HP-71 BASIC MADE EASY

by
Joseph Horn

HP-71 BASIC MADE EASY

by Joseph Horn

P.O. Box 1080
Berkeley, CA 94701-1080
U.S.A.

Printed in the United States of America

HP-71 BASIC MADE EASY
by Joseph Horn

Published by: **SYNTHETIX**

Library of Congress Card Catalog Number: 84-51753
ISBN: 0-9612174-3-X

This electronic form of the book was last edited by the author on **18** Novem**ber 2014**. Please inform him of typos and errors so that they can be corrected:
joehorn@holyjoe.net

TABLE OF CONTENTS

INTRODUCTION

Hewlett-Packard calls the HP-71 an "open" machine. They have made all its Internal Design Specifications available to the public. Trouble is, "the public" is not the average HP-71 user! The documentation comes in huge indigestible volumes, and they cost an arm and a leg. Even if you can afford these esoteric tomes, you probably don't want to spend the rest of your life trying to understand them. So your HP-71 is still not "open" to you...

Hence this book. Unlike the unwieldy HP Internal Design Specifications, "HP-71 BASIC Made Easy" was written with you in mind. Its purpose is not to document every byte in the HP-71, but to give you just those facts and tips about the HP-71 that you can put to immediate use, secrets that the Owner's Manual and Reference Manual didn't tell you. The book also has a collection of programs that show how to put these secrets to real-world use. If you have any questions that remain unanswered by this book, at the end you'll find a terrific offer: the HP-71 Answer Man service, free of charge!

Please remember that this book is not intended to replace the Owner's Manuals. It complements them. If you found those books incomprehensible, begin reading this one and they will become clearer as you progress in HP-71 expertise. On the other hand, if the HP Manuals were pretty clear already, skip the easy stuff in this book and *enjoy* the wealth of PEEK$ and POKE information near the end, and browse through the BASIC and machine-language programs. You'll be glad you did!

Although the contents have been carefully checked, the author makes no guarantee about this book. HP has every right to revise the HP-71 operating system, and it is possible that future HP-71's will not respond the way this book indicates. If version-independent programming is essential to you, the only practical way is with the excellent HP-71 FORTH/ASSEMBLER ROM

combined with the HP-71 IDS (Internal Design Specification), which allows you to use the HP-71 function entry points, which HP has promised will remain the same in all future ROM revisions.

Thanks must go first to my brother James L. Horn, who first got me interested in computers and introduced me to PPC, and is always a source of moral support and intellectual challenge. This book is dedicated to him.

A warm thank-you goes to Karen Dodson, who put up with all the times I crashed the college computer back in the mid-1970's. Karen exemplifies for me what a "Real Programmer" is; one day I proudly put before her an uncommented listing of my latest program, quite sure that it was the most convoluted and wonderfully unreadable code ever concocted. With one glance, she not only saw what the purpose of the program was, but even suggested a shorter and faster way of doing it.

Thanks also to the late James Davidson, who wrote the famous HP-25 Library, and taught us all what a truly "elegant solution" is, in 49 bytes or less.

Without Richard Nelson (founder of PPC and CHHU, and editor of small-computer magazines for over ten years), this book would certainly not exist. My heartfelt thanks go to Richard for all his work and encouragement; may his tribe increase.

Jeremy Smith deserves a jolly thank-you for all his enthusiasm which is so contagious that sometimes I wonder if my excitement with these gadgets is my own or actually some of his which has rubbed off!

Finally, I thank Keith Jarett for publishing the book, and having patience with my schedule. (This book is a concrete example of Hofstadter's Law: "Things take longer than you expect, even when you take Hofstadter's Law into account".) His patience allowed me

to prune the verbosities and clarify the obfuscations. So the reader owes Keith a word of thanks too!

Joseph K. Horn [CHHU Member #13]
St. Michael's Prep.
1042 Star Route
Orange, California USA 92667

```
SUB ANYTIME @ DISP "(714) 633-2041" @ END SUB
```

CHAPTER 1:

THE THREE MODES OF OPERATION

When you first got your new HP-71, I'll bet that you thought you were just getting one machine! (I'm not alluding to the fact that most 71 owners wind up buying all kinds of peripherals!)

Fact is, the HP-71 is really three machines inside one case. And you can use whichever one you need, or all three if you wish!

CALC MODE

The first machine that the HP-71 behaves like is an awesome scientific calculator with more features than any other calculator, period. When you put the HP-71 into "CALC mode", it becomes a calculator that's so simple to use that you can use it right away with no special training; yet it's so powerful that after reading the CALC MODE chapter in this book, you'll be a keyboard wizard!

CALC Mode doesn't let you write programs, but you can solve formulae for several different values and only have to type the formula once! You can solve problems that are similar, without having to retype the whole thing (just change what's different). What you may have been using a programmable calculator for in the past, you can now do faster and easier in CALC mode, and all without writing a single program!

KEYBOARD BASIC MODE

The normal state of the HP-71 (the state after the batteries are first put in) is KEYBOARD BASIC mode. In this mode, you can type anything at all into the display, and when you press the big

[ENDLINE] key, the HP-71 tries to figure out what you want, and it does it. And when that's done, it goes idle again, ready for you to type another "command". You are Aladdin, and the HP-71 is the Genie from the lamp, and it's awaiting your every command (you get more than three !).

Of course, you have to type your commands in the right way. For example, you must spell them correctly. And you must use only the words that the HP-71 knows. For example, if you type the command BEEP OR ELSE, the HP-71 will not be able to obey your command because that's not how the BEEP command works on the HP-71.

The language the HP-71 "speaks" is called BASIC. BASIC is tremendously easy to learn, which is how it got its name. It's so basic that a kid could learn it (and they do!). So if the HP-71 seemed frightening, put your fear right out of your mind, because it really is eagerly awaiting your every command, and learning its language is as easy as reading a fun little book! (This one.)

PROGRAM BASIC MODE

If you take a bunch of BASIC commands and string them together, you have what's called a "program". Just as the programs they sell at a concert give a list of the pieces of music in the order they are to be played, a computer program is nothing other than a list of commands in the order they are to be done. Once you learn how to give commands to the HP-71 in KEYBOARD BASIC mode, you are ready to write programs!

This book will tell you everything you need to know about programming. You'll find out how to figure out which commands to use, then how to write them on paper in program form, then how to get them all neatly stored into the HP-71's memory.

Then the magic begins! After you get the commands stored in memory, all you have to do is press the [RUN] key on the HP-71, and all those commands get done *ZAP* in a flash! And to do them again, just press RUN again!

The real beauty of PROGRAM BASIC mode isn't the time it saves you, though that's nice. The best thing of all is that these collections of commands (programs) work on ANY HP-71. So what, you ask? "I only have one HP-71," you say. Ah! But there are thousands of other people who also have one HP-71, and just think of all the programs they're writing! It is probable that another HP-71 owner has already written a program to do exactly what you want! That's why membership in a club that supports the HP-71, and membership in the HP-71 User's Library, is more than smart.

Matter of fact, many folks use their HP-71 only in PROGRAM BASIC mode, because they filled it up with other people's programs, and just run them all the time. They've customized their HP-71 into a personalized machine, perfect for precisely their needs. This sort of thing is possible in PROGRAM BASIC mode. Just think how personalized your HP-71 will be by time you finish this book!

USING ALL THREE MODES

This book is attempting a mighty task. Not only does it explain how to use CALC mode thoroughly, and how to speak in KEYBOARD BASIC mode, and how to use PROGRAM BASIC mode, but it also will try to make clear the places where they overlap!

The HP-71 is truly schizoid. It isn't really three machines. The HP-71 merely seems like several machines in one, to make life easier. After a while, you'll understand how you can calculate in KEYBOARD BASIC mode. You can even run BASIC programs from CALC mode! By then, you will be able to think of the HP-71 not as three machines, but as one machine with a splittable

personality.

And as soon as you can use the HP-71 as one powerful machine, rather than as a collection of three less powerful machines, then at that time you can be called an HP-71 expert. You may not know what's happening under the hood, but you know how to drive it on every terrain at full speed, and that, by golly, makes you an expert driver!

Here's my guarantee: If you read this book cover to cover, and do the examples shown, I personally guarantee that you'll be an HP-71 expert!

Let's start with CALC mode.

CHAPTER 2:

CALC MODE

First, turn on the HP-71 by pressing the [ON] key in the lower left corner. After the HP-71 turns on, the [ON] key is no longer called [ON] (because it's already on!), but is called [ATTN] which stands for "attention". Type the number 12345 (with the number keys), and press [ATTN]. The number goes away! Think of [ATTN] as the "panic button"; when you hit it, whatever is happening gets zapped.

Right now, you should see a > character in the display, followed by a square block that's blinking. The > tells you that you're in KEYBOARD BASIC mode. The blinking block is called the "cursor". The cursor tells you where you're going to type next. When the cursor is shaped like a block (■), it is called the "replace cursor" because whatever you type replaces what used to be there.

Try typing your name into the display. The [SPC] key is used to put spaces in (like the space bar on typewriters and computer keyboards). Before you press [ATTN] to clear it out, press the left-arrow and right-arrow keys (between the [RUN] and [SPC] keys) to get used to moving the cursor around. Notice that pressing the arrow keys alone just moves the cursor one place left or right, but if you press the blue [g] key first, the arrow keys move the cursor all the way to the right or left end of the line. And if you press the gold [f] key first, the left arrow key backspaces and erases the letter it lands on, whereas the right arrow key deletes the letter it's on now and moves everything else up to fill in the hole.

From now on in this book, keystrokes will be written in abbreviated form. Rather than saying "Press the gold [f] key and then press the left arrow key", I will say [f][LEFT]. More

accurately, this keystroke sequence is called [f][BACK] or simply [BACK] because the gold [f] key always uses the words written in gold above the keys; see how "BACK" is written above the left arrow key? So too, the blue [g] key activates the blue things on the front of the keys. The letter keys have nothing printed in blue; the [g] key is the upper/lowercase shift key for the letters, just like on a typewriter. If it seems backwards, press [f][up-arrow] to switch it to normal. This is the Lowercase Lock key; it flip-flops between upper and lower case.

Notice that you can press [f][I/R] to change the shape of the cursor from a block (■) to an arrow (➧). The arrow cursor is called the "insert cursor" because it inserts what you type into the line, not just on top of it like the replace cursor does. The insert cursor is handy when you accidentally leave out a letter; you can insert it without having to retype the whole line.

To turn off the HP-71, press [f][ATTN], which is the [OFF] key. If you power-down in KEYBOARD BASIC mode, then the next time you turn the HP-71 on it will still be in KEYBOARD BASIC mode.

THE HP-71 CALC MODE

To put the HP-71 into CALCulator mode, press the gold [f] key, and then press the comma [,] key, which has the word "CALC" printed above it in gold. From now on, this keystroke sequence will be abbreviated [f][CALC] or simply [CALC]. You enter and exit CALC mode by pressing [CALC]. If you turn the HP-71 off while in CALC mode, you will still be in CALC mode next time you turn the HP-71 back on.

Let's try some problems in CALC mode. Type 12+34+56 and look at the display. Your keystrokes and the display are:

12+34+56

46+56

(46 is the intermediate answer of 12+34, of course). Now suppose we realize that we made a mistake, and it wasn't supposed to be 56, but 52. We could press [ATTN] and start all over. But there's a better way!

Just press [BACK]. And now look at the display:

[BACK]

46+5

so now press 2 and you'll see exactly what we wanted:

2

46+52

So correcting an error near the right end of the display is easy with the [BACK] key.

Here's a handy hint: if you want to press [BACK] more than once, don't press the gold [f] key each time; just press and hold it down, then go ahead and press [BACK] as many times as you want, while still holding [f] down. Matter of fact, if you hold the [BACK] key down for more than a second or so, it automatically repeats, rapid-fire, so you don't have to wear out your finger hitting it!

The gold [f] key and the blue [g] act two ways: as true prefix keys (press and release to change the meaning of your next keystroke) AND as true shift keys (press and hold down

to change the meaning of your next keystrokes while held down).

The problem now in the HP-71 is 12+34+52. How can we tell? Here's another handy hint: at any time during a calculation, if you want to see the whole problem, just press the up-arrow key, then press [ENDLINE] (NOT down-arrow!) to get back to where you were before:

[UP]

12+34+52

[ENDLINE]

46+5

Now suppose that we realize that we made another mistake. It wasn't supposed to be 12; It should have been 19, say. What now? Don't use [BACK]! That would take too long and besides, it would wipe out the whole calculation! Just press the up-arrow key again, and use the left and right-arrow keys to move the cursor to the offensive digit, and type the correct one. Simple!

[UP][RIGHT] 9

19+34+52

Of course, you need only press [ENDLINE] to continue!

[ENDLINE]

53+52

which is just what we wanted. To get the final answer, press

[ENDLINE]:

[ENDLINE]

105

USING THE LAST RESULT IN THE NEXT CALCULATION WITH RES

Now suppose you want this answer, 105, in your next calculation. There are four ways of doing it. The first and obvious way to use 105 in your next calculation is to type it in every time you want it. This works, but is the tough way.

The second way to get 105 in your next calculation is to use the RES function. RES (for "result") is automatically set to the last answer you got. In our case it's 105. (RES is similar to the "LASTX" function found on HP RPN calculators).

You can type RES from the keyboard (that takes three keystrokes) or you can press the [RES] key by pressing the gold [f] key and then the [ENDLINE] key that has "RES" printed above it in gold (that takes two keystrokes). Either way works the same, even if you type res in lower-case letters!

Let's try to evaluate $\boxed{105+5*\left(\frac{105}{105-30}\right)}$ using RES:

RES+

105+

See how the value of RES is pulled out automatically? Let's keep going:

5*RES/

105+525/

and finish up with:

(RES-30

105+525/(105-30)

If you see 110–30 instead, you forgot to open the parenthesis! If so, press up-arrow and insert the (into the calculation by pressing [I/R].

If we pressed [ENDLINE] right now, we'd get the final answer. But let's go slower to see what's happening. Notice that the cursor is blinking on top of a right parenthesis. It "knows" that we are supposed to close a parenthesis! Let's look at the intermediate answer by closing that parenthesis:

)

105+525/(75)

Now here's a special trick of CALC mode! The [RUN] key is magical. Instead of pressing [ENDLINE] and crashing full-steam into the final answer, watch what happens when you press [RUN] instead:

[RUN]

105+7

How about that! We are looking at the intermediate answer of $\frac{525}{75}$.

Let's finish up:

```
[ENDLINE]
```

```
112
```

USING THE VALUE OF RES BY TYPING ()

A nifty feature of CALC node is that an empty pair of parentheses `()` are automatically filled with the value of the last RESult. Since RES is 112 now (our result above), let's set it to 105 (like before). To set RES to 105, just type `105` and press `[ENDLINE]`:

```
105 [ENDLINE]
```

```
105
```

Now let's calculate, like before, $\boxed{105+5*\left(\frac{105}{105-30}\right)}$. But this time, use `()` instead of `RES`:

```
()+
```

```
105+
```

See how `()` gets filled with 105? Keep going:

```
5*()/
```

```
105+525/
```

This next one is tricky. You have to open the parentheses for the denominator (105–30), but you also are using `()` for 105. So you

need to press (twice:

(()-30

105+525/(105-30)

Doesn't that look familiar! Press [ENDLINE] to get the answer, 112 again.

This () feature of CALC mode can be used all kinds of ways. It is especially nice when the HP-71 supplies parentheses automatically.

For example, we now have 112 as our last RESult. Suppose we want to find the square root of this. All you have to do is press:

[SQR][ENDLINE]

10.583005244 3

If that happened too fast, try this. Type 112 [ENDLINE] to make 112 the RESult again. Now let's do it this way:

[SQR]

SQR()

with the cursor blinking on top of a right parenthesis. Now press:

)

SQR(112)

See how 112 gets put between the parentheses? Now press:

[RUN]

10.583005244 3

This brings up a very important point about the [RUN] key in CALC mode. We have the answer to $\sqrt{112}$ in the display, but it's still on the calculation side (right side) of the display. The HP-71 is still waiting to continue with more math, and doesn't know we're done. That's because we haven't pressed [ENDLINE] yet, of course.

But notice what that means about RES. We haven't changed it yet; it's still 112! To prove that, press [ON] to clear the display, then:

()

(112)

This can be used to good advantage! If you want to perform a bunch of calculations on RES without changing its value, then just be sure to get each answer by pressing [RUN], and never press [ENDLINE]. That way, RES will never change, because it only changes when you press [ENDLINE]!

Here's another nice feature of CALC mode. You don't even need RES or () to see the value of RES. Just clear the display by pressing [ATTN], and then press [ENDLINE]:

[ATTN][ENDLINE]

112

USING VARIABLES IN CALCULATIONS

If you really want to keep a value available for multiple use, you don't have to use RES or `()`. You can save any number under its own name, and use that name any time you need the number.

For example, suppose you want to use $\sqrt{2}$ several times. You certainly are not going to type in 1.41421356237 each time! Nor do you need to type `SQR(2)` each time, either! All you need to do is save it under a name (a letter) and then use that name.

Let's call it "S" for "S"quare root. All you need to do is:

```
S=SQR(2 [ENDLINE]
```

```
1.41421356237
```

(Remember, CALC mode closes all pending parentheses for you when you press `[ENDLINE]`! You don't need to close them yourself).

We've just done three things: calculated $\sqrt{2}$, displayed it, and saved it under the name "S". It's the "S=" above that did the saving; the rest did the calculating. Any time you want to save a number, just type its name followed by an = and then the number (or calculation).

Any time you want to use $\sqrt{2}$ now, you may merely type the letter S. Let's find $\boxed{\sqrt{2}+17-\frac{\sqrt{2}}{12}}$. Using S for $\sqrt{2}$, that's expressed as:

```
S+17-S/12 [ENDLINE]
```

```
18.2963624322
```

Did you notice how S changed into 1.41421356237 every time we used it? That's a lot easier than typing the number in, isn't it! We have assigned S that value, and it'll keep it as long as we want, even if you turn off the HP-71. (Exiting CALC mode and running a BASIC program may stomp it out, though).

We can change the value of S to anything we want any time by just saying S=. Since its value can vary like this, it's called a "variable", as in algebra.

Note for advanced thinkers: In the above example, S contains the number 1.41421356237. It does not contain the expression SQR(2). When you use S, it merely pulls its value off a shelf, it doesn't re-calculate it each time. So if you define S=A+B, S gets assigned the value of the current value of A plus the current value of B; it does not get assigned to always and everywhere be A+B. Otherwise, changing the value of A would change the value of S! That would not only be a confusing mess, but it would make variables work slower. If you really do wish to have variables that change value when other variables change, then what you want are called "user defined functions", which we'll discuss when we get into BASIC. User defined functions do work in CALC mode!

Variables in CALC mode can be any letter of the alphabet. If that's not enough (!), then you can also use letters followed by a single digit 0 through 9. So L, U, K and E are all perfectly fine variables because they are single letters. So are C3, P0, R2 and D2, because they are a single letter followed by a single digit. But AA is no good, because it's two letters. If you try to use AA in CALC mode it'll freak out. Likewise 3M is no good, because the digit is in front. And T42 won't work because there are two digits. Try assigning lots of variables lots of things, and using them in calculations of your own devising.

Subscripted variables, like X(8), can also be used, but they gobble up memory and are cumbersome in CALC mode. We'll discuss

their better uses when we get into BASIC.

If you ever want to save the current RESult into a variable, you don't need to use X=RES or X=(). Since merely pressing [ENDLINE] gives the value of RES, all you have to do is type X= and press [ENDLINE]! Try it: type 1+2 [ENDLINE] to put 3 into RES. Now type X= and press [ENDLINE]. You just saved 3 into X! After messy calculations, this is a handy thing to know.

NEGATIVE NUMBERS

How would you calculate 12 times negative 5? You could use 12*(-5), but that's wasting a keystroke. It's easier to use:

```
12*-5 [ENDLINE]
```

```
-60
```

It looks funny to see 12*-5, but why not? Division by negative numbers works just as nicely: 12/-5.

BUT!!! (And this is a big "but".) Don't try to raise numbers to negative powers this way! 12 raised to the negative 5th power cannot be typed in as 12^-5. Look what happens:

```
12^-
```

```
WRN:Illegal Context
```

(If the error message flashed by too quickly for you to read it, then press and hold down the [ERRM] key. It's a very handy key!)

To use negative powers, you must enclose the power in parentheses:

12^(-5 [ENDLINE]

4.01877572016E-6

This result, of course, is so small that the HP-71 displays it in scientific notation. It means 4.081877572016 times 10 to the –6 power, which is written the normal way like this:

0.000004018775720 16

but the poor HP-71 can't handle numbers that long! The reason 12^{-5} is so small is that 12^{-5} is the same as saying $\frac{1}{12^5}$ which is equal to $\frac{1}{248832}$, a very small number.

If you are wondering why you can multiply and divide by negative numbers but not raise to negative powers without resorting to parentheses, it's all because of the order in which the HP-71 does its math. Here's the scoop.

How would you calculate <u>on paper</u> this problem: $1+2\cdot3^4$? Of course, you'd first raise 3^4, getting 81; then you'd multiply 2 times that, getting 162, and finally you'd add 1 to that, getting 163, the answer. Notice that you did not calculate this problem left-to-right, even though left-to-right is the normal way of doing things. Why? Because powers are more important than multiplications, and multiplications are more important than additions. You simply know that powers "come before" multiplications, and multiplications "come before" additions.

This "coming before" idea is called algebraic hierarchy, or the order of operations. The HP-71 has algebraic hierarchy built in, and you must be familiar with it to use it well. In the following chart, the operations at the top are the "most important" ones, and

they "come before" the ones lower on the chart.

THE HP-71 ALGEBRAIC HIERARCHY

(1) **( )** Parentheses. (Nested ones first, from the inside out).
(2) Functions (**SQR**, **SIN**, **MOD**, etc.).
(3) **^** (Powers).
(4) **–** (Negative), **+** (Positive), and **NOT**.
(5) ***** (Multiplication), **/** (Division), **%** (Percent), and **** or **DIV** (Integer Division).
(6) **+** (Addition) and **–** (Subtraction).
(7) **<** (Less Than), **=** (Equal To), **>** (Greater Than), **#** (Not Equal To), **?** (Unordered With), and any combination of these.
(8) **AND**.
(9) **OR**, and **EXOR**.

Notice that this answers the negative powers dilemma. `12*-5` works fine because negation comes before multiplication. So the HP-71 first negates the 5, getting `12*(-5)`, and then it multiplies.

But powers come before negation. `12^-5` doesn't work because the HP-71 tries to perform the power raising before negating the 5. `12^-5` makes as much sense to the HP-71 as does `12^*5`. It makes no sense at all!

The reason powers come before negations is to make sure that expressions like `-12^4` evaluate correctly. `-12^4` does not mean negative twelve, raised to the 4th power. `-12^4` means 12 raised to the 4th power, negated. Try it; `-12^4` gives –20736, as it should. That's different from `(-12)^4`.

Although the **+** (Positive) function is listed in the table above, it is a totally useless function. Pressing `12*+5` is okay, but it works just as

well if you leave the + out. Matter of fact, if you key redundant + signs into a BASIC program, the HP-71 politely removes them!

MULTIPLE-ARGUMENT FUNCTIONS

Most functions, like LOG, SIN, FACT and so on, operate on just one number. For example, press:

```
[FACT] 14 [ENDLINE]
```

```
87178291200
```

Here we see the factorial of 14 (written as 14!). The FACT function took 14 and gave out 14!, a very big number.

