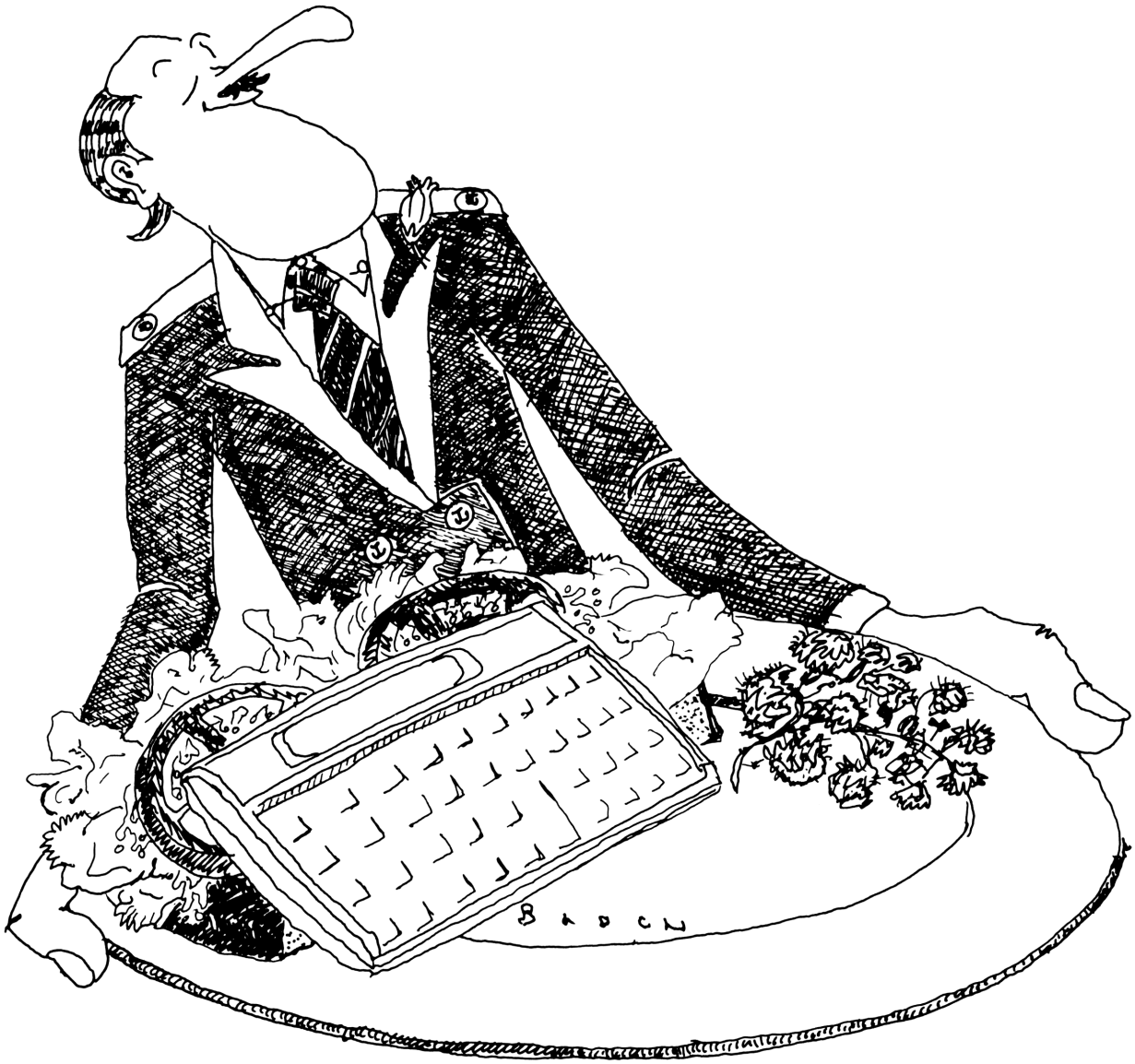
An Easy Course in Using



The HP-16C

by Edward M. Keefe
Illustrated by Robert L. Bloch

AN EASY COURSE IN USING THE HP-16C

By Edward M. Keefe

Illustrated by Robert L. Bloch

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

Acknowledgement

Thanks and appreciation go once again to the Hewlett-Packard Company for continuing to produce such top quality products and documentation.

