level 2, and 1 onto Level 1. Then execute GLABEL.* Finally, press ◀.

Move the cursor to just under the 0. Now press ×, then ▶ repeatedly to move the cursor to the end of the label. Press EDIT LINE to underline the label (you could also use DOT+ to do all this, but the canned shape routines are faster in a program and give more predictable results—use them as much as possible).

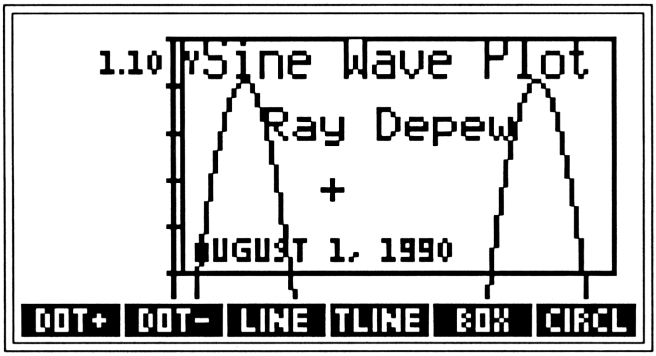
*Remember the hazards of differing PPAR's in different directories (see the footnote on page 163).

Your grob should now look like this.



Hmm…in a presentation-quality plot, the title block should probably be enclosed in some kind of box, no?

All right: Press the arrow keys to get the cursor above and to the left of the title, Sine Wave Plot. Press ⊠. Now move the cursor below the date and to the right of the title and your name. Press EDIT BOX, and you should see your title block as shown below.



Save this as BIGSINE (in PICS) again.

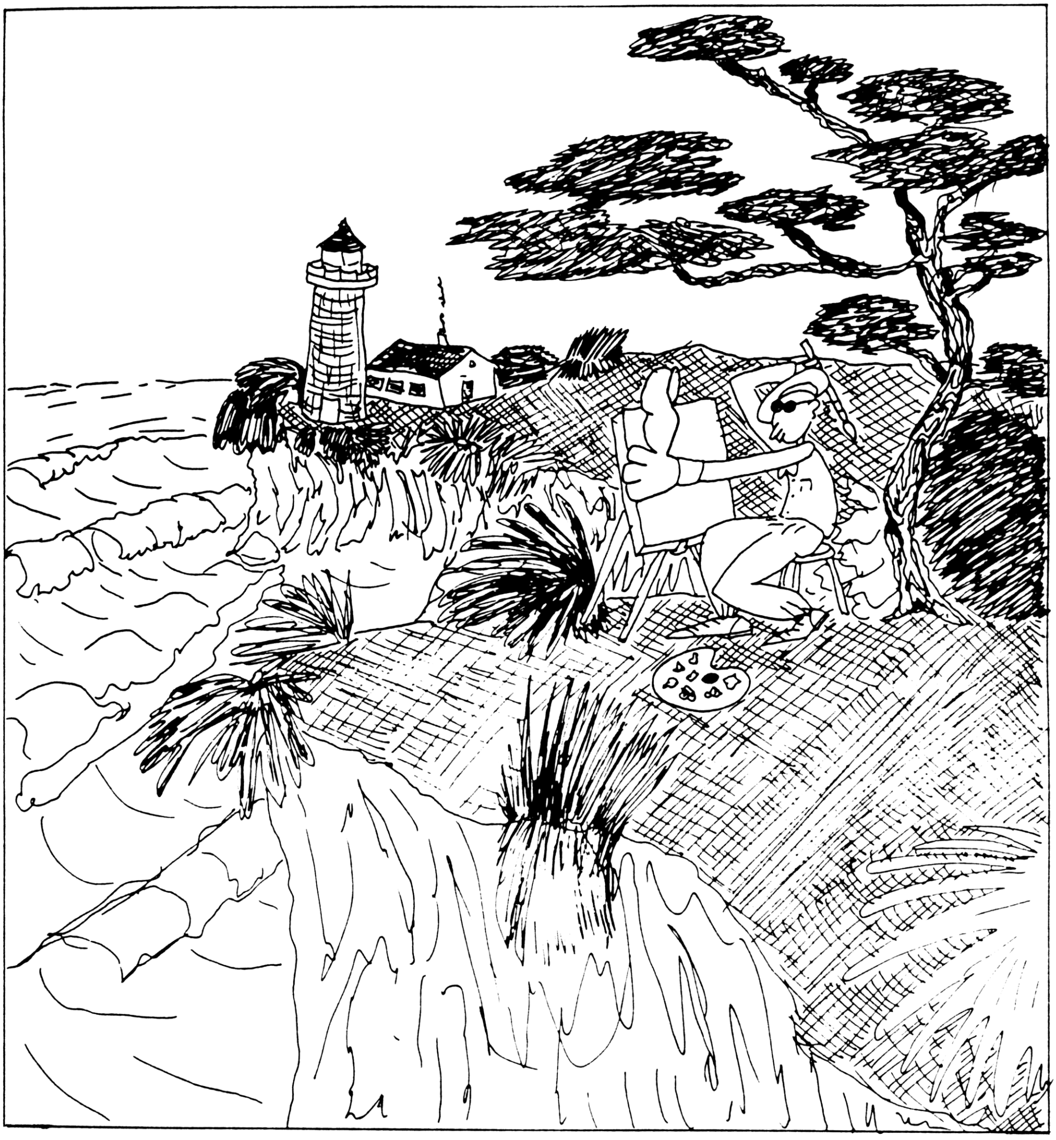
Review

In this chapter you learned how to manipulate the PLOT functions to display your plot the way *you* want to see it. You learned how to display the axis labels in different numeric formats.

You also created some programs to place text—of various sizes—anywhere on a plot. These programs, GLABEL, GL↓, GL→ and ADDB, are important additions to your toolbox.

You then used some of the *shape* commands (e.g. **BOX**, **CIRCL**, **LINE**, **TLINE**) to accent your plot. This is what the shape functions were originally intended for.

In fact, from now on, you can refer to the shape commands as "freehand drawing figures." Together with the freehand drawing commands **DOT+**/**PIXON** and **DOT-**/**PIXOF**, they form the core of the 48's tremendous graphics capability. And that's what the next chapter is devoted to—freehand drawing.



8: Freehand Drawing

How to Do It

What if you could turn on your 48, or start a program, and see an opening display like this?

With freehand drawing, you can create graphics to give your programs more pizzazz, simplify and clarify user interaction, or produce more intuitively understandable, pictorial outputs.

This chapter shows you how to do it.

The procedure for creating freehand graphics is this:

1. Use BLANK or ERASE to create a blank grob—your drawing board.

2. Use XRNG and YRNG, or PDIM—or even CENT and SCALE—to define your user units. Or, just work in pixels.

3. Use BOX to draw a single- or double-line around your grob.

4. Use LINE, CIRCL, etc. as much as possible, and PIXON / DOT+, PIXOF / DOT- only when the shapes won't do. In the Welcome picture at the start of the chapter, for example, all parts of the calculator except the keys were drawn with LINE and ARC. The keys were DOT+ work. The text was done with GL↓ and GLABEL.

5. Periodically during your creation (and of course, when it's done), save your drawing by typing 'TITLE' (or any other name), then STOPIC. Remember that your grob is only an object, which can be lost with a single keystroke.

Now use this program, named OFF1 (store it in your HOME directory: Checksum: # 38534d Bytes: 68):

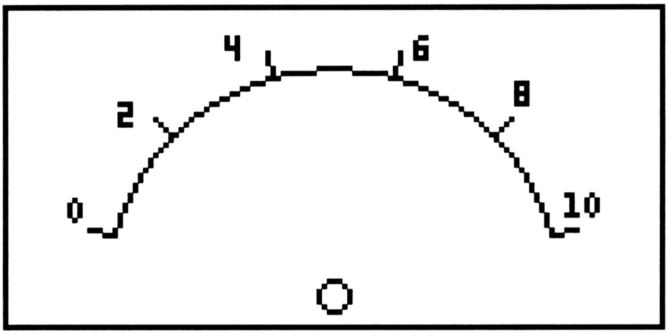
```
« { HOME TOOLS PICS TITLE }
  RCL PICT STO OFF
  { } PVIEW
»
```

You can add it to your CUSTOM menu, or assign the program to the ⮡ OFF key. Then, whenever you use OFF1 to turn the calculator off, you'll see your own TITLE grob.*

*With everything else the 48 has, it's a pity HP didn't include (or at least document) an AUTOSTART feature—a flag to activate a user program whenever the machine is turned on.

Drawing a Voltmeter Face

As another example, here's how to use freehand drawing figures and user units to create the face of an analog instrument meter, such as a voltmeter. You should end up with a grob that looks like this:



Press [←][PLOT] PPAR RESET to create a blank 131×64 grob. Then press [◄] to get to the graphics environment, and put a frame around the grob by drawing a box: [→][◄][→][▲] [×][→][►][→][▼][→][▼] EDIT BOX.

Now define your drawing area in user units. To make it easier, call the pivot point of the needle the origin, or (0, 0).

Give the arc on the numeric scale a radius of 0.9 unit from the origin. Then, allowing for tic marks and lettering, your maximum meter height will be 1.14 units, and your minimum meter height will be –0.12 units. For now, use a meter width of 2.6 units.

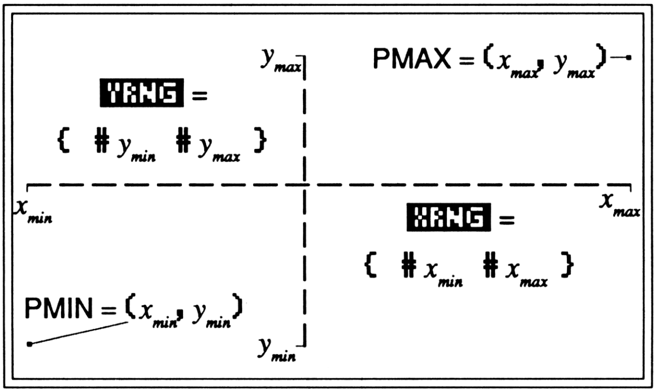
Note that you are using arbitrary units right now. When creating a strip chart or a bar graph, you'll probably want to use more meaningful units, like dollars/month or thousands of barrels per day, etc.

You can set your user units in two ways:

- Specify the lower-left and upper-right corners via PMIN and PMAX:
  `(-1.3,-.12) PMIN (1.3,1.14) PMAX` ENTER
- Or, specify the *x*- and *y*- ranges, using XRNG and YRNG:

  `-1.3 1.3 XRNG -.12 1.14 YRNG`

Either approach works fine. What you're doing is setting the plotting limits in terms of your own units. This diagram illustrates the relationship between PMIN / PMAX and XRNG / YRNG:



Now draw a small circle at the pivot point. You can do this from the Stack or from the PICTURE environment.

From the PICTURE environment, use **(X,Y)** or (+) to find the pixel closest to (0,0), then (×)(▶)(▶)(▶) ((−) gets the menu back), then **EDIT** **CIRCL**.

Or, to draw the pivot circle from the Stack, place these arguments on the Stack:

4:	(0,0)	center of the circle
3:	.03	radius of the circle
2:	0	start angle of the circle
1:	360 or 6.2832	end angle of circle (° or rad)

Then press (PRG) **PICT** **ARC** (the **CIRCL** command doesn't work on the Stack, and **ARC** doesn't work in the graphics environment.)

Next, draw the meter arc, by using (PRG) **PICT** **ARC**, with these Stack arguments:

4:	(0,0)	center of the arc
3:	0.9	radius of the arc
2:	15 or 0.2618	arc start angle (π/8 RADians)
1:	165 or 2.8798	arc end angle (7π/8 RADians)

Have a look at it so far: (◀); then prepare for the next step: **EDIT**.

Now draw the 6 tic marks in the graphics environment, by "eyeballing" their locations (you could calculate their locations exactly, but you'll get equally good resolution using the interactive commands): Move the cursor to the point on the arc where the tic mark originates; press (×). Then move the cursor to the other end of the tic mark, and press **LINE**. Repeat this for all six tic marks, evenly spaced.

Now use the GLABEL utility from Chapter 7 to label the tic marks. You want to label the tic marks 0, 2, 4, 6, 8 and 10.

For each label, follow this procedure:

1. Press [←][PICTURE] or [◄] to get the graphics environment. Move the cursor to the point above the tic mark where the label belongs, and press [ENTER].

2. Press [CANCEL] to exit graphics. Put the label on Level 1 as a string, i.e. "0", "2", "4", etc. Press [1], then execute GLABEL.*

At the end, your grob should look like the figure shown on page 171. Store this grob by entering 'METER' STOPIC.

Later, you will see how this versatile grob can be used in conjunction with the RS-232C interface to simulate a wide variety of measurement instruments.

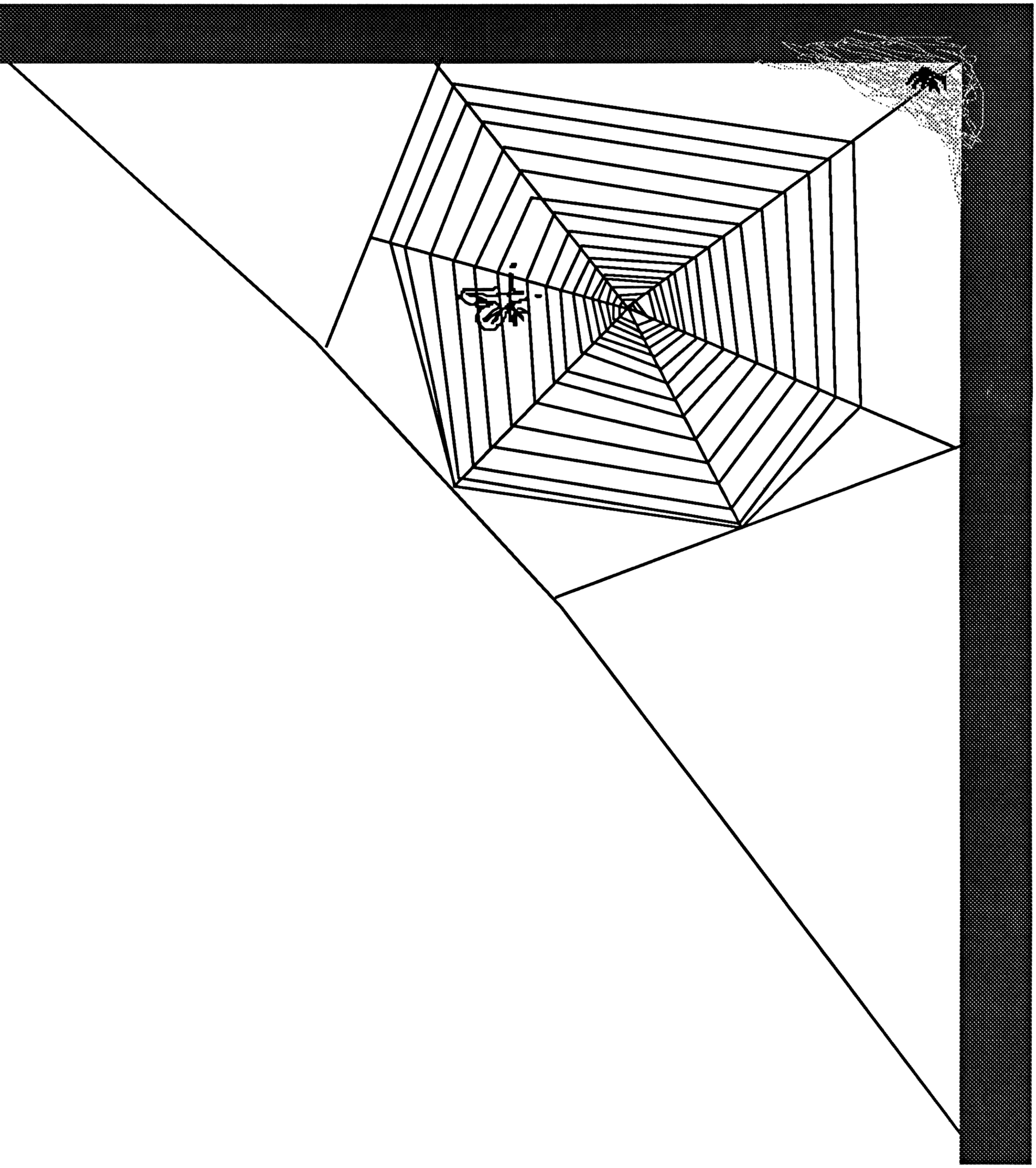
*Keep in mind that GOR, GXOR and REPL use the plotting limits in the current directory when they add data to PICT. This can give you unexpected results if you execute GLABEL from a directory with a different PPAR than what you intend.

Review

In this chapter you've seen the freehand drawing tools and a few examples for using them to create your own grobs, not necessarily tied to the normal PLOT routines. You should feel free to explore any other uses for grobs you can think of.

Keep in mind that a freehand grob can also be created programmatically, by using the commands from within « ». Or, you can use RCLPIC to recall the (previously stored) grob, or SEE if the grob is on the Stack. And any grob on the Stack can be turned into a program by placing it on the Command Line and enclosing it in « » brackets.

Now you're ready to see some real applications—examples of how you might put together everything you've learned here so far....



9: Programmable Graphics Applications

Introduction

In this chapter you're going to see several graphics applications. Some are meant to be used "as is," while others are given simply as examples of what you can do with graphics—to be modified or finished to fit your needs.

Each application begins with a description of the program(s). Then follows a list of subroutines and other variables, then a complete set of program listings, along with checksums, byte counts, Stack argument listings (where appropriate), notes and/or comments (where appropriate). Occasionally, too, you may see multiple versions of a program—just to show you how different your approaches can be.

*The checksum and byte counts given are for a Rev. K machine. To compare checksum and byte count, enter the program and store it under the indicated name. Then put that name onto Stack Level 1 and press ⬅ MEMORY BYTES.

Programmable Scanning Inside a Big Grob

These programs automate scanning inside a large grob—say, 300×200.

Descriptions

PSCAN: To display only certain, predetermined parts of the grob, you can use PSCAN from within a program to display those parts.

SCAN: To examine the grob yourself, use SCAN as a versatile alternative to the built-in PICTURE scrolling mode, moving by pixel, ten pixels, or across the entire grob.* SCAN treats the 48 display as a window onto the grob and redefines the numeric keypad as a window control pad; each numeric key, except [5] and [0], indicates a direction for movement:

- The [7] key, for example, moves the grob *one pixel* up and to the left (that is, it moves the window one pixel down and to the right).
- [⬅][7] moves the grob *ten pixels* up and to the left.
- [➡][7] moves the grob to the upper left corner of the window.
- Similarly, the other numeric keys move the grob in their directions: [3] to the lower right, [6] to the right, etc. ([5] does nothing).
- [0] exits SCAN in an orderly fashion. [CANCEL] is OK for emergencies, but it will leave the directory cluttered with extra objects.

*You may wish to disable the clock display (clear system flag –40) when using SCAN. A strange feature causes the clock display to appear on the top edge of the grob, where it scrolls off- and on-screen, as part of the grob. Interestingly, the clock even keeps "ticking" as it moves around.

Subroutines

PSCAN, SCAN and these subroutines should all be in the same directory.

SETUP: Creates temporary variables and initializes the 48 properly for SCAN and PSCAN.

NUDGE: "Nudges" the graphics display the distance and direction given in Level 1.

MV1: Moves 1 pixel in the direction indicated.

MV10: Moves ten pixels in the direction indicated.

MVall: Moves across the entire grob, in the direction indicated.

ADDB: Adds two lists of the form { # *rrr* # *ccc* } (see page 159).

Alternate Approach

These routines offer another solution, for the sake of comparison.

PSCN An alternate version of PSCAN.

SCN An alternate version of SCAN.

MV Combines the functions of NUDGE, MV1, MV10 and MVall above. Moves the distance indicated (1 pixel, 10 pixels or all the way) in the direction indicated.

SCAN

```
« SETUP                                   (Initialize PICT and variables)
  Cursor PVIEW
  DO 0 WAIT DUP FP
    → ky kfp
    « CASE
        kfp .1 SAME                       (Unshifted)
          THEN ky MV1
          END
        kfp .2 SAME                       (⬅-shifted)
          THEN ky MV10
          END
        kfp .3 SAME                       (➡-shifted)
          THEN ky MVall
          END
      END
      ky
    »
  UNTIL 92.1 SAME                         (Key zero—exit)
  END
  { Cursor PSIZE } PURGE                  (Remove global variables)
»
```

Checksum: # 47364d
Bytes: 257.5

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: SCAN uses PICT.

PSCAN

```
« OBJ→          (Break down list into locations; use list size as a counter)
  DO DUP 1 + ROLL PVIEW          ("Roll up" to the next location,
     .5 WAIT 1 -                   use it and discard it)
  UNTIL DUP 0 SAME
  END
»
```

Checksum: # 29420d
Bytes: 67.5

	Stack Arguments	Stack Results
1:	{ loc_1 loc_2 loc_3 … loc_n }	(none)

Notes: PSCAN uses PICT.

The Stack argument may be given either in user units (complex numbers) or pixel locations { # *rownum* # *colnum* }. Each set of coordinates in the list represents a location on the grob that will successively be passed to PVIEW in the program.

SETUP

```
« PICT SIZE DUP2 2 →LIST
  'PSIZE' STO                          (Save PICT size)
  IF # 64d ≤ SWAP         (If PICT is no bigger than the default...
    # 131d ≤ AND
  THEN              ...offer to view without scrolling or aborting)
    IF "GROB is smaller than■   (■ is NEWLINE; press [→][↵])
      display! Look anyway?"
      { { "YES" « 1 CONT » }
      "" "" "" ""
      { "NO" « 0 CONT » } }
      TMENU PROMPT 0 MENU
    THEN { } PVIEW           (Press [CANCEL] to exit from this)
    END
    CONT                     (CONT breaks out of SCAN here)
  ELSE { # 0d # 0d }
   'Cursor' STO                    (Initialize the cursor)
  END
»
```

Checksum: # 22047d
Bytes: 311.5

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: SETUP initializes SCAN and PSCAN.

NUDGE

```
« Cursor ADDB                         (Add increment to Cursor)
  → cursor
  « IFERR cursor PVIEW
    THEN 300 .2 BEEP
     DROP
    ELSE cursor
      'Cursor' STO                    (Update Cursor for next time)
    END
  »
»
```

Checksum: # 60163d
Bytes: 143

Stack Arguments	Stack Results
1: { # *column-increment* # *row-increment* }	(none)

Notes: NUDGE moves the grob according to the increment given in Level 1.

The increment must be given in binary integers.

NUDGE is called by MV1 and MV10.

MV1

```
« → ky
  « CASE
      ky 62.1 SAME                          (Key [7], up and left)
        THEN { # 1d # 1d } NUDGE
        END
      ky 63.1 SAME                          (Key [8], straight up)
        THEN { # 0d # 1d } NUDGE
        END
      ky 64.1 SAME                          (Key [9], up and right)
        THEN { #18446744073709551615d # 1d } NUDGE
        END
      ky 72.1 SAME                          (Key [4], left)
        THEN { # 1d # 0d } NUDGE
        END
      ky 73.1 SAME                          (Key [5], nowhere)
        THEN { # 0d # 0d } NUDGE
        END
      ky 74.1 SAME                          (Key [6], right)
        THEN { #18446744073709551615d # 0d } NUDGE
        END
      ky 82.1 SAME                          (Key [1], down and left)
        THEN { # 1d #18446744073709551615d } NUDGE
        END
      ky 83.1 SAME                          (Key [2], straight down)
        THEN { # 0d #18446744073709551615d } NUDGE
        END
      ky 84.1 SAME                          (Key [3], down and right)
        THEN
          { # 18446744073709551615d
             # 18446744073709551615d }
          NUDGE
        END
    END
  »
»
```

Checksum: # 48305d
Bytes: 652.5

	Stack Arguments	Stack Results
1:	*keycode*	(none)

Notes: MV1 moves the grob 1 pixel at a time.