The number that a function takes in parentheses is called its "argument". Here, the argument of the FACT function was 14.

Most functions have only one argument. But several have two or more arguments, and it is usually important to put the right number in the right order.

For example, the RMD function gives the remainder after a division. Suppose you cut a large pizza into fifteen pieces. Four people each have the same number of pieces until there are too few left to go around. How many pieces are left? This is the remainder of 15 divided by four. In HP-71 BASIC (and CALC mode), it is expressed as `RMD(15,4)`:

```
RMD(15,4 [ENDLINE]
```

```
3
```

So there are three pieces of pizza left, and four poor starving

college students sitting there watching them get cold. (They are going to college to figure out how to solve this mind-boggling math problem).

Notice that we wrote `RMD(15,4)`. The order of the numbers is important! Try `RMD(4,15)` and see what happens. Of course, it gives you the remainder of 4 divided by 15, which is 4.

THE COMMAND STACK

As we have seen, pressing the up-arrow button in CALC mode lets you edit the current calculation. What it really does is put you in the bottom (most recent level) of the "command stack". Every time you press `[ENDLINE]` in CALC mode, not only does RES change, but so does the command stack. Your entire calculation, no matter what it was, gets shoved up onto the bottom of the command stack, lifting the older commands there higher, until the one on the top gets bumped off the top of the stack and falls into the hungry black hole that gobbles up commands too old to stay on the stack.

Try pressing up-arrow several times right now. You'll see the most recent thing you did, first. Above that you'll find the next most recent command line. Each higher line in the stack is an older and older command, until you reach the top of the stack (be careful! Don't fall in the black hole!) which contains your least recently used command.

The HP-71 not only allows you to look at these commands, but to re-use them however you wish! If you wish to re-execute an old calculation, simply get it into the display by pressing the arrow keys until you see it, and then press `[ENDLINE]`. It'll not only get executed, but it'll get pulled out of the stack and placed nicely on the bottom for you!

Even better, you can edit old commands too. If you want to re-execute an old calculation with just one number changed, then find it in the stack, edit it to look the way you want, then press [ENDLINE]. No need to re-type the whole calculation!

The command stack is five commands high in a normal HP-71. But there are ways of making the stack bigger, so that it can hold more commands. It can be expanded to hold up to 16 commands! The theory behind that, and a program to do it, are presented later in this book. But first we have to understand BASIC, which is our next topic.

HP-71 CALC mode. It beats AOS, and it even beats RPN once you get used to using the command stack. Several hours of practice will pay off in many more hours of time saved as you fly through calculations that before gave you the willies. Everything's visible; no hidden numbers, no hidden operations. Everything's logical; no bizarre mixture of pre-and postfix notation, like AOS (Texas Instruments' calculator logic). Everything's clear; no bizarre new logic to learn, like RPN (Hewlett-Packard's calculator logic).

Ready to try some real-world examples?

REAL-WORLD EXAMPLES OF CALC-MODE USAGE

EXAMPLE 1. Silas Farmer delivers tomatoes to the cannery twice daily. On Monday he delivered 25 metric tons and 27 tons. On Tuesday he delivered 19 tons and 23 tons. Both days the cannery paid him $55 per ton, minus 2% because of blight on the tomatoes. On Wednesday, Silas delivered 26 tons and 28 tons, and the cannery paid him $57.50 per ton, minus 3%. What is Farmer's total net income for Monday through Wednesday?

Solution: Write the problem out in algebraic form:

Monday's & Tuesday's Tomato Value: $\boxed{M = 55 \cdot (25 + 27 + 19 + 23)}$
Deduction for Monday's & Tuesday's tomato blight: $\boxed{2\%M}$
Wednesday's Tomato Value: $\boxed{W = 57.5 \cdot (26 + 28)}$
Deduction for Wednesday's tomato blight: $\boxed{3\%W}$
Total Net Income: $\boxed{M - 2\%M + W - 3\%W}$

Next, key the calculations in the order indicated:

1: `M=55*(25+27+19+23)`
2: `W=57.5*(26+28)`
3: `M-2%M+W-3%W` → 8078.45

Answer: $8078.45 net income.

EXAMPLE 2. Engineer P.C. Bord has determined that in an RC circuit, the total impedance is 77.8 ohms and the voltage lags current by 36.5°. What are the values of the resistance, and the capacitive reactance in the circuit?

Solution: Use the formulae $\boxed{X = R \cdot \cos(\alpha)}$ & $\boxed{Y = R \cdot \sin(\alpha)}$ for the conversion of polar to rectangular form. In this case:

R=77.8 (the total impedance; the polar magnitude here)
A=36.5 (the current lag; the angle here)
Resistance = $R \cdot \cos(A)$
Capacitive Reactance = $R \cdot \sin(A)$

Key the calculations in the order indicated:

1: `R=77.8`
2: `A=36.5`

3: `R*COS(A)` → 62.54
4: `R*SIN(A)` → −46.28

Answer: Resistance is 62.54 ohms, and reactance is −46.28 ohms.

Alternate Solution: If you have the Math Pac ROM plugged into your HP-71, you can convert polar to rectangular form in one step:

1: `RECT((77.8, -36.5))` → (62.54,−46.28)

EXAMPLE 3. Five students took a test, and earned grades of 95%, 90%, 88%, 94%, and 93%. What was the average grade?

Solution: $Average = \frac{sum\ of\ items}{number\ of\ items}$.

1: `(95+90+88+94+93)/5` → 92

Answer: 92 is the average grade.

EXAMPLE 4. You wish to find the sum of the numbers from 1 to 50, and you forget the formula. Rather than waste time trying to rediscover the formula, you grab your HP-71 and start adding in CALC mode `1+2+3+4+5`...

Unfortunately, when you hit 36, the HP-71 beeps, says "Line Too Long", and seems to freak out.

The HP-71 can only handle lines up to 96 characters long. This applies to CALC mode too. If you need to perform a calculation longer than that, break it up into shorter pieces, and then use RES or `()` to connect the pieces.

In this example, we would add chunks of the desired sum, and use `()` to get the subtotals:

1: `1+2+3+4+5+6+7+8+9+10` → **55**
2: `()+11+12+13+14+15+16+17+18+19+20` → **210**
3: `()+21+22+23+24+25+26+27+28+29+30` → **465**
4: `()+31+32+33+34+35+36+37+38+39+40` → **820**
5: `()+41+42+43+44+45+46+47+48+49+50` → **1275**

Answer: **1+2+3...+48+49+50=1275.**

Alternate Solution: You suddenly remember the formula for the sum of the numbers from 1 to N is $\frac{N}{2}(N+1)$:

1: `50/2*(50+1)` → **1275**

This is considerably easier! In general, if you get the "Line Too Long" message, you are doing things the hard way. You should look for the simpler way of doing it.

Note: If you get the "Line Too Long" message, the HP-71 will do its best to let you save time. It automatically puts you into the bottom of the command stack, where you will see the whole long line you just typed. Press `[g][right-arrow]` and edit the right end of the line, then press `[ENDLINE]` to get a subtotal (or intermediate result), and then use RES or `()` to continue.

EXAMPLE 5. You wish to evaluate $7x^5 - 12x^4 + 54x^3 - 22x^2 + 11x - 1$ with values of x {0, 0.1, 0.2}. You know how to do it on an HP hand-held calculator with RPN logic; you would use Horner's Method to avoid powers and round-off errors. But the HP-71 method is not obvious. How to do it?

Solution: Use Horner's Method! Just subtract one from the highest power, and type that many parentheses, and go from there. In our example, the highest power is 5, so we type four parentheses and go from there:

1: `X=0`
2: `(((7*X-12)*X+54)*X-22)*X+11)*X-1` → −1
3: `X=.1`
4: `[UP][UP][ENDLINE]` → −.06713
5: `X=.2`
6: `[UP][UP][ENDLINE]` → .73504

Answer: $f(0)=-1$, $f(0.1)=-0.06713$, $f(0.2) = 0.73504$.

Note: We used the command stack in a tricky way here. We used a variable (X) to stand for 0, 0.1, and 0.2 to avoid having to type the whole equation again. We then set X to different values, and used the formula intact from the command stack. Whenever you have to repeat a calculation with slightly different values, do it this way. You'll soon be addicted to the command stack!

EXAMPLE 6. Derek Lobos, the interior decorator, has four picture hangers in his living room, and he owns seven framed paintings. To keep a fresh look, he wants to re-arrange the paintings every month, and once in a while bring a new painting out of the closet and retire one of the ones displayed. This offers many possible permutations! How long will it be before every possible arrangement has been used, when Derek must go shopping for a new painting?

Solution: The formula for the permutations of X things taken Y at a time is $\boxed{\frac{X!}{(X-Y)!}}$. The exclamation point stands for the factorial function, which on the HP-71 is written `FACT`. Since Derek wants

to re-arrange every month, and there are 12 months in a year, we must divide the number of permutations by 12 to see how many years Derek has before he must go shopping:

1: `FACT(7)/FACT(7-4)/12` → **70**

Answer: Derek is all set for the next 70 years!

Alternate solution: You could also use a user-defined function nicely here.

EXAMPLE 7. The Fonch Hotel in San Placebo, California, catches on fire. The top of a very sturdy 22-foot long rainspout comes loose from the side of the hotel. It falls forward, pivoting at ground level, until it crunches to a halt propped against the side of the Clutch Cargo Bank across the street. Amazingly, the pipe does not break, or even bend. The street is 18 feet wide. A fire truck, 7 feet tall and 8 feet wide, needs to pass under the pipe. Can it make it?

Solution: The pipe forms a hypotenuse with the street as one leg of the right triangle. Pythagoras tells us how to find the other leg, which is the height of the pipe on the bank wall. We can use a proportion to calculate the height of the pipe 8 feet away from that wall (the width of the truck), and if it's more than 7 feet, the truck will fit under the pipe.

1: `W=SQR(22^2-18^2)` → **12.65**
2: `(18-8)*W/18` → **7.03**

Answer: Yes. The pipe rests on the bank wall 12.65 feet up, and the height of the pipe 8 feet away from the wall is 7.03 feet up, which gives just enough clearance for the fire truck. The Famous Fonch Hotel is saved, and they all live happily ever after.

FUNCTIONS THAT WORK IN CALC MODE

The following list of functions is just a fraction of the number of commands that the HP-71 can use. But when it's in CALC mode, these are the only ones you can use. To really get good at CALC mode usage, go through this list and try all the examples given. Once you know these functions, you know CALC mode! (If you have the MATH PAC ROM or other plug-ins that add more functions to CALC mode, check the Quick Reference Guide for the ROM for examples of each function.)

+ performs the addition of two numbers.

`1+2` gives 3.

− performs the subtraction of two numbers, or negates one.

`7−3` gives 4.

`−7` gives −7.

`−7−3` gives −10.

`−7−−3` gives −4 (negative 7 minus negative 3 = −7+3 = −4)

***** performs the multiplication of two numbers.

`3*4` gives 12.

`(1+2)*(3+4)` gives 21 (3 times 7).

`12/2*2` gives 12.

`12/(2*2)` gives 3.

/ performs the division of two numbers.

`12/2/2` gives 3. (12 divided by 2, all divided by 2)

`12/2^2` gives 3. (12 divided by 2 squared = 12/4)

`12/2+2` gives 8. (12 divided by 2, plus 2)

`(3+4+6+7)/(1+2+3+4)` gives 2.

^ raises the first number to the power of the second number.

`2^5` gives 32 (2 to the 5th power)

`FACT(10)/2^8` gives 14175 (10 factorial divided by 2 to the 8th power)

Note that you can use ^ to find roots:

`65536^(1/4)` gives 16 (because 65536 raised to the ¼ power

means the same thing as the 4th root of 65536).

% is the same as * but divides the answer by 100.

6%24 gives 1.44 (6 percent of 24 is 1.44)

24+6%24 gives 25.44 (24 plus 6%)

ABS(x) returns the absolute magnitude of x.

ABS(-3) gives 3

ABS(0) gives 0

ABS(5) gives 5

ACOS(x) returns the angle that has x as its cosine.

ACOS(.5) gives 60 (in DEGREES mode)

ACOS(0) gives 90

ACOS(-1) gives 180

ACS(x) is the same as ACOS above.

AF returns the current Accuracy Factor of the clock.

AF(x) does the same but then sets it to x.

AF(1537) gives the old AF and then sets it to 1537.

AND performs a logical AND on two numbers:

0 AND 0 gives 0

0 AND 2 gives 0 (2 or any non-zero number)

3 AND 0 gives 0

4 AND 5 gives 1

AND means "Exactly both"

ANGLE(x,y) returns the angle from the x-axis to point (x,y).

ANGLE(3,4) gives 53.13 (in DEGREES mode)

ANGLE(.5,SQR(3)/2) gives 60

ANGLE(-1,0) gives 180

ASIN(x) returns the angle that has x as its sine.

ASIN(.5) gives 30 (in DEGREES mode)

ASIN(0) gives 0

ASIN(-1) gives −90

ASN(x) is the same as ASIN above.

ATAN(x) returns the angle that has x as its tangent (slope).

ATAN(1) gives 45 (in DEGREES mode)

ATAN(SQR(3)) gives 60

`ATAN(INF)` gives 90

`ATN(x)` is the same as `ATAN` above.

`CEIL(x)` returns the integer ceiling above x.

`CEIL(-1.7)` gives −1

`CEIL(-1)` gives −1

`CEIL(1)` gives 1

`CEIL(1.7)` gives 2

`CLASS(x)` returns the "class" code of x.

`CLASS(INF)` gives 4, the code for +INF

`CLASS(MINREAL)` gives 2, the code for denormalized numbers

`CORR(x,y)` returns a correlation coefficient.

`CORR(2,4)` gives the correlation between the current stat array variables #2 and #4. There must be a statistic array defined and containing data. The creation of stat arrays is not done in CALC mode.

`COS(x)` returns the cosine of angle x.

`COS(60)` gives .5

`COS(180)` gives −1

`COS(0)` gives 1

`DATE` returns the date in YYDDD format; YY=year, DDD=day#.

`DEG(x)` converts x radians into degrees.

`DEG(PI)` gives 180 because PI radians = 180°.

`DIV` divides two numbers, returns only the quotient, and throws away the remainder.

`12 DIV 5` gives 2 (5 goes into 12 twice)

`DATE DIV 1000` gives the current year.

Note: the backslash "\" character (ASCII code 92) may be assigned to a key as a typing aid and used as DIV.

`DVZ` returns −7, the number of the Division-By-Zero flag.

`DVZ` gives −7

`FLAG(DVZ)` gives 1 if a division-by-zero error has occurred, 0 if not.

`FLAG(DVZ,0)` does the same but also clears the flag.

`FLAG(0,FLAG(DVZ))` makes it visible in flag 0.

`TRAP(DVZ)` gives the current division-by-zero trap value.

`TRAP(DVZ,2)` does the same but also sets it to 2.

EPS returns 1.E−499, the smallest normalized positive number.

`ABS(CLASS(X)=2)=(ABS(X)<EPS)` returns 1, because if a number is smaller than EPS, it is denormalized, and therefore has a class of 2.

ERRL returns the BASIC program line number of the last error.

`ERRL` gives 0 if no BASIC program errors have occurred.

ERRN returns the number of the last error.

`ERRN` gives 4 (if TRAP(DVZ)=0) after trying to calculate TAN(90) because error #4 is "TAN=Inf".

`ERRN` gives 11 (if TRAP(IVL)≠2) after trying to calculate ASIN(2).

EXOR performs a logical Exclusive OR on two numbers.

`0 EXOR 0` gives 0

`0 EXOR 1` (or any other non-zero number) gives 1

`1 EXOR 0` gives 1

`1 EXOR 1` gives 0

EXOR means "one or the other but not both"

EXP(x) returns *e* (2.71828182846) raised to the power x.

`EXP(1)` gives 12 significant digits of *e*.

`EXP(27.6310211159)` gives 999999999971.

EXPM1(x) is the same as `EXP(x)-1`, but more precise.

`EXPM1(PI/1E7)` gives 3.14159314707E−7 (12 digit accuracy)

`EXP(PI/1E7)-1` gives .00000031416 (5 digit accuracy).

Use EXPM1 when x is close to zero.

EXPONENT(x) returns the exponent (decapower) of x.

`EXPONENT(10)` gives 1

`EXPONENT(100)` gives 2

`EXPONENT(999)` gives 2

`EXPONENT(1000)` gives 3

`EXPONENT(x)` is more accurate than `IP(LOG10(x))`:

`EXPONENT(10/3*3)` gives 0, which is correct.

`IP(LOG10(10/3*3))` gives 1, which is wrong.

`FACT(x)` returns the factorial of x.

`FACT(6)` gives 720, which is $6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$.

`FACT(8)` gives 40320, which is $8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$.

`FACT(0)` gives 1 (by definition).

`FLAG(x)` returns the value (0 or 1) of flag x.

`FLAG(0)` gives the value of flag 0, shown in the display.

`FLAG(INX)` gives the value of the Inexact flag.

`FLAG(-3)` gives the value of the Continuous On flag.

`FLAG(x,y)` returns flag x and then sets it to the value of y.

`FLAG(-2,0)` enables the beeper.

`FLAG(-2,1)` disables the beeper.

`FLAG(-10,1)` sets RADIANS mode.

`FLAG(-10,0)` sets DEGREES mode.

`FLOOR(x)` returns the integer floor below x.

`FLOOR(-1.7)` gives −2.

`FLOOR(-1)` gives −1.

`FLOOR(1)` gives 1.

`FLOOR(1.7)` gives 1.

`FNvar`, with "var" being any valid variable; user-defined.

`FP(x)` returns the fractional part of x.

`FP(-1.7)` gives −.7

`FP(-1)` gives −0

`FP(1)` gives 0

`FP(1.7)` gives .7

`INF` returns the value Inf (mathematical infinity).

`SQR(INF)` gives Inf

`1/0` gives Inf after `TRAP(DVZ,2)`

`FACT(255)` gives Inf after `TRAP(OVF,2)`

`INT(x)` returns the greatest integer ≤x. Same as `FLOOR(x)`.

`INX` returns −4, the value of the Inexact Result flag.

Used just like `DVZ`.

`IP(x)` returns the integer part of x.

`IP(-1.7)` gives −1.

`IP(-1)` gives −1.
`IP(1)` gives 1.
`IP(1.7)` gives 1.

`IVL` returns −8, the number of the Invalid Operation flag.
Used just like `DVZ`.

`KEYDOWN` returns 1 if a key is being pressed; 0 if not.
Useful in CALC mode only inside FN functions.

`LET` assigns a value to a variable.
`X=5` is the same as `LET X=5` (LET may be omitted).
`X=1+2+3+4` gives 10 and sets variable X equal to 10.
`F3=FLAG(3)` stores flag 3's setting into variable F3.
`N(3,4)=153` stores 153 into N's 3rd row, 4th column.

`LGT(x)` returns the common logarithm (base 10) of x.
`LGT(10)` gives 1
`LGT(100)` gives 2
`LGT(1000)` gives 3
`LGT(10^5)` gives 5

`LN(x)` returns the natural logarithm (base *e*) of x.
`LN(10)` gives 2.3
`LN(100)` gives 4.6
`LN(1000)` gives 6.9
`LN(EXP(5))` gives 5

`LOG(x)` is the same as `LN(x)`; it is <u>not</u> log base 10!

`LOGP1(x)` is the same as `LOG(x+1)`, but more precise.
`LOGP1(PI/1E12)` gives 3.14159265359E−12 which is right.
`LOG(PI/1E12+1)` gives 0, which is wrong.
Use `LOGP1` whenever x is close to zero.

`LOG10(x)` is the same as `LGT(x)`.

`MAX(x,y)` returns the larger number, x or y.
`MAX(2,3)` gives 3
`MAX(3,2)` gives 3 (order doesn't matter)
`MAX(-7,-8)` gives −7
`MAX(A,MAX(B,C))` gives the MAX of A & B & C
`MAX(X,0)` is the same as `X*(X>0)`

MAXREAL returns the largest positive finite number.

MAXREAL gives 9.99999999999E499

MEAN returns the average of your accumulated statistics.

MEAN gives the mean of the first variable in your statistic array. There must be a statistic array defined and containing data. The creation of stat arrays is not done in CALC mode.

MEAN(x) returns the average of variable number x. See MEAN.

MEM returns the number of unused RAM bytes in main memory.

MEM(x) is like MEM but for RAM or ROM bytes in :PORT(x).

The number of unused ROM bytes is useless information.

MIN(x,y) returns the smaller number, x or y.

MIN(2,3) gives 2

MIN(3,2) gives 2 (order doesn't matter)

MIN(-7,-8) gives −8.

MIN(A,MIN(B,C)) gives the MIN of A & B & C

MIN(X,0) is the same as X*(X<0)

MINREAL gives the smallest possible positive number.

MINREAL is a denormalized number.

MINREAL generates an error if TRAP(UNF) is not 2.

MINREAL gives 0.00000000001E−499 after TRAP(UNF,2)

MOD(x,y) returns x modulo y (defined as x-y*INT(x/y)).

MOD(x,y) always gives a number between 0 and y.

MOD(12,5) gives 2 (because when you keep subtracting 5 from 12, the number you get between 0 and 5 is 2).

MOD(-12,5) gives 3 (because when you keep adding 5 to −12, the number you get between 0 and 5 is 3).

MOD(12,-5) gives −3 (because when you keep adding −5 to 12, the number you get between −5 and 0 is −3).

NAN returns NaN (Not-a-Number), a value that has no value.

0/0 returns an error when TRAP(IVL) is not 2.

0/0 returns NaN after TRAP(IVL,2).

NOT x returns the logical NOT of x.

NOT X gives 1 if X=0; 0 if X is not 0.

FLAG(0,NOT FLAG(0)) flip-flops flag 0.

OR performs a logical OR on two numbers.

`0 OR 0` gives 0

`0 OR 2` (or any non-zero value) gives 1

`3 OR 0` gives 1

`4 OR 5` gives 1

OR means "either one or the other or both"

OVF returns −6, the number of the Overflow flag.

Used just like `DVZ`.

PI returns 3.14159265359, twelve significant digits of pi.

`PI*R^2` gives the area of a circle with radius R.

Note: `PI` in the HP-71 is not equal to the mathematical constant π. `PI` stops at the twelfth digit; π is an irrational number that never stops. So in RADIANS mode, sin(π)=0 but `SIN(PI)` isn't 0.

PREDV(x) returns a statistically predicted value based on x.

`PREDV(x)` uses the current statistical array and selected linear regression variables to predict y based on x. There must be a statistic array defined and containing data, and the `LR` function must be performed. The creation of stat arrays and the `LR` initialization are not done in CALC mode.

RAD(x) converts x degrees into radians.

`RAD(180)` gives PI because 180° = PI radians.

RED(x,y) returns x reduced by y (defined as x-y*IROUND(x/y)).

`RED(x,y)` always gives a number between -y/2 and y/2.

"IROUND" in the definition is the "nearest" integer.

`RED(12,5)` gives 2 because when you keep subtracting 5 from 12, the nearest you get to 0 is at 2.

`RED(13,5)` gives −2 because when you keep subtracting 5 from 13, the nearest you get to 0 is at −2.

RES returns the value of the last result.

RES is set every time you press `[ENDLINE]` in CALC mode.