Printed in The United States of America
First Printing -- July, 1987

ISBN 0-931011-16-7

TABLE OF CONTENTS

WELCOME!

to the world of the HP-16C Calculator. It's quite a strange place, really--all filled with nothing but zeros and ones. It's the world of computer science and engineering, full of integrated circuits and digital readouts.

And it's a world where people speak binary, hexadecimal, and octal as fluently as they do English (...well...in some cases, at least).

This language isn't an easy one for beginners to master, and so some of the people who live and work in that world have invented a very handy tool to make it all a snap. These folks--at Hewlett-Packard--have built a calculator they call the "Computer Scientist." Computer scientists call it the HP-16C (and you can call it anything you like--Fred, Martha, whatever).

The HP-16C is very handy, partly because it slips conveniently into a briefcase or shirt pocket--and because it fits very neatly in the palm of your hand--but its real value is in what it does for you.

And what exactly does the HP-16C do for you? Well, first and foremost, it's a language translator--helping you move freely between the languages of decimal, binary, hexadecimal and even octal numbers. And beyond that, it helps you learn about the math and logic that people use in this strange world.

Because of this, the HP-16C will let you waltz your way through most of the introductory material in computer-math courses. And it will stay with you as you move from classroom theory to the practice of designing computer software and/or hardware on your own.

In fact, just about the only thing this calculator *won't* do is teach you how to use it. And, like a lot of other people, you've probably found that the Owners' (Users') Manual that comes with the HP-16C is a very good reference manual (i.e. it's handy to use--once you know what you're doing). But it wasn't written as a learning guide; it assumes you already "speak the language."

And of course, that's not the case for people who are new to the world of computers....

Aha!...that's where a book like this comes in:

If you're a beginner in computer math or with the HP-16C, then this book has you in mind. Not only will you learn how to use the HP-16C, but you'll also learn the theory and language of the subject.

This book is a self-teaching course; to get the most out of it, just follow the directions as you work through it. If you know some of the material already, no problem--there will be places where you're allowed to skip ahead to the next part of the book. So you'll be learning at your own pace (and you can even repeat the course, if you like--no extra charge for this).

But here's one word of caution:

Beware the Button-Pushing Syndrome!

The dreaded BPS will strike when you are tired or mentally distracted. Here's how to recognize it:

There you are, merrily pushing all the right buttons on your HP-16C--and you're getting the same answers as shown in the book ("gee, this is easy!")....

...But you haven't the foggiest idea what you've done--or why the answers are correct.

The best cure for this is to set the HP-16C aside for a time and get some sleep or relaxation. Then come back to this course when you're refreshed and mentally alert.

And by all means, read the sections of this book that give some of the theory behind the HP-16C. That way, by the time you've learned the theory, you'll also know how to convert numbers from one base to another *in your head or on paper.* This will at least immunize you against Terminal BPS.

You do know what that is, don't you? TBPS occurs most often during computer math tests. You think you understand all the material for the test, and [so the rationalizing goes] even if you don't, you always have your HP-16C. No sweat.

Just then, the professor announces: "No calculators during my tests!" Yes sweat.

That's Terminal BPS. And if you've ever seen someone go down with it during a final exam.....well, it's not a pretty sight.

So please: Take the time to really digest this course. Then you'll have the skill to use your HP-16C--*and* the knowledge that you could go "cold turkey" without it. It's a tool, not a crutch. OK? Fine. Then it's time to get going...