You cannot create the large binary integer in MV1 via # 1d [+/-] while editing the program. You'll get « ... # 1d NEG ... », which won't work in the program. And # 1 [+/-] [α] [←] [D] causes an Invalid Syntax error at « ... # -1d ... ».

To get the large integer, you must either key it in digit-by-digit each time (not too thrilling a prospect) or put it onto the Stack before keying in the program, then pull it into the program during editing via [←] [EDIT] ↑STK. This seems far easier, since the number is just the negative of a smaller, more familiar integer: # 1 [ENTER] [+/-] Result: # 18446744073709551615d

Then, while creating your program, put the insert cursor (✦) in the space to the right of where you want to place the integer. Press [←] [EDIT] to get the EDIT menu and ↑STK to get to the selection environment. Use [▲] and [▼] to select the integer, and then ECHO [ENTER]. You'll return to the program editing, with the integer in the right place.*

*Or, alternatively, you can add the number to your CST menu and enter it from there: If you already have a CST menu, press [→] [CST] # 1 [ENTER] [+/-] [+] [←] [CST]; if you don't already have a CST menu, press # 1 [ENTER] [+/-] 'CST' [STO].

MV10

```
« → ky
  « CASE
      ky 62.2 SAME                          (Key ⬑7, up and left)
        THEN { # 10d # 10d } NUDGE
        END
      ky 63.2 SAME                          (Key ⬑8, straight up)
        THEN { # 0d # 10d } NUDGE
        END
      ky 64.2 SAME                          (Key ⬑9, up and right)
        THEN { #18446744073709551606d #10d } NUDGE
        END
      ky 72.2 SAME                          (Key ⬑4, left)
        THEN { # 10d # 0d } NUDGE
        END
      ky 73.2 SAME                          (Key ⬑5, nowhere)
        THEN { # 0d # 0d } NUDGE
        END
      ky 74.2 SAME                          (Key ⬑6, right)
        THEN { #18446744073709551606d # 0d } NUDGE
        END
      ky 82.2 SAME                          (Key ⬑1, down and left)
        THEN { #10d #18446744073709551606d } NUDGE
        END
      ky 83.2 SAME                          (Key ⬑2, straight down)
        THEN { # 0d #18446744073709551606d } NUDGE
        END
      ky 84.2 SAME                          (Key ⬑3, down and right)
        THEN
          { # 18446744073709551606d
            # 18446744073709551606d }
          NUDGE
        END
    END
  »
»
```

Checksum: # 38008d
Bytes: 653.5

	Stack Arguments	Stack Results
1:	*keycode*	(none)

Notes: MV10 moves the grob 10 pixels at a time.

As with MV1, to get the large integer here, you must either key it in digit-by-digit each time or put it onto the Stack before keying in the program, then pull it into the program during editing via (⬅)(EDIT) ↑STK. Again, this seems far easier, since the number is just the negative of a smaller, more familiar integer:

10d (ENTER)(+/-)   Result: # 18446744073709551606d

Then, while creating your program, put the insert cursor (♦) in the space to the right of where you want to place the integer. Press (⬅)(EDIT) to get the EDIT menu and ↑STK to get to the selection environment. Use (▲) and (▼) to select the integer, and then ECHO (ENTER). You'll return to the program editing, with the integer in the right place.*

*Or, alternatively, you can add the number to your CST menu and enter it from there: If you already have a CST menu, press (➡)(CST) # 10d (ENTER)(+/-)(+)(⬅)(CST); if you don't already have a CST menu, press # 10d (ENTER)(+/-) 'CST' (STO).

MVall

```
« → ky
  « CASE
      ky 62.3 SAME                    (Key ⮣7, up and left)
        THEN PSIZE
          { # 18446744073709551485d
            # 18446744073709551552d }
          ADDB
        END
      ky 63.3 SAME                    (Key ⮣8, straight up)
        THEN Cursor OBJ→ DROP2
          PSIZE OBJ→ ROT DROP2
          # 64d - 2 →LIST
        END
      ky 64.3 SAME                    (Key ⮣9, up and right)
        THEN # 0d PSIZE OBJ→ ROT
          DROP2 # 64d - 2 →LIST
        END
      ky 72.3 SAME                    (Key ⮣4, left)
        THEN PSIZE OBJ→ DROP2
          # 131d - Cursor OBJ→
          ROT DROP2 2 →LIST
        END
      ky 73.3 SAME                    (Key ⮣5, nowhere)
        THEN Cursor
         END
      ky 74.3 SAME                    (Key ⮣6, right)
        THEN # 0d Cursor OBJ→
          ROT DROP2 2 →LIST
        END
      ky 82.3 SAME                    (Key ⮣1, down and left)
        THEN PSIZE OBJ→ DROP2
          # 131d - # 0d 2 →LIST
        END
      ky 83.3 SAME                    (Key ⮣2, straight down)
        THEN Cursor OBJ→ DROP2
          # 0d 2 →LIST
        END
```

```
      ky 84.3 SAME                         (Key ⇁ 3, down and right)
        THEN { # 0d # 0d }
        END
      Cursor                               (If no other case is true)
    END                                                        (CASE)
  »
  DUP 'Cursor' STO PVIEW
»
```

Checksum: # 44757d
Bytes: 674

	Stack Arguments	Stack Results
1:	*keycode*	(none)

Notes: MVall moves the grob all the way to one side or corner. As with MV1 and MV10, to get the large integers here, you must either key them in digit-by-digit each time or put them onto the Stack before keying in the program, then pull them into the program during editing via ↰ EDIT ↑STK. That seems easier: they are just the negatives of smaller, more familiar integers:

131d ENTER +/− Result: # 18446744073709551485d
64d ENTER +/− Result: # 18446744073709551552d

Then, while creating your program, put the ◆ to the right of the integer's desired location. Then press ↰ EDIT ↑STK, and use ▲ and ▼ to select the integer, then ECHO ENTER.*

*Or you can add the number to your CST menu and enter it from there: If you already have a CST menu, press VAR ⇁ CST # 131d ENTER +/− # 64d ENTER +/− 2 PRG LIST →LIST + VAR ↰ CST; if you don't have a CST menu, press # 131d ENTER +/− # 64d ENTER +/− 2 PRG LIST →LIST 'CST' STO.

Listings for Alternate Approach

Often you may first solve a programming problem in the way clearest to you, only to discover later that you could have accomplished the same task more simply, or with less code, less memory usage, better execution speed, etc. In fact, the very act of creating and documenting the first version often reveals the possibilities for improvement.

This application is a good example of that process. After studying the previous version, you'll see how this version "streamlines" it somewhat (though the effective speed is about the same either way):

PSCN

```
« OBJ→ 1
  FOR j j ROLL
    PVIEW .5 WAIT -1
  STEP
»
```

Checksum: # 12373d
Bytes: 58

	Stack Arguments	Stack Results
1:	{ loc_1 loc_2 loc_3 ... loc_n }	(none)

Notes: PSCN is very similar to PSCAN (page 181).

SCN

```
« { # 0d # 0d } PVIEW                    (Display PICT)
  RCLF 'Flags' STO 64 STWS               (Save current flag settings
                                         before messing with them)
  PICT SIZE 64 - B→R 'PY' STO            (Re-size PICT if
  131 - B→R 'PX' STO                     it's too small)
  0 'CX' STO 0 'CY' STO                  (Initialize variables)
  WHILE 0 WAIT DUP IP 92 ≠               (Get keycode)
  REPEAT DUP IP                          (Dissect it into two
     SWAP FP 10 *                        arguments for MV)
     MV                                  (Do the move and display the result)
  END
  DROP Flags STOF                        (Restore previous flag settings)
  {Flags PX PY CX CY } PURGE             (Clean up)
»
```

Checksum: # 2224d
Bytes: 291

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: SCN behaves like SCAN (page 180).

MV

```
« { 1 10 1E12 } SWAP GET
  → f
  «
    { { 1 1 } { 0 1 } { -1 1 }
      { 1 0 } { 0 0 } { -1 0 }
      { 1 -1 } { 0 -1 } { -1 -1 } }
    { 62 63 64 72 73 74 82 83 84 }
    ROT POS GET EVAL
    f * CY + PY MIN 0 MAX 'CY' STO
    f * CX + PX MIN 0 MAX 'CX' STO
  »
  CX R→B CY R→B 2 →LIST PVIEW
»
```

Checksum: # 7919d
Bytes: 372

	Stack Arguments	Stack Results
2:	*keycode (integer portion)*	(none)
1:	*keycode (tenths digit)*	(none)

Notes: MV moves the grob as indicated by the two keycode arguments it receives from SCN. Compare this with NUDGE, MV1, MV10, and MVall on pages 183-189. Note, too, that since only SCN calls MV—and only once—you could certainly incorporate MV into SCN with no loss of efficiency.

Generating a Stripchart

Here are two programs which allow you to display data in a stripchart format. Astripchart recorder is a mechanism that drags a strip of paper at a constant speed under a pen being activated by a signal from an instrument or sensor. Usually the signal is a 0-5-volt or 4-20-milliamp signal.

Now, with the advent of low-power signal conditioning modules, you can read an analog signal input, then convert it to a real number and transmit it via datacomm lines to a digital computer.*

The 48 has a unique position as a portable instrument controller or data logger: On the ⬅ I/O NXT SERIA menu are some low-level commands with which you can configure your 48 to communicate with any serial device in the world. These stripchart programs and the VM program which follows, are intended to demonstrate this capability.

*Signal conditioning modules that do this are available from Omega Engineering, DGH, Onset Computer Corp., Keithley-Metrabyte, Inc., and many other sources. Most modern test and measurement instruments are now sold with a built-in or optional serial interface.

Descriptions

STRIP: This program displays an animated (rolling) stripchart on the display. It may be halted by pressing any key.

PSTRIP: This program prints a stripchart on the infrared printer. The output is very elementary, but the program is easily modified to add more detail to the output. It may be halted by pressing any key.

STRIP and PSTRIP do not take their input from the Stack. Instead, they look for a list called DApar ("Data Acquisition parameters"), of the form { *minimum-value maximum-value title time-interval* }, where

minimum-value and *maximum-value* (real numbers) are the chart limits.

title (a character string) is the chart title.

time-interval (a real number) is the minimum interval between measurements (not used in STRIP). This is given in HMS format—as *hh.mmss*, where *hh* is the number of hours, *mm* the minutes, and *ss* the seconds. The routine Nxtime uses this time interval to compute the time until the next measurement. The minimum useful time interval varies from machine to machine, and depends on how long it takes to execute READY and print the results.

If the programs do not find any list object named DApar, then they use this default DApar:

```
{ 0 1 "" 0 }
```

Note that in a real setting, where the 48 would be connected to a voltmeter or other signal conditioning module, the routine READY would query that instrument or module, and the commands within READY would typically look like this:

```
«  ..."#1RD" XMIT DROP
   REPEAT
   UNTIL BUFLEN DROP
   END
   SRECV DROP...
»
```

Here, however, for the purposes of these demonstration programs, the input of a real meter is simulated with a random number generator. Therefore READY becomes simply

```
« RAND
»
```

Subroutines

STRIP and PSTRIP use several subroutines. The main programs and the subroutines should all be stored in the same 48 directory.

READY:	Program to collect the data from the serial- or infrared-equipped sensor or instrument.
MkAxis:	Draws a *y*-axis for PSTRIP paper output.
Now?:	Performs an elapsed-time (true-false) test.
Pr8:	Prints eight pixel rows to the infrared printer.

Variables

DApar:	The data-acquisition parameter list
δt (delta-t):	The time interval, in ticks, between measurements.
Nxtime:	PSTRIP uses a DO ... UNTIL loop to time readings, rather than alarms; the current time (in ticks) is incremented by δt to generate the value Nxtime. But in a remote application, PSTRIP could be modified to set alarms and turn itself off, rather than use such a DO ... UNTIL loop.

STRIP

```
« RCLF 'Flags' STO 64 STWS                    (Save current status)
  IF DApar DUP TYPE 5 ≠                       (Find or create DApar)
  THEN { 0 1 "" 0 } DUP ROT STO
  END
  DUP 2 GET SWAP 1 GET DUP2 -                 (Extract parameter values
  → hi lo diff                                        from DApar)
  « PICT PURGE                                (Draw the stripchart recorder)
    { # 0d # 0d } { # 130d # 63d } BOX
    { # 20d # 11d } { # 120d # 54d } BOX
    20 120
    FOR z z R→B # 55d
      2 →LIST
      PIXON 20
    STEP
    { # 0d # 0d } PVIEW STO                   (Show the stripchart recorder)
    PICT { # 20d # 57d }
    lo 1 →GROB GOR                            (Label the reticle)
    PICT hi 1 →GROB DUP
    SIZE DROP NEG # 121d + # 57d
    2 →LIST SWAP GOR
    PICT { # 2d # 2d }
    IF DApar 3 GET DUP SIZE NOT               (Draw the title)
    THEN DROP "Press any key to quit."        (Default title)
    END
    1 →GROB GOR
    DO                                        (The data acquisition loop)
      READV lo MAX hi MIN lo - diff /
      PICT { # 21d # 12d } { # 119d # 52d } SUB
      PICT { # 21d # 13d } ROT REPL
      PICT { # 21d # 12d } GROB 99
      1 0000000000000000000000000 REPL
      100 * 20 + R→B # 12d 2 →LIST PIXON
    UNTIL KEY
    END
```

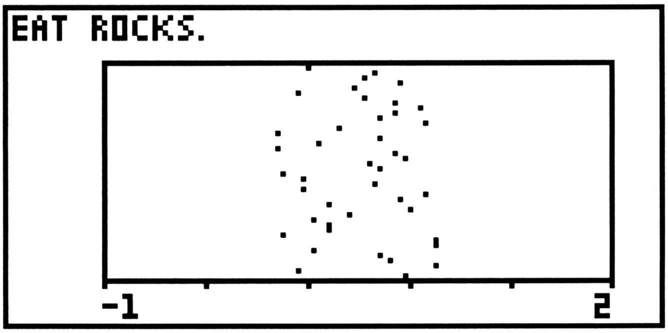
```
      DROP
    »
    Flags STOF                          (Restore status)
    { Flags } PURGE                     (Delete global variables)
»
```

Checksum: # 20905d
Bytes: 899

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: STRIP generates an on-screen stripchart.

DApar may be modified before running the program. On machines with black LCD pixels, the default DApar may cause the data to scroll by too quickly to be seen. If so, then adjust the time interval parameter to slow down the data display. A setting of 0.00001, or 1/10th of a second, should work fine. Machines with blue LCD pixels (version K) won't have this problem.



PSTRIP

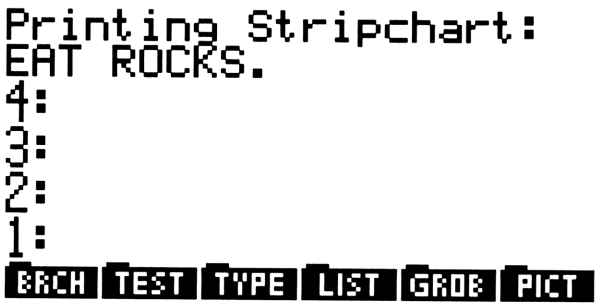
```
« "Printing Stripchart:" 1 DISP
  IF DApar DUP TYPE 5 ≠                        (Find or create DApar)
  THEN { 0 1 "" 0 } DUP ROT STO
  END
  OBJ→ DROP HMS→ 29491200 * 'δt' STO           (Calculate δt)
  DUP
  IF SIZE                                      (Print and display the chart title...
  THEN PR1 2 DISP
  ELSE DROP                                    ...unless there isn't one)
  END
  DUP2 XRNG -56 7 YRNG                         (Set up PICT, draw & print y-axis)
  PICT PURGE
  PICT { # 0d # 0d } MkAxis GOR
  → lo hi
  « TICKS δt + 'Nxtime' STO                    (Increment the timer)
    DO 7 0
      FOR rowcounter                           (Printer can print 8 rows at once)
        DO
        UNTIL Now?
        END                                    (An idle loop: Now? is a T/F test)
        READV                                  (Read the "voltage")
        lo MAX hi MIN                          ("Peg the meter" limits)
        rowcounter R→C PIXON
        IF rowcounter NOT
        THEN Pr8
        END
        -1
      STEP
    UNTIL KEY
    END                                        (End of DO loop)
    "Stripchart completed" 1 DISP Pr8 DROP
  »
  { δt Nxtime } PURGE                          (Delete global variables)
»
```

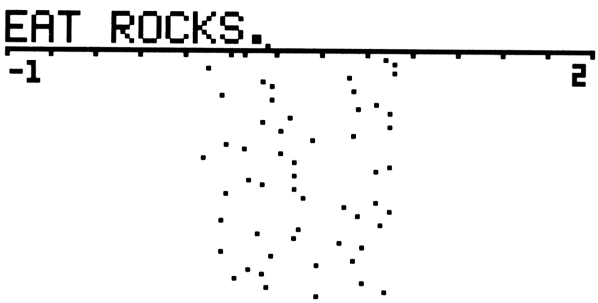
Checksum: # 45726d
Bytes: 472.5

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: PSTRIP generates a stripchart on the HP 82240B infrared printer.

DApar may be modified before running the program.





READY

```
« RAND
»
```

Checksum: # 51900d
Bytes: 22

	Stack Arguments	Stack Results
1:	(none)	a real number

Notes: READY reads a voltmeter or other serial output device. In this demonstration case, it's a simple random number generator; in real applications, this routine would contain the appropriate commands to read the device.

Now?

```
« TICKS
  IF Nxtime ≥ DUP
  THEN δt 'Nxtime' STO+
  END
»
```

Checksum: # 63658d
Bytes: 70.5

Stack Arguments		Stack Results	
1:	(none)	1	(if it's time to take another measurement, or...)
		0	(...if it's not)

Notes: Now? updates (increments) the value in Nxtime and returns a 1 or 0 to the Stack.

MkAxis

```
« PPAR OBJ→ 6 DROPN                    (Get PMIN, PMAX)
  SWAP RE SWAP IM R→C AXES             (Calculate axis intersection)
  ERASE DRAX LABEL                     (Draw axis)
  PICT { # 0d # 2d }
  GROB 1 6 000000000000 REPL
  PICT { # 0d # 0d }
  { # 130d # 7d } SUB                  (Cut out axis for printing)
»
```

Checksum: # 32330d
Bytes: 177

	Stack Arguments	Stack Results
1:	(none)	grob for the *y*-axis

Notes: MkAxis creates the grob for the *y*-axis of the stripchart.

Pr8

```
« PICT
  { # 0d # 0d } { # 130d # 7d }
  SUB PR1 DROP ERASE
»
```

Checksum: # 55076d
Bytes: 92

	Stack Arguments	Stack Results
1:	(none)	(none)

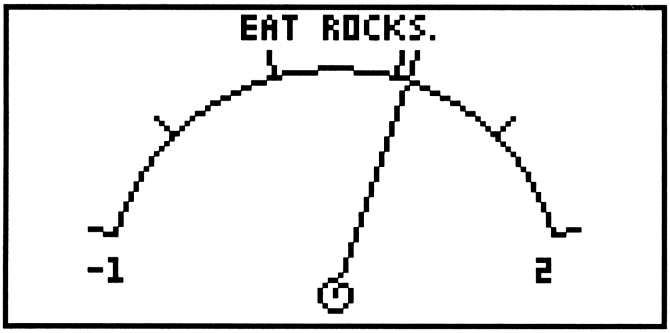
Notes: Pr8 sends the top 8 pixel rows of PICT to the printer and then erases PICT.

An Analog Voltmeter

This is a versatile application that lends itself to infinite modification. Using the same DApar and READY as used for the stripcharts, the 48 display becomes an analog meter with a swinging needle. With an analog display, your brain can immediately analyze data without taking the time to translate from digital representation to a quantitative "picture." This is probably why digital car dashboards have disappeared, and the reason for the return of the "old-fashioned" dial—now called "analog" (UGH!)—wristwatch.

Description

The VM application can be used in lieu of the stripchart, when you want instantaneous display of a signal in analog form. VM will draw a voltmeter face in the graphics display, label the display according to the parameters it finds in the list named DApar, and then swing a needle back and forth, using a routine called POINT. The needle's position will reflect the values it receives from the "voltage-reading" routine, READY.



Simply press any key to halt VM. The program and display are simple enough that you can add other features, such as Out of Range indicators, auto-ranging, secondary digital readout, etc.

VM takes no input from the Stack. Instead, it looks for a list called DApar ("Data Acquisition parameters"), of the form { *minimum-value maximum-value title time-interval* }, where

minimum-value and *maximum-value* (real numbers) are the meter limits.

title (a character string) is the meter title.

time-interval (a real number) is the minimum interval between timed measurements (not used in VM).

If the program does not find any list object named DApar, then it uses this default DApar: `{ 0 1 "" 0 }`

Note that in a real setting, where the 48 would be connected to a voltmeter or other signal-conditioning module, the routine READY would query that instrument or module, and the commands within READY would typically look like this:

```
« ..."#1RD" XMIT DROP
  REPEAT
  UNTIL BUFLEN DROP
  END
  SRECV DROP ...
»
```

Here, however, for the purposes of these demonstrations programs, the input of a real meter is simulated with a random number generator. Therefore READY becomes simply

```
« RAND
»
```

Subroutines

VM uses the following subroutines, which should be stored in the same directory as VM:

MAKEFACE:	Draws the meter face, except for the needle, title and scale labels.
READY:	Program to collect the data from the serial device, IR device, or whatever else.
POINT:	Erases and redraws the needle, using TLINE.
CTR:	Centers text around a point in a grob.

Variables

DApar:	The data-acquisition parameter list

VM

```
« RCLF → f                                   (Save current status)
  « -16 SF -19 SF DEG 64 STWS                (Set flags as needed)
    (0,.5) CENTR .2 DUP SCALE                (Set graphics parameters)
    IF DApar TYPE 5 ≠                        (Find or create DApar)
    THEN { 0 1 "" 0 } 'DApar' STO
    END
    MAKEFACE PICT                            (Draw the meter face)
    { # 21d # 50d } DApar 1 GET CTR PICT
    { # 104d # 50d } DApar 2 GET CTR PICT
    { # 66d # 2d } DApar 3 GET CTR
    DApar 1 GET DUP POINT                    (Put the needle at far left)
    DO READV DUP ROT POINT POINT             (Move the needle)
    UNTIL KEY
    END
    DROP2 f STOF                             (Restore previous status)
  »
»
```

Checksum: # 4616d
Bytes: 417.5

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: VM generates a working analog meter in the 48 display. DApar may be modified before running the program.