`()` is another way to get the value of RES.

`13^PI` gives 3159.04819859, then:

`RES^(1/PI)` gives 13 (if done right after `13^PI`).

RMD(x,y) returns the remainder of x/y (defined x-y*IP(x/y)).

RMD(x,y) always gives a number between 0 and y*SGN(x). The sign of the answer is the same as the sign of x.

RMD(12,5) gives 2 because 12 divided by 5 leaves a remainder of 2.

RMD(−12,5) gives −2 because 12 divided by 5 leaves a remainder of 2, and −12 is negative.

RND returns a random number between 0 and 1.

RND<RND gives 0 half the time, and 1 the other times.

SDEV returns the sample standard deviation for variable #1.

SDEV gives the standard deviation (Sn-1) of the sample data accumulated under the first variable in your statistic array. If the data is not a sample but an entire population, the standard deviation (Sn) may be calculated with SDEV after ADDing the MEAN. For SDEV to work, there must be a statistic array defined and containing data. The creation of stat arrays and the accumulation of data into them is not done in CALC mode.

SDEV(x) is the same but it works on variable #x. See SDEV.

SGN(x) returns the sign of x.

SGN(−37) gives −1

SGN(0) gives 0

SGN(−0) gives −0

SGN(153) gives 1

SIN(x) returns the sine of angle x.

SIN(30) gives .5 (in DEGREES mode)

SIN(90) gives 1

SIN (180) gives −0

SIN(270) gives −1

SQR(x) returns the square root of x.

SQR(9) gives 3

SQR(25) gives 5

SQR(12345654321) gives 111111

SQRT(x) is the same as SQR(x).

TAN(x) returns the tangent of angle x.

`TAN(45)` gives 1 (in DEGREES mode)

`TAN(90)` gives an error, or MAXREAL, or Inf, depending on the current setting of TRAP(DVZ).

TIME returns the number of seconds since last midnight.

`TIME` gives a number between 0 and 86399.99

`RMD(TIME,60)` gives the sweep second hand reading.

`RMD(TIME,3600) DIV 60` gives the minute hand reading.

`TIME DIV 3600` gives the hour hand reading (24 Hr. clock).

TOTAL returns the total of the first statistic variable.

`TOTAL` gives the sum of the data accumulated under the first variable in your statistic array. For `TOTAL` to work, there must be a statistic array defined and containing data. The creation of stat arrays and the accumulation of data into them is not done in CALC mode.

TOTAL(x) does the same but for variable #x.

TRAP(x) returns the current value of trap number x.

`TRAP(INX)` gives 0 if the Inexact Result Trap is 0; it gives 1 if the trap is 1, and 2 if the trap is 2.

TRAP(x,y) does the same but also sets the trap to y.

`TRAP(INX,2)` gives the value of the Inexact Result Trap, then sets it to 2.

UNF returns −5, the number of the Underflow flag.

`UNF` gives −5

`FLAG(UNF)` gives 1 if an underflow has occurred; else 0.

`FLAG(UNF,0)` does the same but also sets the flag to 0.

`TRAP(UNF,2)` gives the value of the Underflow Trap and then sets it to 2.

CHAPTER 3:

KEYBOARD BASIC MODE

When you are in CALC mode, there is a little "CALC" sign in the display. This is called the "CALC annunciator" because it announces that you are in CALC mode. There are other annunciators, but we'll jump off that bridge when we get to it. For now, please exit CALC mode if you're in it, by pressing `[f][CALC]`. The CALC annunciator should turn off.

Whenever the CALC annunciator is turned off, you are in BASIC mode. Just as CALC mode was expecting you to key in mathematical calculations, BASIC mode expects you to key in commands in the BASIC language.

If you are in BASIC mode and need to perform a calculation, you have two options. You may enter CALC mode, perform your calculation, and go back to BASIC mode. Or, you may simply perform the calculation in BASIC mode!

For example, let's do the same calculation we did before. It was:

```
105+5*105/(105-30)
```

To do this in BASIC mode, simply type the whole thing in as you see it, but don't press `[ENDLINE]` yet. You'll see

```
>105+5*105/(105-30)
```

in the display. Remember, the `>` means that you're in KEYBOARD BASIC mode. Notice that BASIC isn't giving any intermediate answers, as CALC did. Here's a general rule:

KEYBOARD BASIC IGNORES YOU UNTIL YOU END THE LINE!

Since `105+5*105/(105-30)` is a command in HP-71 BASIC, as soon as you press `[ENDLINE]` the HP-71 interprets it and obeys your command:

`[ENDLINE]`

112

If you wish to use the command stack in BASIC mode, you can't just press the `[UP]` key as you do in CALC mode; you have to press `[g][CMDS]` first to get into the command stack, and then you can move up and down the stack with the arrow keys. Try that now. Pressing `[ENDLINE]` when you're in the command stack will re-execute the command displayed. And you can use the `[LEFT]` and `[RIGHT]` arrows to move the cursor around on a command, allowing you to change it before executing it! Don't forget the insert cursor (press `[f][I/R]` to get it) that allows you to insert things!

Here's another rule to remember:

THE HP-71 IS EASY TO USE BECAUSE IT HAS A COMMAND STACK!

So you the sooner you get used to jumping into the command stack and using what's there, the sooner you'll find the HP-71 a breeze to use.

THE HP-71 BASIC LANGUAGE

The display should look like this:

The > means "Hi, I'm BASIC; what do you want?" The blinking blob (the replace cursor) means "Type on top of me".

Before we try any commands, be sure that the Lower Case Lock is on. It is easy to do this. Press any letter. If it is a capital (upper case) letter, press the [f][LC] key. Press [ATTN] to clear the display. From now on in this book, I will assume that you leave the Lower Case Lock on.

Here we go! Press these letter keys: [B][E][E][P]. You should see:

This is a command in BASIC that tells the computer to make a sound that a tone-deaf person might mistake for a musical note! Then why doesn't the computer beep? Remember, the command is ignored until you press [ENDLINE]! Go ahead and press [ENDLINE] now, and listen to the beautiful beep.

Notice that BEEP is on the keyboard, printed in gold above the [L] key. As you may have guessed, you can get the command BEEP without typing it by merely pressing the gold [f] key and then pressing the [L] key. This is abbreviated [f][BEEP] or merely [BEEP]. Notice that this "typing aid" key types BEEP in upper case letters, regardless of the setting of the Upper Case Lock. This way you'll be able to tell in this book whether you have to type it yourself (it'll be in lower case letters) or whether it has a typing aid key (it'll be in upper case letters).

Since you have to press [ENDLINE] for any command to get carried out, from now on I won't bother to tell you to press

[ENDLINE]. If I say to execute the BEEP command, or if I simply say that you should BEEP, I mean that you should type BEEP and press [ENDLINE]. I don't, of course, mean that you should beep. You may if you wish, however.

To see the power of BASIC, execute this command:

```
FOR x=1 TO 10 @ x @ NEXT x
```

You just told the HP-71 to count from 1 to 10, which it did! (We will cover FOR, TO and NEXT later). If it seemed awfully slow (about half a second for each number), that's because the HP-71 slowed it down on purpose to give you time to watch the numbers. If you want to speed it up to 1/10th of a second for each number, use this command:

```
delay .1
```

Now enter the command stack by pressing [g][CMDS], and go up the stack until you find the FOR command you typed above, and press [ENDLINE] to re-execute it. See how fast the HP-71 counts to 10 now! Try DELAY 0, and see it really rip! In general, the DELAY command merely tells the HP-71 to slow down displays by the number of seconds you specify.

There are several other commands in BASIC that control how things are displayed. CONTRAST is a nice one that lets you change the contrast of the display. If the display seems too dim, try executing

```
contrast 12
```

or an even higher contrast. If the display is too dark, with all the dots looking like they're on, then try

```
contrast 6
```

or an even lower contrast. The "normal" CONTRAST (after the batteries are first put in) is CONTRAST 9. I keep mine set to CONTRAST 7 because I slouch in my chair. You will no doubt find your favorite CONTRAST too. If you use it both sitting down and standing up, you'll probably set it to a low CONTRAST while sitting and a high one while standing.

So far, all our calculations were done in the HP-71's standard display mode. That means that all 12 digits of the answer are shown, unless there are trailing zeros, which are of course chopped off. For example, `1/4` gave .25 (not .250000000000), but `1/7` gave .142857142857 (not .14).

There are three other display modes available, if you want them. To make answers look like money, you can

```
fix 2
```

What this does is instruct the HP-71 to show two fixed digits after the decimal point, regardless of what they are. So 1/4 still shows 0.25, but 1/7 now shows 0.14 with the rest chopped off. Note well: this "FIX" display mode only affects the display! It does not chop anything off internally. It is just like the FIX function on HP, TI, CASIO, and other calculators.

To go back to the standard display mode, execute

```
std
```

This returns the display to normal. Also available are SCI and ENG, just like scientific calculators have. If you

```
sci 4
```

then answers are displayed in scientific notation with four digits after the decimal point. So too,

```
eng 3
```

sets the display to show all answers in engineering notation (same as scientific notation but the exponent is always a multiple of three), with three digits after the first digit.

Notice that these display modes and the contrast you set all keep working in CALC mode. But to change them, you have to be in BASIC mode.

Until we get to the chapter on PROGRAM BASIC mode, I will assume that you are trying all the examples given, right on the keyboard. We can't begin writing BASIC programs until we know how to speak KEYBOARD BASIC. The next several chapters will show you how to use KEYBOARD BASIC to its utmost.

The lesson of this chapter has been that you're in KEYBOARD BASIC mode when you have to type commands in BASIC and press `[ENDLINE]` to get anything done. Next chapter, we'll look at the fundamental rules of BASIC grammar.

CHAPTER 4:

BASIC VOCABULARY

Unlike English, which has a zillion types of words (nouns, pronouns, verbs, adjectives, adverbs...), HP-71 BASIC has only four types of words. These four types of words are:

(1) STATEMENTS (or COMMANDS). When you type `BEEP`, you are commanding the HP-71 to emit a sound. You aren't looking for an answer to something; you just want to tell the machine to do something internally. All words and expressions that tell the HP-71 to do something, without giving a number or string back to you, are called "statements". I often call them "commands" in this book. The two terms mean the same thing. If an analogy to English helps, you may think of statements (or commands) as VERBS; verbs in the imperative mood: Hey, you! BEEP! Sometimes statements allow numbers after them to modify the action of the command. For example, BEEP 1000 emits a beep twice as high as the normal BEEP. Notice that the 1000 is not in parentheses. Statements never require parentheses.

(2) FUNCTIONS. When you type `SIN(45)`, you are looking for the answer of a math problem: the sine of 45°. The SIN part is the function; the 45 part is the argument of the function, and the parentheses show where the argument starts and stops. So `SIN(45+45)` is very different from `SIN(45)+45`. Using our English analogy, functions are like adjectives, with their arguments being like nouns. When I say "blue house", Blue is an adjective operating on the noun House. The word "blue" is a function that turns its argument blue! SIN is an adjective that modifies the noun 45 into a sine.

Functions that have arguments always require parentheses. Some

functions (like PI, DVZ, RES, etc.) do not need any argument and therefore do not use parentheses.

(3) POSTFIXES. When you type `CFLAG ALL`, you are commanding the HP-71 to clear all the user flags (0 thru 63). The word CFLAG is a statement, as discussed above. The word ALL is not a statement, because by itself it means nothing. Try typing `ALL` by itself and see what happens; it's not a valid expression. It is allowed after CFLAG because it modifies the CFLAG command. Since these type of words always come after statements, I call them "postfixes" throughout this book. In English we modify verbs with adverbs; in BASIC we modify commands with postfixes. CFLAG ALL in English: Hey you! Clear the flags, completely! This analogy is not perfect, however. In English, adverbs can modify verbs and adjectives. In BASIC, postfixes only modify statements, not functions. So don't get hung up on the analogy between postfixes and adverbs!

(4) OPERATORS. The "operators" (`+` , `–`, `*`, `/`, `&`, etc.) are akin to prepositions in English. Like prepositions, they never come at the end of the sentence, and may begin short or complicated phrases. Their use is almost identical to the normal way we write algebra, e.g. 1+2*(3−7) which translates into English as "One with two threes without sevens", sort of.

Unfortunately, as in English, some BASIC words can be used in more than one way. OFF by itself is a statement, but BEEP OFF uses OFF as a postfix! ANGLE is both a function (as in ANGLE(x,y)) and a postfix (as in OPTION ANGLE DEGREES)!

So don't memorize which words are statements, which words are functions, which words are postfixes, and which words are operators. Their use determines what type of words they are. Some words allow more than one type of use.

All these types of words are familiar to you. There is only one that I've been keeping a secret. It is the "at" (@) symbol. It exactly parallels the words "and then" in English. Suppose I want to tell the computer "BEEP, and then BEEP 1000". Rather than typing it in two lines, I can just:

```
BEEP @ BEEP 1000
```

The "@" symbol "concatenates" (joins) the two commands into one long command. You may use @ as often as you wish; you can type commands all strung together up to 96 characters in length. But HEED MY WARNING!!! The @ symbol is just as dangerous as "and then" is in English! Do you know people who tell you a story and then tell you another one and then they bring in some unnecessary details and then they mention how their aunt's doing and then they tell another story and then and then and then? Conversation with people like that is frustrating at best. Run-on sentences have caused more headaches, loss of temper, broken marriages and murder than all other causes combined! If you start talking to your HP-71 that way, be careful! It might not turn on one day!

Seriously, don't overuse @. It's nice. It's even necessary for some things, like keyboard FOR-NEXT loops (more on that later). And it lets you load a whole ton of commands into the command stack! But don't get into the habit of pressing @ when you should press [ENDLINE]. Otherwise, your programs will be written the same way, and that makes debugging them more difficult. More on this when we get to the chapter on PROGRAM BASIC mode.

In this chapter we saw the rules of BASIC grammar. Next we'll see how to use BASIC to manipulate variables in ways not possible in CALC mode.

CHAPTER 5:

VARIABLES IN KEYBOARD BASIC

In CALC mode, we saw how to assign values to variables. For instance, we typed `W=SQR(22*2-18*2)` to place 12.65 into N, the height of the Fonch's fallen rainspout against the bank wall. This also works in BASIC mode. Matter of fact, variables keep their value between BASIC and CALC mode! If you set a variable in one mode, you can use that value in the other mode by using that variable. This is one way to have CALC and BASIC overlap.

But variables are more powerful in BASIC mode than in CALC mode! You can do things in BASIC that you just can't do in CALC mode. For example, in BASIC you can tell the HP-71 to consider X not to be just one number, but to be the name of a whole list of numbers (say 100 numbers) all hiding under the name X. Or you can clear all your variables in one fell swoop! You can dump variables into a "file" in memory, safe from harm, so that later on you can load the variables back from the file (like a built-in disk drive!). And best of all, you can put English words or any other text (not just numbers) into variables too!

DECLARING VARIABLES

In CALC mode, we have been using variables to stand for numbers. These variables, like the PI function, are always 12 digits long. So if you key `X=1/3`, and then key X, you see that X was set to .333333333333, which isn't really ⅓; it's just the first 12 digits of ⅓ (which is as close to the real value of ⅓ as the computer can handle). If you key `X=1/4`, then X is set to .25, which is still 12 digits; the last ten digits are zeros!

So X contains 12 digits. This is the maximum precision that a variable can have on the HP-71. But variables can have less precision if you want them to. If you want a variable to have its full 12-digit precision, then you should tell the HP-71 so. This is called "declaring" the precision of the variable. The way to declare 12-digit precision for the variable X is to type:

```
real x
```

The reason we call it "real" is because as far as the computer is concerned, 12-digit precision gives you the real answer. It doesn't care that it chops off everything after the 12th digit; it did the best it could! So it calls the number a real number. (Later we'll discuss the TRAP and FLAG functions that let you discover exactly when the HP-71 did or did not chop something off past the 12th digit. For most people, 12 digits of precision is more than enough!)

But suppose that 12 digits is too much for you, and you hate seeing all those digits all over the display! Let's say you just want 5-digit precision. You have two options.

One option is to change the display mode to FIX 5, which shows everything with 5 digits precision. This method has two drawbacks. First, it shows everything with 5 digits, even when there are trailing zeros, and that can really get annoying. Second, FIX 5 is only a display mode; inside the machine, the numbers are still 12 digits long, and so what you see is not what you've got, and that can get confusing. For example, in FIX 5 mode, setting X to `1/3` sets X to .333333333333. Now, `3*X` shows as 1.00000, but `RES-1` (which seems like it should be 0) gives −1.00000E−12, which is scientific notation for −0.000000000001. The reason for this "error" is obvious in STD display mode.

The other option is to actually force X to have only 5-digit precision, this way:

```
short x
```

After executing this command, X will be set to 0. Try it, and then set `X=1/3`. Notice that X is not set to the usual .333333333333 (twelve digits) but to .33333 (five digits). This is not a display mode; X was actually set to only the first five digits of ⅓! Now (in STD mode) our problem of `3*X` looking like 1 is solved. `3*X` gives .99999, which is clearly not 1. (The round-off error, almost invisible in REAL precision, becomes more obvious in SHORT precision.)

There is one more type of precision, called INTEGER precision. If you type

```
integer x
```

then X can't hold any fraction digits at all, and can only be integers (whole numbers) from −99999 to +99999. So if you now set `X=1/3`, X contains only 0 (because 1/3 is closer to 0 than to 1). If you set `X=2/3`, however, X becomes 1 (because 2/3 is closer to 1 than to 0).

Important note! You can change the precision of variables any time, but remember three things if you do. First, whenever you change a variable's precision by declaring a new precision, its old value is lost, and its new value is always 0. Second, changing one variable's precision effects only the precision of that one variable, and does not effect anything else whatsoever! For example, if you set a short variable equal to ⅓, then RES gets the full 12-digit value of ⅓, even though the variable only stores 5 digits of it. Variables can have different precisions, but the HP-71 calculations (like the PI function) are always carried out to 12 digits. (Actually, if you care, calculations are carried out to 15-digit accuracy inside, and sometimes to even greater precision, but when the calculating is done the answer gets rounded to 12 digits, so you might as well think of it as 12 digit precision!) Third and most important of all, variables keep their precision (and value) when you enter and exit

CALC mode. So be forewarned (since there is no mention of this in the handbook) that the precision of your CALC calculations depends on the precision of the variables you use! You cannot expect `X=2/3` to work right if X is an INTEGER variable.

Quiz. After keying the following commands, what would the display show in STD mode? And why? Now try it.

```
short x
 x=1/3
 1/3-x
```

DIMENSIONING VARIABLES

So far we've been using X to stand for just one number (albeit a variable number). But suppose you want to store a whole list of numbers in the computer; say, the first-class USA postal rates per ounce. You don't have to use a mess of different variables, like this:

```
A= 20
B= 37
C= 54
D= 71
E= 88
F=105
G=122
 ⋮=⋮
```

You may do it this way if you wish, but there's a better way.

In mathematics, we often use expressions like P_1, and P_2, and P_3, to stand for different numbers. Matter of fact, you can even say P_n to stand for the "nth" number in the "P" list. (In this example, P_3 is the postage for 3 ounces; P_n would be the postage for n ounces.) Such notation is called "subscript notation" because the little

number is written below the variable. And a list of numbers under one name like this is usually called an “array”.

In BASIC, we also have subscripted variables, and they are called “arrays” too. But before you can use an array, you have to declare it. That way, the computer can set up the memory needed to store all the numbers you need. How much memory it takes depends on how long your array is, and that is up to you.

Suppose you need an array containing the seven numbers above. First decide what precision you need. Since all seven are integers, we might as well use integer precision (it saves memory too). The way to declare an integer array called P containing seven numbers is:

```
integer p(7)
```

The number in parentheses after p tells the computer how many numbers will be in the array. Similarly, you type the subscript numbers in parentheses too:

```
p(1)= 20
p(2)= 37
p(3)= 54
p(4)= 71
p(5)= 88
p(6)=105
p(7)=122
```

So if you want to find the postage for 3 ounces, you just type `p(3)`, and there it is: 54 cents. So to find P_4 you just type `p(4)`. And guess what! You can even do what they do in mathematics; you can refer to P_n by typing `p(n)`. Try this: `n=5`, and then `p(n)`. You should see 88, because P_n=88 when n=5.

Some strange people aren’t satisfied with just lists of numbers in a

line, called arrays; they want a whole box of numbers, called a "matrix" (pronounced MAY-tricks). For example, they might want to store this arrangement of numbers

$$\begin{bmatrix} 1 & 2 & -3 & -4 \\ 2 & 3 & -4 & 5 \\ 5 & -6 & 7 & -8 \end{bmatrix}$$

into a single variable, as in heavy mathematics. Since it's a matrix, let's call it M. In math, we'd say the number in the first row and first column is $M_{1,1}$=l, and the number in the third row, fourth column is $M_{3,4}$=−8. Guess what! The HP-71 works with matrices just like with arrays! All you have to do is specify how many rows and how many columns you want. Since matrix work usually needs great precision, let's create a real-precision 3 by 4 matrix and call it M:

```
real m(3,4)
```

To put the values into M, it's just like with the array:

```
m(1,1)= 1
m(1,2)= 2
m(1,3)=-3
m(1,4)=-4
m(2,1)= 2
m(2,2)= 3
m(2,3)=-4
m(2,4)= 5
m(3,1)= 5
m(3,2)=-6
m(3,3)= 7
m(3,4)=-8
```

what is $M_{2,3}$? Just type `m(2,3)` and see −4 displayed.

If a variable with one subscript is called an array, and a variable with two subscripts is called a matrix, what is a variable with three subscripts called? It's not called anything, because the HP-71 can't do it. But notice that HP books refer to the number of subscripts as the "dimension" of the variable. Arrays are one-dimension variables, and matrices are two-dimension variables. So another term for "declaring" an array or matrix is "dimensioning" it. Matter of fact, `REAL X(100)` and `DIM X(100)` are identical commands! You can use DIM in place of REAL if you like. Never forget that DIM does more than just dimension the variable; it also declares its precision.

But what's a regular old, plain, simple variable without any subscripts called? If X=1/3, X isn't an array or a matrix. It's called a "scalar" variable (pronounced SKAY-ler). Scalar variables contain a scalar number: one single number without fringe benefits. Scalars are what most mere mortals need most of the time. If you need an array once in a while, or even a matrix, it's nice to know that the HP-71 handles them easily.

By the way, if you buy the HP-71 MATH PAC ROM, it'll let you work with arrays and matrices in fantastic ways. It lets you add arrays in a single stroke; you can multiply matrices in a single command; invert a matrix in a single command; solve systems of equations in a single command; perform finite fourier transforms, solve for the roots of functions, and more, all in one swell foop. If you are a serious user of arrays or matrices, you might consider buying the Math Pac, because of the time it saves in array processing.

Note: if you try to use a variable in a way contrary to its declared type, you'll get an error message. If P is an array, you can't say `P=4`. If you do, you'll see the "Data Type" error displayed. If you want to clear an array or matrix out of memory entirely, use the DESTROY command. To zap our P array and M matrix, type:

```
destroy p,m
```

The DESTROY command doesn't just clear a variable; it completely destroys it! The variable is no longer a matrix, array, scalar, or anything at all. It no longer exists. If your arrays and matrices start gobbling up memory, you may have to DESTROY one or two of them to avoid getting the dreaded "Insufficient Memory" error. If you are really hard up for memory, you can blow away all of your variables at once by this command:

```
destroy all
```

Even though this command sounds ominous (the only one more frightening is END ALL, discussed later), don't hesitate to use it, because all it clears is your variables. It doesn't destroy anything else!