THE BIG PICTURE

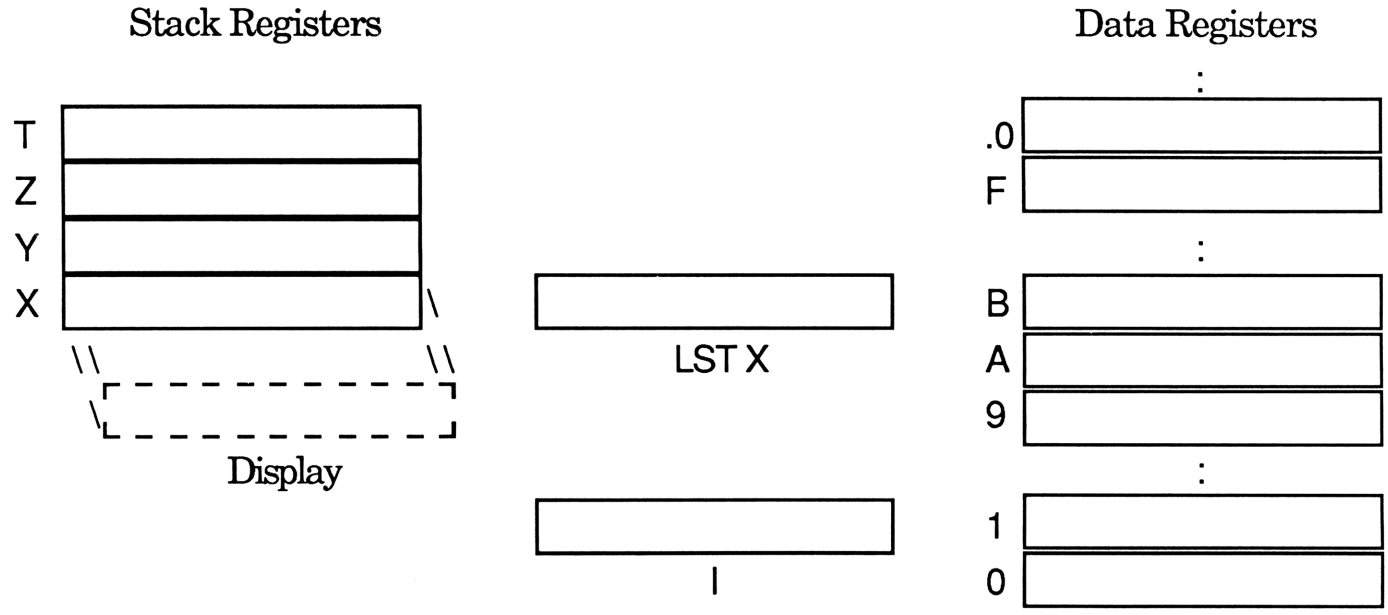
Your Calculator's Memory

It's always best to start with a look at the machine as a whole: What is it? How should you picture it in your mind? What are its various parts?

Well, your HP-16C is really two calculators in one. Of course, it's built with an "Integer Mode" to help you solve your Integer math computer problems. But there's another mode, called "Floating Point mode." There, you have an ordinary, powerful HP calculator, one that will do arithmetic on ordinary decimal numbers, take square roots and reciprocals, etc. (and if you've never used an HP calculator before, you're going to wonder how life has had any meaning at all).

But first you have to learn how to talk to your HP-16C, so that's where to start (actually, if you already understand the registers shown here, go on to page 16).