MAKEFACE

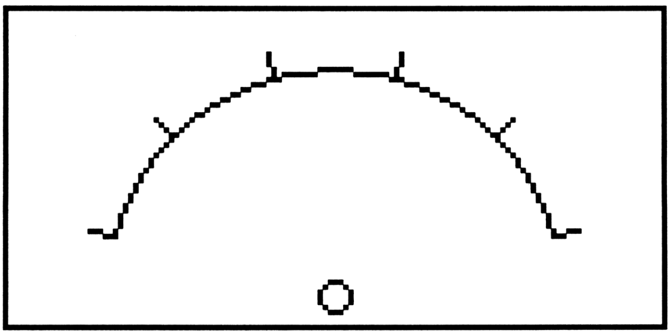
```
« PICT PURGE
  { # 0d # 0d } PVIEW
  { # 0d # 0d } { # 130d # 63d }
  BOX                                        (Meter bezel)
  { # 65d # 57d } DUP
  # 3d 0 360 ARC                             (Needle pivot)
  # 45d 15 165 ARC                                 (Scale)
  165 15
  FOR n 1 n →V2 .9 n →V2 LINE -30
  STEP
»
```

Checksum: # 55665d

Bytes: 294.5

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: MAKEFACE draws the meter face:



POINT

```
« → v
  « 1
    '15+150*MIN(1,MAX(0,(DApar(2)-
      v)/(DApar(2)-DApar(1))))'
    →NUM →V2 (0,0) TLINE
  »
»
```

Checksum: # 6495d
Bytes: 176

	Stack Arguments	Stack Results
1:	*signal level* (a real number)	(none)

Notes: POINT erases and redraws the meter's needle.

A properly formatted DApar should be in the same directory.

CTR

(see page 164)

READY

(see page 195)

Plots with Two Independent Variables

The 48's two-dimensional plotter allows you to plot multiple equations simultaneously, but it allows for only one independent variable.

For example, with the equation 'Z=X+Y', you must store several versions of the equation with different values for either X or Y, then create an EQ list containing all the versions of the equation.

The 48's 3-D tools (particularly YSLICE and WIREFRAME) eliminate some of this inconvenience, but they require that you use evenly-spaced incremental values for the second independent variable (by specifying YRNG and NUMY).

For example, if you're interested in the shape of the function at Y-values of 1, 2, 10 and 36, you can't do it with only YRNG and NUMY.

Description

MULTIPLOT allows you to plot functions such as $z = f(x,y)$ without all the headache. Before executing MULTIPLOT, you do the following:

1. Create the equation just as you would for the PLOT application; any equation or program that works with PLOT will also work with MULTIPLOT. However, you must store it under a global variable name *other than EQ*.

2. Press [←][PLOT] PPAR. Set up the ranges, independent variable and dependent variable appropriately (see Chapter 5 for a reminder on how to do this—or you can create an entirely new PPAR on the Command Line and store it directly.)

3. Onto Stack Level 1 put a list of this form:

   { *eqname* *yname* { y_1 y_2 ... y_n } }   where

   eqname is the name of the equation (or the equation itself);

   yname is the name of the second independent variable;

   y_1, y_2, ...y_n... are the values of that variable to be used in the plot.

MULTIPLOT is remarkably small and simple, since it uses built-in 48 routines to do most of the work—and it works at about the same speed as the Plotter application. Some examples follow the program listing.

You may wish to try your multivariable equation with the built-in Plotter first, to find a good range for the second independent variable. Also, note that you can store and recall the equation lists as desired, effectively saving many different MULTIPLOT applications.

Variables

VALS: a list of values for the second independent variable

SIV: the second independent variable's current value

Listing

MULTIPLOT

```
« 1 GETI STEQ                              (Save equation name in EQ)
  GETI 'SIV' STO GET 'VALS' STO            (Save SIV and VALS)
  ERASE { # 0d # 0d } PVIEW
  DRAX LABEL                               (Draw and label axes)
  1 VALS SIZE
  FOR n                                    (For each value ...
    VALS n GET 'SIV' RCL STO               ...store it in 2nd ind. var....
    DRAW                                   ...and plot the function)
  NEXT
  { VALS SIV } PURGE                       (Clean up)
  7 FREEZE                                 (Freeze the display)
»
```

Checksum: # 18534d
Bytes: 188

	Stack Arguments	Stack Results
1:	{ *eqname* *yname* { y_1 y_2 ... y_n } }	(none)

Notes: MULTIPLOT generates a plot of the function f(x,y). The function is plotted in PICT (which is displayed during the plot), and the program stops with PICT displayed.

Be sure that the PPAR settings are correct.

Example: A Simple Plane

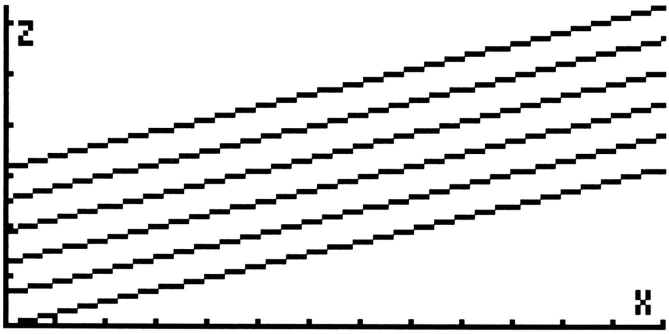
Equation: PLANE: 'Z=X+Y'

Plot parameters:	XRNG:	0 10	YRNG:	0 20
	INDEP:	X	RES:	0
	AXES:	(0,0)		
	PTYPE:	FUNCTION		
	DEPND:	Z		

PPAR: { (0,0) (10,20) X 0 (0,0) FUNCTION Z }

Level-1 Stack argument: { PLANE Y { 0 2 4 6 8 10 } }

Result: A series of lines representing contours on the plane:



Note that in this example and the next, the dependent variable in PPAR does not appear in the algebraic. This simply allows LABEL to label the *y*-axis correctly and does not affect the computation at all. However, in this first example, the dependent variable in PPAR must be the same as the dependent variable in the equation; an equals sign makes a lot of difference.

Example: A Fourier Series of a Full-Wave Rectified Sine Wave

Equation: `FOURIER: '2*A/π-4*A/π*Σ(n=1,Nmax,`
`COS(n*ω*t)/(4*n^2-1))'`
(Checksum: `# 13515d` Bytes: `120.5`)

Variables: `A: 1`
`ω: 1`

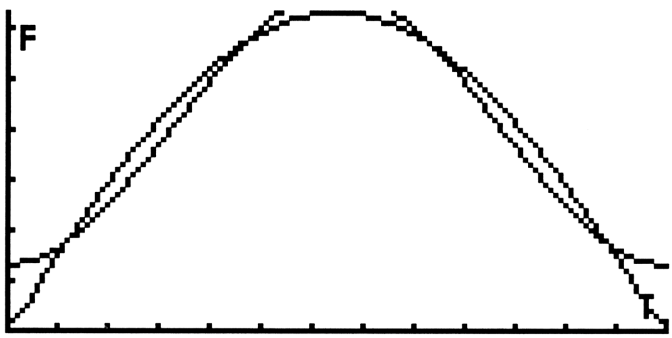
Plot parameters:

XRNG:	0 6.3	YRNG:	0 1
INDEP:	`t`	RES:	0
AXES:	`(0,0)`		
PTYPE:	`FUNCTION`		
DEPND:	`f`		

(PPAR): `{ (0,0) (6.3,1) t 0 (0,0) FUNCTION f }`

Level-1 Stack argument: `{ FOURIER Nmax { 1 10 } }`

Result: A plot of the first several approximations to the Fourier Series representation of a full-wave rectified sine wave:



Compare this with a similar plot of the function `'ABS(SIN(ω*t/2))'`. To see more than one lobe, increase x_{max} from 6.3 to 13 or more.

Example: A Field-Effect Transistor

Equation: IDID0: 'IFTE(VD≤VG-Vp,(VD-2/3*(Vbi-Vp)*
(((VD+Vbi-VG)/(Vbi-Vp))^1.5-
((Vbi-VG)/(Vbi-Vp))^1.5))/
(-Vp-2/3*(Vbi-Vp)*(1-(Vbi/(Vbi-
Vp))^1.5)),(1-VG/Vp)^2)'
(Checksum: # 60795d Bytes: 278.5)

Variables: Vbi: 1
Vp: -2.5

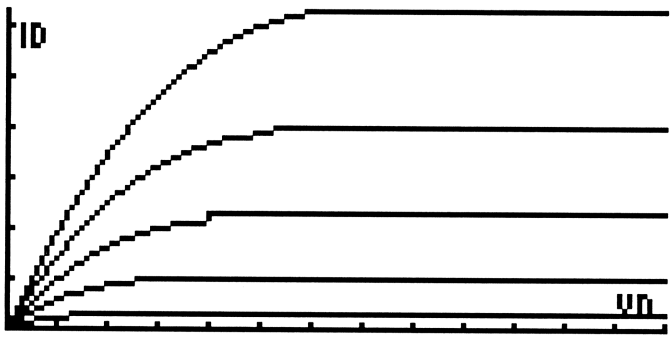
Plot parameters:

XRNG:	0 5	YRNG:	0 1
INDEP:	VD	RES:	0
AXES:	(0,0)		
PTYPE:	FUNCTION		
DEPND:	ID		

(PPAR): { (0,0) (5,1) VD 0 (0,0) FUNCTION ID }

Level-1 Stack arg: { IDID0 VG { 0 -.5 -1 -1.5 -2 } }

Result: A plot of a theoretical ID-VD curve for a FET. The *y*-axis is ID/ID_0, where ID_0 is *ID* at saturation, with zero gate voltage:



Compare this curve with those found in typical electronics textbooks.

An undocumented feature of the HP48 is its ability to use indexing to extract items from lists or matrices: for example, `'AAA(2)' EVAL` will return the third item in a list named `AAA`; and `'AAA(1, 9)' EVAL` will return the number from the row 1, column 9 of an array named `AAA`.

See if you can create an equation using an indexed list, and use this equation with YSLICE to duplicate `MULTIPLOT`'s action with the built-in routines. You may find that `MULTIPLOT` is faster.

A Contour-Plotting Program

In Chapter 6, you were introduced to plotting data in three dimensions. But not all three-dimensional data sets can be reduced to an equation in three variables. Consider, for example, the need to measure current uniformity in a plating tank, or temperature distribution on a heat exchanger fin, or noise levels on a factory floor.

Although such data sets are empirically gathered—not analytically generated—you can nevertheless analyze them with the contour-plot approach by mapping the physical grid of measurements onto an array.

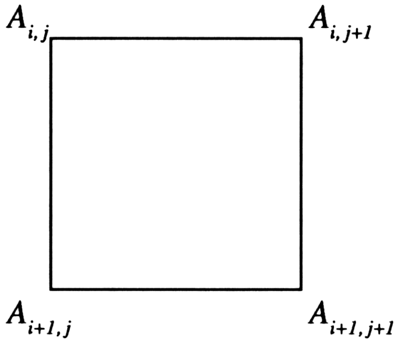
Description

CONTOUR makes a contour plot, taking data contained in an array and displaying it as a three-dimensional surface, as seen from above. The contour lines represent "isovalues"—places on the surface at the same "altitude," or value. An example follows the program listing.

CONTOUR takes all of its arguments from the Stack, including the array of data to be plotted. However, this array will be saved as ARRAY, so that you can modify it after running CONTOUR, if you wish.*

*Note that the easiest way to enter array data into the 48 is through the MatrixWriter, ⮣MATRIX (for more on the MatrixWriter, read chapter 14 in the User's Guide.)

CONTOUR divides the array into squares, with the points in the array being the corners of the squares:



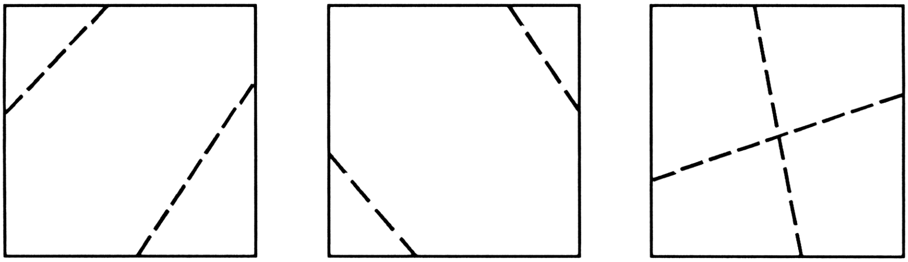
CONTOUR works on one square at a time, cycling through all possible contour values. At each contour value, CONTOUR searches for intersections of the desired contour line with the sides of the square, finding either zero, two or four intersections per square.

If CONTOUR finds zero intersections for a given contour value, it skips to the next value.

If it finds two intersections, it determines which two sides of the square are affected. Simple linear interpolation is used to find the points of intersection, and the contour line segment is drawn in the square.

If it finds four intersections, CONTOUR has encountered a "saddle," where two diagonally opposite corners of the square are higher than the other two corners. Saddles are frequently found in the real world—potato chips, mountain passes, and (of course) a cowboy's saddle.

Saddles are difficult for CONTOUR to draw. It tries to draw a pair of roughly parallel contour lines, closest to the corners whose average value comes closest to the contour value. If the value of the contour is equal to the average of all four corners, then CONTOUR draws two crossing lines in the square.



Contour value closest to average of upper-left and lower-right corners

Contour value closest to average of lower-left and upper-right corners

Contour value equal to average of all four corners

In each case, simple linear interpolation determines the points of intersection. The more points you have, the more accurate CONTOUR is.

Variables

ARRAY: The name in which the given data array will be saved.

Suggestion: Before keying in CONTOUR, store this list into CST in your **TOOLS** directory, and then press CST to use it as a typing aid:

```
{ ARRAY smallest largest lowlimit hilimit stepsize
  range rows cols ii j ul ur ll lr small big top
  bottom left right contour }
```

Listing

CONTOUR

```
« PICT PURGE DUP 'ARRAY' STO
  1 GETI DUP
  → smallest largest  (Local variables for max. and min. values)
  « DO GETI DUP              (Find array's max. and min. values)
       smallest MIN 'smallest' STO
       largest MAX 'largest' STO
    UNTIL -64 FS?C
    END
    DROP2 largest smallest DUP2 -     (Find array's range)
  »
  { # 0d # 0d } PVIEW ARRAY SIZE EVAL
  → lowlimit hilimit stepsize                      (Save array
    largest smallest range rows cols              parameters)
  « 1 rows R→C PMIN
    cols 1 R→C PMAX               (Set drawing boundaries)
    1 rows 1 -
    FOR ii                                   (For each row...
      1 cols 1 -
      FOR j                                  and each column...
        ARRAY ii j 2 →LIST GET               ...work the four cor-
        ARRAY ii j 1 + 2 →LIST GET           ners of the square)
        ARRAY ii 1 + j 2 →LIST GET
        ARRAY ii 1 + j 1 + 2 →LIST GET
        4 DUPN 4 DUPN MIN MIN MIN
        5 ROLLD MAX MAX MAX
        0 0 0 0
        → ul ur ll lr small big
          top bottom left right
        « lowlimit hilimit
          FOR contour               (For each contour value...
            IF 'contour ≥ small           ...if necessary...
                AND contour ≤ big'
            THEN     ...find the number of edge intersections)
              'contour > MIN(ul,ur) AND
                contour < MAX(ul,ur)'
```

```
→NUM 'top' STO
'contour > MIN(ll,lr) AND
  contour < MAX(ll,lr)'
→NUM 'bottom' STO
'contour ≥ MIN(ul,ll) AND
  contour ≤ MAX(ul,ll)'
→NUM 'left' STO
'contour ≥ MIN(ur,lr) AND
  contour ≤ MAX(ur,lr)'
→NUM 'right' STO
'top+bottom+left+right' →NUM
CASE                 (How many intersections?)
  DUP 0 ==              (none...
    THEN DROP           ...skip computations)
    END
  DUP 2 ==              (2 intersections)
    THEN DROP
      IF top
      THEN
        'j+(contour-ul)/(ur-ul)'
        →NUM ii R→C
        IF bottom
        THEN                 (Top-to-bottom)
          'j+(contour-ll)/(lr-ll)'
          →NUM ii 1 + R→C LINE
        ELSE       (Okay, not top-to-bottom)
          IF left           (Top-to-left?)
          THEN
            'ii+(contour-ul)/
              (ll-ul)'
            →NUM j SWAP R→C LINE
          ELSE          (Aha—top-to-right)
            'ii+(contour-ur)/
              (lr-ur)'
            →NUM j 1 + SWAP
            R→C LINE
          END
        END           (IF...bottom...ELSE)
      ELSE  (Not top, so try bottom edge)
        IF bottom
        THEN
```

```
          'j+(contour-ll)/
            (lr-ll)'
          →NUM ii 1 + R→C
          IF left
          THEN          (Bottom-to-left)
            'ii+(contour-ul)/
              (ll-ul)'
            →NUM j SWAP
            R→C LINE
          ELSE          (Bottom-to-right)
            'ii+(contour-ur)/
              (lr-ur)'
            →NUM j 1 + SWAP
            R→C LINE
          END
        ELSE      (Not bottom, either, so...
          'ii+(contour-ul)/
            (ll-ul)'  ...left-to-right)
          →NUM j SWAP R→C
          'ii+(contour-ur)/
            (lr-ur)'
          →NUM j 1 + SWAP
          R→C LINE
        END          (IF...bottom...ELSE)
      END              (IF...top...ELSE)
    END          (Case of 2 intersections)
  4 ==  (Case of 4 intersections—a saddle—
           so calculate those 4 intersections)
    THEN 'j+(contour-ul)/(ur-ul)'
      →NUM ii R→C
      'j+(contour-ll)/(lr-ll)'
      →NUM ii 1 + R→C
      'ii+(contour-ul)/(ll-ul)'
      →NUM j SWAP R→C
      'ii+(contour-ur)/(lr-ur)'
      →NUM j 1 + SWAP R→C
      'ABS(contour-(ul+lr)/2)'
      →NUM
      'ABS(contour-(ll+ur)/2)'
      →NUM DUP2
```

```
                              IF <        (Diagonal to upper right)
                              THEN DROP2 ROT (Closer to ul, lr)
                              ELSE
                                IF >      (Diagonal to upper left)
                               THEN ROT ROT (Closer to ll, ur)
                                END (So crossover is at midpoint)
                              END                   (IF...<...ELSE)
                              LINE LINE
                           END              (Case of 4 intersections)
                      END                                    (CASE)
                   END                   (contour range IF test)
                   stepsize
                STEP                        (Next contour value)
             »
           NEXT                                    (For j loop)
        NEXT                                       (For ii loop)
        smallest "Min value" →TAG
        largest "Max value" →TAG
        lowlimit "Min contour" →TAG
        hilimit "Max contour" →TAG
        stepsize "Contour step" →TAG
     »
  »
```

Checksum: # 21186d
Bytes: 2420.5

	Stack Arguments	Stack Results
5:		*minimum data value* (tagged)
4:	*low limit* (real)	*maximum data value* (tagged)
3:	*high limit* (real)	*lower contour limit* (tagged)
2:	*step size* (real)	*upper contour limit* (tagged)
1:	$n{\times}m$ (real) data array	*contour step size* (tagged)

Notes: Clearly, you could shorten the program with shorter variable names; these were used for clarity. Also, you might explore alternate ways to arrive at the same solution. As you saw with SCAN/PSCAN, there's always more than one way to do things.*

Example

With the Stack set up as follows, use CONTOUR to get the result shown:

```
4:  0
3:  5
2:  1
1:  the following array (use the MatrixWriter):

[[ 0.8 1.3 2.2 0.5 1.3 2.4 1.3 0.5 ]
 [ 0.9 1.5 2.5 0.5 0.9 0.5 0.5 1.5 ]
 [ 1.8 3.0 3.2 1.8 0.5 1.1 2.1 3.0 ]
 [ 1.9 3.2 4.3 1.6 0.8 2.0 2.7 3.3 ]
 [ 1.8 2.1 2.9 1.9 0.5 1.7 2.6 3.7 ]
 [ 1.5 1.4 1.1 0.1 1.5 2.4 2.9 4.0 ]
 [ 1.4 0.9 0.5 1.3 2.1 3.2 3.6 4.2 ]
 [ 1.1 0.9 0.5 1.2 2.8 3.9 4.3 4.8 ]]
```

*Speaking of other ways to do things: Can you write a program using EQ and VPAR to create ARRAY from any three-dimensional function (thereby adding CONTOUR to your suite of 3-D tools)? How about the other way? Can you write a program using indexed values (e.g. ARRAY(X, Y)) to extract values from ARRAY for use in YSLICE, PCONTOUR, SLOPEFIELD or WIREFRAME plots?

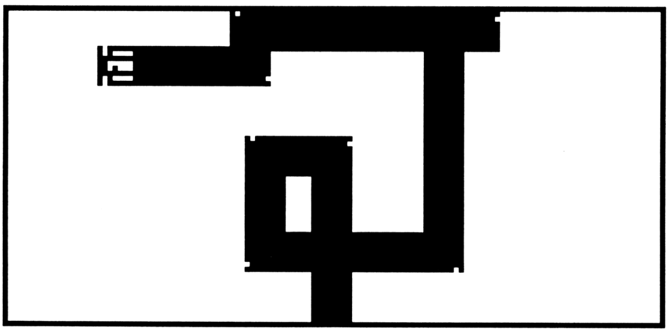
Drive a Bulldozer Around the Display

This is a fun demonstration of using small grobs as "sprites"—objects that you can move around the display at will.

Description

The main program, called BULLDOZER, uses a list called DOZDATA, which, in turn, consists of two sublists. The first sublist is a list of four grobs, showing the bulldozer facing north, east, south and west. The second sublist is a list of four complex numbers representing those directions. Thus if you tire of the bulldozer image, you can always create another 8×8 grob, then make 3 rotated copies, assemble a new DOZDATA, and run the program with your own custom "sprite."

To start the program, just execute BULLDOZER. A bulldozer will appear at the bottom of the display and start plowing a swath towards the top. Use the arrow keys to control its direction (it will stop when it hits the wall at the edge of the display). Note that these arrow keys are not "north, south, east and west." Rather, they are "forward, reverse, left-turn and right-turn."



A speed factor is built into BULLDOZER; you change the bulldozer's speed by increasing or decreasing this number. The speed is stored as a local variable in the program, in case you want to add a "gas pedal" key to the program.

Press [ENTER] to halt the program (if you use [CANCEL], it may leave a spurious KEY output on the Stack).