COMPLEX DATA TYPE

The HP-71 was designed with five basic variable types. We've already seen REAL, SHORT and INTEGER types (all of which permit subscripts). The fourth variable type is called COMPLEX, and represents "complex" numbers like 3–4i (where "i" stands for the square-root of negative one). Unfortunately, when the design team at HP finished writing the operating system for the HP-71, it was 80K long, and they only had room for 64K. So they had to chop out some features from the operating system and put them into optional ROM's that you can buy. One such feature that got the axe was the complex data type. The HP-71 can have complex variables, but unless you have the MATH ROM you can't do anything with them. Matter of fact, the command that declares variables to be of complex precision is itself inside the MATH ROM! So if you need complex (or imaginary) numbers, buy the MATH ROM, which handles them even better than the HP-15C does.

As mentioned before, don't forget that variables keep their precision when you enter and exit CALC mode. They also keep their dimensions. If you need an array in CALC mode, just dimension it in BASIC and go back into CALC. If you have the MATH ROM, you can even use COMPLEX variables in CALC mode!

Final note: if you "re-dimension" an array, that is, if you declare it to have the same precision but a different number of elements, then the array is NOT automatically filled with zeros! It is merely shortened (in which case some values are lost) or lengthened (with zeros). The same goes for matrices; if re-dimensioned, the values in a matrix are re-arranged (row by row, not column by column) but not cleared automatically. There are no restrictions on re-dimensioning arrays or matrices. If the precision is changed, however, all the values are cleared. Try this with a few small arrays and matrices to get the hang of it.

STATISTICS

Nothing significant is said in this book about the built-in statistics functions (STAT, CLSTAT, ADD, DROP, TOTAL, MEAN, SDEV, CORK, LR, and PREDV) because the HP-71 Owner's Manual covers the topic so well. If you need the stat functions, read the Owner's Manual and do all the examples there. They even describe how to fit data to various curves, with examples!

STRING VARIABLES

All the variables discussed above have one thing in common: they store numbers. But what if you want to store your name, or an English word, into a variable?

When you store a number into a real-precision variable, you are

storing how many digits? 12. Always. Even if you just store the number 3 into the real variable X, X still contains 12 digits (11 of which are zeros). You already know this. X is just one number, containing 12 digits.

But if you want to store, say, the word “Hello” into a variable, you don’t want to store digits at all. You want to store five letters all strung together. Since we don’t want a number but a “string” of characters, the variable that stores it is called a “string variable”.

String variables look like regular (numeric) variables except that they have a dollar sign ($) after them. For instance, try this:

```
N$="Hello"
```

N$ (pronounced “N string”) is the string variable, and we are setting it equal to the string “Hello”. To see what N$ contains, do like we’ve been doing for numeric variables; just type the name of the variable and press `[ENDLINE]`.

You may be puzzled that we set N$ to “Hello” including the quotation marks, but when you look at the contents of N$ it merely shows Hello without any quotation marks around it. The quotation marks in our original assignment statement, `N$="Hello"`, are just there to tell the computer that Hello is a string, not a number. If you type `PI`, you get the number 3.14159265359. But if you type `"PI"`, you get the word PI. Try it.

DECLARING STRING VARIABLES

If you don’t declare the precision of a numeric variable, the HP-71 assumes you want REAL precision. In a similar manner, the HP-71 assumes that your string variables should contain 32 characters or less. You can type `N$="Hello"` without dimensioning N$ because “Hello” is less than 32 characters long. But if you were to type this:

```
N$="These are times that try men's souls"
```

you would get a “string overflow” error because N$ cannot hold more than 32 characters. If you want a string to hold more than 32 characters, you must dimension it. If you want N$ to hold up to 100 characters, type:

```
DIM N$[100]
```

Notice the square brackets. Square brackets in the HP-71 refer to the position of a character in a string. So DIM N$[100] means “allot memory for N$ up to its 100th character”. If you need an array of strings, just remember that the dimension goes first, inside parentheses, and the string size goes second, inside square brackets, like this:

```
DIM N$(7)[100]
```

which sets up memory for 7 strings containing up to 100 characters each. They are referred to as N$(l), N$(2) and so on. The HP-71 does not support string matrices, but if you need them we’ll see a way of faking them when we get to the chapter on PROGRAM BASIC. Quick! What does `DIM N$[96],X,Y,Z` do? Right! It declares N$ to 96-character length, and declares X, Y and Z to REAL precision.

SPECIAL STRING FUNCTIONS

Functions that operate on numbers are called numeric functions (surprise!), and functions that operate on strings are called (let me hear you, all together now) string functions! There are a lot fewer string functions than numeric functions. No wonder! The HP-71 is primarily a number-crunching machine. It is happiest taking the SQR(X) or FACT(253). It becomes unhappy when asked to take the SQR(“Fred”) or FACT(“Hello”). Since these are not valid

expressions, the HP-71 says so. Try it. Notice too that CALC mode is strictly for number crunching; it does not allow any strings or string functions.

One very useful string function is LEN. It gives you the length of the string specified. For example, try:

```
len(n$)
```

which gives you the length of n$, which should be 5 because n$ is equal to "Hello" which is 5 characters.

The numeric functions discussed before all have one thing in common: you may use a numeric variable as their argument, e.g. SIN(Y), or you may use a literally spelled-out number, e.g. SIN(17.7). The same is true of string functions. You may use a string variable, e.g. `LEN(N$)`, or you may use a literally-spelled out string (inside quotation marks), e.g. `LEN("Hello")`. A literally-spelled out expression, whether numeric or string, is called a "literal". An example of a numeric literal is 1537. An example of a string literal is `"Hello"`. You may use either the double quotation mark (`"`) or the single quotation mark (`'`) to delineate literal strings, but don't get them mixed up! If you start a string with double quotes, end it with double quotes: if you start it with single quotes, end it with single quotes. The HP-71 sometimes seems to change double quotes into single quotes (we'll see examples of this in key assignments and in program labels) but you may type either one as you please.

Note to philosophers: notice that PI is neither a variable (it cannot vary) nor a literal (it contains no digits). It is a function. A philosopher of mathematics would say "That is a contradiction in terms; functions have to be functions OF something, but PI has no argument and always gives the same value, and therefore is not a function but a CONSTANT." I agree whole-heartedly. PI should be

called a constant, but it isn't. It is called a function. This solves the dilemma of what to call similar functions, like RES. RES acts like a constant (it gives a value, and has no arguments), but it does vary. It can't be called a variable, though, because the proper way to assign values to variables does not work with RES (try `RES=12345` and see what happens). So we just call it a function.

Speaking of RES, notice that RES is never a string. The value of RES is changed every time a numeric variable is assigned a value (or whenever a value is displayed or printed). RES is never changed from a string operation.

SQUARE BRACKETS and SUBSTRINGS

Type these commands:

```
DIM N$[96]
N$="These are limes that try men's souls"
N$[11,24]
```

What do you see? You should see three words. The square brackets at the end of the string are special modifiers that tell how much of the string to use. The 11 means "start at the 11th character" and the 24 means "stop at the 24th character". So we just got part of N$, from the 11th to 24th character. A small string taken from a bigger string is called a "substring".

If you only use one number, it is taken to be the "start" number only. Using the command stack so that you don't have to retype it, try this expression:

```
N$[11]
```

Here's the best part about substrings. Suppose you don't want "limes" in that string but "times". Using substrings, you can change part of N$ into something else:

```
N$[11,11]="t"
N$
```

Now you see “These are times that try men’s souls”. Try this:

```
N$[ll,ll]="the T"
N$[26,26]="d"
N$[38,39]="le"
N$
```

You should see a new sentence, about the poor man’s wet weather doormat!

Although not mentioned in the HP literature, notice that square brackets can modify not just strings but even substrings! Try this:

```
N$[LEN(N$)/2][3,5]
```

This means that you want the 3rd through 5th character of the second half of N$, which is “hat”. Since this involves two substring concepts (“second half” and “3rd through 5th character”), doing it with double square brackets is logical. Doing it according to the HP books, we’d need

```
N$[LEN(N$)/2+2,LEN(N$)/2+4]
```

to do it! Note well, however, that assignment statements can only have one set of brackets on the left side of the equals sign. `N$[10,20]="XYZ"` is okay, but `N$[10,20][2,4]="!"` is not valid.

Here’s another tidbit not in the manual. You might wish to insert something into the middle of a string. To do that, specify a substring that starts where you want but has no length:

```
N$[10,0]="INSERTED TEXT"
```

To insert text at the very beginning of a string, you may use N$[1,0] or N$[0,0], whichever you prefer.

Quiz! Suppose I$="123456789". What is I$[3,6]? What is I$[6]? What does `I$[3,7]="-"` do to I$? Guess. Try it.

Note to the curious: The HP-71 is consistently algebraic in CALC mode; all functions come before the number(s) involved, and all operators come between. The same is true for all HP-71 functions, even the string functions... except for the square brackets. They always come at the end. If you are used to other BASIC's way of doing substrings (with functions like MID$, LEFT$ and RIGHT$), square brackets may take a bit of getting used to. If you've never worked with BASIC before, you'll find square brackets easy to use.

AMPERSAND (&)

As square brackets allow us to chop up strings into smaller strings, the ampersand (&) allows us to join strings together into longer strings. For example,

```
"ABC"&"123"
```

gives a result of "ABC123". The "&" can be thought of as an "and"; it tacks the two strings together. Some people think of the "&" as a "+", as if it "adds" the two strings together. This can be misleading. When you add 1+2, you don't get 12; but `"1"&"2"` gives `"12"`. Rather than +, the & symbol is more like the @ symbol, the statement concatenator, which joins multiple statements together in one line. So we'll call "&" the "string concatenator".

Notice that you can combine square brackets and ampersands in amazing ways. For example,

```
"ABCDEFG"[2,6]&"1234567"[3,5]
```

gives “BCDEF345” because we concatenated the 2nd through 6th characters of “ABCDEFG” with the 3rd through 5th characters of “1234567”. But notice that

```
("ABCDEFG"[2,6]&"1234567")[3,5]
```

gives “DEF” because that’s the 3rd through 5th characters of `"ABCDEFG"[2,6]&"1234567"`. Using parentheses like this is the only way to have square brackets refer to more than the immediately previous string. Remember the algebraic hierarchy we learned for the mathematical functions? Multiplication “comes before” addition. Same here; square brackets “come before” ampersands in string evaluation. I guess we could call it “string function hierarchy”, but nobody else does, so I won’t. Just remember that parentheses are more important than brackets, and brackets are more important than ampersands.

CHR$ and NUM

CHR$ (“Character string”) is a handy function that changes a number into a single-character string. For instance, try this: `CHR$(74)&CHR$(72)`. You should see “JH” in the display. That’s because the 74th character that the HP-71 knows is “J”, and the 72nd character is “H”. These numbers are called the ASCII (pronounced “ASK-ee”) codes of their corresponding characters. When the HP-71 stores a string, it actually is storing a string of ASCII codes; when you display the string, it converts each code into the appropriate character. I did not include an ASCII code table in this book because an excellent one comes with your HP-71, on pages 41-45 of the HP-71 Quick Reference Guide (“Pocket Guide”) which you ought to keep in the 71’s carrying case at all times.

You may object that you can get “JH” by typing “JH”, so what do

you need CHR$ for? You need it for characters that you can't type. For example, the backslash character (\) is not on the keyboard. It is ASCII code 92. If you want to display it, just use CHR$(92).

So CHR$ takes an ASCII code number and gives you the corresponding character. The inverse function is called NUM ("Number"). NUM takes a character and gives you its ASCII code number. For example, `NUM("J")` gives 74, because the ASCII code number for "J" is 74. Notice that upper-case letters have different codes than lower-case letters. NUM only looks at the first character of its argument, so you can send it long strings. For example, `NUM("Jack Horner")` gives 74 too. Everything after the first character (J) is ignored by NUM.

You can use CHR$ and NUM to pack a lot of numbers into very little memory. For example, if you have 100 numbers to store, ranging from 0 to 255, then you can store them all in a string as ASCII characters. This takes only a fraction of the memory that would be required by an array.

Example: Let's do it with an array first, then see how it's done with a string. To make an array named "A" of 100 integers, you would type `INTEGER A(100)`. To set the 17th one equal to 153, you'd type `A(17)=153`. To get the 17th one back out again, you'd just type `A(17)`. Now let's do it with a string. To create a string named "A$" 100 characters long, you type `DIM A$[100]`. To set the 17th character equal to ASCII code 153, you type `A$[17,17]=CHR$(153)`. To get the ASCII code for the 17th character back out, you type `NUM(A$[17])`.

STR$ and VAL

Have you noticed that every time you display a number, the HP-71 adds a space onto the front and back of it? Try this:

```
X=153 @ "Price = $";X
```

What we want, of course, is

Price = $153

but that's not what we get. Instead, we see

Price = $ 153

with a space between the $ and the 153. We can get rid of the space by using the STR$ ("String string") function. The STR$ function takes a number and turns it into a string that looks exactly like the number as it would be displayed, except that it does not add those dumb spaces! So type

```
"Price = $";STR$(X)
```

and you'll see

Price = $153

Using STR$ just to eliminate spaces, however, is like using the HP-71 in just CALC mode. It's okay, but a waste of power! The real power of the STR$ function lies in the way it allows you to use numbers as strings. Many functions only work on strings, so if you want to use that function on a number, use STR$ first. For example, the LEN function only tells you the length of strings, not the length of numbers. Suppose you want to find out how long the display of X would be. You can display it and count the display positions. Or you can use `LEN(STR$(X))`. Try it. This principle applies to all string functions; to use them on numbers, just use STR$ first, to turn the number into a string.

The opposite function is VAL ("Value"). VAL takes a string, and

turns it back into a number. For example, type `A$="153"`. Now suppose you want to get the number 153 back out of A$. You can't use NUM; that just gives ASCII code number of the first character, which we don't want at all. Use `VAL(A$)`; this gives us 153 back. But VAL is a lot more powerful than that! It can also evaluate entire mathematical expressions. For example, if `A$="1+2*3^4"`, then `VAL(A$)` is 163, same as if you typed `1+2*3^4` in CALC mode!

But here's the real clincher of the VAL function. You can send it strings that contain variables, and it evaluates them correctly! For example, set `A$="PI*R^2"`. That's the formula for the area of a circle with radius R. To find the area of a circle with radius 7, just set `R=7` and then type `VAL(A$)` to see 153.938040026, the area. To use long, complicated strings with lots of variables, be sure that you `DIM`ension the string first. This ability of VAL to evaluate expressions is the heart of some of the most powerful programs written for the HP-71.

Note: even though we have not yet discussed programming, we have seen two ways of "programming" the HP-71 to do repetitive calculations. The first method is in CALC mode, and uses the command stack. Expressions using variables are evaluated in CALC mode, then the variables are changed, and then the expression is recovered from the command stack and re-evaluated. The second method was just discussed; expressions are placed into strings, and evaluated with the VAL function, then the variables are changed, then the string is re-evaluated with the VAL function.

Of these two methods, CALC is easier for one or two functions. But the command stack is not high enough if you have many functions to be evaluated. In this case, use the string and VAL method. There is practically no limit to the number of string expressions you can have in memory at once. For example, you can store one expression in A$, another in B$, and so on. The trick will be

remembering the name of each one! You can always look at A$, of course! All you have to do is type `A$` and press `[ENDLINE]`.

POS, UPRC$, VER$

There are only three more string functions, and they're unrelated to each other. The first is POS ("Position"), and is used to find substrings inside longer strings.

Type `A$="These are times that toy men's souls."` Notice the typo: it should be "try", not "toy". We have already seen how to change the "o" into an "r", using square brackets, but it requires knowing where the "o" is inside the string. Doing it by hand is a payne. So let's have the computer do it! Type `POS(A$,"o")` and you'll see 23. This means that "o" is the 23rd character in A$. POS looks for the string you specify, and tells you where it is first found. Luckily, the "o" we want is the first one in the string! To change it to "r" as it should be, type:

```
A$[RES,RES]="r" @ A$
```

I used RES because RES's memory is better than mine! Hint: get used to using RES a lot. It's a handy function. Just as good HP-41 owners use LASTx all over the place, good HP-71 owners should use RES a lot. I have yet to see a program that puts RES to significant use. This will no doubt change as '71 users become more experienced.

Now, suppose you want to find not the first "m" in a string, but the second? There are two ways of doing this. You can specify just a portion of the string in the POS function by using square brackets; `POS(A$[15],"m")` looks for "m" in just the 15th through last characters of A$. The other way is by having POS itself specify which portion of the string to search; `POS(A$,"m",15)` searches

for “m” in A$ starting at the 15th character of A$. So to find the second “m” in A$, type `POS(A$,"m",POS(A$,"m")+1)`, which should give 26, because we told the computer to find the first “m”, add 1, and then search for “m” starting at that position.

Never forget that functions don’t have to have timid numbers as their arguments; they can be awesome expressions of any complexity you can dream up! As your BASIC expertise increases, you’ll actually find it easier to use nested arguments like this, just as in English we easily use grammatical structures that are no doubt difficult for the foreign student to understand. Thank goodness that the HP-71 speaks BASIC better than any of us ever will!

Gloat note: The HP-41 has limited ASCII string ability. If you create an ASCII (TEXT) file in its extended memory consisting of a single record containing 253 “A”s followed by a single “B”, and then use the POSFL function to search for 23 “A”s followed by a single “B”, it takes the HP-41 one minute and 7 seconds to find it. The HP-71 eats strings for breakfast. Using the POS function to find 23 “A”s followed by a “B” in a string consisting of 253 “A”s followed by a “B” takes less than three tenths of a second! If you are an HP-41 graduate, don’t worry about string functions taking a long time on the HP-71. They don’t. They’re even faster than string functions in Microsoft® BASIC as found on most home computers from the TRS-80® to the IBM® PC.

Another handy string function is UPRC$ (“Uppercase String”). If you want to see all the lowercase letters in A$ in uppercase (capitalized), just use UPRC$(A$). Note well: UPRC$ does not change its argument; it merely returns the uppercase version of it. If you want to change A$ itself into uppercase, you have to type `A$=UPRC$(A$)`. This applies to all functions; they don’t change the argument, but merely return something based upon the argument.

Suppose we want to find the first “e” in a string, regardless of whether it’s upper or lowercase. Here’s how you do it:

```
POS(UPRC$(A$),"E")
```

which looks through the uppercase version of A$ for “E”. Notice that A$ itself isn’t changed to uppercase! This method is often used in HP-71 programs.

One more string function: VER$ (“Version String”). Type VER$ and see what it gives. If you see only HP71:1BBBB, then you have the basic, bare-bones, original HP-71 with nothing plugged into the ports. (If you see other things after the HP71 version string, it indicates that you have other LEX files plugged into ports or floating around in RAM, and you can probably guess by looking at VER$ what they are!) The “BBBB” stands for the version of the four internal ROMs. The first marketed version of the HP-71 was the BBBB version (if you have an AAAA version, you do not need to read this book!). An HP-71 bought later may have higher version letters. Later models of HP-71 (e.g. HP-71C) will certainly have higher version letters. For example, the HP-41C went through several versions, and then they came out with the HP-41CV which had more RAM inside, and then they released the HP-41CX which had more RAM and more ROM. My HP-41CX is the original CX, and its three ROMs’ version letters are NFL! So VER$ will become an important function as time goes by. HP will update your ROMs to newer version ones for a nominal fee.

Do you understand the string functions? Programs and routines and hints that utilize them all can be found after the chapter on PROGRAM BASIC.

LET

So far, when we wanted to assign a value to a variable, we simply

stated the equality. For example, to set X equal to PI, we simply said `X=PI`. There is another way of saying the same thing, and beginners find it less confusing:

```
LET X=PI
```

The command "LET" is a BASIC statement which means what it says, somewhat as in the sentence "Let the show begin!" or "Let them eat cake!" Personally, I think it should have been called SET. And it should rain beer. But it doesn't!

How would you add 1 to X? Of course, you'd type `X=X+1`. But such a statement would give a mathematician the shakes; how on earth can `X=X+1`? If you are a purist, you may type `LET X=X+1` instead, which makes more sense. Operationally, however, the two statements are identical. The LET command is always optional.

FOR / NEXT

Suppose you want to do a command several times. You have two options. The obvious solution is to go ahead and do it several times yourself. For example, to BEEP ten times, you may type `BEEP` and press `[ENDLINE]` ten times. Of course, you'd use the command stack to shorten your work.

There is a better way. All we need is a "counter" that counts from 1 to 10, and BEEPs along the way. You can use any variable as the counter. Let's use X, because it's my favorite variable. Here's the scenario. I tell the HP-71 to BEEP for all values of X from 1 to 10. The HP-71 then sets X to 1, BEEPs, goes to the next X (2), BEEPs again, goes to the next X (3), BEEPs, and so on, until it BEEPs for an X value of 10, then stops. Try this:

```
FOR X=1 TO 10 @ BEEP @ NEXT X
```

(The spaces here, as always, are optional.) Remember that the "@" symbol is the "statement concatenator" that strings commands together. We have three commands here. `FOR X=1 TO 10` means "Use X as a counter from 1 to 10." By itself, this would only set X equal to 1. But if followed on the same line by "NEXT X", the X will count from 1 to 10, and all the statements concatenated between the FOR and the NEXT will get executed along the way! Try changing the 10 to other numbers (use the command stack!). Try changing the BEEP to other commands (try just X).

This ability to "loop" from the keyboard is very useful, especially when you use the counter value itself. Suppose you wish to clear a portion of an array. Do you see why

```
FOR X=10 TO 20 @ M(X)=0 @ NEXT X
```

would clear M(10), M(11), M(12), ... M(20) to zero?

Besides specifying the starting and stopping place for the counting, you can even specify how much to count by:

```
FOR X=200 TO 2000 STEP 100 @ BEEP X @ NEXT X
```

sings a silly ditty. Try it and drive your cat mad. What was happening, of course, was the HP-71 BEEPing at a frequency of 200 Hz, then 300 Hz, then 400 Hz, and so on, up to 2000 Hz. "STEP 100" means "add 100 to the counter each time you go to its NEXT value." If no STEP is specified, as before, then the default step of 1 is used.

Sometimes an infinite loop is useful. Try this:

```
FOR X=1 TO INF @ TIME$ @ NEXT X
```

How about that! A running digital clock, and we didn't even write a program! If the display looks jumpy (some seconds lasting too long,

others too short) then type

```
delay 0,0
```

That tells the HP-71 to use no delay between things that are displayed. Normally there is ½ a second between displays. To set the display back to its normal delay, type

```
delay .5
```

The FOR / NEXT loop is one of the most powerful devices BASIC offers. There is no limit to the uses for it. Suppose you want to add the numbers from 1 to 100, and have forgotten the formula, and don't want to look it up, and don't want to bother adding them yourself. Try this!

```
T=0 @ FOR X=1 TO 100 @ T=T+X @ NEXT X @ T
```

This means "Clear the running total (T), then add the numbers 1 to 100 to the running total, then display the running total." In less than two seconds the total (5050) is displayed!