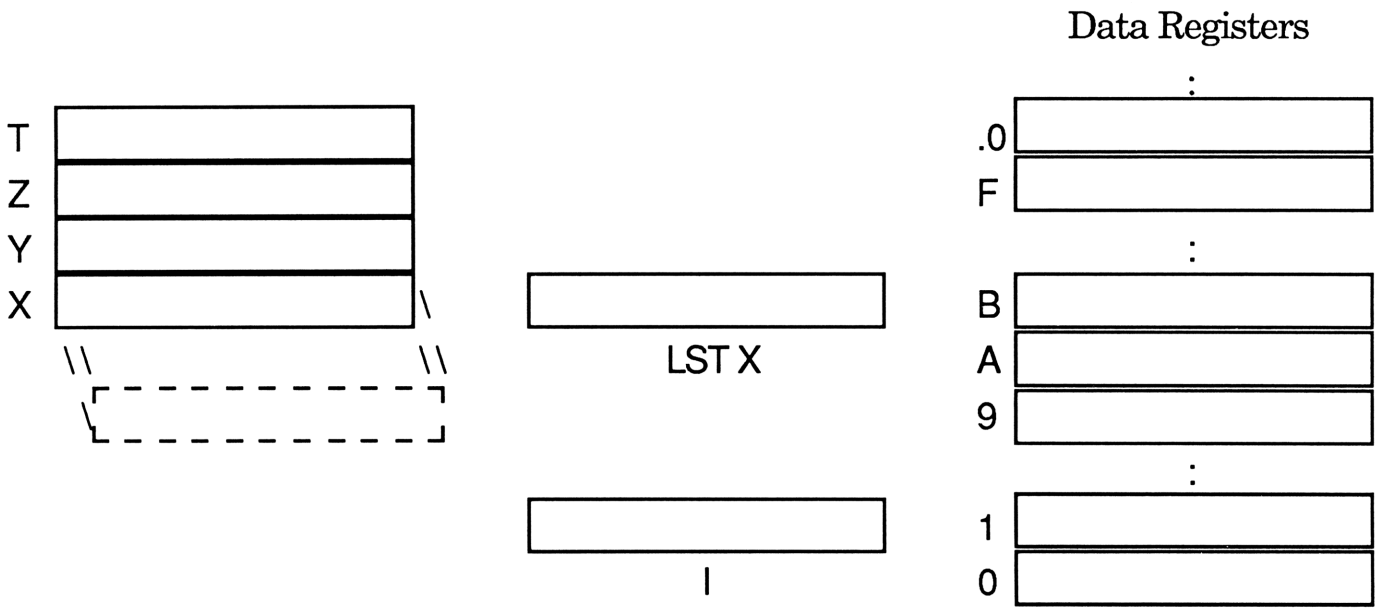
Imagine, if you will, this picture of the memory of the HP-16C:



Each of these boxes represents a location in your calculator's memory. And each of these memory locations ("registers") is associated with a number or letter.

Data Registers

Those boxes on the far right are called data registers. As you might guess, they hold data (numbers)--one per register.