Variable

DOZDATA: The grob data for BULLDOZER:

```
{ { GROB 8 8 FFC37EDA5A5A5AFF   (Dozer north)
    GROB 8 8 FB1AFF1D1CFF1AFB   (Dozer east)
    GROB 8 8 FF5A5A5A5B7EC3FF   (Dozer south)
    GROB 8 8 DF58FF38B8FF58DF } (Dozer west)
  { (0,1) (1,0) (0,-1) (-1,0) } }
                  (North, East, South and West
                        in complex numbers)
```

(Checksum: # 33345d Bytes: 172.5)

Listing

BULLDOZER

```
« PICT PURGE { # 0d # 0d } PVIEW
  0 131 XRNG 0 63 YRNG (0,0) (131,63) BOX (Define area)
  DOZDATA 1 GET 1 GET (61,8) 1 10 (0,1) RCLF
  → cat locn gear speed direction flags
  « 50 CF PICT locn cat REPL
    DO 'gear*direction+locn' EVAL C→R
      8 MAX 62 MIN SWAP 1 MAX 123 MIN SWAP
      R→C 'locn' STO PICT locn cat REPL
      .3 speed / WAIT
    UNTIL
      IF KEY
      THEN → k
        « CASE
            'k==25'                              (Forward)
              THEN 1 'gear' STO
              END
            'k==35'                              (Reverse)
              THEN -1 'gear' STO
              END
            'k==34'                              (Left turn)
              THEN DOZDATA OBJ→ DROP
                DUP direction POS 1 -
                IF DUP 0 ==               (You can't turn
                THEN DROP 4                past 0°)
                END
                SWAP OVER GET
                'direction' STO GET 'cat' STO
              END
            'k==36'                              (Right turn)
              THEN DOZDATA OBJ→ DROP
                DUP direction POS 1 +
                IF DUP 5 ==               (You can't turn
                THEN DROP 1                past 360°)
                END
```

```
                    SWAP OVER GET
                    'direction' STO GET 'cat' STO
                  END
                'k==51'
                  THEN 50 SF                    (Quit)
                  END
              END                               (CASE)
            »
        END                                     (IF...KEY)
        50 FS?
      END                                       (DO...UNTIL)
      flags STOF                                (Clean up)
  »
»
```

Checksum: # 6914d
Bytes: 933

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: The bulldozer leaves some "litter" when it turns. And different grobs will leave different garbage (the culprits here are the little cutouts behind the dozer's blade). This is because the program turns, increments the position and *then* writes to the display. A commercial game machine would fix this by using a separate sprite for the tracks and/or a "mask" sprite under the bulldozer. But both approaches are slow here and make the dozer flicker. So for this demo, just ignore the litter.*

*But in case you're interested in exploring other solutions here's an observation: A sprite with an all-black border always leaves tracks; if it has an all-white border, it never leaves tracks.

A Friendly Game of Checkers

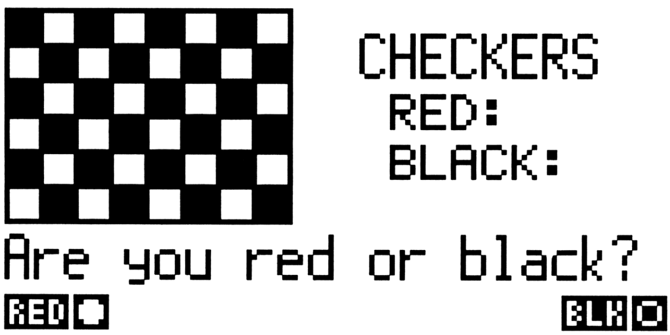
Here is a checkers game to be played by two 48's—via the Infrared interface or wired serial ports.

This is the book's largest application. If you've been working through Chapter 9 nonstop to this point, **Stop!** Go get some cookies and milk. Give your brain a rest. Then come back.

Description

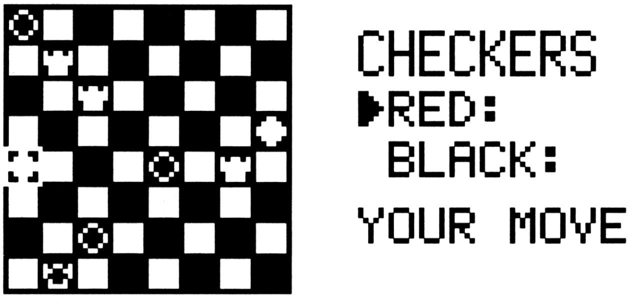
You start the game by executing `CHKRS`.

The title screen should appear, with two menu keys to choose RED or BLK (okay, so it's white and blue—give HP a few more years....)

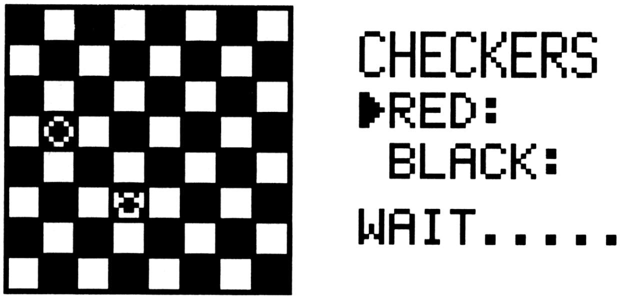


After someone has chosen a color, the other player's color is set, and the 48's set up their playing boards accordingly.

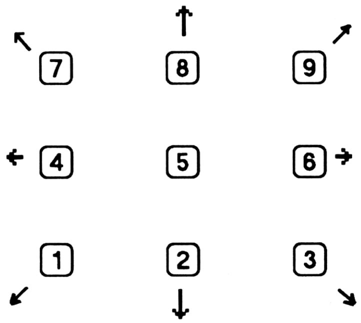
Red moves first, and the two players take turns...



...until one player is out of pieces.



In CHKRS, the numeric keypad becomes a "selector control pad." As with SCAN, the 5 key is the neutral center of the pad, and the other non-zero keys act as arrow keys:



When it's your move, the 48 will highlight a suggested piece to move. Its selections are not very smart, so use the numeric keys to move the highlight to the piece you want to move, and press ENTER. Then press one of the diagonal-move keys (1, 3, 5 or 7) to indicate the direction you wish to move.

If you choose an invalid move, the piece you selected remains highlighted and you must re-select the move. If it's a valid move or jump, the 48 will update the board display, and send the move information to your opponent's machine. It will also crown your piece if that move sends it to the 8th row.

When your move is over, the 48 passes control to your opponent's machine. At the end of each player's turn, the 48 checks to see if both of you are still in the game, and then goes through the selection and movement procedure again. This cycle continues until one or the other of the players has no more pieces on the board, at which time both machines declare the winner.

The checkerboard layout is contained in an 8×8 array, appropriately called LAYOUT, which is updated during the game to reflect each move. The graphic checkerboard is stored in a grob called BOARD. If you accidentally erase BOARD, don't worry. The STARTUP routine checks for the existence of BOARD, and if it doesn't find it, calls a routine called MAKEBOARD to generate a new one. The pieces themselves are stored as 8×8 grobs called RPIECE, BPIECE, RKING and BKING.

This is indeed a "friendly" game of checkers. A complete and ruthless game would probably require an entire chapter in this book, so this version has the following limitations:

- It won't do multiple jumps (but notice that flag 58 has been left in reserve—for indicating "multiple jump allowed"—so if you're ambitious, go for it).

- The forced-jumping rule is not in effect: If you're in a position to jump, then you are *not* forced to "jump or lose the piece."

- There's no "boss key" to quickly save the current game status as your boss walks up. To abort the game, you must press CANCEL and risk leaving junk on the Stack.

Subroutines

CHKRS is organized in a modular fashion. This keeps each routine short, easy to understand, and tightly focused.

STARTUP: A routine called initially by CHKRS to check for the existence of a checkerboard grob called BOARD. If it doesn't find BOARD, then STARTUP calls MKBOARD to create one.

STARTUP also prompts the user to choose sides, and waits for input from either the keyboard or the I/O port.

REDRAW: A routine that maps the contents of LAYOUT onto PICT.

MYMOVE: The busiest module in the application, MYMOVE calls SELECT to suggest a piece to play. It accepts key input on the direction to move the piece of your choice, sending this information to a routine called VALID.

VALID: The routine that determines whether your proposed move is legal: You may move only to diagonally adjacent, unoccupied squares, unless you are jumping. You may jump only an opponent in a diagonally adjacent square, and only if the square beyond your opponent's piece is empty. Also, only kings may move or jump backwards.

THMOVE: A routine that waits for an "M", "J", "K" or "D" string from the other machine, then translates the move information and calls MOVEIT to update LAYOUT and PICT. When a "D" is received, THMOVE sets flag 59 and exits.

SELECT: This routine simply searches LAYOUT for the first occurrence of your playing pieces as its suggestion for your next move. Fortunately, it doesn't commit to any square until you press [ENTER] with the square highlighted (The highlight can be on any square—even an empty one or one occupied by an opponent—so if the chosen square is not occupied by one of your pieces, the highlight remains). SELECT will not move past the board edges.

MOVEIT: This routine takes the parameters of the validated move and the piece to be moved and performs the manipulations on LAYOUT and PICT.

MKBOARD: The routine that generates the checkerboard inside a 57×57 grob—called by STARTUP when necessary.

WHOZAT: A small routine that determines which player (if any) is occupying a given square.

C→L: A utility (quite generally useful) that converts a complex number (*x*, *y*) into a list of the form { # *row* # *col* }.

GL↓: A text formatting routine (see page 158).

GLABEL: A text formatting routine (see page 157).

Variables

LAYOUT: An 8×8 array listing the entire layout of the checkerboard, created by STARTUP. Row 1 of the array is the bottom row of the checkerboard. Element values:

0 = empty   1 = red piece   2 = black piece
             3 = red king   4 = black king

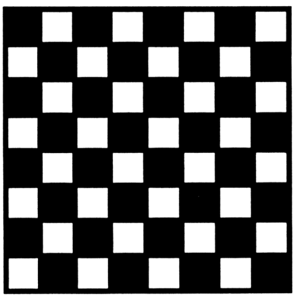
Elements on red squares are always zero. Red squares are identified by adding the row and column indices. The sum is always even for red, odd for black.

Initial values (red player's values are shown; exchange 1's and 2's for black players initial values):

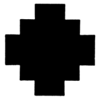
```
[[ 0 1 0 1 0 1 0 1 ]
 [ 1 0 1 0 1 0 1 0 ]
 [ 0 1 0 1 0 1 0 1 ]
 [ 0 0 0 0 0 0 0 0 ]
 [ 0 0 0 0 0 0 0 0 ]
 [ 2 0 2 0 2 0 2 0 ]
 [ 0 2 0 2 0 2 0 2 ]
 [ 2 0 2 0 2 0 2 0 ]]
```

Checksum: LAYOUT is dynamic; checksums change.
Bytes: 537.5

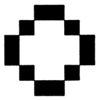
BOARD: 57×57 grob of blank checkerboard, created by MKBOARD.
Checksum: #31247d   Bytes: 475.5



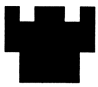
RPIECE: Grob of a red piece: GROB 8 7 0081C3E7E7C381



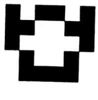
BPIECE: Grob of a black piece: GROB 8 7 00814224244281



RKING: Grob of a red king: GROB 8 7 0000A5E7E7C3C3



BKING: Grob of a black king: GROB 8 7 0000A5662442C3



CHKRS

```
« RCLF 'Flags' STO                              (Save defaults)
  -40 CF                               (Turn off clock display)
  STARTUP REDRAW                  (Initialize game, choose sides…
  IF 57 FS?                          …draw board—red goes first)
  THEN 59 SF                          (Flag 57 set means: "I'm red")
  ELSE 59 CF                          (Flag 59 set means: "My turn")
  END
  DO
    PICT { # 70d # 40d }
    #54 #8 BLANK REPL
    IF 59 FS?                                       (My turn?)
    THEN { # 70d # 40d } "YOUR MOVE"
      2 GLABEL MYMOVE
    ELSE { # 70d # 40d } "WAIT....."
      2 GLABEL THMOVE
    END
  UNTIL
    IF LAYOUT →STR DUP "1" POS             (Game ends when…
      SWAP "3" POS OR NOT DUP                reds are gone…
    THEN "BLACK WINS" SWAP
    END
    IF LAYOUT →STR DUP "2" POS            …or blacks are gone)
      SWAP "4" POS OR NOT DUP
    THEN "RED WINS" SWAP
    END
    OR
  END
  Flags STOF                          (Restore previous states)
  'Flags' PURGE                                     (Clean up)
»
```

Checksum: # 19875d
Bytes: 538.5

	Stack Arguments	Stack Results
1:	(none)	"RED WINS" or "BLACK WINS"

Notes: CHKRS is the main program. Be sure both players have the same I/O setup. This means checking the status of IOPAR, and clearing system flags –33, –34 and –38.

The layout data is stored in the 8×8 array, LAYOUT. Pieces on squares are identified by number:

0 = empty	1 = red piece	2 = black piece
	3 = red king	4 = black king

Row 1 in LAYOUT is the first row of the array; Row 1 of the checkerboard is the bottom row of the board—the row nearest you. This makes for faster computing. Notice also that the sum of the row and column numbers of a red square is an even number, while the sum of row and column numbers of a black square is an odd number. This fact speeds up execution time.

Since the game is played only on the black squares, an 8×4 array could also be used. But this would require monitoring of zigzag movements, and the additional code would far outweigh any memory savings from using the smaller array.

All the red squares in the array contain 0's. You could use the red squares for storing game status, etc., if you incorporate a "boss key" into your game, but be aware that some sections of the application check *all* squares for zeros—you can't use the red squares for temporary storage during a game.

These user flags are used:

57	SET: You are red.	CLEAR: You are black.
58	(reserved for use in multiple jumping)	
59	SET: Your move.	CLEAR: Their move.

After initialization, CHKRS checks flag 57. Since red always goes first, for the first move CHKRS sets user flag 59 to match flag 57. It then enters a DO...UNTIL loop, which can be exited only when one player runs out of pieces (or via [CANCEL]). Throughout the game, depending on the status of flag 59, CHKRS calls either MYMOVE or THMOVE ("THeir MOVE").

When it's your opponent's move, the 48 monitors the input buffer for any activity. As soon as some information enters the buffer, the 48 analyzes it and updates LAYOUT and the display.

To communicate between the two machines, the 48 relies on the commands XMIT, BUFLEN and SRECV.

XMIT takes a string from Level 1 and transmits it over the current I/O port. If the transmission is successful, then a 1 is returned to the Stack; otherwise the unsent fragment of the string is put into Level 2, and a 0 into Level 1. Use ERRM to see the cause of the error.

BUFLEN returns the number of characters in the I/O buffer to Level 2 and puts a 1 to Level 1 if no framing errors or UART overruns occur. If an error does occur, then BUFLEN returns the number of characters received before the error to Level 2, and a 0 to Level 1.

SRECV takes the number specified in Level 1, returns that number of characters from the I/O buffer to Level 2, and returns a 1 to Level 1 if the data were retrieved successfully. If an error occurs during SRECV, then Level 2 contains the data received before the error, and Level 1 contains a zero. Execute ERRM to see the cause of the error.

CHKRS does not use the error-trapping capability of these commands, so in order to keep transmission errors to a minimum, CHKRS uses a small number of short messages to communicate between machines. Each message is transmitted as a list inside a string—the most efficient way of passing a variable number of parameters. Valid messages are:

"{ (x_1, y_1) (x_2, y_2) "M" }"	Move the piece at (x_1, y_1) to (x_2, y_2).
"{ (x_1, y_1) (x_2, y_2) "J" }"	Jump the piece at (x_1, y_1) to (x_2, y_2), capturing the opposing piece en route.
"{ (x, y) "K" }"	Crown the piece at (x, y), replacing it with a king of that color.
"{ "D" }"	Done. It's the opponent's turn.

The only exception to this "list in a string" rule is the "R" or "B" that is transmitted at the start of the game, when players are choosing sides.

STARTUP

```
« IF BOARD TYPE 11 ≠                         (Does BOARD already exist?)
  THEN MKBOARD                                (If not, then make it)
  END
  BOARD PICT STO                              (Draw board)
  (1,-1) PMIN (19.5714285714,8) PMAX          (Set user limits)
  { # 0d # 0d } PVIEW                         (Display board)
  { # 70d # 5d } "CHECKERS" 3 GL↓             (Title labels)
  " RED:" 2 GL↓
  " BLACK:" 2 GLABEL
  PICT RCL                                    (Set up prompt to choose color)
  PICT { # 0d # 43d } # 57d # 14d BLANK REPL
  { # 0d # 45d } "Are you red or black?" 3 GLABEL
  PICT { # 0d # 57d }
 GROB 21 7 FFFDF1919D815D5501915501 5D5501519D81FFFDF1
                                              ("RED" menu key)
  REPL PICT { # 110d # 57d }
 GROB 21 7 FFFDF19D5D815D55719D95715D5571915D81FFFDF1
                                              ("BLACK" menu key)
  REPL
  OPENIO                   (Necessary to receive input from the other 48)
  DO
  UNTIL
    IF KEY
    THEN DUP
      CASE
        11 SAME                               (User chooses red....
          THEN DROP "R" "B" XMIT               ... tell opponent)
          END
        16 SAME                               (User chooses black....
          THEN "B" "R" XMIT                    ...tell opponent)
          END
        0
      END
    ELSE 0
    END
```

```
      IF BUFLEN DROP DUP                        (Opponent chose first)
      THEN SRECV ROT                            (What user gets)
      END
      OR  (DO UNTIL loop ends when one of the 3 options is satisfied...
   END        ...user chooses red or black, or opponent chooses)
   CLOSEIO                                      (To save battery life)
   SWAP PICT { # 0d # 0d } ROT REPL             (Remove prompt)
   IF "R" SAME
   THEN 57 SF                                   ("I'm red")
      [[ 0 1 0 1 0 1 0 1 ]                      (Red's startup LAYOUT)
      [ 1 0 1 0 1 0 1 0 ]
      [ 0 1 0 1 0 1 0 1 ]
      [ 0 0 0 0 0 0 0 0 ]
      [ 0 0 0 0 0 0 0 0 ]
      [ 2 0 2 0 2 0 2 0 ]
      [ 0 2 0 2 0 2 0 2 ]
      [ 2 0 2 0 2 0 2 0 ]]
      { # 70d # 17d }
   ELSE 57 CF                                   ("I'm black")
      [[ 0 2 0 2 0 2 0 2 ]                      (Black's startup LAYOUT)
      [ 2 0 2 0 2 0 2 0 ]
      [ 0 2 0 2 0 2 0 2 ]
      [ 0 0 0 0 0 0 0 0 ]
      [ 0 0 0 0 0 0 0 0 ]
      [ 1 0 1 0 1 0 1 0 ]
      [ 0 1 0 1 0 1 0 1 ]
      [ 1 0 1 0 1 0 1 0 ]]
      { # 70d # 27d }
   END
   PICT SWAP 134 CHR
   2 →GROB REPL          (Put a "selection arrow" beside user's color)
   'LAYOUT' STO
»
```

Checksum: # 42104d

Bytes: 1927.5

	Stack Arguments	Stack Results
1:	(none)	a real number

Notes: STARTUP draws the checkerboard in PICT, prompts the user to choose a color, communicates this choice to the opponent's 48, and sets up pieces on the board to start the game.

If the user chooses a color from the keyboard, then a single-character string identifying the opposite color ("R" or "B") is transmitted to the opponent's 48. If the user doesn't choose a color before a "R" or "B" is received from the other machine, then the 48 acts on that string.

If the user is red, the 48 sets user flag 57 (the "I'm red" flag), initializes LAYOUT with red pieces in the first three rows, and calls REDRAW to put the pieces from LAYOUT in the right places on the board. Similarly, if the user is black, the 48 clears user flag 57, initializes LAYOUT with black pieces in the first three rows, and calls REDRAW.

REDRAW

```
« PICT { # 0d # 0d } BOARD REPL  (Redraw a blank board)
  1 8
  FOR y
    1 8
    FOR x
      IF x y + 2 MOD                (Only check black squares)
      THEN PICT x y R→C             (Calculate square location)
        'LAYOUT' OVER C→L GET       (Check array contents)
        CASE
          DUP 1 SAME                (1 is a red piece)
            THEN DROP RPIECE GXOR
            END
          DUP 2 SAME                (2 is a black piece)
            THEN DROP BPIECE GXOR
            END
          DUP 3 SAME                (3 is a red king)
            THEN DROP RKING GXOR
            END
          4 SAME                    (4 is a black king)
            THEN BKING GXOR
            END
          DROP2
        END                         (CASE)
      END                           (IF)
    NEXT                            (FOR x loop)
  NEXT                              (FOR y loop)
»
```

Checksum: # 25345d   Bytes: 296.5

Stack Arguments: (none)   Stack Results: (none)

Notes: REDRAW redraws the pieces on the checkerboard, according to the contents of LAYOUT. It assumes that BOARD already exists and redraws part of PICT.

MYMOVE

```
« WHILE 59 FS?                          (Loop to find and complete valid move)
  REPEAT SELECT 0 WAIT VALID            (Select, validate movement)
    CASE
      DUP "X" SAME                      (Invalid move—try again)
        THEN DROP
        END
      DUP "D" SAME                      (End of move)
        THEN DROP 59 CF
        END
      DUP "M" SAME OVER "J" SAME OR     (Move or jump)
        THEN 3 DUPN 3 →LIST →STR
          XMIT DROP                     (Tell the other machine...
          MOVEIT 59 CF                  ...update LAYOUT, display, end move)
          IF DUP IM 8 ==                (If a piece reaches row 8...
          OVER WHOZAT 2 ≤ AND           ...and it isn't a king...)
          THEN "K" DUP2 2 →LIST →STR    ..."king me")
            XMIT DROP                   (Tell the other machine...
            MOVEIT 59 CF                ...update LAYOUT and display)
          ELSE DROP
          END                           (IF)
        END
    END                                 (CASE)
  END                                   (WHILE ... REPEAT loop)
  IF 59 FC?                             (End of turn?)
  THEN { "D" } →STR XMIT DROP           (Pass token to other 48)
  END
»
```

Checksum: # 8080d
Bytes: 343

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: MYMOVE prompts user to select the piece to move, validates the move, communicates it to the opponent's 48 (sends "M", "J", "K", or "D"), updates LAYOUT and the display, and passes the turn to the opponent (clears flag 59). Notice that if MYMOVE gets an "X" from VALID, it repeats SELECT and VALID until you make a valid move.

THMOVE

```
« OPENIO                                   (Necessary to receive data)
  DO
    IF BUFLEN DROP DUP                       (Check buffer for input)
    THEN SRECV DROP OBJ→ EVAL         (Read buffer, evaluate list)
      → move                      (Store only Level 1 as local variable)
      « CASE
          move "D" SAME          (Other 48 passes token to me...
            THEN 59 SF             ...therefore, it's my turn)
            END
          move "K" SAME                                  ("King me")
            THEN (9,9) SWAP -               (Rotate coordinates)
              move MOVEIT          (Update LAYOUT and display)
            END
          move "M" SAME
          move "J" SAME OR                           (move or jump)
            THEN (9,9) ROT -
              (9,9) ROT -                   (Rotate coordinates)
              move MOVEIT DROP(Update LAYOUT, display)
            END
        END                                               (CASE)
      »
    ELSE DROP                             (No input in buffer yet)
    END                                              (IF BUFLEN...)
  UNTIL 59 FS?                           (that is, UNTIL my turn)
  END                                                (DO ... UNTIL)
  CLOSEIO                              (To conserve battery power)
»
```

Checksum: # 35460d

Bytes: 322.5

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: THMOVE receives the data string from the opponent's 48, translates it and updates LAYOUT and the display accordingly (and sets flag 59). It does not validate the opponent's moves.