You can count backwards in a loop too! Just use a negative STEP. After typing `DELAY 1`, type the following all on one line and then press `[ENDLINE]`:

```
FOR X=10 TO 1 STEP -1 @ "T MINUS";X;"SECONDS" @
NEXT X @ "LIFT OFF!"
```

The only trouble was "1 seconds", but what do you want, good grammar or a good time? Can you figure out how to fix it to say "1 second"? An inelegant solution is:

```
FOR X=10 TO 2 STEP -1 @ "T MINUS";X;"SECONDS" @
NEXT X @ "T MINUS 1 SECOND" @ "LIFT OFF!"
```

Try inventing wild and crazy FOR / NEXT loops on your own. Most heavy users of variables inevitably wind up becoming heavy users of FOR / NEXT loops. They are not only useful time savers, they're also fun!

In this chapter we've seen how to use variables to hold data from the keyboard. In the next chapter we'll see another, more permanent way of storing data in the HP-71.

CHAPTER 6:

HP-71 DATA FILES

So far we've used the HP-71 as a calculator. It hasn't really kept much information in memory for us except in the form of variables. Admittedly, this can be a lot of information, especially if you have arrays or matrices floating around. But variables have two big drawbacks. First, their number is limited (I use X, Y, Z, J and K, and A$, B$ and C$ almost exclusively), and it's easy to wipe out the value of a variable by using it for something else. It's like a blank cassette that you record music onto, only to realize too late that it wasn't really blank. Variables have no "erase prevention tab" like cassettes do! Variables were designed to vary. Their second big drawback is that if you get a lot of them haunting your machine, and you want to DESTROY a few to recover the memory they're using up, there is no function that tells you whether a variable exists or not. You have to DESTROY ALL. And that wipes out ALL of your variables, whether they were important or not.

To solve this dilemma, HP provided us with a way of saving data in a more permanent block of memory. These blocks of memory are immune to the DESTROY command, and are called "data files". The HP-71 can be thought of as a filing cabinet, with each drawer being a file full of data.

Before we can put data into a data file, we have to first create the data file. It's sort of like dimensioning an array; we have to tell the computer to set aside enough memory for the data.

There are several kinds of files that the HP-71 can have, and each is for specific purposes:

FILE TYPE	INTENDED CONTENTS
SDATA	Numbers
TEXT	Strings
DATA	Numbers and Strings
KEY	Key assignments
BASIC	BASIC programs
BIN and LEX	Machine-Language programs
FORTH	FORTH-language programs

Let's cover these one at a time.

SDATA FILES

One of the most confusing chapters of the HP-71 Owner's Manual is Section 14, "Storing and Retrieving Data," all about how to use "files". After reading that section, I was convinced that the book was written by somebody who never used an HP-71! The book even implies that there are two kinds of files: sequential-access files, and random-access files, which is not at all true!

WHAT IS AN SDATA FILE?

"SDATA" stands for "stream Data" and is a file type designed for compatibility with the HP-41. But don't let this suggest that SDATA files are only for HP-41 owners! In fact, I use SDATA files more than DATA files, yet I've never placed HP-41 data into them! You may even find SDATA files the most useful file type the '71 offers.

HOW TO CREATE AN SDATA FILE

If you want to keep an array (say, a 100-element array called A) in a file, then an SDATA file is your answer. To create a 100-element SDATA file called MYDATA, use the command:

```
CREATE SDATA MYDATA,100
```

This creates 100 "records" numbered 0 through 99, and fills them with zero for you.

HOW TO ASSIGN A CHANNEL NUMBER

Rather than referring to your SDATA file by its full name, the HP-71 allows you to assign a number to it, and then refer to the file by that number. This handy "nickname" number is called the file's "channel number." Use television channel numbers as an analogy: you don't have to dial a station's full name (e.g. KNBC, WWTV, etc.), but can merely dial channel #2.

To assign a channel number to your file, use the command:

```
ASSIGN #1 TO MYDATA
```

From now on, you don't have to refer to "MYDATA" again, but can merely refer to channel #1. (Note: Other versions of the BASIC language have an "OPEN" command. In HP-71 BASIC, "ASSIGN" performs the same operation.) You can have more than one channel assigned at one time, each one to a different file.

HOW TO RECALL A RECORD

The 100 numbers in "MYDATA" (whoops; I mean in #1) are called "records", one number per record (the HP-41 calls them "registers", and the British call them "stores"). The records are themselves numbered sequentially starting at 0 (for the first record) on up. It would seem that the first record ought to be called record 1, but it isn't, and I can't tell you why. That's just life.

To read the value of any particular record, you must specify three things: the channel number of your file, the record number, and a variable that you wish to "catch" the record's value. For example,

```
READ #1,10;X
```

reads the value of record 10 (in channel #1) and stuffs it into the variable X. You can then PRINT X or do whatever you want with it. Don't worry; this does not "assign" X to record 10! Changing the value of X will not change the value of record 10.

HOW TO STORE A RECORD

Storing a value into a record is similar:

```
PRINT #1,10;X
```

"prints" the value of X into record 10, replacing whatever had been there. Note that this does not "insert" X into your file, like inserting a card into a deck. It replaces record 10's old value with the value of X. (We'll see an easy way of inserting/deleting records below). You don't have to use a variable; you can use a literal number instead of "X".

HOW TO FIND HOW MANY RECORDS EXIST

When PRINTing or READing records to/from an SDATA file, you may use any record number at any time, provided that it exists. This may seem obvious, but it isn't true of DATA files! DATA files can be a pain! As I said, SDATA files are very nice. If you forget how many records your SDATA file contains, an easy way to find out is to CAT it, and divide its byte length by 8. For example, if you type CAT MYDATA (sorry, CAT doesn't use channel numbers!), you'll see

```
MYDATA SDATA 800
```

in the display. This means that MYDATA is an SDATA file, and is 800 bytes long. Divide 800 by 8, and you get 100, the number of

records it contains.

I personally always reserve record 0 to contain how many records there are, and then use the other records for my actual data.

STORING/RECALLING MORE THAN ONE NUMBER AT A TIME

There are some tricks I haven't mentioned. If you want to store or recall more than one number at once, you can do it in a single command! For example,

```
READ #1,12;X,Y,Z,T
```

reads record 12 into X, record 13 into Y, record 14 into Z and record 15 into T! Likewise,

```
PRINT #1,12;X,Y,Z,T
```

stores X into record 12, Y into record 13, and so on. This is very handy when you have a handful of variables that you wish to store and recall as a group.

But even better is the ability to store and recall whole arrays in one fell swoop. Suppose you have the array of 100 numbers mentioned above, from DIM A(100). You wish to recall MYDATA into the entire array. All it takes is:

```
READ #1,0;A()
```

to read all 100 values! (The parentheses indicate that A is an array, but they are optional. I never use 'em myself.) The HP-71 recognizes A as an array, and simply keeps reading from channel #1 until the array is filled. Of course, if the array is bigger than the file, you'll get an error message. By the way, you can mix array variables and regular variables in a single READ command, like `READ #1,15;J(),K,G(),P(),Q`. Just make sure you don't run off

the end of the file!

PRINTing arrays is just as easy:

```
PRINT #1,0;A()
```

stores all of array A into the file, starting at record 0. Again, the parentheses are optional. If you wish to store or recall a matrix, put a single comma between the parentheses like this (,) to indicate a matrix. Or you can omit the parentheses altogether, which I find less confusing.

Since an SDATA file is simply a linear list of numbers, you can store an array and recall it as individual numbers, or mix & match however you like. The fact that you stored single numbers, or arrays, or matrices, is remembered only by you. The file only contains the numbers themselves.

SEQUENTIAL ACCESS

So far, when PRINTing or READing to/from an SDATA file, we've been specifying the record number desired. This is called "random access" to the file, because we are able to access any record we randomly desire. But notice something!

When we `READ #1,12;X,Y,Z,T` we didn't tell the '71 to place record 13 into Y; it did it automatically. We specified record 12, which placed the "file pointer" there. After reading record 12 into X, the '71 automatically moved the pointer forward one record, and thus read record 13 into Y. And so on, until it read record 15 into T, and once again moved the pointer forward one record. So after this command is finished, the pointer is sitting on record 16. Suppose we then give the command:

```
READ #1;L
```

What do you think will happen? We didn't specify which record to use! But the pointer is on record 16, so of course record 16 gets read into L, and then the pointer is moved ahead to record 17. This is called "sequential access" because it accesses the records sequentially, one after the other. Notice that there is no such thing as a sequential file or a random-access file! Any file can be accessed randomly or sequentially.

HOW TO MOVE THE FILE POINTER

To place the file pointer wherever you want it, use the RESTORE command. To point at record 57, for example:

```
RESTORE #1,57
```

And from there you can access the file sequentially to your heart's delight. But you can't READ past the end of the file, of course! If you try, you'll get an error.

HOW TO EXPAND A FILE

BUT!! Here's the magic of sequential access. You can PRINT sequentially past the end of the file! Suppose we do the following:

```
READ #1,99;X
```

That reads record 99 (the last record) into X, and moves the file pointer to record 100 (which doesn't exist). Then if we

```
PRINT #1;PI
```

the '71 performs a miracle! It does not give an error! It actually expands the file to 101 records long, and places the value of PI into record 100 (and moves the pointer to record 101)! Now if you CAT MYDATA, you'll see that it is 808 bytes long. All the hassle of memory shifting etc. is automatically done for you! PRINTing

more than one number (as a list or as an array) also works past the end of the file!

Notice, however, that RESTORE is a random-access type of animal, and so you cannot RESTORE the pointer to a non-existing record. The only way to set the pointer "above" the file is by READing or PRINTing the highest-numbered record. This makes sense, since this is normally done by a previous sequential PRINT.

HOW TO INSERT/DELETE A RECORD

When you combine the ability to expand a file with the ability to PRINT a whole array at once, you get the ability to insert a new record into the file. Suppose our file is 100 records long, and we wish to insert a new record, X, between records 52 and 53 (thus becoming the new record 53, and raising the old record 53 up to record 54 and so on). All it takes is:

```
DIM T(100-53)
READ #1,53;T
PRINT #1,54;T
PRINT #1,53;X
```

which takes about 0.7 seconds to execute! No need for a special INSERT command! READING an array and PRINTING it somewhere else allows not only insertion, but also deletion, rotation, and more!

HOW TO DE-ASSIGN A CHANNEL NUMBER

To "close" a file and de-assign its channel number, you may use the ASSIGN command like this:

```
ASSIGN #1 TO ""
```

This assigns channel #1 to "nothing" and makes MYDATA

unavailable to READ and PRINT commands until assigned again. Notice that the END command also closes all open files. Of course, PURGEing an open file not only closes it (for good!) but also de-assigns its channel number.

Note well! Whenever HP-71 memory is "reconfigured," all open channels get closed! This is not obvious in the Owner's Manual. Memory reconfiguration occurs in three instances: when FREE PORT is executed, when CLAIM PORT is executed, and when the contents of any plug-in ports are changed (except plugging or unplugging the card reader, which is not a soft-addressed device). So don't do any of these when files are open, or they'll get closed, possibly resulting in a suspended program failing to be able to CONTinue.

SDATA TIDBITS

There are dire warnings in the Owner's Manual about "end-of-file markers" and "loss of data" and other bizarre things. Pay no attention to it. None of it applies to SDATA files, just DATA files. As I said, SDATA files are very nice!

One thing was left unsaid about SDATA files: You can only store numbers in them, not strings (through "normal" operations). If you have an HP-41 and its HP-IL module, you can have fun dumping numbers AND alpha strings into SDATA files on the HP-71!

It may be obvious by now that manipulating files from the keyboard is not too easy. It was really meant to be done from a program. However, saving an array for safekeeping in a file is a common practice, as well as using a FOR / NEXT loop to manipulate the file. To reverse a 100-element array A, for example:

```
                PRINT #1,0;A
FOR X=0 TO 99 @ READ #1,X;A(100-X) @ NEXT X
```

In general, the complexity of file usage is outweighed by the fact that files are safe places to put data. The slings and arrows of outrageous functions like DESTROY do not affect files.

FILES IN GENERAL

Now that you have some data in an SDATA file, you can do some wonderful things with it. There are a lot of BASIC statements that deal with files. If you type CAT ALL, you'll see the top of the "catalog" of files in memory. Pressing the up and down arrow keys allows you to scroll through the catalog, and pressing the right and left arrow keys lets you see the entire catalog entry for each file. The display shows the important facts: the file's name, its security type, its file type, and its size in bytes.

File names are always 1 to 8 letters long. The first character must be a letter, but after that you can use numbers and/or letters. To change the name of a file, just type

```
RENAME FRED TO BILLY
```

That would give the FRED file the new name BILLY. To kill a file from memory, all it takes is

```
PURGE BILLY
```

and BILLY is cleared from memory. Be sure you really want to do this before doing it, because there is no easy way to retrieve a purged file! It is as good as gone.

But suppose you have a really important file that should never be purged? Then you can protect it! Type

```
SECURE FRED
```

and the FRED file's security type shown in the CATalog changes to

"S". This means that you can read from the file but you can't write to it or purge it. To write new data into it, just

```
UNSECURE FRED
```

and write whatever you like. A good idea is to UNSECURE a file, ASSIGN it a channel number, then SECURE it right away. That way you can write to it (because it was UNSECURE when you opened it) but you can't purge it (because it is SECURE).

If you wish to make a copy of the file in memory, so that you can play with one copy and have the other copy as a backup for safety's sake, then simply

```
COPY FRED TO BILLY
```

This makes a new file in memory called BILLY that is identical to FRED in all respects except name.

If you have a card reader, tape drive, disk drive or other mass memory device, then you can even COPY the file onto a mass memory medium. This allows you to keep truckloads of data on file, and COPY it back into the HP-71 as needed. The proper BASIC syntax for the operation of these optional devices is not described in this book.

TEXT FILES

Unlike SDATA files, which can only store numbers, TEXT files can only store strings, and can only store sequentially. Other than these two huge differences, SDATA and TEXT files work very much alike. To create a TEXT file, you

```
CREATE TEXT MYWORDS
```

This creates a TEXT file called MYWORDS. To open the file,

```
ASSIGN #1 TO MYWORDS
```

and to write data into it,

```
PRINT #1;"This is a test"
```

Even though you can only write sequentially to TEXT files, you may read either sequentially or randomly.

Even though TEXT files store everything as text, you can pretend to store numbers in them and read numbers back out. For example, you could

```
PRINT #1;12345
```

and later `READ #1;X` and find the number 12345 in X. But you could also `READ #1;A$` and find "12345" in A$. You see, TEXT files store numbers as text, and if read back into numeric variables, the text is translated back into a number. Numbers in text form generally take up more memory than regular numbers, so don't use text files to store a lot of numbers!

DATA FILES

As you recall, SDATA files can only store numbers, not words, names, street addresses, and so on. If you need to store a mixture of numbers and text in a file, then a DATA file is what you need!

You also remember that an SDATA file can only hold one number in each record. Guess what! DATA files can hold as many items (numbers or words) as you wish. This allows you to keep logically related items all together in one record.

HOW TO CREATE A DATA FILE

Unlike SDATA files, whose records are always 8 bytes long, DATA

files' records can be as short or as long as you like. Unfortunately, this means that you have to figure out how long you want them to be! It's not that difficult; you just figure how many numbers and words you wish to store in each record. Here's an example.

Let's say we want to write a Telephone Number program that stores the names of friends and their telephone numbers in a DATA file. Since DATA files allow multiple items in each record, let's put each person's name and telephone number in one record. So we would like to store "Richard J Nelson" and his telephone number 7145490581 all in one record of a DATA file.

We must first answer the vital question: <u>What is the maximum length of the text we wish to store in each record?</u> If you can't answer that because you have no idea how long your text might be, then you shouldn't use a DATA file; use a TEXT file, which doesn't care how long your text is.

In our case of a telephone number program, let's agree to limit the names to 21 letters long. The rule for record length in a DATA file is:

Total Maximum String Length PLUS
3 bytes for each string PLUS
8 bytes for each number.

Since we have one string of 21 characters, and one number, our telephone DATA file needs 21 + 3 + 8 bytes per record, for a grand total of 32 bytes! (Your own custom DATA files will of course have different record lengths).

Let's name our phone data file "PDATA". But before you create a file you have to decide how many records you want! Let's reserve room for 50 names and phone numbers. RECAP: we want a DATA file called PDATA with 50 records of 32 bytes each. Here's how it's done:

```
CREATE DATA PDATA,50,32
```

Just remember to put the number of records first, then the record size. Since the number of records is logically more important, it is easy to remember that it comes first. Also, remember that when you create SDATA files, you only specify the number of records, because record length is automatically set to 8 bytes (enough for one number).

HOW TO ASSIGN A CHANNEL NUMBER

You assign a channel number to DATA files in exactly the same way as SDATA files:

```
ASSIGN #1 TO PDATA
```

From now on, you refer not to "PDATA" but to #1.

RANDOM ACCESS STORAGE

To store a record, write its entire contents to the file at once. For example,

```
PRINT #1,5;"Phineas McLanagan",1234567
```

writes Finny's name and phone number into record #5. Notice that the first item is a string, and the second is a number. You must keep track of which items are of what type, so that recalling them into the right type of variable is easy. Unlike TEXT files, DATA files insist that numbers get read into numeric variables, and strings get read into string variables, and never strings into numbers or vice versa.

SEQUENTIAL ACCESS STORAGE

It can be done, but it makes a mess out of the file, with strings spanning records and other nastiness. Unless you know precisely

what you're doing, don't try it.

RANDOM ACCESS RECALL

To pull out the contents or a particular record:

```
READ #1,5;N$,P
```

reads record 5's contents into two variables, N$ and P. If the record does not contain a string and then a number, you'll get an error when you try this! If you did the above random access storage example, then N$ now contains Finny's name, and P is his phone number.

Unlike SDATA files, attempting to recall a DATA file record that hasn't yet been written to yields an error. To avoid this, some programmers like to fill new files with blank data right away. I prefer to keep track of which records have contents and which don't.

SEQUENTIAL ACCESS RECALL

As with SDATA files, failure to specify which item you wish to recall results in the recalling of the next item. Unlike SDATA files, however, DATA file records can contain more than one item! For example,

```
READ #1;N$,P
```

would now read record #6, since we just read record 5 above. On the other hand,

```
READ #1;N$
```

would read only the name in record 6. Note well! After doing this, the file pointer is left in limbo, floating between two items in a

record. Then we could

```
READ #1;P
```

to get that person's phone number.

STORING/RECALLING MORE THAN ONE RECORD AT A TIME

Sorry! It can't be done. Remember how we could READ a whole bunch of records at once from SDATA files? You can't do that with DATA files. If you try, you'll get an error when it hits the end of the record.
Well, I lied. You can sort of do it if your data is all strings or all numbers. Then you can read an array or matrix as described for SDATA files. But remember that storing an array or matrix into a file is a form of sequential access, and that is guaranteed to mess up a DATA file! I had an application once that used the first half of a file randomly and the second half sequentially, but when I tried to figure out how it worked after several months of disuse I almost had a mental hernia.

MOVING THE FILE POINTER

RESTORE works on DATA files exactly the same way as it does on SDATA files. It places the pointer at the beginning of the specified record. For example,

```
RESTORE #1,5
```

places the pointer at the beginning of record 5.

Unfortunately, there is no way to RESTORE the pointer within a record. To place the pointer within a record, you must recall some items from it without recalling the entire record, as described above. This leaves the pointer sitting at the end of the last item

recalled. Although it is possible, leaving pointers floating around within records is of dubious utility and is a Bad Thing because you'll never understand what's happening even if you did it.

DATA TIDBITS

The warning against sequential access storage is because the HP-71 has a nasty feature called the "end-of-file marker." After every sequential access PRINT to the file, the HP-71 stores an end-of-file marker in the record just written. This prevents sequential recalling beyond that point. Luckily, it does not affect random access operations. The Owner's Manual overstates the dangers of the end of file marker. It does NOT result in the loss of all data beyond it! You can read sequentially up to, and after, an end-of-file marker! It just prevents a sequential READ to go through that point. And it is ignored by random access READS.

If you try to store too much in a record, you'll get an error. For example, if we tried to store a record into our PDATA file with a name longer than 21 letters long, the HP-71 wouldn't have enough room to do it. If you keep getting this sort of error, re-CREATE the file with longer records. Unfortunately, there is no way of lengthening records directly. If you wish to do this, CREATE another file with the new record length, then loop through the old file, storing its contents into the new file. Then the old file can be PURGEd.

Examples of the above ideas about DATA files can be found in a program called "PHONE". See Chapter 9 for the listing, instructions, and comments.

KEY FILES

On of the handiest features of the HP-71 is its ability to redefine what the keys do. I had to bite my tongue to keep from telling you about it until now!

Suppose you use the DESTROY command a lot (I do!). And suppose you almost never use the RETURN key (I don't!). Rather than letting RETURN (the gold-shifted D key) gobble up keyboard space, and typing DESTROY yourself all the time, you can actually change the RETURN key into a DESTROY key!

Try this. Type exactly as shown:

```
KEY "fD","DESTROY ";
```

and of course press [ENDLINE]. Now press the USER key (gold-shifted 0 key) until you see the little "USER" light up in the lower left corner of the display.

Now press the RETURN key. Magic!! Instead of RETURN, the display shows DESTROY, with the cursor following it ready for you to continue typing!

This key assignment only works in USER mode (i.e. when the USER annunciator is on). If you want to clear it altogether, type

```
KEY "fD"
```

and the RETURN key will operate normally whether or not you're in USER mode.

In general, to assign anything to a key, type the statement KEY followed by the key's name in quotation marks, then a comma, and then the assignment in quotation marks. If the key name or assignment is in a string variable, you can use it instead of the literal string.

Notice that the assignment above ended with a semicolon (;) . That meant that the cursor was supposed to stay there and let you keep typing. There are two other kinds of assignments.

If you omit the semicolon from an assignment, then pressing the key will type the assigned string and then press [ENDLINE] automatically. For instance, I use CAT ALL and DESTROY ALL so much that I assigned ALL to the FACT key (above =) . But since ALL is always the last thing I type before pressing [ENDLINE], I let the HP-71 do it for me:

```
KEY "f=","ALL"
```

So in USER mode, I get CAT ALL and DESTROY ALL in two keystrokes!

The third type of key assignment is spooky and should be used with caution. If you end an assignment with a colon (:) instead of a semicolon, the assignment is done without even showing up in the display! Try this:

```
KEY "f(","TIME$":
```

and now press the AUTO key (above the open parenthesis key) in USER mode. Notice how the time appears immediately without "TIME$" being typed in the display? This type of assignment, since it works mysteriously, should only be used when you are certain that you'll remember what's going on. Pressing a key and seeing a result without having a clue why you got that result can be disconcerting.

If you ever forget what a key's assignment is, just press VIEW (above the period) and then press and hold the desired key. Its assignment will be shown, preceded by a semicolon, colon, or space to indicate which type of assignment it is.

To change a key's assignment, the best way is with the FETCH command. For example, `FETCH KEY "f("` would bring

```
DEF KEY 'f(',TIME':
```

into the display with the cursor waiting for you to type over it. (The DEF is optional, somewhat as LET is optional).

To see all current key assignments, the best way is to set the DELAY to 8 (to hold each key in the display) and then type

```
LIST KEYS
```

Press the right and left arrow keys to see any assignment longer than the display, and press any other key to move on to the next key. After all the assignments are seen, be sure to reset the DELAY to your normal setting (.5 is standard).