SELECT

```
« IF 57 FS?                                                   (If I'm red...
  THEN 1 3                                     ...then search for red pieces...
  ELSE 2 4                                ...otherwise, search for black pieces)
  END
  → p1 p2                                           (The search for the pieces)
  « 'LAYOUT' 1                                          (Initialize the search)
    DO GETI                                                          (Search...
    UNTIL DUP p1 == SWAP
      p2 == OR                                         ...until a piece is found)
    END
    1 -                                           (Index is 1 count too high)
  »
  SWAP DROP DUP 8 / CEIL SWAP 8 MOD                        (Convert counter...
  IF DUP 0 ==
  THEN DROP 8
  END
  SWAP R→C                              ... into a square location—a complex #)
  DO HILITE 0 WAIT                         (Highlight the square and wait for...
  UNTIL
    → loc key                                                     ...key input)
    « loc HILITE
      CASE
        'key==83.1'                                  (Key [2]—down 2 squares)
          THEN C→R 2 - OVER 2 MOD 1 + MAX R→C 0
          END
        'key==63.1'                                    (Key [8]—up 2 squares)
          THEN C→R 2 + OVER 2 MOD 7 + MIN R→C 0
          END
        'key==72.1'                                  (Key [4]—left 2 squares)
          THEN C→R SWAP 2 - OVER 2 MOD
          1 + MAX SWAP R→C 0
          END
        'key==74.1'                                 (Key [6]—right 2 squares)
          THEN C→R SWAP 2 + OVER 2 MOD
          7 + MIN SWAP R→C 0
          END
```

```
          'key==64.1 AND RE(loc)<8
          AND IM(loc)<8'                    (Key [9]—up and right)
            THEN (1,1) + 0
            END
          'key==62.1 AND RE(loc)>1
          AND IM(loc)<8'                    (Key [7]—up and left)
            THEN (-1,1) + 0
            END
          'key==82.1 AND RE(loc)>1
          AND IM(loc)>1'                    (Key [1]—down and left)
            THEN (1,1) - 0
            END
          'key==84.1 AND RE(loc)<8
          AND IM(loc)>1'                    (Key [3]—down and right)
            THEN (1,-1) + 0
            END
          'key==51.1'    ([ENTER] key—select highlighted square)
            THEN DUP C→L 'LAYOUT' SWAP GET
              DUP DUP 1 == SWAP 3 == OR     (If the piece
              57 FC? XOR AND        on the square is my color...
            END                 ...return its location to Stack)
           0                                (Otherwise,...
         END
      »
    END                                     (Repeat the search)
  »
```

Checksum: # 43360d
Bytes: 984

	Stack Arguments	Stack Results
1:	(none)	*location of selected piece* (complex)

Notes: SELECT searches LAYOUT for the first occurrence of the user's piece and suggests it as the piece to move. By redefining the numeric keypad as a direction control pad, it also allows the user to move the selector around the board to choose a piece to move. Then, with the highlight on a valid piece, ENTER selects the piece. SELECT uses HILITE to draw an inverted box around the indicated square.

Note that to make them applicable to either color, many routines use the XOR command, as in this sequence from SELECT:

```
« ... DUP 1 == SWAP 3 == OR
  57 FC? XOR AND ...
»
```

This says: "If the square has a red piece and I'm red, OR if the square has a black piece and I'm black ... ", thus eliminating the need for:

```
« ...
  IF 57 FS?
  THEN DUP 1 == SWAP 3 == OR
  ELSE DUP 2 == SWAP 4 == OR
  END ...
»
```

HILITE

```
« PICT OVER
  GROB 8 8 FF1818181818181 8FF
  GXOR
»
```

Checksum: # 4202d
Bytes: 46

	Stack Arguments	Stack Results
1:	*square location* (complex)	*same square location* (complex)

Notes: HILITE highlights the indicated square by drawing an inverse box around it. It also "un-highlights" the square.

VALID

```
« OVER DUP
  → oldloc key newloc jumploc
  « CASE
      'key==62.1'                          (Key [7]—up and left)
        THEN (-1,1)
        END
      'key==64.1'                          (Key [9]—up and right)
        THEN (1,1)
        END
      'key==82.1' oldloc WHOZAT
      2 > AND                  (Key [1]—down and left—kings only)
        THEN (-1,-1)
        END
      'key==84.1' oldloc WHOZAT
      2 > AND                 (Key [3]—down and right—kings only)
        THEN (1,-1)
        END
      "X"                                          (Invalid key)
    END                                                 (CASE)
    IF DUP TYPE 1 ==          (Complex type means a valid key)
    THEN
      → inc                                     (Save increment)
      « oldloc inc + DUP C→R          (Calculate new location)
        IF DUP 0 > SWAP 9 < AND SWAP (If in bounds ...
          DUP 0 > SWAP 9 < AND AND
        THEN 'newloc' STO
          IF newloc WHOZAT NOT ... and if nobody's there...
          THEN oldloc newloc "M"    ... then do the move)
          ELSE newloc DUP                  (Somebody's there)
            'jumploc' STO
            inc + DUP C→R
            IF DUP 0 > SWAP 9 < AND
              SWAP DUP 0 > SWAP 9 <
              AND AND      (If it's a jump ...and in bounds...
            THEN 'newloc' STO
              IF newloc WHOZAT NOT...far side is vacant
```

```
                jumploc WHOZAT 2 MOD...and ctr. piece
                57 FS? XOR AND       ...is the other guy...
              THEN oldloc newloc "J"     ...then jump)
              ELSE "X"      (Otherwise, not a valid jump)
              END                  (IF far side is vacant)
            ELSE DROP "X"          (Jump is out of bounds)
            END                    (IF jump is in bounds)
          END                          (IF nobody's there)
        ELSE DROP "X"              (Move is out of bounds)
        END                         (IF move is in bounds)
      »
    END                                     (IF valid key)
  »
»
```

Checksum: # 16646d
Bytes: 781

	Stack Arguments	Stack Results
3:		*starting location* (complex)
2:	*starting location* (complex)	*ending location* (complex)
1:	*keycode for move direction*	"J" or "M" or "X"

Notes: VALID validates the proposed move passed to it from MYMOVE. The contents of the string output at Stack Level 1 depend on whether the move is a valid Jump, a valid simple Move, or an invalid proposed move ("X"). In the case of an invalid move proposal, no location values are returned in Levels 2 and 3. VALID doesn't check for "king me" opportunities; MYMOVE does.

VALID uses WHOZAT to determine the target square's current occupant.

MOVEIT

```
« 0
  → move piece
  « IF move "M" SAME
       move "J" SAME OR                          (Move or Jump)
    THEN
       → oldloc newloc         (Store start and end locations)
       « 'LAYOUT' oldloc C→L       (Get piece from LAYOUT)
         DUP2 GET 'piece' STO
         0 PUT                (Blank out old LAYOUT location)
         'LAYOUT' newloc C→L
         piece PUT          (Put piece in new LAYOUT location)
         CASE                   (Select the appropriate grob)
            'piece==1'
              THEN RPIECE
              END
            'piece==2'
              THEN BPIECE
              END
            'piece==3'
              THEN RKING
              END
            'piece==4'
              THEN BKING
              END
         END                                          (CASE)
         'piece' STO          (Store the grob in place of the #)
         PICT oldloc piece GXOR   (Blank out old location)
         PICT newloc piece GXOR (Put piece in new location)
         IF move "J" SAME   (Extra work needed for jumps...
         THEN oldloc newloc + 2 /   ...find jumped square
            'LAYOUT' OVER C→L 0 PUT ...blank its LAYOUT
            PICT SWAP # 8d # 8d              location and its
            BLANK NEG REPL                   board location)
         END
         newloc      (Dummy Stack value—killed by ... END)
       »
    END                                     (IF Move or jump)
```

```
IF move "K" SAME                                    ("King me")
THEN
  → loc                                             (Store location)
  « 'LAYOUT' loc C→L                       (Get piece from LAYOUT...
    DUP2 GET DUP
    'piece' STO 2 + PUT              ...and replace it with a king)
    PICT loc # 8d # 8d
    BLANK NEG REPL                      (Blank out board location...
    PICT loc
    CASE
      'piece==1'          ... and replace it with a red king...
        THEN RKING
        END
      'piece==2'                              ...or a black king)
        THEN BKING
        END
    END                                                     (CASE)
    GXOR                                 (The actual replacement)
  »
END                                                (IF "king me")
»
»
```

Checksum: # 56746d
Bytes: 780.5

	Stack Arguments	Stack Results
3:	*starting location* (complex)	
2:	*ending location* (complex)	
1:	"J" or "M" or "K"	(none)

Notes: MOVEIT updates LAYOUT and PICT according to the move data received from other processes. For a "K" ("king me"), the piece's location is the Level-2 argument, with no Level-3 argument.

WHOZAT

```
« 'LAYOUT' SWAP C→L GET
»
```

Checksum: # 5341d
Bytes: 46.5

	Stack Arguments	Stack Results
1:	*square location* (complex)	value of LAYOUT there (0-4)

Notes: WHOZAT determines "who's at" a given location on the board.

C→L

```
« C→R SWAP 2 →LIST
»
```

Checksum: # 34716d
Bytes: 27.5

	Stack Arguments	Stack Results
1:	*square location* (complex)	array index { # *row* # *col* }

Notes: C→L converts a complex number to an array index.

MKBOARD

```
« PICT PURGE                              (Start with a clean slate)
  (0,-7) PMIN (131,56) PMAX                       (Set user limits)
  { # 0d # 0d } PVIEW           (Just for fun, show it being built)
  (0,0) (56,56) BOX                          (Outline of the board)
  7 56
  FOR y
     0 49
     FOR x
        IF x y + 2 MOD NOT  (Sum of row and column of black
        THEN PICT x y R→C   square is not odd in this case)
        GROB 8 8 FFFFFFFFFFFFFFFF GOR         (Fill black square)
        END
        7
     STEP
     7
  STEP
  PICT (0,0) (56,56) SUB
  'BOARD' STO                                      (Store as BOARD)
»
```

<u>Checksum</u>: # 65383d　　　<u>Bytes</u>: 315.5

<u>Stack Arguments</u>: (none)　　　<u>Stack Results</u>: (none)

<u>Notes</u>: MKBOARD makes a blank checkerboard and stores the grob under the variable name BOARD.

A Calendar Demo

With its time and date functions, the 48 is certainly equipped to be a time management tool. One of the features in most electronic time managers is some kind of perpetual calendar, usually presented in the classic seven-column format. As a final little demo, here's an example of what you could do.

Description

The program `CALEND` displays the current month in seven-column format, offering unshifted menu keys to increment the day, month and year; and ⬅-shifted menu keys to decrement the day, month and year. Press the **EXIT** key to exit the program.

`CALEND` uses `DISP` to build the calendar, then turns it into a grob via `LCD→`. The grob's contents are stored in `PICT`, and the graphics display is frozen—with the custom menu line displayed—via `PVIEW -1 WAIT`.

Note that `CALEND` doesn't use `PICT STO` to store the calendar in `PICT`. When the HP 48 executes `PICT STO`, it resizes `PICT` to zero, then to the size of the new grob. If your graphics display is active during this time (for example, during a `PVIEW`), you will see "snow" fill your screen momentarily. This is a graphical representation of part of the HP 48's memory and is displayed while the machine is re-sizing `PICT`.

However, since the `REPL` command does not cause `PICT` to be re-sized. `CALEND` uses `PICT { #0d #0d } ROT REPL`, instead of `PICT STO`, thus avoiding the "snow."

The DAY and ⬅ DAY menu keys in CALEND are not active, although "hooks" (entry points) are included in here so that you can use them to increment/decrement the days as you wish.

Of course, CALEND could also be embellished to do other useful things: set and clear appointments, create "to-do" lists, and do other time-management tasks.

Subroutines

MYR: is the major subroutine behind CALEND. Note that its algorithm uses DISP and not PVIEW to do the display. MYR was written and modified by several members of the CHIP HP48 user's group. The version presented here was developed by Ron Johnson and is used with his permission—and with much appreciation.

CALEND

```
« RCLMENU
  DATE DUP IP SWAP FP 100 * DUP IP SWAP FP 1E4 *
  → menu m d y
  « IFERR
      DO
        m y MYR                              (Create the calendar)
        LCD→ PICT { #0d #0d } ROT REPL        (Avoid snow)
        { { "DAY" } { "MON" } { "YR" } { } { }
          { "EXIT" } }
        TMENU { #0d #0d } PVIEW -1 WAIT       (Disp. menu)
        → key                                (Wait for keystroke)
        « CASE
            'key==11.1'
              THEN "Not used" DROP           (Increment day)
              END
            'key==11.2'
              THEN "Not used" DROP           (Decrement day)
              END
            'key==12.1'
              THEN                           (Increment month)
                IF 'm==12'
                THEN 1 'm' STO 'y' 1 STO+
                ELSE 'm' 1 STO+
                END
              END
            'key==12.2'
              THEN                           (Decrement month)
                IF 'm==1'
                THEN 12 'm' STO 'y' 1 STO-
                ELSE 'm' 1 STO-
                END
              END
            'key==13.1'
              THEN 'y' 1 STO+                (Increment year)
              END
```

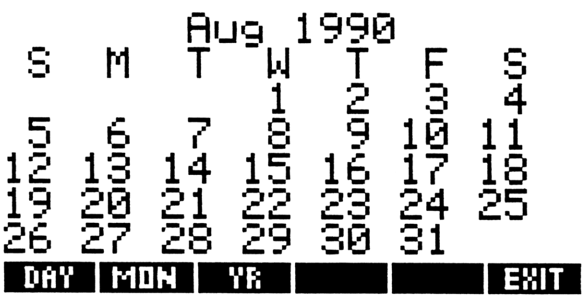
```
            'key==13.2'
              THEN 'y' 1 STO-         (Decrement year)
              END
            'key==16.1'
              THEN 0 DOERR            (Create exit condition)
              END
            1760 .1 BEEP   (Otherwise, beep—invalid key)
          END                                    (CASE)
        »
      UNTIL 0
      END
    THEN menu MENU
    END
  »
»
```

Checksum: # 29788d

Bytes: 828

	Stack Arguments	Stack Results
1:	(none)	(none)

Notes: CALEND displays a perpetual calendar in classic seven-column format. It uses the current system date to determine the first month displayed.



MYR

```
«
  «                                                  (Local function g)
    "  1  2  3  4  5  6  7  8  9 10 11 "             (Build a
    "12 13 14 15 16 17 18 19 20 21 22 "              week string)
    "23 24 25 26 27 28 29 30 31" + +
    ROT 3 * 2 - ROT 3 * 1 - SUB
  »
  «                                                  (Local function p)
    IF DUP TYPE 7 ==                                 (Display the week string)
    THEN INCR OVER SWAP DISP
    END
    DROP
  »
  RCLF 0 0 0 1 0 0
  → m y g p f d n i b e r
  « y 1000000 / m + .01 + DUP 'd' STO
    10.171582 SWAP DDAYS 7 MOD 'i' STO               (Day of week:
                                                     0=Sun, 6=Sat)
    IF m 12 ==                                       (Figure number of days in month…
    THEN 31                                          …where December is a special case)
    ELSE d DUP 1 + DDAYS
    END
    'n' STO CLLCD "       "                          (Month-year string—7 spaces)
    "JanFebMarAprMayJunJulAugSepOctNovDec"
    m 3 * DUP 2 - SWAP SUB + " " + STD y + 'r'
    p EVAL
    "  S  M  T  W  T  F  S"                          (Days-of-week header)
    IF n i + 35 ≤                                    (Leave it out if it doesn't fit)
    THEN 'r'
    END
    p EVAL 7 i - 'e' STO i 3 *
    "         " DUP + 1 ROT SUB                      (First row—9 spaces)
    b e g EVAL + 'r' p EVAL                          (Display first row)
    DO e 1 + 'b' STO e 7 + n MIN
       'e' STO                                       (Build subsequent rows)
       b e g EVAL 'r' p EVAL                         (Display subsequent rows)
```

```
    UNTIL e n ==
    END
    3 FREEZE f STOF
  »
»
```

Checksum: `# 61525d`
Bytes: `844.5`

	Stack Arguments	Stack Results
2:	*month (real number from 1 to 12)*	
1:	*year (real number ≥ 1582)*	(none)

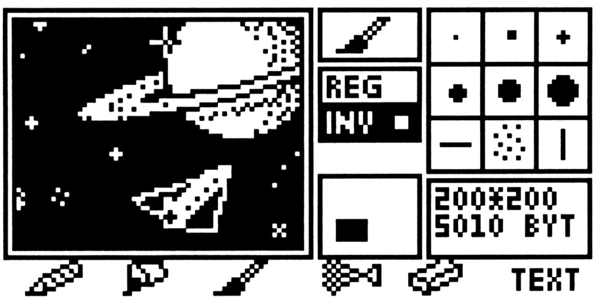
Notes: MYR draws the calendar for any given month and year (the earliest allowable month is November, 1582).

More Suggestions

Now that you've seen some working examples of 48 graphics, you may be speculating on the infinite possibilities. Here's a suggestion or two:

- The 48 has enough graphics power that you could come up with a great PAINT program or grob editor for it, with a display similar to the one shown below. At a menu line, the user would select from the available tools—and submenus would select different brush or fill patterns for each respective tool. A vertical menu on the right side could be used, via the arrow keys, for object/variable management or other purposes. Then the rest of the display would be a window into the grob, which could be scanned as needed. The current grob would not reside in PICT, but portions of it would be displayed in PICT when being edited.

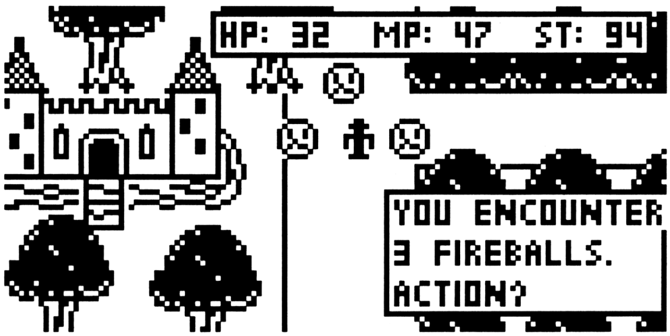
  PAINT would use KEY and WAIT to redefine the keyboard as appropriate. And note that several of the routines developed in this book could be incorporated into PAINT, too.



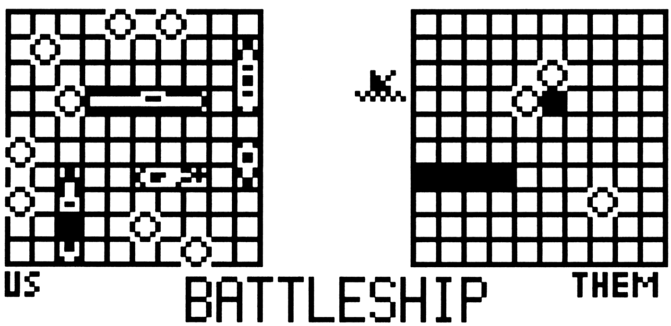
  The only drawbacks—as with all graphics routines—are memory use and speed. Consider those your challenges. After all, you're the judge as to what's acceptable and usable.

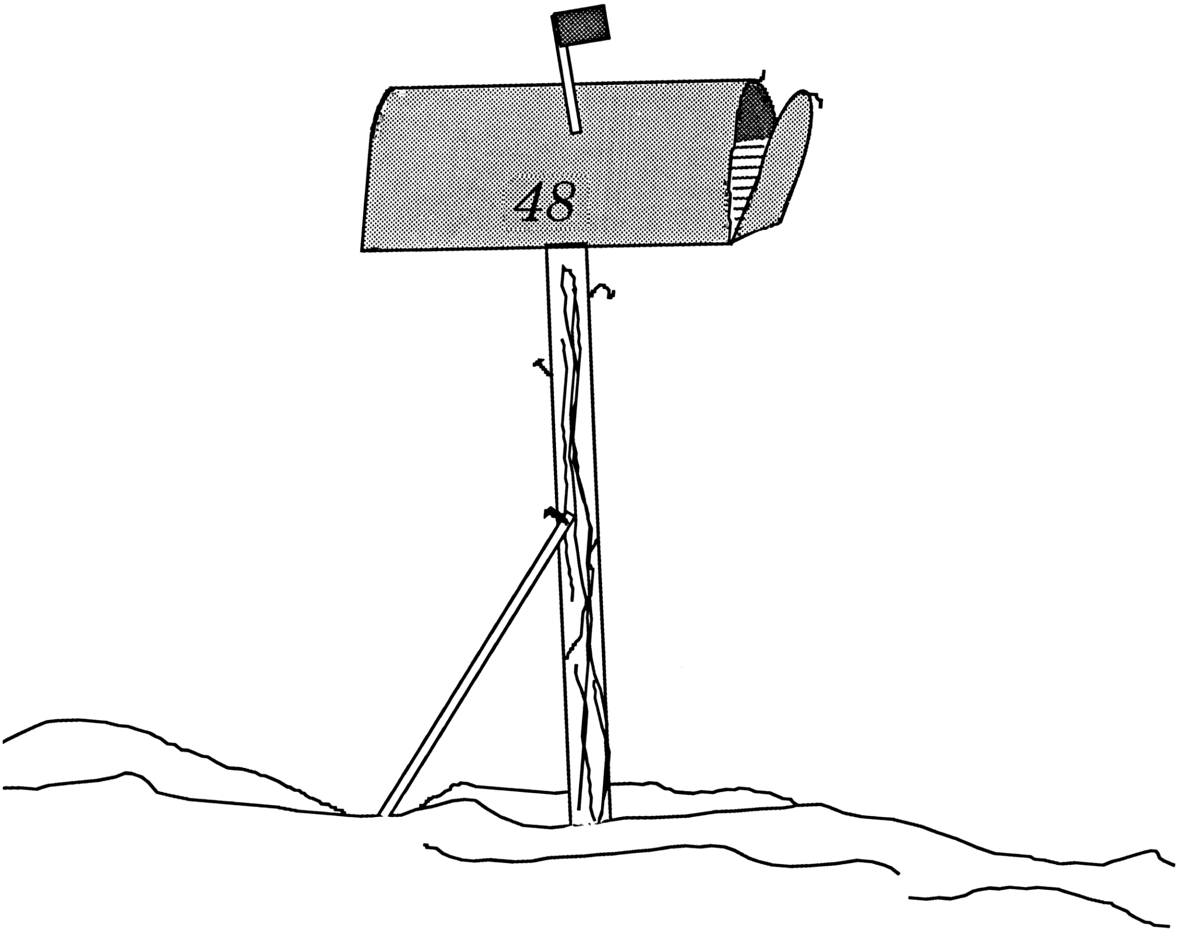
- Some of the most intriguing home video games are the role-playing adventure games, where the hero negotiates some large playing field, encountering monsters and other baddies.

  Such a game on the 48, for example, could use an intricately detailed 800×800 grob as the playing field, and dozens of little 8×8 grobs for the hero and the baddies. It wouldn't be hard.



- You've seen a checkers game. How about other familiar games (Battleship, Tetris, hangman, cards, etc.)? Your only limits are your imagination (and spare time).





10: Graphics Beyond the 48

(Or, "What's That Funny Hole in the Top of my Calculator?")

Of course, graphics on the 48 are nice in and of themselves, but their utility increases when you can transfer them to other machines.

Printing Graphics on the Infrared Printer

Although it is possible to send low-level graphics commands to the HP82240B infrared printer, it is faster and more efficient to use the built-in commands PR1 and PRVAR.

PR1 prints the grob in Stack Level 1. PRVAR prints the grob whose variable name appears in Level 1. To print more than one grob, you can use a list of variable names as the PRVAR argument. Note that PRVAR prefaces each object with a blank line and the variable name.

The HP 82240B printer can print only 166 dot columns. For a grob wider than 166 pixels, the printer will print the graphic in strips, with "cut here" dotted lines separating the strips, so you can paste them together later. You can avoid this problem if you have an Epson-compatible or PCL-compatible printer (keep reading...).

To print the text representation of a grob, (GROB *x y ddd...*), it's best to convert the grob to a string, a list or a program, and print it via PR1 (or, better yet, upload it to a personal computer and print it from there).

Printing Graphics on a Larger Printer

To print a graphic on a larger printer, you must translate the grob from 48 language into a language that the larger printer can understand. Recall from Chapter 4 that a grob is an object of the format

`GROB` *x* *y* *bbbbbb....*

where *x* and *y* are the width and height, respectively, in pixels, and *bbbbbb....* is a hexadecimal bitmap of the grob—in the 48's "reversed" notation.

Before you can print the grob, you must separate these three pieces of information for the printer. This program takes a grob from Stack Level 1 and separates the information into its three parts on the Stack:

DISSECT

```
« →STR DUP SIZE 6 SWAP SUB
  0 1
  FOR n
    DUP DUP " " POS SWAP OVER
    1 - 1 SWAP SUB OBJ→
    ROT ROT 1 + OVER SIZE SUB
  NEXT
»
```

Checksum: `# 48062d` Bytes: `102`

	Stack Arguments	Stack Results
3:		*x* (a real number)
2:		*y* (a real number)
1:	`GROB` *x* *y* *bbbbbb....*	*bbbbbb....*(a string)

Now, you'll also recall from the discussion in Chapter 4 (see page 97) that each nybble in the bitmap is presented with the bits reversed from the normal convention.

Here's a table that shows the translation between the 48 bitmap and a "right-reading" bitmap:

48 nybble hex value	bit pattern	reversed bit pattern	"right-reading" hex value
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	C
4	0100	0010	2
5	0101	1010	A
6	0110	0110	6
7	0111	1110	E
8	1000	0001	1
9	1001	1001	9
A	1010	0101	5
B	1011	1101	D
C	1100	0011	3
D	1101	1011	B
E	1110	0111	7
F	1111	1111	F

Notice the symmetry in the table: E translates to 7, and 7 translates to E, for example. Also, 0, 6, 9 and F translate into themselves, because their bit patterns are symmetrical.

From the translation table given above, you can assemble a string to represent the translated bitmap. The string is composed of the entries in the "right-reading" column of the table: "084C2A6E195D3B7F". Thus, in a program, translating a nybble becomes as simple as

```
« ... "0123456789ABCDEF"
  "084C2A6E195D3B7F"
  ROT POS DUP SUB ...
»
```

And you can build this sequence into a routine for translating bitmaps of any size. The following program will take a bitmap string from Stack Level 1 and replace it with a translated string:

TRANSLATE

```
« DUP SIZE
  → map len
  « 1 len
    FOR j
      "0123456789ABCDEF" "084C2A6E195D3B7F"
      map j j SUB POS DUP SUB
      map j ROT REPL 'map' STO
    NEXT
    map
  »
»
```

Checksum: # 58829d Bytes: 171.5

	Stack Arguments	Stack Results
1:	*bbbbbb....*(a string)	*bbbbbb....*(a string)

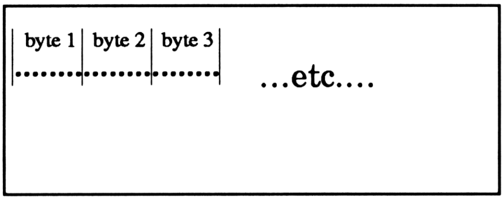
Note: To get your original string back again, just execute TRANSLATE a second time—the translation table is symmetrical.

Formatting Output for the Printer

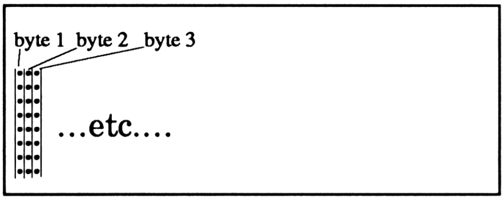
The most common printer protocols in use today are Epson and PCL. Most printers—including laser printers—offer Epson compatibility, either built-in or as an option. PCL is the Printer Control Language used by all HP printers, including the HP LaserJet and DeskJet. Most laser printers offer built-in PCL compatibility.

The main difference between the two protocols is that PCL uses *raster* graphics—receiving data in 8-dot *rows*—while Epson uses *column* graphics—receiving data in 8-dot *columns*:

PCL-Protocol Printers



Epson-Protocol Printers



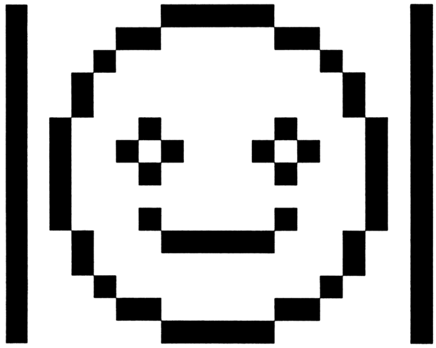
Each byte here represents 8 dots* of graphic output.

In PCL, each bit represents one dot in a *row*, with the least significant bit on the right. Bytes are sent to the printer as characters, so a *row* of four black dots followed by four white dots would have a character value of # 11110000b (that's # F0h or # 240d).

By contrast, in Epson, the least significant bit goes at the bottom of a *column* of bits. Bytes are sent to the printer as characters, so a *column* of four black dots atop four white dots would have a character value of # 11110000b (that's # F0h or # 240d).

*Dots are *printer* data, as opposed to pixels, which are *display* data.

So suppose you wanted to print this 19×15 graphics object:



On the 48, you would describe this object as

```
GROB 19 15
18F040 160340 110440 900840 900840
540150 5A8250 540150 500050 540150
98F840 900840 110440 160340 18F040
```

(rows are separated for clarity)

Running the bitmap string through `TRANSLATE` would then give you:

```
81F020 860C20 880220 900120 900120
A208A0 A514A0 A208A0 A000A0 A208A0
91F120 900120 880220 860C20 81F020
```

To successfully print the grob, a PCL printer would need to see a string of the form "x̄ ð ▶■ ...", where

x̄ is `CHR(129)` or `81h`
ð is `CHR(240)` or `F0h`
 is `CHR(32)` or `20h (<space>)`
▶ is `CHR(134)` or `86h`
■ is `CHR(12)` or `0Ch (<Form Feed>)`

As you can see, the PCL data string can be readily obtained directly from the TRANSLATE'd bitmap string (compare for yourself).

On the other hand, an Epson printer would expect to see a string of the form "Ÿ■■■ ..." where

Ÿ is `CHR(255)` or `FFh`
■ is `CHR(0)` or `00h (<NUL>)`
■ is `CHR(7)` or `07h (<BEL>)`
■ is `CHR(24)` or `18h (<CAN>)`
 is `CHR(32)` or `20h (<space>)`

This Epson string is not so easy to obtain from the TRANSLATE'd string. In fact, it's probably easier to write an Epson print program on the 48 which stores the grob in PICT and builds the Epson data string by testing individual pixels.

Printer Control Codes

When printing graphics, you must send control codes to the printer, warning it that the next batch of data it receives is graphics data instead of text. Otherwise, your printer will act unpredictably.

For PCL printers, use these commands, each sent as a string:*

"<ESC>*rA"	*(Start raster graphics)*
"<ESC>*b*n*W..."	*(Print the next "*n*" bytes as graphics data. For your 19×15 grob, you'd repeat this string 15 times—once for each row. The first part of the command, then, would be* "<ESC>*b3Wx̄ö <ESC>*b3W▶■..."
"<CR><LF>"	*(Print the buffer, advance to the next line and return to the left margin)*
"<ESC>*rB"	*(End raster graphics)*

These PCL control codes are for the HP ThinkJet, QuietJet, DeskJet and LaserJet printers, and any other printers which understand PCL.

Keep in mind that your display grobs printed at 300 dpi will become postage-stamp size. But on some printers, (for example, the DeskJet and LaserJet), you can select from different dot pitches. To change dot pitches in PCL printers, use these commands.

"<ESC>*t75R"	*Set dot pitch to 75 dpi—DeskJet or LaserJet only)*
"<ESC>*t100R"	*Set dot pitch to 100 dpi—DeskJet or LaserJet only)*
"<ESC>*t150R"	*Set dot pitch to 150 dpi—DeskJet or LaserJet only)*
"<ESC>*t300R"	*Set dot pitch to 300 dpi—DeskJet or LaserJet only)*
"<ESC>*t96R"	*Set dot pitch to 96 dpi—QuietJet only—default)*
"<ESC>*t192R"	*Set dot pitch to 192 dpi—QuietJet only)*

*<ESC> is CHR(27) ("Escape"); <CR> is CHR(13) ("Carriage Return"); <LF> is CHR(10) ("Line Feed").

For Epson printers, use these commands, each sent as a string:*

"<ESC>A8"	*(Set the line spacing to 8-dot rows)*
"<ESC>K*nm...*"	*(Print the next "*n+*(256×*m*)" bytes as graphics data. For the 48, usually you'll have less than 256 bytes per row, so* m=0*. In the example grob, you have 19 columns of data, so* n *will be* `CHR(19)`*; you have 15 rows of data, so you'll have to send such a string twice:* "<ESC>K■■Ÿ■■■ ..." and "<ESC>K■■..." *The first two* ■■ *in each string are* `CHR(19)` *and* `CHR(0)`*, respectively, and then the actual data commences—with* Ÿ■■■ ...*, for example, in the first string, as shown on page 277)*
"<CR><LF>"	*(Print the buffer, advance to the next line and return to the left margin)*
"<ESC>2"	*(Reset the line spacing to 6 lines per inch)*

These Epson control codes are for printers that print at 96 dpi in "single-density" mode (<ESC>K selects "single-density" printing). The codes will work with printers of other dot pitches, also—even with the 300-dpi Epson emulation on most laser printers. But as you know, at that resolution, your 131×64 display-sized grobs start looking like postage stamps. You'll need to modify your printing program to print a square of several dots for each pixel in your grob.

For more information on printer control codes, consult the owner's manual for your printer.

*<ESC> is `CHR(27)` ("Escape"); <CR> is `CHR(13)` ("Carriage Return"); <LF> is `CHR(10)` ("Line Feed").

The basic algorithm for a printer driver is as follows:

1. Clear system flag –33, to route non-printing I/O through the infrared port, and set system flag –34, to route printer output through the serial port.

2. **Epson:** Set the line spacing on your printer—typically 8 for most Epson printers. **PCL:** Set the dot pitch, if applicable; enable raster graphics.

3. **PCL:** Use the "translation string" to translate the grob data to a "right-reading" bitmap. **Epson:** Store the grob in PICT and extract data, 1 column of 8 pixels at a time.

4. Build the graphics data string for the first row of data. Preface it with the appropriate printer control code (see previous page).

5. Build data strings for all subsequent rows of data. Preface each string with the appropriate printer control code, and append them to the data string (for every case with the 48, the printer control codes will be identical).

6. Send the data string to the printer, making sure to end the line with a `<CR>` only. Note that on the 48, the `<CR><LF>` is automatic. But you can disable the `<LF>` by setting system flag –38, executing `0 TRANSIO`, and then storing a null string (`""`) in the fourth field of PRTPAR.

7. **Epson:** Reset the line spacing to 6 lines per inch. **PCL:** End or disable raster graphics; reset the dot pitch, if necessary.

8. Restore system flags, if necessary.

Avoiding Problems

Laser printers don't print to the paper until they receive a `<Form Feed>`, which is `CHR(12)`. If you're printing to a laser printer, you won't see any output until either the end of the page has been reached, or you send a `CHR(12)` to the printer.

However, if you store this program, FF, in your HOME directory, then you can send a `<Form Feed>` simply by executing FF, or by including it in any program:

```
FF:    « 12 CHR PR1 DROP
       »
```

Checksum: # 22456d Bytes: 34.5

It is strongly recommended that you use handshaking on both your printer and the 48. This gives the printer a chance to say "wait a minute, I'm busy" without either the 48 or the printer losing any data. You can select `XON/XOFF` handshaking on the 48 by setting the fourth parameter in the IOPAR reserved variable to 1 (for more information on using IOPAR, see chapter 27 of the User's Guide).

Two Sample Printing Programs

Combining all the above information into one place, you should be able to create a program to suit your needs and your printer. Use these two programs as examples.

PRGROB1

```
« DUP SIZE PICT RCLF                                  (Save defaults)
  STD                               (Select standard numeric notation)
  27 CHR "A8" +                                 (Set dot pitch to 8)
  27 CHR "K" +                            (Beginning of data string)
  27 CHR "2" +                          (Reset dot pitch to default)
  0                                    (Temporary storage variable)
  → gr x y pictx flags dp8 dat re t
  « gr PICT STO
    -33 CF -34 SF -38 SF   (IR I/O, serial printing, auto LF)
    dp8 PR1 DROP
    x B→R 256 MOD CHR dat
    OVER + 'dat' STO                   (Build <ESC>K to <ESC>Kn)
    x B→R SWAP NUM - 256 / CHR
    dat SWAP + 'dat' STO             (Build <ESC>Kn to <ESC>Knm)
    ""                                     (Initialize data string)
    0 y B→R 8 / CEIL
    FOR bigrow
      dat +                                  (Initialize line data)
      0 x B→R
      FOR col
        0 't' STO                          (Initialize column data)
        0 7
        FOR row                                   (Test each pixel)
          col R→B
          bigrow 8 * row + R→B
          2 →LIST PIX?                              (Returns 1 or 0)
          2 7 row - ^ * 't' STO+       (Increment col. data)
        NEXT                                            (Next row)
```

```
        t CHR +
      NEXT                                    (Next column)
    NEXT                                      (Next big row)
    PR1 DROP re PR1 DROP                      (Print grob, reset printer)
    pictx PICT STO flags STOF                 (Restore previous states)
  »
»
```

Checksum: # 61444d
Bytes: 549

	Stack Arguments	Stack Results
1:	GROB *x y bbbbbb....*	(none)

Notes: PRGROB1 prints a grob on an Epson-compatible printer, destroying PICT in the process.

PRGROB2

```
« DISSECT TRANSLATE                  (Get width, height and bitmap)
  RCLF                                        (Save previous states)
  STD                            (Select standard numeric notation)
  27 CHR "*t75R" +        (Set dot pitch to 75 dpi—96 for QuietJet)
  27 CHR "*rA" +                          (Begin raster graphics)
  27 CHR "*rB" +                            (End raster graphics)
  27 CHR "*b" +                         (Beginning of data string)
  0                                  (Temporary storage variable)
  → x y map flags dp75 begrg endrg dat t
  « -33 CF -34 SF -38 SF                              (IR I/O,...
    0 TRANSIO 'PRTPAR' DUP
    3 1000 PUT 4 "" PUT             ...serial printing, disable LF)
    endrg PR1 DROP              (Garbage collection on the printer)
    dp75 PR1 DROP                                  (Set dot pitch)
    begrg PR1 DROP                        (Begin raster graphics)
    map SIZE y /
    DUP 't' STO                       (Data string length per row)
    dat SWAP 2 / + "W"
    + 'dat' STO                        (Build <ESC>b to <ESC>bnW)
    ""                                    (Initialize data string)
    1 y
    FOR row
      dat +                                 (Initialize line data)
      row 1 - t * 1 + row t *
      FOR char
        map char
        DUP 1 + SUB                (Read bitmap for next 8 bits)
        "#" SWAP + "h" + OBJ→
        B→R CHR +                            (Add to data string)
        2
      STEP                                       (Next character)
    NEXT                                               (Next row)
    PR1 DROP endrg PR1 DROP (Prt. grob, end raster graphics)
    12 CHR PR1 DROP                         (Form feed—optional)
    flags STOF                           (Restore previous states)
  »
»
```

Checksum: # 23770d
Bytes: 595

	Stack Arguments	Stack Results
1:	GROB *x y bbbbbb....*	(none)

Notes: PRGROB2 prints a grob on a PCL-compatible printer.

The program assumes that PRTPAR already exists in the current directory.

The Hard Work's Already Done

Fortunately, HP has already provided print routines that do all this for you, in the form of two public-domain libraries called `EPSPRINT.LIB` and `PCLPRINT.LIB`.

These libraries are available on the HP 82208C Serial Interface Kit disk, or are downloadable from the HP Calculator Bulletin Board System (BBS). Instructions for using the libraries are located in two other files called `EPSPRINT.TXT` and `PCLPRINT.TXT`.*

Using EPSPRINT

Once installed, the EPSPRINT library appears in the Library menu as **EPPRT**. When selected, it shows this menu: **EPOFF** **EPON** **MAG**

Pressing **EPON** modifies PRTPAR and system flags –33 and –34 to send all printer output to an Epson-compatible printer over the serial interface, using `XON/XOFF` flow control. It uses a "hook" in the 48's operating system to activate the Epson graphics printer driver. Text is output in the printer's current font, and graphics is output at 60 dpi (you can modify PRTPAR to set it to 120 or 240 dpi, but 240 dpi is not recommended).

Pressing **EPOFF** returns PRTPAR and flags –33 and –34 to their turn-on states, allowing you to continue using the infrared printer. You may ignore **EPOFF** if you don't use an infrared printer.

*For more information on the HP BBS, contact HP Calculator Technical Support at (503) 757-2004, or look on the inside back cover of your User's Guide.

Pressing **MAG** with an argument of 1, 2 or 4 causes EPPRT to use the given magnification factor in printing graphics (the default is 2). For example, 4 **MAG** causes every pixel in the grob to be printed as a square, 4 dots × 4 dots.

All 48 printing commands *except* [ON]-[I/O] work normally with EPPRT. [ON]-[I/O] does unpredictable nasties with your printer and should not be used. Use PRLCD instead. Also, you can automate your Epson printing somewhat by storing these routines in your **HOME** directory:

```
EPR1:       « EPON PR1 EPOFF
            »
```

Checksum: # 6487d   Bytes: 32

```
EPRVAR:     « EPON PRVAR EPOFF
            »
```

Checksum: # 13016d   Bytes: 34

Using PCLPRINT

The PCLPRINT library appears in the Library menu as **HPPRT**. When you select it, you see this menu: **HPOFF** **HPON** **DPI** **MAG**

Similar to **EPON** in EPPRT, **HPON** also modifies PRTPAR and system flags –33 and –34, but it does so in order to send all printer output to a PCL-compatible printer over the serial interface, using XON/XOFF flow control. It, too, uses a "hook" in the HP-48's operating system to activate the PCL graphics printer driver. Text is output in the printer's current font.

HPOFF acts much like **EPOFF**, allowing you to continue using the infrared printer (and likewise, you may ignore **HPOFF** if you aren't using an infrared printer).

DPI takes an argument from Stack Level 1 and uses it to set the printer to the proper dot pitch. This could be 75, 150 or 300 dpi for a DeskJet or LaserJet (doesn't apply to other printers).

Unlike the **MAG** in EPPRT, the **MAG** in HPPRT can take any integer as an argument for the magnification factor. Entering *n* **MAG** causes every pixel in the grob to be printed as a square, *n* dots × *n* dots (no default is given, but it appears to be 1).

For a 300 dpi printer, 1 **MAG** will give you a postage-stamp sized image of a 131×64 grob. A grob printed at 2 **MAG** is about the same scale as an HP82240B printout, and a grob printed at 6 **MAG** is about the same scale as the 48's LCD display.

All 48 printing commands *except* [ON]-[I/O] work normally with HPPRT. [ON]-[I/O] has the same problems in HPPRT as in EPPRT.

However, when printing to a LaserJet series printer, note that the LaserJet prints to a *buffer*, not directly to the paper. The buffer is printed onto the paper either when the buffer is full, or when a form-feed character (ASCII `# 12d`) is sent to the printer. So if you're putting several graphics on one page, be sure to send a `CR` (there's a CR key in the PRINT menu) after each grob to provide some white space.

When you're ready to eject the page, you'll need to send a `<Form Feed>` character to the printer (you can use your FF program to do this).

Also, you can automate your PCL printing somewhat by storing the following two routines in your HOME directory.

```
HPR1:      « HPON PR1 HPOFF FF
           »
```

Checksum: # 32965d   Bytes: 37.5

```
HPRVAR:    « HPON PRVAR HPOFF FF
           »
```

Checksum: # 32180d   Bytes: 39.5

You may omit the FF's in these two routines if you're not using a LaserJet, or if you wish to put multiple printouts on one page.

Printing Graphics on a Pen Plotter

With the advent of high-resolution, wide-carriage, color dot-matrix printers, pen plotters seem to be disappearing quickly. Still, a pen plotter can be used as a graphics output device. The algorithm for a plotter driver is very simple—and fast, since pixels can be printed "on the fly," without waiting to build large graphics command strings.

The basic algorithm for a plotter driver is as follows:

1. Set the pen width and pixel spacing for the plotter—typically 0.3 mm or 0.65 mm.

2. ***Either*** use TRANSLATE to translate the grob's data to a "right-reading" bitmap, and then process the bitmap; ***or*** store the grob in PICT, and scan PICT, pixel by pixel.

3. With pen UP, scan the paper, row by row. At each pixel location, put the pen DOWN if the pixel is "dark" in that location, and draw a small square. Then put the pen UP again to resume scanning.

You may also wish to draw an outline box around your grob after it is completed.

Grobs and Other Computers

Since integrated text and graphics are taken for granted on computers these days, it would be nice to be able to include grobs in your computer work.

For example, if you're writing a lab report on your PC and have some important data stored in your 48, you can upload the numeric data to your computer, but you might also want to include the impressive graph you made on the 48 to avoid having to duplicate it in a spreadsheet.

Or suppose your report contains several long, involved equations like the ones in Chapter 3 in this book. Using the two-dimensional EW version is an easy way to get "textbook" notation in your report without having to buy the mathematics add-on for your word processor.

By virtue of their (admittedly) superior raw computing power, conversion of raw grobs to computer-format graphics is best done by the computers. DISSECT and TRANSLATE are trivial on a PC, but the grob-to-graphics conversion problem is complicated by the fact that there doesn't yet exist a standard computer graphics format.

Here, Hewlett-Packard comes to the rescue again. HP has developed programs called `GROB2TIF.EXE` and `TIF2GROB.EXE` for MS-DOS computers, `GROBer` for Macintosh computers, and GRAB48.EXE for MS Windows®.

`GROB2TIF.EXE` converts grobs to TIFF files, which can be used, or at least converted into something else, by the most popular word-processing and desktop-publishing programs. `TIF2GROB.EXE` converts TIFF files to grobs for use on the 48.

The `GROBer` allows you to convert grobs to Macintosh graphics for use with any Macintosh package, and to convert Macintosh graphics to grobs. Some of the finest 48 graphics to appear to date were taken from the Macintosh.

`GRAB48` turns your PC into a "virtual HP82240B printer," one that receives 82240B graphics commands and turns them into an image in MS Windows®. You can then print the image, save it in a variety of graphic file formats, or cut it and paste it into other Windows applications. If you have `GRAB48.EXE`, you may not need the HP82240B infrared printer, the `EPSPRINT.LIB` or `PCLPRINT.LIB` libraries, or the `GROB2TIF.EXE` utility—and `GRAB48.EXE` is free!