During a CAT ALL, you will notice a file called “keys”. Its file type is “KEY”. This is the file that contains your key assignments. You can COPY, PURGE, SECURE and perform any other file command to “keys” that you can do to any other file. However, be warned that the HP-71 will only consider “keys” to be the active assignments. If you RENAME KEYS TO FRED, you’ll lose your key assignments! To revive them, just RENAME FRED TO KEYS. This allows you to have more than one key assignment file in memory, and then select which one you wish to be active by renaming it to “keys”.

BASIC FILES

Finally! We have arrived at BASIC files! And you thought you’d get there on the first page!

DATA files are collections of data. TEXT files are collections of text. KEY files are collections of key assignments. So BASIC files are obviously collections of BASIC commands.

The value of keeping a collection of BASIC commands in memory is not so that we can look at it and type the commands ourselves all over again. The magic of a BASIC file is that the HP-71 can be told to execute the entire collection of commands all at once!

There is a special name for a collection of commands which a computer executes all at once. It is called a "program". The art of putting together such collections of commands is called the art of programming. The person who does it is called a programmer. If the commands are in the BASIC language, then you have a BASIC program, written by a BASIC programmer. The HP-71 is the ultimate tool for BASIC programmers. Besides knowing BASIC, it can collect BASIC commands into a BASIC file, and then execute the entire file full of commands at once. All this in a handheld machine!

As you did a CAT ALL, you saw a file called "workfile" of file type "BASIC". Let's play with that one.

To get into the "workfile", simply type the command `EDIT`. (It is above the Z key). After that, to make sure it's empty, type the command `PURGE`. (Above the V key). We now have an empty "workfile" to play with.

Although the intricacies of programming will be explained in a later chapter, here's a foretaste of its joys. Type these lines, pressing `[ENDLINE]` after each:

```
10 "I AM A PROGRAMMER!"
20 BEEP 800
30 "THIS IS EASY!"
40 BEEP 650
50 GOTO 10
```

This is a program, a list of five commands. Notice how they did not get executed as you typed them. The reason was because they

started with numbers (called “line numbers”). Whenever you type a command but precede it by a number, the HP-71 figures that you don’t want it executed but instead you want it stored as a line of a program. In our case, these lines got stored in “workfile”. Press the up and down arrow keys to move up and down through the program, and press the [RUN] key to execute the program. Press [ON] to stop it.

The “workfile” is designed as a BASIC programmer’s scratch pad. Never put important programs into the workfile! Real programs should have their own names. To create a BASIC file with a real name, all you do is

```
EDIT ULAMCONJ
```

to create a BASIC file called ULAMCONJ. You may RENAME, PURGE, SECURE etc. any BASIC file. More on that later!

FORTH FILES

FORTH is a computer language that the HP-71 can speak if you plug in the optional HP 82441A FORTH/Assembler ROM, or the HP 82490A HP-41 Translator Pac, or any other module that contains the FORTH kernel. With such a module installed, you can write and run FORTH programs. Some say that FORTH is faster than BASIC. All I know is, when it comes to complex tasks I find BASIC a lot faster to write. Maybe FORTH runs faster, but BASIC is sure friendlier!

BIN and LEX FILES

The HP-71 doesn’t really speak BASIC. I don’t really speak Greek, but I could translate a Greek newspaper article into English if I could use my dictionary and grammar book and if you gave me enough time. That’s what the HP-71 does. It doesn’t really speak

BASIC; it translates it slowly, bit by bit, into the language it really does understand.

The native language of the HP-71, as for all computers, is called Machine Language. It's a horrible gobbledygook mess of jumbled up numbers that makes no sense to normal humans. But it's what the machine speaks, and so it's very fast. A program written completely in machine language runs circles around the same program written in BASIC. Trouble is, a complex BASIC program that takes 10 hours to write might take 10 years to write in machine language. It's possible, but a pain. Machine language programs live in BIN and LEX files.

A BIN file (it stands for BINary) is a complete program, like a BASIC file. A LEX file (it stands for Language EXtension) adds new functions and commands to BASIC. Both can be written through an intermediate language called Assembly Language. It is comprehensible by gifted humans and those with nothing else to do. If you have the FORTH/Assembler ROM, you can write assembly language programs yourself, assemble them into machine language, and then run them. If you're gifted and have nothing else to do!

Although this book is supposed to be about BASIC, at the end we'll see a way of keying LEX and BIN files into memory without buying the FORTH module! Several wonderful LEX files are floating around in the HP-71 backwaters, and the technique presented at the end of the book will allow you to use them without having to buy anything else.

CHAPTER 7:

THE HP-71 CLOCK & CALENDAR

Of all the functions in the HP-71, the internal real-time clock is probably the simplest to use, yet the hardest to explain! The Owner's Manual was little help to me. I hope my explanation is useful to you.

Right now, set the date, using the SETDATE command. It works like this:

```
setdate "85/07/04"
```

sets the date to 4th of July, 1985. Remember to arrange the year, month and day in descending order. HP used this order (not a standard way to write the date!) because it is logical, just like numbers, with the most important number at the left and least significant number at the right.

To get the date back out of the computer, just type

```
date$
```

("Date String"). You'll see the date just as you set it. Of course, tomorrow it'll be tomorrow's date. By the way, you may use "1985" instead of "85", but you don't need to. Matter of fact, if you use "00" it'll know you mean "2000"! And so on up to 2059. If you are actually reading this book after 2059 or before 1960 (equally unlikely), then you must spell out the whole year with all four digits.

Before you set the time, please be aware of two things.
First, all clocks drift away from the real time. Given enough time, any clock will get a little bit ahead or behind true time. The speed

at which it gets wrong is called its “rate of drift” or simply its “drift”. Lousy clocks have erratic drifts; they speed up and slow down depending on the temperature, humidity, chance, and other unpredictable factors. Good clocks have predictable drifts; they are always slow or always fast, and by a precise amount. So you can correct a good clock by adding or subtracting a precise “adjustment factor” to compensate for the drift.

Second, the HP-71 has a very good clock in it. It drifts, but by a very predictable amount. And best of all, the HP-71 has a variable “adjustment factor” built right in, which allows you to automatically add or subtract the clock’s drift, and obtain a practically perfect timepiece! Of course, if you lock your HP-71 in the freezer for a week and then lock it in the glove compartment of your dune buggy and go blazing through the desert for a week, I guarantee that the clock will become somewhat inaccurate, if not stop altogether! Dependable use of the clock and the adjustment factor (called “accuracy factor” in some HP literature) assumes a reasonably constant overall environment. So if you bought your HP-71 from a previous user, don’t trust the adjustment factor he or she swears is the right one.

Having said that, here’s how to set the clock and adjustment factor.

When starting out from scratch, always reset the adjustment factor:

```
reset clock
```

This doesn’t clear the time; it merely clears the adjustment factor and any accumulated drift.

Next, set the time, like this:

```
settime "19:00:00"
```

using the right hours, minutes and seconds. (NB: The HP-71 uses "military" 24-hour time. The hour after midnight is 00, not 12. If it's PM, add 12 to the clock time to get 24-hour time. Midnight is 00:00:00; the previous second is 23:59:59). Remember, the command will not be carried out until you press [ENDLINE], so type a time a little bit in the future. About half a second before that time arrives, press [ENDLINE]. A trick I use is to type in a time like

```
settime "09:00:01"
```

which is one second after the hour; then, when the radio gives the 9 o'clock tone, I press [ENDLINE]. This gets it close enough!

To see the time (anytime) just use the TIME$ function:

```
time$
```

Now that we have the clock set to an exact time, tell the HP-71 that it really is the exact time:

```
exact
```

The EXACT command doesn't change the time any. It merely stores current time and date safely in memory, for use next time you set the time.

Now let a week or two go by. Don't worry about the drift; the HP-71 calculates it for you. Just set the time once again (making sure it's right!) and when it's exact, execute EXACT once again. The second execution of EXACT sets the adjustment factor automatically, which begins correcting the clock automatically. You should now have a very accurate clock! Of course, after several months or so you may notice a little drift; just set the exact time and execute EXACT once again to really make the clock exact!

If you change time zones or change daylight savings time, don't reset the time. Use the ADJUST command. If the time becomes an hour greater, use:

```
adjust "01:00:00"
```

and if it becomes an hour less, use:

```
adjust "-01:00:00"
```

Adjusting by whole hours or multiples of 30 minutes does not affect your adjustment factor, so using ADJUST is better than using SETTIME to correct the time when the time change is not due to drift but time zone or daylight savings time change. If you specify an adjustment that isn't a multiple of 30 minutes, then the HP-71 will consider that a drift, and will modify the accumulated drift. It is better to use SETTIME to correct for drift automatically. You may use ADJUST to do it, if you know the exact amount of drift.

If you have your clock adjusted correctly (its drift is now compensated for by the appropriate adjustment factor), but then find out that the time standard you set it by was off by a certain amount, you can't use SETTIME or ADJUST to correct the time, because SETTIME changes the accumulated drift, and ADJUST does too if the adjustment is small. In this case, use the ADJABS ("Adjust Absolute") command.

```
adjabs 0.3
```

adds exactly 3 tenths of a second to the running clock without affecting the adjustment factor in any way. In my case, after getting my clock exact by the radio, I subtract eight tenths of a second to account for the time between hearing the time tone and the pressing of `[ENDLINE]` (about .3 seconds) and the amount of time it takes SETTIME to work (about .5 seconds):

```
adjabs -.8
```

As soon as you have an adjustment factor that seems to be working well for you, be sure to find out what it is and write it down. To see your currect adjustment factor, type

```
af
```

and jot it down somewhere. That way, if you ever get a Memory Lost (which clears the time, date, and adjustment factor), you can set the adjustment factor manually without waiting weeks or months first. If your AF is 1000, then you can set it to 1000 manually by typing

```
af(1000)
```

This not only tells you what the AF is but also sets it to 1000. Remember to execute EXACT first!

There are two other time/date functions, of dubious utility. TIME gives the number of seconds since last midnight, in hundredths of a second. This can be used to time things, by subtracting the starting TIME from the ending TIME, as long as you don't run past midnight!

DATE gives the year and day-of-year as a single number. For example, if DATE gives 85231, then it's the 231st day of 1985 (my 30th birthday!). Some business applications require knowing the day of year; `MOD(DATE,1000)` gives it. Unfortunately, if your HP-71's VER$ is 1BBBB, then your DATE has a bug in it that makes the day of year unreliable. Years whose second-to-last digit is odd, and whose last digit is 2 or 6, give the wrong day of year for dates beginning with March 1st (they are 1 too low). Hopefully later versions of the HP-71 ROMs will correct this bug.

Note: the HP-71 clock is more reliable than the HP-75 clock, which

became notorious for its seemingly random errors. The reason for its errors was simple. The time is stored in memory, and the computer has to pull it out one number at a time (just like any other data). But suppose the time "kerchunks" (changes) right in the middle of the process of pulling it out? You then get half of the old time and half of the new time strung together. The HP-71 design team was aware of this possibility, and de-kerchunked its clock by performing two clock reads in a row and comparing them to see if a kerchunk had occurred. Hence a reliable clock.

CHAPTER 8:

PEEK$ AND POKE!

Of all the keywords in HP-71 BASIC, the two which receive the least treatment in the Owner's Manual and Reference Manual are PEEK$ and POKE. And no wonder! These two keywords allow you to romp through memory, discovering what HP would rather leave unseen, and even change anything anywhere in memory! A programmer without PEEK$ and POKE who suddenly comes into their possession is like graduating from a barber to a brain surgeon!

Enough praise. What are some practical uses of PEEK$ and POKE?

The primary value of PEEK$ is that it lets you get certain values that are otherwise unavailable. Suppose your program thrashes flags 0 through 63. You could save them in 64 variables, but that would take a lot of memory and time.

PEEK$ to the rescue. The flags 0 through 63 reside in 16 nibbles of memory at hex address 2F6E9. So to save all 64 flags, we just say

```
F$=PEEK$("2F6E9",16)
```

and they are all stuffed into F$ in no time. When we're done playing with the flags, we can just

```
POKE "2F6E9",F$
```

to restore all 64 flags to their original position!

The primary use of POKE is for control purposes. There is simply no way to do certain things without POKing the HP-71 right in the

ribs. Suppose you want to have a program keep running, and not be interrupted, even if somebody presses the [ON] key? The only way is to

```
POKE "2F441","F"
```

which disables the ON key from interrupting programs. It is re-enabled by `POKE "2F441","0"`. Control like this is not available any other way.

Although the programs in this book contain a wealth of PEEK$ and POKE examples, here is a table of the primary hex addresses that are useful or interesting to PEEK$ and POKE.

CAUTION!! POKEing around indiscriminately is unwise, as it can easily result in a Memory Lost, or even worse, a crash so bad that you'll have to send it in to get fixed. Simple caution and a little common sense should suffice to avoid such problems.

How do PEEK$ and POKE work? When you store a number in memory, say 1234 in X, each digit takes up a piece of memory called a "nibble". Whereas most computers are "byte" (8-bit) oriented machines, the HP-71 is nibble (4-bit) oriented. Every nibble in memory has its own unique address, from 0 to 1048575. To simplify addresses, PEEK$ and POKE don't refer to addresses as such huge numbers. Instead they use the hex version of them. Use the HTD and DTH$ functions to get an idea of what numbers look like in hex. For example, `DTH$(112233)` tells us that the decimal number 112233 is represented as 1B669 in hex (try it!), and `HTD("1B669")` converts it back to decimal.

You already know that when you perform the CONTRAST command, the display's contrast changes. You can set it to anything from 0 through 15. In hex, that's 0 through F. This suggests that the display contrast is stored somewhere in memory

as a single nibble, and in fact that's exactly true. It resides in the nibble whose address number is 189438. In hex (use DTH$ to see this) that number is 2E3FE. So we should be able to use PEEK$ and POKE to look at and change the contrast nibble.

Try this. `CONTRAST 7` to set the contrast to 7. Then `PEEK$("2E3FE",1)` and you'll see 7, the contrast nibble! Now `CONTRAST 10` and peek again (use the command stack). This time it shows A. Why A? Because "A" is hex for 10. Use HTD to see this:

```
HTD(PEEK$("2E3FE",1))
```

and see 10!

We can also use POKE to change the contrast nibble. It is silly, because we can change it with the CONTRAST command, but this is instructive and immediately recognizable. Try this:

```
POKE "2E3FE","4"
```

Notice that the contrast immediately changes to 4! That's because we told the HP-71 to poke a "4" into memory starting at nibble address "2E3FE" which is the contrast nibble. CAUTION! Don't try other addresses, and don't try poking things longer than one digit long! That'll mess up memory and possibly cause a serious crash.

Here is a partial table of useful addresses in hex. Some of these are only for PEEKing at, other for POKING, and some for both. Just be careful!

ADDR	#NIBS	WHAT IT IS
2C014	(1)	Card Reader present (0 if NOT present)
2E100	(2)	Annunciators on left side of display
2E34C	(2)	Annunciators on right side

ADDR	#NIBS	WHAT IT IS
2E350	(16)	Display row drivers (reset by INIT: 1)
2E3FE	(1)	Display contrast
2E3FF	(1)	Display controller (1=on, 2=blink)
2F438	(4)	Cold Start Constant (Memory Lost if changed)
2F43C	(5)	Interrupt vector (Bad crash if changed)
2F441	(1)	ATTN-key disabler (0=enabled)
2F443	(1)	Key buffer pointer
2F444	(15)	Key buffer (2 nibs per keycode)
2F471	(2)	Number of characters to the left of window
2F473	(2)	Length of WINDOW minus 1
2F47C	(2)	Buffer pointer to first displayed character
2F47E	(2)	Buffer pointer to cursor
2F480	(96*2)	Display buffer (2 nibs per character)
2F540	(96/4)	Display readability mask (1 bit per character)
2F576	(5)	Address of top of Command Stack
2F58A	(5)	Address of bottom of Command Stack
2F59E	(5)	FOR / NEXT stack address
2F5A3	(5)	GOSUB stack address
2F5A8	(5)	Address of active variable space
2F5AD	(5)	CALL stack address
2F5B2	(5)	Address of absolute end of memory
2F5B7	(5)	Parameter chain address
2F5BE	(26*7)	Variable chain (7 nibs for A vars, 7 for B...)
2F6C6	(5)	INPUT buffer address
2F6CB	(4)	AUTO increment
2F6D9	(64/4)	System flags (-1 to -64), 1 bit per flag
2F6E9	(64/4)	User flags (0 to 63), 1 bit per flag
2F6F9	(5)	IEEE traps (INX, UNF, OVF, DVZ, IVL)
2F6FE	(15)	Random number seed for RND
2F761	(2)	Bitmap of pending alarms
2F763	(12)	Accumulated time drift
2F76F	(12)	Time that the clock was last set
2F77B	(12)	Time that the adjustment factor was last set
2F787	(6)	Adjustment factor
2F7AD	(3)	Name of the STAT array variable
2F7B0	(1)	Trace mode (0=OFF, 2=FLOW, 4=VARS, 6=both)
2F7B2	(8*2)	LOCK password (2 nibs per character)
2F7C2	(34)	RES (holds any result, even COMPLEX numbers)
2F7E4	(4)	ERRN (Error number)
2F7E8	(4)	Current line number (of a non-running program)

ADDR	#NIBS	WHAT IT IS
2F7EC	(4)	ERRL (Last line on which an error occurred)
2F870	(1)	Multi-line FN flag (0=not multi-line)
2F946	(2)	Character rate (the second DELAY argument)
2F948	(2)	Line rate (the first DELAY argument)
2F94F	(2)	WIDTH setting
2F956	(2)	Current PRINT column position
2F958	(2)	PWIDTH setting
2F95A	(1)	Length of ENDLINE string (0, 1, 2 or 3)
2F95B	(3*2)	ENDLINE string
2F967	(2)	Key definition string length (2 nibs per byte)
2F969	(1)	Key type
2F96A	(5)	Address of key definition string
2F96F	(2)	Current channel number being accessed
2F976	(1)	Number of Command Stack entries minus one
2F977	(5)	Clock speed /16
2F986	(1)	Complex IMAGE flag (0=not complex)

That ought to be enough to keep you busy for a while!

Notice that a program can grab all the user flags in one step, for example `U$=PEEK$("2F6E9",16)`, then use the flags any way at all, and then nicely reset them all in one step again, `POKE "2F6E9",U$`. The same applies to other settings, like the TRAPS, WIDTH and so on.

There is no normal way to get the current random number. Every time you use RND, it generates a new one. One way to get the old one out is:

```
POKE "2F7C2","999"&PEEK$("2F701",12)&"0" @ R=RES
```

This places the current random number seed into the RES register and grabs it from there into the variable R.

Note well that numbers are stored backwards in HP-71 memory! For example, type PI and then peek at RES by:

```
PEEK$("2F7C2",16)
```

and you'll see PI backwards:

0009535629514130

The leading three zeros are the exponent (backwards), and the trailing zero is the positive sign (9 is negative). The moral is this: if you want to store anything in memory with POKE, be sure it's going in backwards!

This can complicate things. Suppose you want to pull out the clock speed, divide it by two, and store it back in memory. You have to PEEK$ it, reverse the entire string, convert it to a number with HTD, divide by two, change it back to hex with DTH$, reverse the string again, and finally POKE it into place. If you have REV$ in memory, you can:

```
POKE "2F977",REV$(DTH$(HTD(REV$(PEEK$("2F977",5)))/2))
```

but you have to have the LEX file that gives you REV$. By the way, the result of the above is that all BEEPs speed up twice as fast and sound twice as high!

The most exciting use I've found for PEEK$ and POKE is the ability to enter LEX files from the keyboard! HP said in the Owner's Manuals that it couldn't be done, and in some other documentation they admitted that maybe it could. See the program called MAKELEX (in the next chapter) that lets you add REV$ and other fantastic new machine-language functions to your HP-71!

CHAPTER 9:

PROGRAM BASIC

Here's what you've been waiting for! And I hate to disappoint you, but you have learned so much in these past 124 pages that I haven't really got much left to teach you! You have already typed a program into the HP-71 and run it. What more could you want?

Complete mastery of BASIC programming comes with experience, not by reading more about it. However, one can accelerate one's experience by perusing others' programs. So let me here just mention a final few meatballs of wisdom before launching into a baker's dozen of exemplary programs for you to pick through to see how it's really done.

First of all, if you wish to start typing in a program that is already written, don't type it into the workfile! Rather, type

```
EDIT PASSWORD
```

if it's going to be called PASSWORD, or whatever the name might be. Remember, the workfile is reserved for your scratchpad work!

If you key in a program and it doesn't seem to work, don't forget that pressing the up and down arrow keys lets you look at the program and change what's in them. The commands `TRACE VARS`, `TRACE FLOW`, and `TRACE OFF` are very helpful in finding bugs in programs.

Little extra niceties, like the AUTO statement, are not covered by this book, because they are so simple and are treated well enough by the owner's manual. Now that you know BASIC well enough to converse with the HP-71, be sure to sit down and read both

manuals thoroughly. You'll be amazed that what used to seem impossible to comprehend is now very simple. Even the very complex topics (like `IMAGE` and `PRINT USING`) are best understood by doing HP's examples.

And since learning a language is best done by imitation, here is a potpourri of BASIC programs selected not only to give you a good cross-section of various types of programming but also because many of them may prove useful to you.

You should first key in the program as listed. Then you should try out the program and plow through it line by line and figure out how it works. Then try to modify it to work some slightly different way. Or try a major overhaul; you've got nothing to lose, and BASIC to gain!

CALORIES – A Jogger's Calorie Calculator

```
10 DIM U,D,T,T1,C @ INPUT "Lbs? ";W
20 IF W<120 OR W>220 THEN 120
30 INPUT "Miles? ";D
40 IF D<=0 THEN 120
50 INPUT "Min.Sec? ";T
60 IF T<=0 THEN 120
70 Tl=IP(T)+FP(T)/.6
80 IF T/D<5.33 THEN BEEP @ DISP "Too Fast" @ END
98 IF T/D>10.67 THEN BEEP @ DISP "Too Slow" @ END
100 C=(W*(.735993-(T1/D*.01325))+3.6)*D
110 DISP "Calories Used:";IP(C+.5) @ END
120 BEEP @ DISP "Out of Range"
```

After taking a healthy jog, run the above program. It asks you how much you weigh, how far you jogged and how long it took you. It then calculates how many calories you burned off by your exercise. This is a simple program that illustrates IF / THEN comparisons, and the concept of screening out unwanted inputs.

CMDSTK – A Command Stack expander / compressor

```
CMDSTK     BASIC     265
************************
10 SUB CMDSTK(X) @ POKE "2f441","1" @ STD
20 DEF FNR$(A$)=A$[5]&A$[4,4]&A$[3,3]&A$[2,2]&A$[1,1]
30 X=IP(MIN(MAX(X,1),16))
40 DIM S$[X*6] @ A=HTD(FNR$(PEEK$("2F576",5)))
50 FOR Y=1 TO X @ S$=S$&"000300" @ NEXT Y
60 E$=FNR$(DTH$(A+X*6)) @ POKE "2F580",E$&E$&E$
70 POKE DTH$(A),S$ @ POKE "2F976",DTH$(X-1)[5]
80 POKE "2F441","0" @ END SUB
```

You can't RUN this program; you have to CALL it like this:

```
CALL CMDSTK(5)
```

When you do, the command stack will be made to have 5 entries, which is the normal number. Using 10 instead of 5 creates a command stack that has ten entries! Up to a 16-high stack is possible, and the minimum is 1.