`GROB2TIF.EXE` is available on the HP82208C Serial Interface Kit disk for MS-DOS machines. The `GROBer` is available on the HP82209 Serial Interface Kit disk for Macintoshes. Both of these programs are also available from the HP Calculator BBS (see the footnote on page 286).

`TIF2GROB.EXE` and GRAB48.EXE are available only from the HP Calculator BBS.

Graphics Between Two 48's

It's hard to think of a serious use for two-machine graphics besides games or cool-looking demos, but some people take their games and their demos very seriously.

As you've seen with the CHKRS program, it is quite straightforward to create some two-player games on the 48, with two machines connected via IR or the serial port.

A well-behaved game program shows the board from the player's point of view and passes a token to keep track of whose move it was. A *skilled* game program checks for invalid moves (such as moving backwards in checkers) and allow for complex moves (such as double-jumping in checkers), and—of course—it would keep score.

Final Thoughts

This book is only the beginning. It has shown you just a few of the great graphics tricks the 48 can do, and how you can use these graphics tricks to your advantage. And in the process, hopefully, you've become more comfortable with the machine, by working through the exercises and trying the applications (and maybe you also have a better idea of how to use the EquationWriter, the Solvers and the Plotter).

All that remains is for you to find real uses for these tools—applications in your job, studies or hobbies. As you use the 48, you will undoubtedly become more skilled with it and thus it will become the more useful to you in return. Again, remember what your high school band teacher told you:

"Proficiency comes through practice."

Above all, have fun!

Famous Oatmeal Cookies*

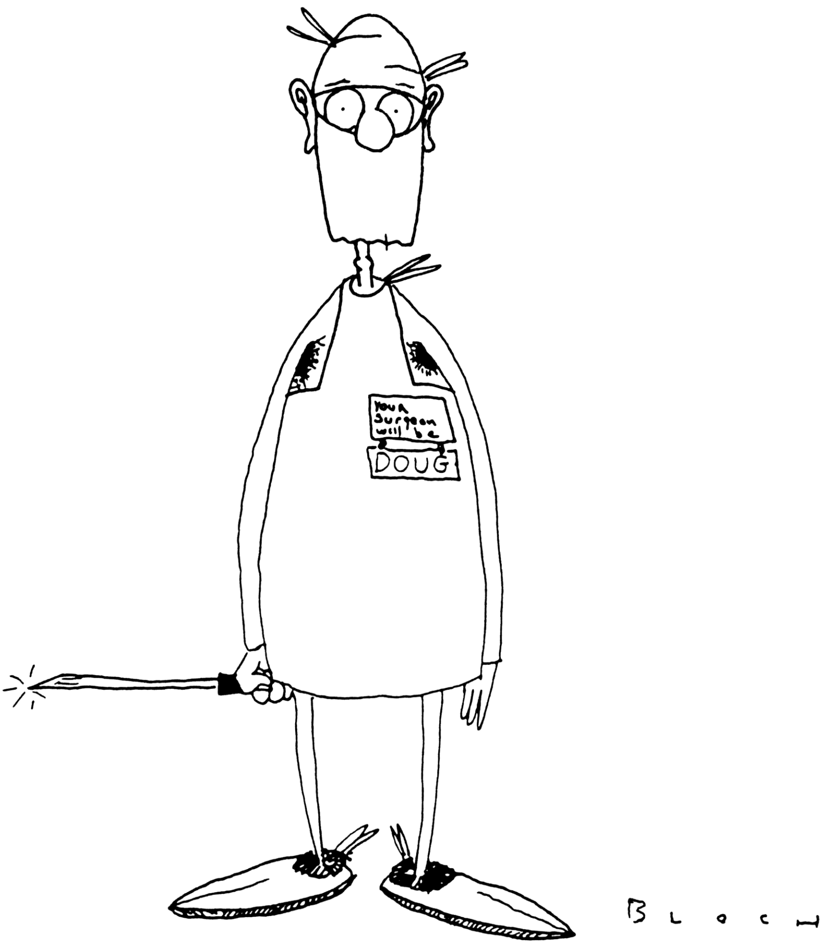
3/4 Cup vegetable shortening
1 Cup firmly packed brown sugar
1/2 Cup granulated sugar
1 egg
1/4 Cup water
1 teaspoon vanilla
3 Cups rolled oats, uncooked
1 Cup all-purpose flour
1 teaspoon salt (optional)
1/2 teaspoon baking soda

Preheat oven to 350°F. Beat together shortening, sugars, egg, water and vanilla until creamy. Add combined remaining ingredients; mix well.

(Finally, Valerie says to fold in 1 Cup of semi-sweet chocolate chips.)

Drop by rounded teaspoonfuls onto greased cookie sheet. Bake at 350°F for 12 to 15 minutes.

* Recipe courtesy of the Quaker Oats company.



APPENDICES

A: Review of the Hexadecimal Number System

"Hexadecimal" is a word derived from the Latin roots for *six* ("hexa-") and *ten* ("decimal"). It is a form of expressing numbers in base sixteen. "Hexadecimal" is often abbreviated to "hex."

The Decimal System as an Example of Counting Systems

Most human beings count in the *decimal*, or base-ten, number system (though you may have heard also of the *binary*, or base-two, number system). In base ten, you use the numerals from 0 to 9. To count past nine, you need some way to indicate the overflow, so you use a second digit—the "tens" digit—to count the "number of overflows." Likewise, when you run out of digits to express the "overflows," you add a third digit—a "hundreds" digit—to count the "overflows of overflows." And so on, until you have enough digits to express any given number.

So, proceeding from right to left, the first digit represents the number of "ones," or 10^0, in the number; the second digit represents the number of whole sets of ten (10^1); the third digit represents the number of whole sets of a hundred (10^2), etc. Thus, the *n*th digit represents the number of whole sets of 10^{n-1} in the number.

So you could think of the number 3401 as:

$$3\times10^3 + 4\times10^2 + 0\times10^1 + 1\times10^0$$

Significant Digits

Obviously, changing the leftmost digit in the number has a greater effect on the number than changing the rightmost digit. That is, the leftmost digit is the *most significant digit*; and the rightmost digit is the *least significant digit*. For example, if you see a house selling for $93,499 and one selling for $93,500, you'd say they both cost the same. One dollar isn't very significant compared to ninety thousand dollars.

The right-to-left order of increasing significance is a convention used in other place-value numbering systems, including binary and hexadecimal.

Hexadecimal Values

Computers count in binary, using only the numerals 0 and 1. That's difficult for humans to comprehend and uses a lot of space in displays and printouts. A more convenient way to organize binary data is to group the binary digits (*bits*) together in groups of four, and assign each group a single value.

Look at the table on the opposite page. You'll see that a group of four bits can range from 0000, with a value of zero, to 1111, with a value of fifteen. That's sixteen values, which is why sixteen—hexadecimal—is such a convenient number base to use when working with computers.

Of course, when expressing number values, you have only ten conventional Arabic numerals (0-9). But when counting in hexadecimal, you must go all the way to fifteen before adding a second numeral as a "counter of overflows." So the *letters* A-F are used as numerals to represent the values ten through fifteen in hexadecimal.

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

In the 48, integer objects can be expressed as binary, decimal, hex or octal (base eight). The # sign before the number means that it's an integer, and the b/d/h/o suffix indicates its number base. You can convert these integer number formats from one base to another using the 48's MTH BASE menu, or use the following table (for the corresponding 48 display characters, use → CHARS, or see page 2-5 in the UG.):

Binary	Decimal	Hex.
# 00000000b	# 000d	# 00h
# 00000001b	# 001d	# 01h
# 00000010b	# 002d	# 02h
# 00000011b	# 003d	# 03h
# 00000100b	# 004d	# 04h
# 00000101b	# 005d	# 05h
# 00000110b	# 006d	# 06h
# 00000111b	# 007d	# 07h
# 00001000b	# 008d	# 08h
# 00001001b	# 009d	# 09h
# 00001010b	# 010d	# 0Ah
# 00001011b	# 011d	# 0Bh
# 00001100b	# 012d	# 0Ch
# 00001101b	# 013d	# 0Dh
# 00001110b	# 014d	# 0Eh
# 00001111b	# 015d	# 0Fh
# 00010000b	# 016d	# 10h
# 00010001b	# 017d	# 11h
# 00010010b	# 018d	# 12h
# 00010011b	# 019d	# 13h
# 00010100b	# 020d	# 14h
# 00010101b	# 021d	# 15h
# 00010110b	# 022d	# 16h
# 00010111b	# 023d	# 17h
# 00011000b	# 024d	# 18h
# 00011001b	# 025d	# 19h
# 00011010b	# 026d	# 1Ah
# 00011011b	# 027d	# 1Bh
# 00011100b	# 028d	# 1Ch
# 00011101b	# 029d	# 1Dh
# 00011110b	# 030d	# 1Eh
# 00011111b	# 031d	# 1Fh

Binary	Decimal	Hex.
# 00100000b	# 032d	# 20h
# 00100001b	# 033d	# 21h
# 00100010b	# 034d	# 22h
# 00100011b	# 035d	# 23h
# 00100100b	# 036d	# 24h
# 00100101b	# 037d	# 25h
# 00100110b	# 038d	# 26h
# 00100111b	# 039d	# 27h
# 00101000b	# 040d	# 28h
# 00101001b	# 041d	# 29h
# 00101010b	# 042d	# 2Ah
# 00101011b	# 043d	# 2Bh
# 00101100b	# 044d	# 2Ch
# 00101101b	# 045d	# 2Dh
# 00101110b	# 046d	# 2Eh
# 00101111b	# 047d	# 2Fh
# 00110000b	# 048d	# 30h
# 00110001b	# 049d	# 31h
# 00110010b	# 050d	# 32h
# 00110011b	# 051d	# 33h
# 00110100b	# 052d	# 34h
# 00110101b	# 053d	# 35h
# 00110110b	# 054d	# 36h
# 00110111b	# 055d	# 37h
# 00111000b	# 056d	# 38h
# 00111001b	# 057d	# 39h
# 00111010b	# 058d	# 3Ah
# 00111011b	# 059d	# 3Bh
# 00111100b	# 060d	# 3Ch
# 00111101b	# 061d	# 3Dh
# 00111110b	# 062d	# 3Eh
# 00111111b	# 063d	# 3Fh

Binary	Decimal	Hex.
# 01000000b	# 064d	# 40h
# 01000001b	# 065d	# 41h
# 01000010b	# 066d	# 42h
# 01000011b	# 067d	# 43h
# 01000100b	# 068d	# 44h
# 01000101b	# 069d	# 45h
# 01000110b	# 070d	# 46h
# 01000111b	# 071d	# 47h
# 01001000b	# 072d	# 48h
# 01001001b	# 073d	# 49h
# 01001010b	# 074d	# 4Ah
# 01001011b	# 075d	# 4Bh
# 01001100b	# 076d	# 4Ch
# 01001101b	# 077d	# 4Dh
# 01001110b	# 078d	# 4Eh
# 01001111b	# 079d	# 4Fh
# 01010000b	# 080d	# 50h
# 01010001b	# 081d	# 51h
# 01010010b	# 082d	# 52h
# 01010011b	# 083d	# 53h
# 01010100b	# 084d	# 54h
# 01010101b	# 085d	# 55h
# 01010110b	# 086d	# 56h
# 01010111b	# 087d	# 57h
# 01011000b	# 088d	# 58h
# 01011001b	# 089d	# 59h
# 01011010b	# 090d	# 5Ah
# 01011011b	# 091d	# 5Bh
# 01011100b	# 092d	# 5Ch
# 01011101b	# 093d	# 5Dh
# 01011110b	# 094d	# 5Eh
# 01011111b	# 095d	# 5Fh

Binary	Decimal	Hex.
# 01100000b	# 096d	# 60h
# 01100001b	# 097d	# 61h
# 01100010b	# 098d	# 62h
# 01100011b	# 099d	# 63h
# 01100100b	# 100d	# 64h
# 01100101b	# 101d	# 65h
# 01100110b	# 102d	# 66h
# 01100111b	# 103d	# 67h
# 01101000b	# 104d	# 68h
# 01101001b	# 105d	# 69h
# 01101010b	# 106d	# 6Ah
# 01101011b	# 107d	# 6Bh
# 01101100b	# 108d	# 6Ch
# 01101101b	# 109d	# 6Dh
# 01101110b	# 110d	# 6Eh
# 01101111b	# 111d	# 6Fh
# 01110000b	# 112d	# 70h
# 01110001b	# 113d	# 71h
# 01110010b	# 114d	# 72h
# 01110011b	# 115d	# 73h
# 01110100b	# 116d	# 74h
# 01110101b	# 117d	# 75h
# 01110110b	# 118d	# 76h
# 01110111b	# 119d	# 77h
# 01111000b	# 120d	# 78h
# 01111001b	# 121d	# 79h
# 01111010b	# 122d	# 7Ah
# 01111011b	# 123d	# 7Bh
# 01111100b	# 124d	# 7Ch
# 01111101b	# 125d	# 7Dh
# 01111110b	# 126d	# 7Eh
# 01111111b	# 127d	# 7Fh

Binary	Decimal	Hex.	Binary	Decimal	Hex.
# 10000000b	# 128d	# 80h	# 10100000b	# 160d	# A0h
# 10000001b	# 129d	# 81h	# 10100001b	# 161d	# A1h
# 10000010b	# 130d	# 82h	# 10100010b	# 162d	# A2h
# 10000011b	# 131d	# 83h	# 10100011b	# 163d	# A3h
# 10000100b	# 132d	# 84h	# 10100100b	# 164d	# A4h
# 10000101b	# 133d	# 85h	# 10100101b	# 165d	# A5h
# 10000110b	# 134d	# 86h	# 10100110b	# 166d	# A6h
# 10000111b	# 135d	# 87h	# 10100111b	# 167d	# A7h
# 10001000b	# 136d	# 88h	# 10101000b	# 168d	# A8h
# 10001001b	# 137d	# 89h	# 10101001b	# 169d	# A9h
# 10001010b	# 138d	# 8Ah	# 10101010b	# 170d	# AAh
# 10001011b	# 139d	# 8Bh	# 10101011b	# 171d	# ABh
# 10001100b	# 140d	# 8Ch	# 10101100b	# 172d	# ACh
# 10001101b	# 141d	# 8Dh	# 10101101b	# 173d	# ADh
# 10001110b	# 142d	# 8Eh	# 10101110b	# 174d	# AEh
# 10001111b	# 143d	# 8Fh	# 10101111b	# 175d	# AFh
# 10010000b	# 144d	# 90h	# 10110000b	# 176d	# B0h
# 10010001b	# 145d	# 91h	# 10110001b	# 177d	# B1h
# 10010010b	# 146d	# 92h	# 10110010b	# 178d	# B2h
# 10010011b	# 147d	# 93h	# 10110011b	# 179d	# B3h
# 10010100b	# 148d	# 94h	# 10110100b	# 180d	# B4h
# 10010101b	# 149d	# 95h	# 10110101b	# 181d	# B5h
# 10010110b	# 150d	# 96h	# 10110110b	# 182d	# B6h
# 10010111b	# 151d	# 97h	# 10110111b	# 183d	# B7h
# 10011000b	# 152d	# 98h	# 10111000b	# 184d	# B8h
# 10011001b	# 153d	# 99h	# 10111001b	# 185d	# B9h
# 10011010b	# 154d	# 9Ah	# 10111010b	# 186d	# BAh
# 10011011b	# 155d	# 9Bh	# 10111011b	# 187d	# BBh
# 10011100b	# 156d	# 9Ch	# 10111100b	# 188d	# BCh
# 10011101b	# 157d	# 9Dh	# 10111101b	# 189d	# BDh
# 10011110b	# 158d	# 9Eh	# 10111110b	# 190d	# BEh
# 10011111b	# 159d	# 9Fh	# 10111111b	# 191d	# BFh

Binary	Decimal	Hex.
# 11000000b	# 192d	# C0h
# 11000001b	# 193d	# C1h
# 11000010b	# 194d	# C2h
# 11000011b	# 195d	# C3h
# 11000100b	# 196d	# C4h
# 11000101b	# 197d	# C5h
# 11000110b	# 196d	# C6h
# 11000111b	# 199d	# C7h
# 11001000b	# 200d	# C8h
# 11001001b	# 201d	# C9h
# 11001010b	# 202d	# CAh
# 11001011b	# 203d	# CBh
# 11001100b	# 204d	# CCh
# 11001101b	# 205d	# CDh
# 11001110b	# 206d	# CEh
# 11001111b	# 207d	# CFh
# 11010000b	# 208d	# D0h
# 11010001b	# 209d	# D1h
# 11010010b	# 210d	# D2h
# 11010011b	# 211d	# D3h
# 11010100b	# 212d	# D4h
# 11010101b	# 213d	# D5h
# 11010110b	# 214d	# D6h
# 11010111b	# 215d	# D7h
# 11011000b	# 216d	# D8h
# 11011001b	# 217d	# D9h
# 11011010b	# 218d	# DAh
# 11011011b	# 219d	# DBh
# 11011100b	# 220d	# DCh
# 11011101b	# 221d	# DDh
# 11011110b	# 222d	# DEh
# 11011111b	# 223d	# DFh

Binary	Decimal	Hex.
# 11100000b	# 224d	# E0h
# 11100001b	# 225d	# E1h
# 11100010b	# 226d	# E2h
# 11100011b	# 227d	# E3h
# 11100100b	# 228d	# E4h
# 11100101b	# 229d	# E5h
# 11100110b	# 230d	# E6h
# 11100111b	# 231d	# E7h
# 11101000b	# 232d	# E8h
# 11101001b	# 233d	# E9h
# 11101010b	# 234d	# EAh
# 11101011b	# 235d	# EBh
# 11101100b	# 236d	# ECh
# 11101101b	# 237d	# EDh
# 11101110b	# 238d	# EEh
# 11101111b	# 239d	# EFh
# 11110000b	# 240d	# F0h
# 11110001b	# 241d	# F1h
# 11110010b	# 242d	# F2h
# 11110011b	# 243d	# F3h
# 11110100b	# 244d	# F4h
# 11110101b	# 245d	# F5h
# 11110110b	# 246d	# F6h
# 11110111b	# 247d	# F7h
# 11111000b	# 248d	# F8h
# 11111001b	# 249d	# F9h
# 11111010b	# 250d	# FAh
# 11111011b	# 251d	# FBh
# 11111100b	# 252d	# FCh
# 11111101b	# 253d	# FDh
# 11111110b	# 254d	# FEh
# 11111111b	# 255d	# FFh

B: Graphics Operations and Commands

Setting/Checking Graphics Parameters

Operation (Interactive)	Command (Programmable)	Description
PRG LIST ELEM SIZE PRG GROB NXT SIZE	SIZE	Returns the height and width of the grob, in pixel units (page 107).
↰ PLOT NXT 3D VPAR NXT RESET ↰ PLOT PPAR RESET ↱ PLOT NXT RESET ▼ OK	« PPAR' PURGE PICT PURGE PICT DROP »	Resets plot parameters to defaults (page 112).
↰ PLOT PPAR INDEP ↱ PLOT OPTS	INDEP	Specifies independent variable (page 114).
	« PPAR 3 GET »	Recalls independent variable (page 114).
↱ PLOT DEPN ↰ PLOT PPAR DEPN	DEPND	Specifies dependent variable (page 114).
	« PPAR 7 GET »	Recalls the dependent variable (page 114).
↰ PLOT PPAR RES ↱ PLOT OPTS ▲	RES	Specifies the plot resolution (page 117).
	« PPAR 4 GET »	Recalls plot resolution (page 117).
↰ PLOT NXT FLAG CNCT ↱ PLOT OPTS ▼ ✓CHK	« -31 CF »	Enables curve filling (page 118).

Operation (Interactive)	Command (Programmable)	Description
⬅ PLOT NXT FLAG CNC■ ➡ PLOT OPTS ▼ ✓CHK	« -31 SF »	Disables the curve filling (page 118).
⬅ PLOT PPAR NXT AXES	AXES	Specifies intersection of axes (pp. 108, 116).
	« PPAR 5 GET »	Recalls intersection of axes (pp. 108, 116).
⬅ PLOT PPAR NXT CENT	CENTR	Specifies the center of PICT (page 115).
	« PPAR OBJ→ 6 DROPN DUP2 - 2 / DUP RE - PICT SIZE SWAP DROP B→R 1 - / ROT ROT + 2 / + »	Recalls center of PICT (page 115).
⬅ PLOT PPAR NXT SCALE	SCALE	Sets the x and y plotting scales (page 115).
	« PPAR OBJ→ 6 DROPN SWAP - 10 * C→R PICT SIZE 1 - B→R ROT SWAP / ROT ROT B→R 1 - / SWAP »	Recalls x and y plotting scales (page 115).
⬅ PLOT PPAR XRNG ➡ PLOT ▼ ▶	XRNG	Sets x-range (page 115).
	« PPAR 1 2 SUB RE EVAL »	Recalls x-range (p. 115).
⬅ PLOT PPAR YRNG ➡ PLOT ▼ ▼ ▶	YRNG	Sets y-range (page 115).

Operation (Interactive)	Command (Programmable)	Description
	« PPAR 1 2 SUB IM EVAL »	Recalls y-axis range (page 115).
	PMIN	Sets PMIN (page 116).
	« PPAR 1 GET »	Recalls PMIN (page 116).
	PMAX	Sets PMAX (page 116).
	« PPAR 2 GET »	Recalls PMAX (page 116).
PRG PICT PDIM	PDIM	Changes PICT size or user units (page 123).
← PLOT PPAR NXT *W	*W	Changes x-rng. (p. 117).
← PLOT PPAR NXT *H	*H	Changes y-rng. (p. 117).

Creation/Manipulation of Grobs

(PICTURE) STO (PICTURE) EDIT NXT NXT PICT→ PRG PICT PICT → RCL	« PICT RCL »	Puts PICT onto Stack (pages 95, 119).
(EW) STO	« 0 →GROB »	Turns equation into a grob (pages 95, 119).
PRG GROB NXT LCD→	LCD→	Turns Stack display into a grob (a "snapshot") (pages 95, 119).

Operation (Interactive)	Command (Programmable)	Description
PRG GROB →GRO	→GROB	Turns any object into a grob (pages 95, 119).
PRG GROB BLAN	BLANK « GROB x y 0 »	Creates a blank grob (pages 95, 119).
PRG GROB GOR	GOR	Superimposes one grob upon another, OR'ing pixels (page 120).
PRG GROB GXOR	GXOR	Superimposes one grob upon another, XOR'ing pixels (page 120).
PRG GROB REPL PRG LIST REPL (PICTURE) EDIT NXT NXT REPL	REPL	Superimposes one grob upon another, replacing target grob pixels (pages 120, 121).
PRG GROB SUB PRG LIST SUB (PICTURE) EDIT NXT NXT SUB	SUB	Creates subgrob from parent grob (pages 120, 121).
(PICTURE) EDIT NXT DEL (PICTURE) DEL		Erases ("blanks out") part of grob (page 121).
← PLOT (*or* → PLOT) ERASE (PICTURE) ← CLEAR (PICTURE) EDIT NXT ERASE	ERASE	Erases (blanks out) all of PICT (page 112).
(Stack) +	+	Adds (GOR's) two grobs of same size (page 126).
(Stack) +/−	NEG	Inverts a grob, toggling each pixel (page 126).