TOKENIZE

```
TOKENIZE     BASIC     142
*************************
1 ! Any line here
10 SUB A @ N=256 @ DIM A$[N] @ A$=ADDR$(CAT$(0))
20 A$=PEEK$(DTH$(HTD(A$)+55),N)
30 A$=A$[1,POS(A$,"0F0100")-1]
40 PLIST 1,9 @ DISP A$;" (";STR$(LEN(A$));")"
```

This is a self-referential program. All it does is list its own first line, and then show it in its internal representation and its memory size in nibbles. For example, type

```
1 BEEP 153
```

and then `CALL A` (two keystrokes). First you see 1 BEEP 153, then 8EB0351 (7). This means that BEEP 153 is "tokenized" internally as 8EB0351 (8E is BEEP, and B0351 is 153), a total of 7 nibbles. This serves two purposes. It is fascinating to use this program to figure out how the HP-71 tokenizes various functions and statements. But it is especially useful when you think of two similar ways to program something and don't know which one takes less memory. For example:

```
1 DIM M(3) @ M(1)=1 @ M(2)=2 @ M(3)=3
```

and

```
1 DIM M(3) @ READ M @ DATA 1,2,3
```

both do exactly the same thing. Which takes less memory? Enter each as TOKENIZE's first line, and then CALL A to find out each one's nibble count.

SIEVE – The Ancient Sieve of Eratosthenes

```
10 REAL K,Q,T,X @ DESTROY P @ OPTION BASE 1
20 INPUT "Primes up to? ";Q @ T=(Q-1) DIV 2 @ INTEGER P(T)
30 DISP "2" @ FOR K=1 TO (SQR(Q)-1)/2 @ IF P(K) THEN 50
40 DISP STR$(K+K+1) @ FOR X=3*K+1 TO T STEP X-K @ P(X)=1 @ NEXT X
50 NEXT K @ FOR K=K TO T @ IF P(K)=0 THEN DISP STR$(K+K+1)
60 NEXT K @ END
```

Run the above program, and input a number (say, 100 or 1000) where you want it to stop. The program will list all the prime numbers up to that limit amazingly fast. It goes slow at first, but then the primes simply pour out. If they aren't, maybe your DELAY is not 0. This program illustrates array usage and the trick of using the counter variable of a FOR / NEXT loop in the STEP statement.

ERRORS

```
10 DIM X @ DELAY 0,0 @ POKE "2F7E6","00"
20 POKE "2F443","0" @ INPUT "Error #? ";X
30 X=MOD(X,249)
40 IF X>97 AND X<228 THEN X=229 ELSE IF X=228 THEN X=97
50 POKE "2F7E4",DTH$(MOD(X,16))[5]&DTH$(X DIV 16)[5]
60 DISP STR$(X);'"';ERRM$;'"'
70 IF NOT KEYDOWN THEN 70
80 IF KEYDOWN("#51") THEN X=X+1 @ GOTO 30
90 IF KEYDOWN("#50") THEN X=X-1 @ GOTO 30
100 IF KEYDOWN THEN 20 ELSE 70
```

RUN ERRORS, and you will be asked for an error number. Input something like 10 or 20, and see the error message that corresponds to that number. Now press the down arrow key to move down through the "error catalog", or press up-arrow to move up. Pressing any other key re-prompts for an error number input. Notice how the program grabs a key and sees if it is held down to allow it to repeat just as CAT ALL does. The POKE in line 10 is necessary in case the last error before running the program was a LEX-file or ROM error, in which case this POKE clears it.

PHONE – A telephone directory program

```
10 DIM N$[21],S$[21],P,N,R,M
20 ON ERROR GOTO 370
30 UNSECURE PHDAT @ ASSIGN #1 TO PHDAT @ READ #1;N,M
40 DISP "Ready" @ ON ERROR GOTO 50
50 GOTO KEY$
60 'Q': PRINT #1,0;N,M @ DISP "Done" @ END
70 'P': IF R=0 THEN 90
80 DISP USING "'('3*')'3*'-'4*";P @ GOTO 50
90 DISP "No such record" @ GOTO 50
100 'N': IF R=0 THEN 90
110 DISP N$ @ GOTO 50
120 'D': IF R=0 THEN 90
130 IF R=N THEN 150
140 READ #1,N;N$,P @ PRINT #1,R;N$,P
150 R=0 @ N=N-1 @ DISP "Deleted" @ GOTO 50
160 'A': IF N=M THEN DISP "No room" @ GOTO 50
```

```
170 N$="" @ S$="" @ N=N+1 @ R=N
180 ON ERROR GOTO 190
190 LINPUT "Name: ",N$;N$ @ IF N$="" THEN 190
200 ON ERROR GOTO 210
210 INPUT "Phone #: ",S$;S$ @ P=VAL(S$)
220 PRINT #1,R;N$,P @ GOTO 40
230 'S': IF N=0 THEN 90 ELSE LINPUT "Search: ";S$ @ S$=UPRC$(S$)
240 FOR R=1 TO N @ READ #1,R;N$,P
250 IF POS(UPRC$(N$),UPRC$(S$))=0 THEN 270 ELSE DISP N$;"?"
260 ON POS("YN",KEY$)+1 GOTO 260,80,270
270 NEXT R @ R=0 @ GOTO 90
280 'Z': DISP "Zap ALL";N;"names?"
290 ON POS("YN",KEY$)+1 GOTO 290,300,40
300 PURGE PHDAT @ GOTO 370
310 'F': IF N=0 THEN 90 ELSE R=R+1 @ IF R>N THEN R=1
320 READ #1,R;N$,P @ GOTO 110
330 'B': IF N=0 THEN 90 ELSE R=R-1 @ IF R<1 THEN R=N
340 GOTO 320
350 'U': IF R=0 THEN 90 ELSE S$=STR$(P) @ GOTO 180
360 'M': DISP "A,B,D,F,M,N,P,Q,S,U,Z?" @ GOTO 50
370 OFF ERROR @ INPUT "How many records? ";M @ N=0
380 CREATE DATA PHDAT,M+1,32 @ ASSIGN #1 TO PHDAT
390 PRINT #1;N,M @ GOTO 40
```

This program does a lot of work in a small space. It stores names and telephone numbers in a DATA file, and lets you search for names to find a phone number quickly. When you first run the program, it asks you how many names you'll want MAXIMUM in your directory. Give a reasonable number, say 30. To Add a name & phone number, just press A.

Note: input phone numbers as numbers; no spaces, parentheses, dashes or stuff like that. So you'd input my phone number as 7146332041.

To browse Forward or Backwards through the names, press F or B. To see the Phone number of a displayed name, press P. To see their Name again, press N.

To Update a displayed person, press U and correct their name and/or phone number. To Delete a person altogether, press D when they are displayed.

To Search for somebody, press S. Input a few crucial letters in that person's name. A name will appear followed by a question mark. If it's the right person, press Y; if not, press N to continue searching.

To Zap the entire directory and start with a clean slate, press Z. You'll be asked if you really meant it; press N if you didn't, or Y if you did.

To Quit for the day, press Q. (DON'T press ON!)

If you ever forget which items are in the above Menu of commands, just press M. The active letters will appear on the screen.

This program illustrates many tricks. Program labels are directly paired to keystrokes for programming ease (look at line 50!). Menu-driven programs are always more enjoyable to use than ones in which you have to remember everything or answer a hundred questions before anything gets done. Error trapping weeds out most common input errors. Deletion by file shrinking is rare in programming, but the technique used here of replacement by the last record is fast and easy.

MAKELEX – A Lexfile Creator

Note: The string in line 60 contains exactly 16 dashes.

```
10 SFLAG -1 @ ON ERROR GOTO 30 @ DESTROY ALL
20 PURGE DUMMY
30 ON ERROR GOTO 220
40 INPUT '# of bytes: ';N
50 CREATE TEXT DUMMY,N
60 A=HTD(ADDR$('DUMMY')) @ A1=A @ P$="----------------"
70 Q=1 @ X=0 @ INPUT '000: ',P$;A$ @ C$=A$ @ GOSUB 200
80 Q=2 @ X=1 @ GOSUB 190
90 A$=A$&C$ @ A=A+37 @ N=N*2+31 @ Q=3 @ SFLAG 5
100 FOR X=2 TO N DIV 16-1
110 GOSUB 190
120 IF FLAG(5) THEN C$=C$[6]
130 POKE DTH$(A),C$ @ A=A+16-5*FLAG(5,0) @ NEXT X @ Q=4
```

```
140 DISP DTH$(X)[3]; @ INPUT ': ',P$[1,MOD(N,16)];C$
150 GOSUB 200
160 L=LEN(C$) @ IF C$[L,L]='-' THEN C$=C$[1,L-1] @ GOTO 160
170 POKE DTH$(A),C$ @ POKE DTH$(A1),A$
180 OFF ERROR @ CFLAG -1 @ END
190 DISP DTH$(X)[3]; @ INPUT ': ',P$;C$
200 DISP DTH$(X)[3]; @ INPUT ' ck ','--';C1$
210 S=0 @ FOR Z=1 TO LEN(C$) @ S=IP(NUM(C$[Z,Z])*Z+S) @ NEXT Z
220 IF C1$=DTH$(MOD(S,256))[4] THEN RETURN
230 DISP 'Checksum Error' @ BEEP @ POP @ ON Q GOTO 70,80,110,140
240 DISP 'Error: '&ERRM$ @ BEEP @ GOTO 180
```

This program was written by Stephen Tobiasson and Thomas Fange. It is a gem of BASIC code! It lets you take the listing of a LEX file or BIN file and type it directly into the HP-71. This allows you to do what ordinarily costs many dollars: enter valuable LEX files without a card reader, HP-IL mass storage device, or barcode reader.

Run the program. First input the number of bytes that is printed at the top of the LEX file listing. Then key in the hex code of the file, starting with line 000. Don't type the spaces; they are merely for visual benefit. After each line of code, next input its checksum byte as printed to the right of the line of code. Proceed thus until the last line, after which the program will automatically end. TURN THE HP-71 OFF AND BACK ON to link the new LEX file into the LEX file chain. Its functions will now be available!

Example: The following is a listing of the LEX file that adds the REV$ function to BASIC. Try inputting it with the above program:

```
REVLEX    ID#5D    36 bytes

     0123 4567 89AB CDEF ck
000: 2554 65C4 5485 0202 B1
001: 802E 0052 0010 1000 2D
002: 8400 0D51 0100 0000 3D
```

```
003: F710 0000 0000 0000 A7
004: 0710 00F7 2554 6542 FE
005: 101F F411 048F E83B C7
006: 18DC 32F0           0E
```

Once REVLEX is in memory, the REV$ function becomes available from the keyboard and in all programs. Type:

```
REV$("This is a test of string reversal")
```

and see “lasrever gnirts fo tset a si sihT” in a flash! REV$ can reverse a 14000-byte long string in one second! That is so much faster than BASIC it isn’t even funny. Good thing that BASIC is powerful enough to let us create REVLEX!

Here’s another useful function: KEYWAIT$.

```
KEYWAIT   ID#52    58 bytes

     0123 4567 89AB CDEF ck
000: B454 9575 1494 4502 BB
001: 802E 0053 9080 2058 AB
002: 3700 0251 0100 0000 D2
003: F710 0000 0000 0000 A7
004: 0E10 00FF B454 9575 9C
005: 1494 4542 101F F001 5C
006: 3610 8137 1098 F2C6 90
007: 0045 1119 1351 1813 20
008: 48D8 ACA1 8F12 7006 38
009: BDF                 9C
```

This LEX file was written by HP. KEYWAIT$ is like KEY$ in that it returns the most recently pressed key off the top of the key

buffer. But there is one major difference. If the key buffer is empty, KEY$ returns an empty string. KEYWAIT$, as its name implies, instead waits for a key to be pressed. And best of all, the HP-71 is held in a low power-drain mode, so it doesn't eat your batteries! Replacing KEY$ with KEYWAIT$ in the above PHONE program (all 3 places) would substantially reduce the power drain of the program. KEYWAIT$ is used in the following program called "SMATTER".

SMATTER – An unbreakable Coding / Decoding program

```
10 ! S.M.A.T.T.E.R ! (C) 1984 Joseph Horn
20 DIM A$[96],K,X @ DELAY 9,0 @ STD
30 DISP "Code Decode More Quit"
40 A$=KEYWAIT$ @ IF A$="Q" THEN 130 ELSE IF A$="M" THEN 70
50 IF A$="C" THEN K=1 ELSE IF A$="D" THEN K=-1 ELSE 40
60 OPTION ROUND NEG @ INPUT "Key#: ";X @ RANDOMIZE X
70 LINPUT "Text: ";A$ @ IF A$="" THEN 30 ELSE PRINT "{";
80 FOR X=1 TO LEN(A$) @ IF A$[X,X]<" " OR A$[X,X]>"z" THEN 120
90 RANDOMIZE 10^(RND*100)
100 POKE "2f6fe",STR$(FLAG(-12,RND<RND)+1)
110 PRINT CHR$(MOD(NUM(A$[X,X])+IP(RND*92)*K-32,91)+32);
120 NEXT X @ PRINT "}" @ GOTO 30
130 OPTION ROUND NEAR @ RANDOMIZE @ DELAY 0,0 @ DISP "Done" @ END
```

"SMATTER" is an acronymn for "Self Modifying Algorithm Transcryptor That's Essentially Random." It requires the KEYWAIT$ lex file, listed above. When you see the menu on the display, press the first letter of the desired function: Code, Decode, More (of the same message), or Quit. When asked for the key number, input a secret number that only you and your correspondent know. It can be anything at all, like `SQR(150+PI)`. Just don't use zero! Type in one line of text at a time, and continue by typing the next line. Press `[ENDLINE]` by itself to return to the menu. Braces are printed around the coded/decoded lines to make any leading or trailing spaces visible. I think it's an uncrackable code. It doesn't fall to any cryptanalysis scheme I've ever read about. Can you find a way to crack it?

Here are some more LEX files that you might find interesting and useful.

```
LOCKOFF   ID#5C    33 bytes

     0123 4567 89AB CDEF ck

000: C4F4 34B4 F464 6402 B8
001: 802E 0094 4032 0141 F8
002: 6400 0C50 0000 0000 23
003: FE00 0000 0800 001F 7F
004: F200 431D F961 4000 09
005: 342B 7F21 37AF 0151 23
006: 7                   37
```

LOCKOFF doesn't add any new keywords to BASIC, but it does do an amazing thing. It automatically clears the LOCK password as soon as you turn the HP-71 on! College students find this especially nice, because "friends" love setting bizarre passwords on your HP-71, resulting in your vital programs getting blown away. Just keep this LEX file in memory and nobody can LOCK your machine!

```
CURLEX    ID#5E   110 bytes

     0123 4567 89AB CDEF ck

000: 3455 25C4 5485 0202 A0
001: 802E 0004 1231 7041 F3
002: 0E00 0E50 0000 0000 4B
003: FE00 0000 0800 001F 7F
004: F31B 1961 4000 DB10 87
```

```
005: B317 F8FC 4631 4901 D6
006: 1BD7 0011 BD71 188F 24
007: B14B 1DA3 1F99 62D0 3E
008: 310A 962C 0003 1446 C5
009: 7003 134A F0AE AAF2 00
00A: 3141 8FB3 4B1A DA10 87
00B: 3078 F534 B107 8F53 90
00C: 4B10 7108 11B8 F1C3 C6
00D: 2011 BA5E 4801 0B59 6B
00E: E118 068F B14B 1068 5D
00F: FB14 B106 840       AD
```

CURLEX doesn't add any new keywords either, but it makes programming a lot easier! When you wish to look at a long line, it takes a while to move the cursor around. Wouldn't it be nice if the blue-shifted arrow keys moved the cursor not all the way to the end of the line, but just to the other end of the display, in 21-character jumps? Guess what! As long as CURLEX is in memory AND you are in USER mode, then that's exactly what those keys do! Try it. Just don't use 'em in CALC mode, because you can get funny results. This LEX file was written by John Baker, foremost HP-71 LEX file wizard!

```
CLOCKDSP  ID#52   328 bytes

     0123 4567 89AB CDEF ck

000: 34C4 F434 B444 3505 72
001: 802E 0055 4032 0141 E4
002: 4920 0259 2920 0000 95
003: F710 0000 0E10 0000 84
004: 0BD1 00D9 34C4 F434 BF
005: B492 1FF9 6950 5031 6D
006: 1B13 5112 1CB1 378B A4
```

```
007: 6A11 3510 B3B1 4A3B E4
008: 4C43 4021 5DB0 01FC 2F
009: 79F2 1530 A044 4000 44
00A: D48F 3E32 09F6 10CF DC
00B: B11D F241 BFD3 1000 10
00C: 01B1 09F2 1101 5071 75
00D: 6F11 1150 7191 4DB1 09
00E: 4420 313D 8FC4 6314 48
00F: 606A C08F 2B52 1BF6 EB
010: BF68 1E8F 9223 18F2 E1
011: 5231 20AE 6F23 0AF2 63
012: F2AE 9BF2 30AB F2BF BA
013: 2AEB 1B11 9F21 5271 D0
014: 0118 F152 7100 1612 25
015: 7154 0303 1601 5401 AC
016: 600D 5CE1 9102 0310 A8
017: 214C 2EAF 2328 0DAF A8
018: 7201 368F CB91 01B1 A4
019: 74F2 1423 300C 015C A4
01A: 3239 1271 1B87 4F21 FD
01B: 460B 8410 B154 31B1 85
01C: 49F2 146D 7001 B149 D6
01D: F2DB 1443 13D8 F106 5E
01E: 318F 2B52 11B1 49F2 9F
01F: 146D 7001 B149 F2DB E0
020: 1443 13D8 FAF5 3167 21
021: EEA7 000C 7000 14A3 59
022: 10E9 6224 7350 3070 60
023: E061 5101 F174 F233 B6
024: 0051 15D3 1F87 4F21 81
025: 470B 8410 B155 37A6 28
026: F8D8 4A80 7310 3080 BB
027: E0E1 5107 C7F6 5EF1 1B
```

```
028: FC79 F215 3001 8DCF C2
029: 2508 FA2C 200E 1101 5F
02A: EC00 008D B2E2 08D  E4
```

CLOCKDSP is a long LEX file but I can't live without it! As soon as you see it in action, you'll love it too. It adds two new commands to BASIC: CLOCK ON, and CLOCK OFF.

After executing CLOCK ON, the HP-71 will display a running digital clock in the right side of the display, yet magically you can still type and do anything you want, just like normal, in the left side of the display! It even works in CALC mode! CLOCK OFF clears it from the display and returns control of the entire display to you. Since it must protect the clock portion of the display from being typed over, it effectively performs a WINDOW 13 command once every second, thereby making WINDOW a useless function while the clock is on. CLOCK OFF resets the WINDOW to normal, whether the clock is on or not. This LEX file was written by HP.

Postscript:

THE HP-71 ANSWER-MAN SERVICE

Do you have any questions about your HP-71? Here is a terrific offer that is absolutely FREE! (Well, actually it is included in the price of this book, but I won't mention that). If you ever would like a personal answer to your own personal HP-71 questions, here's all you have to do:

(1) Write the question as clearly as possible on a piece of paper.
(2) Place a stamp and your address on a postcard.
(3) Place both your letter and the postcard in an envelope.
(4) Address the envelope to the author:

Joseph K. Horn
1042 Star Rt.
Orange, CA 92667

(5) Put enough postage on the envelope and mail it.

As soon as I can, I'll write the answer on the postcard and mail it back to you. If the above instructions are not carried out as listed, I cannot guarantee a reply. There is no limit to the number of times you may utilize this service.

Please note! Any question whose answer can fit on a postcard is okay! I don't care how mind-bogglingly complex or how mind-bogglingly simple your question may be. The complex ones are a challenge. The simple ones are a relief, answerable sometimes with just a page reference to this book. Either way, you get an answer, as long as the answer can fit on a postcard. Under no circumstances will I try to write a 100-line BASIC program on the back of a postcard! 99, maybe, but not 100. Also, please don't ask me to enclose things like magnetic cards with my reply, because it

is too difficult to enclose things in postcards.

DISCLAIMER: This service is offered solely by the author, not by the publisher or any other party. Do not mail questions or complaints regarding this service to any address other than the one listed above. Using this service implies consent for your question (but not name or address) to be used in future publications without recompense. This service will remain active until five years after the HP-71 has been removed from the official Hewlett-Packard Price List, at which time it will be terminated. Although the answers received from The HP-71 Answer Man will be as accurate as possible, no guarantee is expressed or implied, and under no circumstances will the author be held liable for their inexactness, inapplicability, or for subsequent or consequent damages due to their use. This legal mumbo-jumbo is required because there are mean-spirited people in this world that find it easier to sue than think; pay it no mind if you are a real HP-71 user.

HP-71 Quick Reference Guide

These are the keywords and other words used by the bare-bones HP-71 as well as the HP-71 Math Pac (HP 82480A), HP-71 HP-IL Interface (HP 82401A), HP-71 FORTH/Assembler ROM (HP 82441A), and a few common HP-71 User's Library Lex Files. Keywords in the bare-bones machine are followed by a page reference to the HP-71 Reference Manual that comes with the HP-71. All other keywords are followed by a reference to the ROM or LEX file in which they are found.

Keywords whose meaning or pronunciation is not obvious are followed on the next line by a rough English translation.

The syntax notation used here is the same as the notation used by all HP Quick Reference Guides, except for the use of \ instead of /, and the addition of curly brackets { } when needed for grouping. Example: `ADJUST (seconds \ time string)` means type `ADJUST` spelled just like that, then type either a number of seconds or a time string, but you must use one or the other. Square brackets, as usual, mean it's optional. Example: `CONT [line number \ label]` means type `CONT` spelled just like that, then you may type a line number or a label if you wish, but you don't have to use either one.

`[ ]`	optional
`{ }`	required
\	either/or
UPPERCASE WORD	keyword, to be spelled exactly as seen
`( ) , ;` etc.	to be used exactly as seen

A

ABS `(number)`. p.10
---"Absolute Value"---

ASIN (number). p.22
---"Arc Sine"---

ASINH (number). MATH ROM
---"Inverse Hyperbolic Sine"---

ASN (number). p.22
---"Arc Sine"---

ASSIGN # channel number **TO** {filespec \ ***** \ '' \ **'*'**}. p.23

ASSIGN IO assign code list string. HPIL ROM

@. Statement concatenator: statement @ statement. p.306
---"At"---

ATAN (number). p.25
---"Arc Tangent"---

ATANH (number). MATH ROM
---"Inverse Hyperbolic Tangent"---

ATH (number). p.25
---"Arc Tangent"---

AUTO [start line number [,increment]]. Keyboard Only. p.26

---------- B ----------

BASE. Postfix. See OPTION.

BASIC. Postfix. See TRANSFORM.

BEEP [frequency in Hz [,duration in sec] \ **ON** \ **OFF**]. p.28

BINAND (number, number). HPIL ROM
---"Binary And"---

BINCMP (number). HPIL ROM
---"Binary Complement"---

BINEOR (number, number). HPIL ROM
---"Binary Exclusive Or"---

BINIOR (number, number). HPIL ROM
---"Binary Inclusive Or"---

BIT (number, bit number). HPIL ROM

BSTR$ (number, base number). MATH ROM
---"Base String"---

BVAL (string, base number). MATH ROM
---"Base Value"---

BYE. p.30

C

CALL [subprogram name [(actual parameter list)] [**IN** filespec]]. p.31

CARD. Postfix. See CAT, COPY.