Accessing, Viewing/Displaying Grobs

Operation (Interactive)	Command (Programmable)	Description
(Stack) [◀] (Stack/CL) [←][PICTURE]	PICTURE *or* GRAPH	Enters graphics environment (page 105).
[←][PLOT] DRAW [→][PLOT] DRAW	DRAW	(Draws all or some of PICT (pages 95, 112).
(PICTURE) [←][◀] (EW) [←][◀]	« { } PVIEW »	Enters scrolling mode (pages 32, 106).
(Scrolling) [◀],[▲],[▼],[▶]		Scrolls through grob. (pages 106, 129-130).
(Scrolling) [→][◀],[→][▲],[→][▼],[→][▶]		Jumps to edge of display or grob (pages 106, 129-130).
(Scrolling) [←][◀]		Exits scrolling mode to EW or graphics (pages 106, 129-130).
(Scrolling) [CANCEL]		Exits scrolling mode to EW or Stack (page 130).
[PRG] OUT TEXT (PICTURE) [CANCEL]	TEXT	Exits graphics environment (pages 105, 125).
[PRG] OUT PVIEW	PVIEW	Views selected portions of PICT (page 106).
[PRG] GROB [NXT] →LCD	→LCD	Displays grob in Stack display (page 100).
[PRG] GROB [NXT] ANIM	ANIMATE	Displays grob sequence (pp. 121-122, 150-152).

Editing/Drawing on Grobs

Operation (Interactive)	Command (Programmable)	Description
⮐ PLOT AUTO ⮑ PLOT ▼ ▼ ◀ ✓CHK	AUTO	Automatically rescales y-axis prior to DRAW (page 113).
⮐ PLOT DRAW ⮑ PLOT DRAW	DRAW	Plots a curve in PICT. When used in a program, DRAW does not erase PICT or draw axes (pages 95, 112).
⮐ PLOT DRAX	DRAX	Draws the x- and y- axes (page 112).
⮐ PLOT LABEL (PICTURE) EDIT NXT LABEL	LABEL	Labels x- and y- axes (or PICT boundaries), using current number format (page 112).
PRG PICT NXT PX→C	PX→C	Converts pixel coordinates into user units (page 123).
PRG PICT NXT C→PX	C→PX	Converts user units into pixel coordinates (page 123).
PRG PICT BOX (PICTURE) EDIT BOX	BOX	Draws a box in PICT (page 123).

Operation (Interactive)	Command (Programmable)	Description
PRG PICT LINE (PICTURE) EDIT LINE	LINE	Draws a line in PICT (page 123).
PRG PICT TLINE (PICTURE) EDIT TLINE	TLINE	Draws a line in PICT, toggling pixels (page 123).
PRG PICT ARC (PICTURE) EDIT CIRCL	ARC	Draws a circle or arc in PICT. CIRCL isn't programmable; use a 360° arc (page 124).
PRG PICT NXT PIXON (PICTURE) EDIT DOT+	PIXON	Turns a pixel on (page 124).
PRG PICT NXT PIXOF (PICTURE) EDIT DOT-	PIXOFF	Turns a pixel off (page 124).
PRG PICT NXT PIX?	PIX?	Tests pixel status: 1 = on 0 = off (page 124).

Printing Graphics

Operation (Interactive)	Command (Programmable)	Description
⬅ I/O PR1 ⬅ I/O PRINT NXT PR1	PR1	Prints grob in Level 1, in graphics mode (page 271).
⬅ I/O PRINT PRVAR	PRVAR	Prints grob(s) named in Level 1, in graphics mode (page 271).
⬅ I/O PRINT PRST ⬅ I/O PRINT PRSTC	PRST PRSTC	Prints the contents of Stack—grobs in compact mode: `Graphics` *n*×*m*

	`« ... →STR PR1 ... »`
	
	`« ... 1 →LIST PR1 ... »`
		Prints a grob in text mode. Note that a list uses less memory than a string.

⬅ I/O PRINT PRLCD ON-I/O	PRLCD	Prints display. **Note: Do not use** ON-I/O with `EPSPRINT.LIB` or `PCLPRINT.LIB` (pages 287, 289).

Miscellaneous Graphics Commands

Operation (Interactive)	Command (Programmable)	Description
(PRG) PICT PICT	PICT	Specifies the current graphics object. Use « ...PICT RCL... » to put contents onto the Stack (page 123).
(PRG) OUT CLLCD	CLLCD	Clears (blanks out) the display (page 125).
(PRG) OUT DISP	DISP	Displays a line of text (page 125).
(PRG) OUT FREEZ	FREEZE	Freezes all or part of the display until next keystroke (page 125).
(PRG) OUT MSGB	MSGBOX	Displays a message box (page 125).
(←)(PLOT) PTYPE (←)(PLOT)(NXT) 3D PTYPE (→)(PLOT)(▲) CHOOS		Offers a selection of plot types; the PTYPE menu selections are programmable (page 114).
(PICTURE) (ENTER) (PICTURE) EDIT (NXT)(NXT) X,Y→		Returns cursor coordinates to Stack (p. 165).
(PICTURE) (+) (PICTURE) (X,Y)		Displays cursor coordinates in user units. (+), (−) or any menu key will restore menu (page 173).

Operation (Interactive)	Command (Programmable)	Description
(PICTURE) [−] (PICTURE) EDIT [NXT] MENU		Hides/restores Graphics menu. [−] or any menu key restores the menu.
(PICTURE) [×] (PICTURE) EDIT [NXT] MARK		Marks current cursor location for BOX, LINE, etc.
(PICTURE) [+/−] (PICTURE) EDIT [NXT] +/−		Toggles cursor style—overwrite vs. invert.
(PICTURE) FCN		Menu of graphic Solver functions (page 71).
[←] [STAT] PLOT BARPL [←] [STAT] PLOT HISTP [←] [STAT] PLOT SCATR	BARPLOT HISTPLOT SCATRPLOT	Generates statistical plots. Refer to the HP User's Guide (UG), chapter 21, "Statistics," for more information.

Setting/Checking 3-D Graphics Parameters

Operation (Interactive)	Command (Programmable)	Description
⬑ PLOT NXT 3D VPAR XVOL ⮫ PLOT OPTS	XVOL	Sets x-range of view volume (page 138).
	« VPAR 1 2 SUB EVAL »	Recalls x-range of view volume (page 138).
⬑ PLOT NXT 3D VPAR YVOL ⮫ PLOT OPTS ▼	YVOL	Sets y-range of view volume (page 138).
	« VPAR 3 4 SUB EVAL »	Recalls y-range of view volume (page 138).
⬑ PLOT NXT 3D VPAR ZVOL ⮫ PLOT OPTS ▼ ▼	ZVOL	Sets z-range of view volume, for WIREFRAME, YSLICE and PARSURFACE only (page 138).
	« VPAR 5 6 SUB EVAL »	Recalls z-range of the view volume, for WIREFRAME, YSLICE and PARSURFACE only (page 138).
⬑ PLOT NXT 3D VPAR XXRN ⮫ PLOT OPTS ▼ ▼	XXRNG	Sets x-range of sampling grid, GRIDMAP and PARSURFACE only (page 139).
	« VPAR 7 8 SUB EVAL »	Recalls x-range of sampling grid, GRIDMAP and PARSURFACE only (page 139).

Operation (Interactive)	Command (Programmable)	Description
⬅ PLOT NXT 3D VPAR YYRN ➡ PLOT OPTS ▼ ▼ ▼ ➡ PLOT XX.YY	YYRNG	Sets *y*-range of sampling grid, GRIDMAP and PARSURFACE only (page 139).
	`« VPAR 9 10 SUB EVAL »`	Recalls *y*-range of sampling grid, GRIDMAP and PARSURFACE only (page 139).
⬅ PLOT NXT 3D VPAR NXT EYEPT ➡ PLOT OPTS ▼ ▼ ▼	EYEPT	Sets *x*-, *y*- and *z*- coordinates of eyepoint, for WIREFRAME and PARSURFACE only (p. 139).
	`« VPAR 11 13 SUB EVAL »`	Recalls *x*-, *y*- and *z*- coordinates of eyepoint, WIREFRAME and PARSURFACE only (p. 139).
⬅ PLOT NXT 3D VPAR NXT NUMX ➡ PLOT ▼ ▶	NUMX	Sets number of *x* intervals to be plotted (page 139).
	`« VPAR 14 GET »`	Recalls the number of *x* intervals to be plotted (page 139).
⬅ PLOT NXT 3D VPAR NXT NUMY ➡ PLOT ▼ ▼ ▶	NUMY	Sets number of *y* intervals to be plotted (page 139).
	`« VPAR 15 GET »`	Recalls the number of *y* intervals to be plotted (page 139).

C: User-Named Objects

Alphabetically (objects named by *other objects* are also listed here, among the References)

Name	PATH	References	Page
ADDB	{ HOME TOOLS }		159
AMRT	{ HOME G.CH3 }	PV, FV, N, I, PMT TVoM	84
ARROW	{ HOME TOOLS PICS }		128
BEGEND	{ HOME G.CH3 }	VIEWP	58
BIG	{ HOME TOOLS PICS }		128
BIGSINE	{ HOME TOOLS PICS }		130
BKING	{ HOME TOOLS PICS }		239
BOLOID	{ HOME TOOLS }	X, Y	148
BPIECE	{ HOME TOOLS PICS }		239
BSTEP	{ HOME TOOLS }		151
BULLDOZER	{ HOME TOOLS }	DOZDATA	230
CALEND	{ HOME TOOLS }	MYR	264
CHKRS	{ HOME TOOLS }	Flags, SETUP, REDRAW, GLABEL, MYMOVE, THMOVE, LAYOUT	240
COMBINE	{ HOME TOOLS }		151
CONTOUR	{ HOME TOOLS }	ARRAY	223

Name	PATH	References	Page
CTR	{ HOME TOOLS }		165
C→L	{ HOME TOOLS }		260
DIODE	{ HOME G.CH3 }	I, V, Vb, Io	67
DISPLAY	{ HOME TOOLS PICS }		128
DISSECT	{ HOME TOOLS }		272
DOZDATA	{ HOME TOOLS PICS }		229
EGGS	{ HOME TOOLS }		144
EMPTY	{ HOME TOOLS PICS }		91
EPR1	{ HOME TOOLS }		287
EPRVAR	{ HOME TOOLS }		287
FF	{ HOME TOOLS }		281
FOURIER	{ HOME TOOLS }	A, Nmax, ω, t	218
FOYAY	{ HOME G.CH2 }	f, A, n, Nmax, ω, t	34
Fruit	{ HOME G.CH3 }	CSTA, APPLES, CSTO, ORANGES, TOTAL	42
G.CH2	{ HOME }		25
G.CH3	{ HOME }		41
GAND	{ HOME TOOLS }		126
GLABEL	{ HOME TOOLS }		157
GL↓	{ HOME TOOLS }		158
GL→	{ HOME TOOLS }	ADDB	158

Name	PATH	References	Page
GRAFX	{ HOME TOOLS }	SEE, STOPIC, RCLPIC	128
GSIZE	{ HOME TOOLS }		99
HILITE	{ HOME TOOLS }		255
HPR1	{ HOME TOOLS }	FF	289
HPRVAR	{ HOME TOOLS }	FF	289
HYP	{ HOME TOOLS }	X, Y	142
IdealGas	{ HOME G.CH3 }	P, V, n, R, T	46
IdealGas2	{ HOME G.CH3 }	P, V, n, R, T	51
IDID0	{ HOME TOOLS }	VD, VG, VP, Vbi	219
Shopping	{ HOME G.CH3 }	Fruit, Wagon	59
M1	{ HOME G.CH3 }	v, v0, a, t	62
M2	{ HOME G.CH3 }	x, x0, v0, v, t	62
M3	{ HOME G.CH3 }	x, x0, v, t, a	62
M4	{ HOME G.CH3 }	v, v0, a, x, x0	62
MAKEFACE	{ HOME TOOLS }	MeterFace	180
METER	{ HOME TOOLS PICS }		174
MkAxis	{ HOME TOOLS }		203
MKBOARD	{ HOME TOOLS }	BOARD	261
MOTION	{ HOME G.CH3 }	M1, M2, M3, M4	62
MOVEIT	{ HOME TOOLS }	LAYOUT, C→L, RPIECE, BPIECE RKING, BKING,	258

Name	PATH	References	Page
MQA	{ HOME G.CH3 }	Per, VIEWP	58
MULTIPLOT	{ HOME TOOLS }	SIV, VALS	216
MV	{ HOME TOOLS }	CX, CY, PX, PY	162
MV1	{ HOME TOOLS }	NUDGE	184
MV10	{ HOME TOOLS }	NUDGE	186
MVall	{ HOME TOOLS }	PSIZE, ADDB, Cursor	188
MYMOVE	{ HOME TOOLS }	SELECT, VALID, MOVEIT	248
MYR	{ HOME TOOLS }		266
NORMAL	{ HOME TOOLS PICS }		128
Now?	{ HOME TOOLS }	Nxtime, &t	202
NUDGE	{ HOME TOOLS }	Cursor, ADDB	183
OFF1	{ HOME }	TOOLS, TITLE, PICS	170
PICS	{ HOME TOOLS }		89
PLANE	{ HOME TOOLS }	Z, X, Y	217
PL4D	{ HOME TOOLS }		148
POINT	{ HOME TOOLS }	DApar	181
Pr8	{ HOME TOOLS }		204
PRANIM	{ HOME TOOLS }		150
PRGROB1	{ HOME TOOLS }		282
PRGROB2	{ HOME TOOLS }	DISSECT, TRANSLATE	284

Name	PATH	References	Page
TVoM	{ HOME G.CH3 }	PV, PMT, I, N, FV	49
TVM2	{ HOME G.CH3 }	PV, I, Begin?, Per, PMT, N, FV, VIEWP, MQA, BEGEND	56
VALID	{ HOME TOOLS }	WHOZAT	256
VIEWP	{ HOME G.CH3 }	Per, MQA, Begin?	58
VM	{ HOME TOOLS }	DApar, MAKEFACE, CTR, POINT	208
Wagon	{ HOME G.CH3 }	LOAD, WT.A, APPLES, WT.O, ORANGES	59
WHOZAT	{ HOME TOOLS }	LAYOUT	260
XPAN	{ HOME TOOLS }	VPAR	146
YPAN	{ HOME TOOLS }	VPAR	147
ZPAN	{ HOME TOOLS }	VPAR	147

<u>By Directory</u> *(Last→First)*

Directory PATH	Name
{ HOME }	OFF1
	TOOLS
	G.CH3
	G.CH2
{ HOME TOOLS }	HPRVAR
	HPR1
	EPRVAR
	EPR1
	PRGROB2
	PRGROB1
	FF
	TRANSLATE
	DISSECT
	MYR
	CALEND
	MKBOARD
	C→L
	WHOZAT
	MOVEIT
	VALID
	HILITE
	SELECT
	THMOVE
	MYMOVE
	REDRAW
	STARTUP
	CHKRS
{ HOME TOOLS } (cont.)	BULLDOZER
	CONTOUR
	IDID0
	FOURIER
	PLANE
	MULTIPLOT
	POINT
	MAKEFACE
	VM
	Pr8
	MkAxis
	Now?
	READV
	PSTRIP
	STRIP
	MV
	SCN
	PSCN
	MVall
	MV10
	MV1
	NUDGE
	SETUP
	PSCAN
	SCAN
	CTR
	ADDB
	GL→

Directory PATH	Name	Directory PATH	Name
{ HOME TOOLS } (cont.)	GL↓	{ HOME TOOLS PICS }	BKING
	GLABEL		RKING
	COMBINE		BPIECE
	BSTEP		RPIECE
	SSTEP		DOZDATA
	PRANIM		METER
	BOLOID		TITLE
	PL4D		BIGSINE
	ZPAN		ARROW
	YPAN		DISPLAY
	XPAN		TINY
	TRXY		NORMAL
	EGGS		BIG
	TRYIT		EMPTY
	HYP		SINE
	ROXY		
	GRAFX		
	RCLPIC		
	STOPIC		
	GAND		
	TPIX		
	≠SIZE		
	GSIZE		
	SEE		
	PICS		

Directory PATH	Name
{ HOME G.CH3 }	AMRT
	DIODE
	Step2
	Step
	MOTION
	M4
	M3
	M2
	M1
	Shopping
	Wagon
	BEGEND
	MQA
	VIEWP
	TVM2
	REACTOR
	READP
	IdealGas2
	TVoM
	R
	IdealGas
	Fruit
{ HOME G.CH2 }	RLCPER
	RLCEXP
	RLC
	FOYAY

Index

(Entries do not include user-named objects—see Appendix C)

Reader Comments

We here at Grapevine like to hear feedback about our books. It helps us produce books tailored to your needs. If you have any specific comments or advice for our authors after reading this book, we'd appreciate hearing from you!

Which of our books do you have?

Comments, Advice and Suggestions:

May we use your comments as testimonials?

Your Name: Profession:

City, State:

How long have you had your calculator?

Please send Grapevine catalogs to these persons:

Name ______________________________

Address ______________________________

City ______________ State __________ Zip __________

Name ______________________________

Address ______________________________

City ______________ State __________ Zip__________

If you liked this book, there are others that you will certainly enjoy also (and see also the comments on pages 20-21):

An Easy Course in Programming the HP 48G/GX

Here is an *Easy Course* in true Grapevine style: Examples, illustrations, and clear, simple explanations give you a real feel for the machine and how its many features work together. First you get lessons on using the Stack, the keyboard, and on how to build, combine and store the many kinds of data objects. Then you learn about programming—looping, branching, testing, etc.—and you learn how to customize your directories and menus for convenient "automated" use. And the final chapter is filled with example programs—all documented with comments and tips.

Algebra/Pre-Calculus on the HP 48G/GX

Grab your calculator, grab this book, and you're all set for math class. You'll get lots of lessons, examples and advice on graphing and problem-solving with:

Functions (linear, quadratic, rational, polynomial), trig, coordinate and analytic geometry, conics, equations of lines and planes, inequalities, vectors.

You'll also get great programmed tricks and tips for plotting and solving—all from an experienced classroom math teacher.

Calculus on the HP 48G/GX

Get ready now for your college math! Plot and solve problems with this terrific collection of lessons, examples and program tricks from an experienced classroom math teacher:

Limits, series, sums, vectors and gradients, differentiation (formal, stepwise, implicit, partial), integration (definite, indefinite, improper, by parts, with vectors), rates, curve shapes, function averages, constraints, growth & decay, force, velocity, acceleration, arcs, surfaces of revolution, solids, and more.

For more details on these books or any of our titles, check with your local bookseller or calculator/computer dealer. Or, for a full Grapevine catalog, write, call or fax:

ISBN		Price*
	Books for personal computers	
0-931011-28-0	**Lotus** Be Brief	$ 9.95
0-931011-29-9	A Little **DOS** Will Do You	9.95
0-931011-32-9	Concise and **WordPerfect**	9.95
0-931011-37-X	An Easy Course in Using **WordPerfect**	19.95
0-931011-38-8	An Easy Course in Using **LOTUS 1-2-3**	19.95
0-931011-40-X	An Easy Course in Using **DOS**	19.95
	Books for Hewlett-Packard Scientific Calculators	
0-931011-18-3	An Easy Course in Using the **HP-28S**	9.95
0-931011-25-6	**HP-28S** Software Power Tools: **Electrical Circuits**	9.95
0-931011-26-4	An Easy Course in Using the **HP-42S**	19.95
0-931011-27-2	**HP-28S** Software Power Tools: **Utilities**	9.95
0-931011-31-0	An Easy Course in Using the **HP 48S/SX**	19.95
0-931011-33-7	**HP 48S/SX** Graphics	19.95
0-931011-XX-0	**HP 48S/SX** Machine Language	19.95
0-931011-41-8	An Easy Course in Programming the **HP 48G/GX**	19.95
0-931011-42-6	Graphics on the **HP 48G/GX**	19.95
0-931011-45-0	Algebra/Pre-Calculus on the **HP 48G/GX**	19.95
0-931011-46-9	Calculus on the **HP 48G/GX**	19.95
	Books for Hewlett-Packard financial calculators	
0-931011-08-6	An Easy Course in Using the **HP-12C**	19.95
0-931011-12-4	The **HP-12C Pocket Guide:** Just In Case	6.95
0-931011-19-1	An Easy Course in Using the **HP 19BII**	19.95
0-931011-20-5	An Easy Course In Using the **HP 17BII**	19.95
0-931011-22-1	The **HP-19B Pocket Guide:** Just In Case	6.95
0-931011-23-X	The **HP-17B Pocket Guide:** Just In Case	6.95
0-931011-XX-0	**Business Solutions** on Your HP Financial Calculator	9.95
	Books for Hewlett-Packard computers	
0-931011-34-5	**Lotus in Minutes** on the **HP 95LX**	9.95
0-931011-35-3	**The Answers You Need** for the **HP 95LX**	9.95
0-931011-44-2	**Making Connections**: Data Communications w/the **HP Palmtop**	9.95
	Other books	
0-931011-14-0	**Problem-Solving Situations:** A Teacher's Resource Book	9.95
0-931011-39-6	**House-Training Your VCR:** A Help Manual for Humans	9.95

Contact: **Grapevine Publications, Inc.**
626 N.W. 4th Street P.O. Box 2449 Corvallis, Oregon 97339-2449 U.S.A.
800-338-4331 (503-754-0583) *Fax:* 503-754-6508

**Prices shown are as of 8/6/93 and are subject to change without notice. Check with your local bookseller or electronics/computer dealer—or contact Grapevine Publications, Inc.*

About the Author

Ray Depew is a very normal guy who happens to own an HP 48 and likes to write. Graphics on the HP 48G/GX is his second published work. His other projects in various stages of completion include a compilation of children's stories, additional software for the HP 48, and some musical compositions that may never see the light of day. To make some money on the side, Ray works as an integrated circuit engineer for Hewlett-Packard in Loveland, Colorado, where he lives with his wife, 5 children, and a Dalmatian named LazerJet. When he's not working, writing, or fixing up the house, he likes to spend time in the Rockies, read, make music, play with his family (and the dog), and eat oatmeal-chocolate chip cookies.

If you have comments or suggestions about this book, he would appreciate hearing them. You can write to him in care of the publisher:

Grapevine Publications, Inc.
P.O. Box 2449
Corvallis, Oregon 97339-2449 U.S.A.

Graphics on the HP 48G/GX

Here is a fascinating look at the potential of that big display on your HP 48G/GX. HP engineer Ray Depew shows you how to build graphics objects ("grobs") and then use them to customize displays with diagrams, pictures, labels, titles, multiple plots, games, and menu icons.

The book begins with a good, in-depth review of the Equation-Writer, the SOLVE and the PLOT applications. Next, it guides you through the locations and uses of the 48's built-in graphics commands, including the 3-D commands and animation. Then you learn to build your own grobs and combine them into some extensive application programs. There's even a chapter that discusses transferring your HP 48 graphics to other computers or printers.

So don't miss this insightful—and fun—excursion into the world of Graphics on the HP 48G/GX. It adds a whole new dimension to your use of this powerful machine.

Grapevine Publications, Inc.
626 N.W. 4th St. P.O. Box 2449
Corvallis, OR 97339 U.S.A.

ISBN 0-931011-42-6

51995

9 780931 011429