CAT [**ALL** \ **KEYS** \ **CARD** \ filespec \ **:MAIN** \ **:PORT** \ device]. p.35
---"Catalog"---

CAT$ (file number [,device string]). p.38
---"Catalog String"---

CEIL (number). p.40
---"Ceiling"---

CFLAG {**ALL** \ **MATH** \ flag number list -32 to 63}. p.41
---"Clear Flag"---

CHAIN filespec. p.42

CHARSET string. p.43
---"Character Set"---

CHARSET$. p.46
---"Character Set String"---

CHR$ (number 0 to 255). p.47
---"Character String"---

CLAIM [:]**PORT** (port number p.dd). Keyboard Only. p.48

CLASS (number). p.49

CLEAR [device \ **LOOP**]. HPIL ROM

CLOCK. Postfix. See RESET.

CLSTAT. p.51
---"Clear Statistics"---

CNORM (array). MATH ROM
---"Column Norm"---

COMPLEX var list. MATH ROM

CON. Postfix. See MAT.
---"Constant Matrix"---

CONJ (number). MATH ROM
---"Conjugate"---

CONT [line number \ label]. Keyboard Only. p.52
---"Continue"---

CONTRAST number 0 to 15. p.54

CONTROL {**ON** \ **OFF**} [loop number 1-3]. HPIL ROM

CONTROL. Postfix. See PASS.

COPY [filespec \ **CARD** \ **KEYS**] [**TO** {filespec \ :device \ **CARD** \ **KEYS**)]. p.55

COPY [filespec \ device \ **LOOP**] **TO** [filespec \ device \ **LOOP**]. HPIL ROM

CORR (var number, var number). p.58
---"Correlation"---

COS (number). p.59
---"Cosine"---

COSH (number). MATH ROM
---"Hyperbolic Cosine"---

CREATE {**TEXT** \ **LIF1** \ **DATA** \ **SDATA**} filespec [,file size [,record len]]. p.60

CREATE ALL. THEOLOGY ROM

D

DATA [data list]. Program Only. p.62

DATA. Postfix. See CREATE

DATE. p.65

DATE$. p.66
---"Date String"---

DEBUG. User's Library LEXflle #2, ID=53 hex. With DEBUGGER ROM only.

DEF. Postfix. See END.
---"Definition"---

DEF FNnumeric var [(parameter list)] [= expr].
Program Only. p.67
---"Define Function"--- e.g. DEF FNA(X) is "Define Function A of X"

DEF FNvar**$** [[string len]] [(parameter list)] [=expr]. Program Only. p.67
DEF FNA$(X$) is "Define Function A-String of X-String"

[**DEF**] **KEY** key name string [,assigned string [;\ :]]. p.69
---"Define Key"---

DEFAULT {**EXTEND** \ **ON** \ **OFF**}. p.72

DEG (number of radians). p.73
---"Degrees"---

DEGREES. p.74

DEGREES. Postfix. See OPTION.

DELAY line rate in sec [,scroll rate in sec]. p.75

DELETE (**ALL** \ start line number [,final line number]}. Keyboard Only. p.77

DELETE #channel number, record number. FORTH or TEXT EDITOR ROM

DESTROY {**ALL** \ var list}. p.78

DET or **DETL**. MATH ROM
---"Last Determinant"---

DET (square matrix). MATH ROM
---"Determinant"---

DEVADDR (device). HPIL ROM
---"Device Address"---

DEVAID (device). HPIL ROM
---"Device Accessory ID"---

DEVID$ (device). HPIL ROM
---"Device ID String"---

DIM numeric var [(dim limit 1 [,dim limit 2])]. p.79
---"Dimension"---

DIM var**$** [**(**dim limit**)**] [**[**string len**]**]. p.79

DISP [expr \ **TAB**(number) [{**;**\ **,**} expr]] [**;** \
,]... p.82
---"Display"---

DISP USING image **;** [expr] [**;**\ **,**]... p.84

DISP$. p.86
---"Display String"---

DISPLAY IS {device \ ***}. HPIL ROM

DIV. Operator: number **DIV** number. p.87
---"Integer Divide"---

****. Operator: number **** number. p.311
---"Integer Divide"--- (same as DIV)

DOT (vector, vector). MATH rom

DROP [number list]. p.88

DTH$ (decimal number). p.89
---"Decimal to Hex String"---

DVZ. p.90
---"Division by Zero"---

---------- E ----------

EDIT [filespec]. Keyboard Only. p.91

EDPARSE$ (command string). FORTH or TEXT EDITOR ROM
---"Editor Parse String"---

EDTEXT filename. FORTH or TEXT EDITOR ROM
---"Edit Textfile"---

ELSE. Postfix. See IF.

ENABLE INTR mask number. HPIL ROM
---"Enable Interrupt"---

END [**ALL**]. p.93

END DEF. Program Only. p.94
---"End Definition"---

END SUB. p.94
---"End Subprogram"---

ENDLINE [string]. p.96. Defines characters sent after carriage return.

ENG number 0 to 11. p.97
---"Engineering"---

ENTER {device \ **LOOP**} [**USING** image] [; var list]. HPIL ROM

EPS. p.99
---"Epsilon"---

ERRL. p.100
---"Error Line"---

ERRM$. p.101
---"Error Message String"---

ERRN. p.102
---"Error Number"---

ERROR. Postfix. See `OFF`, `ON`.

ESCAPE `string, key code number`. FORTH ROM

ESCAPE. Postfix. See `RESET`.

EXACT. p.103

!. Same as `@ REM`. p.242
---"Remark"---

EXOR. Operator: `number` **EXOR** `number`. p.105
---"Exclusive Or"---

EXP `(number)`. p.106
---"e to the x power"---

EXPM1 `(number)`. p.107
---"e to the x power, minus 1"---

EXPONENT `(number)`. p.108

EXTEND. Postfix. See `DEFAULT`.

---------- F ----------

FACT `(number)`. p.109
---"Factorial"---

FETCH `[line number \ label]`. Keyboard only. p.110

FETCH KEY `key name string`. Keyboard only. p.111

FGUESS. MATH ROM
---"Function Guess"---

FILESZR (filename string). FORTH or TEXT EDITOR ROM
---"File Size in Records"---

FIX number 0 to 11. p.112

FLAG (flag number [,new value 0 or 1]). p.114

FLOOR (number). p.115

FLOW. Postfix. See TRACE.

FNORM (array). MATH ROM
---"Frobenius Norm"---

FNROOT (first guess, second guess, function of FVAR). MATH ROM
---"Function Root"---

FNvar **=** expr. Inside multi-line FN definition only. p.116
---"Function"--- e.g. FNA=Q is "Function A equals Q"

FNvar [(parameter list)]. p.116
e.g. FNA is "Function A"

FNvar. Postfix. See DEF, LET.

FOR var **=** start number **TO** final number [**STEP** increment]. p.118

FORTH. FORTH ROM. Keyboard only.

FORTH$. FORTH ROM
---"Forth String"---

FORTHF. FORTH ROM
---"Forth Floating-point"---

FORTHI. FORTH ROM
---"Forth Integer"---

FORTHX command string [,parameter list]. FORTH ROM
---"Forth Execute"---

FOUR. Postfix. See MAT.
---"Fourier Transform"---

FP (number). p.121
---"Fractional Part"---

FREE [:] **PORT** (port number p.dd). p.122

FVALUE. MATH ROM.
---"Function Value"---

FVAR. MATH ROM.
---"Function Variable"---

G

GAMMA (number). MATH ROM

GDISP string. p.124
---"Graphically Display"---

GDISP$. p.127
---"Graphic Display String"---

GO {**SUB** \ **TO**}. See GOSUB, GOTO.

GOSUB {line number \ label}. p.129

GOSUB. Postfix. See ON.

GOTO {line number \ label}. p.131

GOTO. Postfix. See ON.

---------- H ----------

HPIL. Postfix. See RESET.
---"H.P.I.L."--- (Hewlett-Packard Interface Loop; HP people say "Pill")

HTD (hex string 0 to FFFFF). p.133
---"Hex to Decimal"---

---------- I ----------

IBOUND. MATH ROM
---"Integration Bounds"---

IDN. Postfix. See MAT.
---"Identity Matrix"---

IF expr **THEN** {statement \ line number \ label} [**ELSE** {statement \ line number \ label}]. p.134

IMAGE [**#** {pg ctrl item \ ,}] format string. p.136

IMPT (number). MATH ROM
---"Imaginary Part"---

IN. Postfix. See CALL.

INF. p.150
---"Infinity"---

INITIALIZE [volume] device [,directory size]. HPIL ROM

INPUT [prompt string [,default string];] variable list. p.151

INSERT #channel number, record number; string. FORTH or TEXT EDITOR ROM

INT (number). p.154
---"Integer"---

INTEGER variable list [**(**dims after each**)**]. p.155

INTEGRAL (low limit, high limit, error limit,
function of IVAR). MATH ROM

INTO. Postfix. See TRANSFORM.

INTR. Postfix. See ENABLE, ON.
---"Interrupt"---

INV. Postfix. See MAT.
---"Inverse Matrix"---

INX. p.157
---"Inexact"---

IO. Postfix. See ASSIGN, LIST, OFF, RESTORE.
---"I. O."--- (stands for "Input/Output")

IP (number). p.158
---"Integer Part"---

IROUND (number). MATH ROM
---"Integer Round"---

IS. Postfix. See DISPLAY, KEYBOARD, PRINTER.

IVALUE. MATH ROM
---"Integration Value"---

IVAR. MATH ROM
---"Integration Variable"---

IVL. p.159
---"Invalid"---

---------- K ----------

KEY key name string [,assigned string [; \ :]]. p.69
---"Define Key"---

KEY. Postfix. See DEF, FETCH.

KEY$. p.160
---"Key String"---

KEYBOARD IS {device \ *}. FORTH ROM

KEYDEF$ (key name string). p.162
---"Key Definition String"---

KEYDOWN [(key name string)]. p.164

KEYS. Postfix. See CAT, COPY, LIST, PLIST, PURGE, RENAME, SECURE, UNSECURE.

KEYWAIT$. User's Library LEXfile »1, ID=52 hex.
---"Keywait String"---

---------- L ----------

LBND (array, {1 \ 2}). MATH ROM
---"Lower Bound"---

LBOUND (array, {1 \ 2}). MATH ROM.
---"Lower Bound"---

LC [**ON** \ **OFF**]. p.166
---"Lower Case"---

LEN (string). p.167
---"Length"---

[**LET**] var **=** expr. p.168

[**LET**] **FN**var **=** expr. Only inside multi-line FN definition. p.168

LGT (number). p.178
---"Log Base Ten"---

LIF1. Postfix. See CREATE, TRANSFORM.
---"Lif One"--- (Logical Interchange Format One)

LINPUT [prompt string [,default string];] var$. p.171
---"Line Input"---

LIST [start line number [,final line number]]. p.173

LIST filespec [,start line or key number [final line or key number]]. p.173

LIST KEYS [,start key number [final key number]]. p.173

LIST IO. HPIL ROM

LN (number). p.176
---"Natural Log"---

LOCAL [device \ **LOOP**]. HPIL ROM

LOCAL LOCKOUT [loop number 1-3]. HPIL ROM

LOCK password string. p.175

LOCKOUT. Postfix. See LOCAL.

LOG (number). p.176
---"Natural Log"--- (same as LN, log base e, not log base 10)

LOG10 (number). p.176
---"Log Base 10"---

LOG2 (number). MATH ROM
---"Log Base 2"---

LOGP1 (number). p.177
---"Natural Log of x-plus-1"---

LR y var number, x var number [,intercept var
[,slope var]]. p.179
---"Linear Regression"---

---------- M ----------

MAIN. Postfix. See CAT.

MAT array **=** array. MATH ROM
---"Matrix"---

MAT array **=** (number). MATH ROM

MAT array **= CON** [(number [, number])]. MATH ROM

MAT array **= IDN** [(number, number)]. MATH ROM

MAT array **= ZER**[**O**] [(number [, number])]. MATH ROM

MAT INPUT array list. MATH ROM

MAT DISP [**USING** image ;] array list. MATH ROM

MAT PRINT [**USING** image ;] array list. MATH ROM

MAT array **= -** array. MATH ROM

MAT array **=** array **+** array. MATH ROM

MAT array **=** array **-** array. MATH ROM

MAT array **= (**number**) *** array. MATH ROM

MAT array **=** array ***** array. MATH ROM

MAT array **= TRN** (array) ***** array. MATH ROM

MAT array **=** **INV** (array). MATH ROM

MAT array **=** **TRN** (array). MATH ROM

MAT array **=** **SYS** (array, array). MATH ROM

MAT complex array **=** **PROOT** (real array). MATH ROM

MAT complex array **=** **FOUR** (complex array). MATH ROM

MATH. Postfix. See CFLAG, SFLAG.

MAX (number, number). p.181
---"Maximum"---

MAXREAL. p.182
---"Maximum Real Number"---

MEAN [(var number)]. p.183

MEM [(port number p.dd)]. p.184
---"Memory"---

MERGE filespec [,start line or key number [,final line or key number]]. p.186

MIN (number, number). p.188
---"Minimum"---

MINREAL. p.189
---"Minimum Real Number"---

-. Operator: number **-** number. p.309
---"Minus"---

MOD (number, number). p.190
---"Modulo"---

MSG$ (message number). User's Library LEXfile #l, ID=52 hex.
---"Message String"---

N

NAME filespec. p.191

NAN. p.192
---"Not-a-Number"---

NAN$ (not-a-number). MATH ROM
---"Not-a-Number String"---

NEAR. Postfix. See OPTION.

NEG. Postfix. See OPTION.
---"Negative"---

- number or NaN. Negative unary operator. p.309
---"Negative"---

NEIGHBOR (number, direction number). MATH ROM

NEXT numeric var. p.118.

NOT number. p.193

NUM (string). p.194
---"Character Number"---

O

OFF. p.195

OFF {**ERROR** \ **TIMER #** timer number}. p.195

OFF. Postfix. See BEEP, CONTROL, DEFAULT, LC, STANDBY, TRACE, USER.

OFF {**INTR** \ **IO**}. HPIL ROM.

ON number {**GOSUB** \ **GOTO** \ **RESTORE**} line number and label list. p.199

ON ERROR {**GOSUB** \ **GOTO**} {line number \ label}. p.197

ON INTR {**GOSUB** \ **GOTO**} {line number \ label}. HPIL ROM
---"On Interrupt"---

ON TIMER # timer number, seconds {**GOSUB** \ **GOTO**} {line number \ label}. p.201

ON. Postfix. See BEEP, CONTROL, DEFAULT, LC, STANDBY, USER.

OPTION ANGLE {**DEGREES** \ **RADIANS**}. p.204

OPTION BASE {0 \ 1}. p.204

OPTION ROUND (**NEAR** \ **NEG** \ **POS** \ **ZERO**). p.204

OR. Operator: number **OR** number. p.206

OUTPUT {device \ **LOOP**} [**USING** image] [; output list]. HPIL ROM

OVF. p.207
---"Overflow"---

---------- P ----------

PACK device. HPIL ROM

PACKDIR device. HPIL ROM
---"Pack Directory"---

PASS CONTROL [device \ **LOOP**]. HPIL ROM

PAUSE. p.208. Program only.

PCRD. Card Reader only.
---"Private Card"---

PEEK$ (hex address string, number of nibbles). p.299
---"Peek String"---

PI. p.210
---"π"---

PLIST [start line number [,final line number]]. p.211
---"Print List"---

PLIST filespec [,start line or key number [,final line or key number]]. p.211

PLIST KEYS [,start key number [,final key number]]. p.211

POKE hex address string, data string. p.213

POLAR (number). MATH ROM

POP. p.215

PORT. Postfix. See CAT, CLAIM, FREE, SHOW.

POS (string being searched, substring looked for [,start number]). p.216
---"Position"---

POS. Postfix. See OPTION.
---"Positive"---

PREDV (number). p.218. Must be initialized by executing LR.
---"Predicted Value"---

PRINT [expr \ **TAB**(number) [{;\ ,} expr]] [; \ ,]... p.219

PRINT USING image ; [expr] [;\ ,]... p.219

PRINT #channel number [, record number]; [data \
array] [,...]. p.223

PRINTER IS {device \ ***}. HPIL ROM

PRIVATE filespec. p.225

PROJ (number). MATH ROM
---"Projective Infinity"---

PROOT. Postfix. See MAT.
---"Polynomial Roots"---

PROTECT. p.226. Card Reader only.

PURGE [filespec \ **KEYS** \ **ALL**]. p.227

PUT keyname string. p.229

PWIDTH number. p.230
---"Printer Width"---

---------- R ----------

RAD (number). p.231
---"Radians"---

RADIANS. p.232

RADIANS. Postfix. See OPTION.

RANDOMIZE [number]. p.233

READ [**#**channel number [, record number] ;] {var \
array} [,...]. p.234,6

READDDC. HPIL ROM
---"Read Device Dependent Command"---

READINTR. HPILROM
---"Read Interrupt"---

REAL var [(number [, number])] [,...]. p.238

RECT (number). MATH ROM
---"Rectangular"---

RED (number, number). p.240
---"Reduce"---

REM. p.242. Abbreviated form: !.
---"Remark"---

REMOTE {device \ **LOOP**}. HPIL ROM

RENAME [filespec \ **KEYS**] **TO** {filespec \ **KEYS**}. p.243

RENUMBER [new start [, increment [, old start [, old final]]]]. p.245

RENUMBER 1,1,1,1. Compiles all branches without renumbering anything.

REPLACE #channel number, record number; string. FORTH or TEXT EDITOR ROM.

REPT (number). MATH ROM
---"Real Part"---

REQUEST status number. HPIL ROM

RES. p.247
---"Result"---

RESET [**CLOCK**]. p.248

RESET HPIL [loop number 1-3]. HPIL ROM

RESET ESCAPE. FORTH ROM

RESTORE [line number \ label]. p.249

RESTORE IO [loop number 1-3]. HPIL ROM

RESTORE #channel number [, record number]. p.250

RESTORE. Postfix. See ON.

RETURN. p.251

RMD (number, number). p.252
---"Remainder"---

RND. p.254
---"Random Number"---

RNORM (array). MATH ROM
---"Row Norm"---

ROUND. Postfix. See OPTION.

RUN [line number \ filespec [, {line number \
label}]]. p.255

---------- S ----------

SCALE10 (number, integer number). MATH ROM

SCI number from 0 to 11. p.257
---"Scientific"---

SCROLL position number. User's Library LEXfile #1, ID=52.

SDATA. Postfix. See CREATE.
---"Stream Data"---

SDEV [(var number)]. p.259
---"Standard Deviation"---

SEARCH (string, col number, start rec, end rec, channel). FORTH or TEXT EDITOR ROM

SECURE [filespec \ KEYS]. p.260

SEND hpil message list. HPIL ROM

SETDATE {date number \ date string}. p.262

SETTIME {time number \ time string}. p.264

SFLAG {**ALL** \ **MATH** \ flag number list -32 to 63}. p.267
---"Set Flag"---

SGN (number). p.268
---"Sign"---

SHORT var [**(**number [, number]**)**] [,...]. p.269

SHOW [:]**PORT**. p.271

SIN (number). p.272
---"Sine"---

SINH (number). MATH ROM
---"Hyperbolic Sine"---

SPOLL (device). HPIL ROM
---"Status Poll"---

SQR (number). p.273
---"Square Root"---

SQRT (number). p.273
---"Square Root"---

STANDBY {**ON** \ **OFF** \ timeout number [, verify interval number]). HPIL ROM

STARTUP command string. p.274

STAT array [(number of vars)]. p.275
---"Statistics"---

STATUS. HPIL ROM

STD. p.277
---"Standard"---

STEP. Postfix. See FOR.

STOP. p.279

STR$ (number). p.280
---"String String"---

SUB subprogram name [**(**formal parameter list**)**]. p.282
---"Subprogram"---

SUB. Postfix. See END.

SYS. Postfix. See MAT.
---"System of Equations"---

---------- T ----------

TAB. Postfix. See DISP, PRINT.
--- (short for "tabulator") ---

TAN (number). p.284
---"Tangent"---

TANH (number). MATH ROM
---"Hyperbolic Tangent"---

TEXT. Postfix. See CREATE, TRANSFORM.

THEN. Postfix. See IF.

TIME. p.285

TIME$. p.286
---"Time String"---

TIMER. Postfix. See OFF, ON.

TO. Postfix. See ASSIGN#, COPY, FOR, RENAME.

TOTAL [(var number)]. p.287

TRACE {**FLOW** \ **VARS** \ **OFF**}. p.288

TRANSFORM [filespec] **INTO** {**BASIC** \ **TEXT** \ **LIF1**}
[new filespec]. p.289

TRAP (flag number [,new value 0 or 1]). p.293

TRIGGER [device \ **LOOP**]. HPIL ROM

TRN. Postfix. See MAT.
---"Transpose Matrix"---

TYPE (number \ string \ array). MATH ROM

U

UBND (array, {1 \ 2}). MATH ROM
---"Upper Bound"---

UBOUND (array, {1 \ 2}). MATH ROM
---"Upper Bound"---

UNF. p.295
---"Underflow"---

UNPROTECT. p.296. Card Reader only.

UNSECURE [filespec \ **KEYS**]. p.297

UPRC$ (string). p.298
---"Uppercase String"---

USER [**ON** \ **OFF**]. p.299

USING. Postfix. See DISP, ENTER, OUTPUT, PRINT.

---------- V ----------

VAL (string). p.300
---"Value"---

VARS. Postfix. See TRACE.
---"Variables"---
VER$. p.301
---"Version String"---

---------- W ----------

WAIT (number of seconds). p.302

WIDTH number. p.303

WINDOW first column number [, last column number].
p.305

---------- Z ----------

ZER. Postfix. See MAT.
---"Zero Matrix"---

ZERO. Postfix. See MAT, OPTION.
---"Zero Matrix"---

USERS' GROUPS

Two international users' groups support the HP-71 calculator. Any serious HP-71 programmer should join one or both of these groups. For further information and membership applications, send $1 to:

Club of HP Handheld Users
2545 W. Camden Place
Santa Ana, CA 92704
U.S.A.

or

PPC
P.O. BOX 9599
Fountain Valley, CA 92728
U.S.A.

GET THE MOST FROM YOUR HP-71!

The HP-71 sets new standards for handheld performance. **HP-71 BASIC Made Easy** supplements your Owner's Manual in several important areas to help you realize your HP-71's full potential, from quick keyboard calculations to BASIC to machine language.

Learn how to calculate efficiently and confidently using CALC mode and the command stack. You may find that you like the HP-71's CALC mode even more than the RPN logic used on the HP-41 and other HP calculators.

General tips on keyboard BASIC are followed by details on how and where to use PEEK and POKE. Several BASIC application programs provide instruction in programming technique besides being useful in themselves.

One application program lets you key in machine language programs from numeric listings. The examples show you just how powerful these machine language programs can be. Carve out a continuous clock display at the right side of the display window; reverse a 14,000 character string in 1 second!

A 20-page alphabetical syntax guide lists hundreds of HP-71 keywords. **HP-71 BASIC Made Easy** is an excellent tutorial and an essential reference. Find out how much fun your HP-71 can be!

ISBN: 0-9612174-3-X