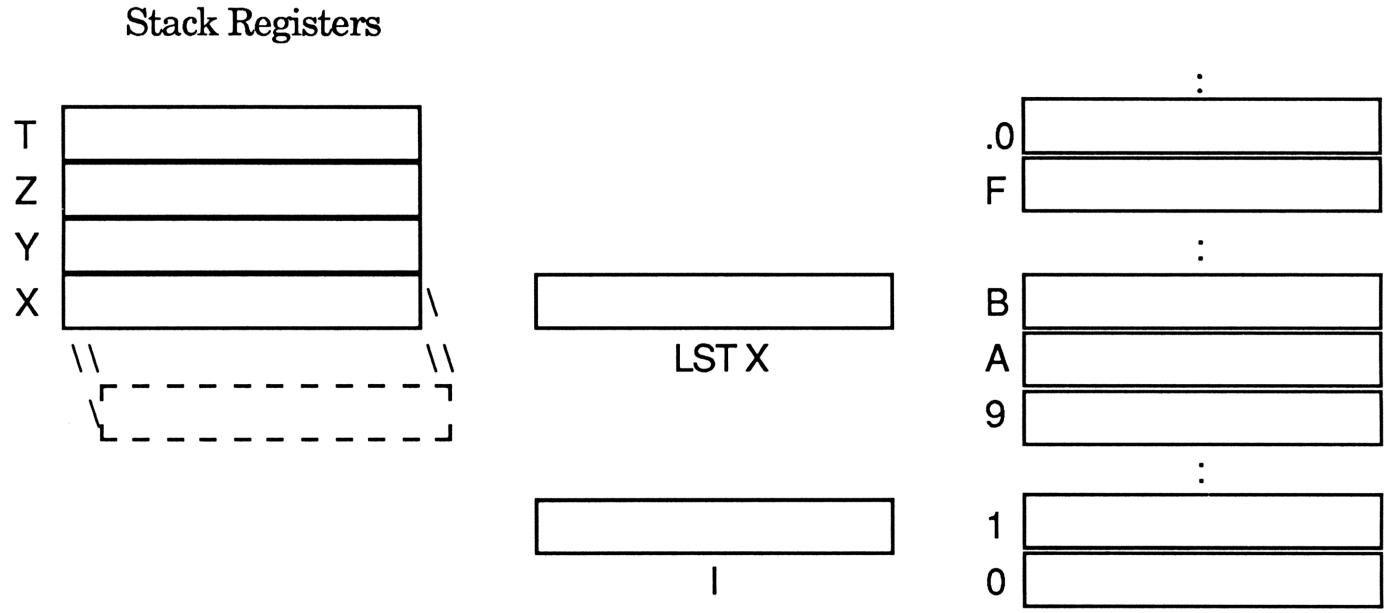
Once you store a number in any given data register in the HP-16C, it's there for good (or at least until the batteries get tired). The only way you can get rid of it is to store a different number there instead (think about it: even when you "clear" a register, you're really just storing a zero there, right?).

Now, as you can see, the data registers are given numbers as names, running from 0 to 9 and then from A to F (A-F are the hexadecimal--base 16-- equivalents of 10 through 15).

Then there is a second set of registers beyond that. HP chose to number these extra register with the codes .0 through .9 and .A through .C.

OK so far? Those are the data registers.

The Stack

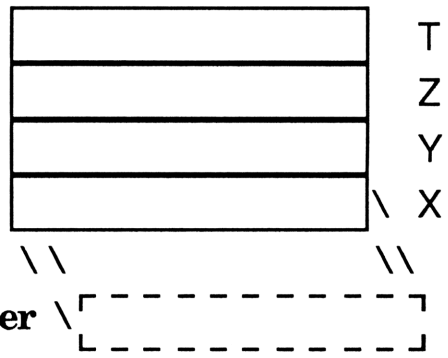


Take a look at those four stack registers there on the left. Actually, they're just ordinary data registers--with one important difference: HP designed them to work together--automatically--in a "stack." This is the key to great arithmetic!

Obviously, the four main stack registers are X, Y, Z, and T; another register is called LST X. If you've never worked with the stack before, you're in for a very pleasant surprise ("so stay tuned").

The I-Register

Oh, yes: Before moving on, it's probably wise to give a slight nod to the I-register. You see it there, on the bottom, in the middle of the diagram? It's really just another data register, but as you'll find out later, it can have special uses when you're programming your HP-16C.



The Display Register

The display register is one unlike any other: it acts as an interpreter between you and your HP-16C. When looking at your HP-16C, you actually see only what's in this display register, and this is *only its interpretation* of the X-register.

But you can instruct the display register exactly how to interpret the X-register. For example, to adjust your HP-16C to show you 2 decimal places, you would press: f FLOAT 2 (go ahead--do it).

The display now shows 2 decimal places. Then to see, say, 6 places, press f FLOAT 6. Get the idea? Notice that the display actually rounds the edited version.

"OK, but do I reduce the accuracy of the calculator if I limit the display?" Nope. All numbers stored in the HP-16C always retain their full values.

Try This: Key in this number: 1.765739492

Now press f FLOAT 0.

Of course, the display will show you only: 2.

But now press f FLOAT 5. You'll see: 1.76574

So remember! The *display* register is doing this rounding for you. The X-register (and every data register) retains full 10-digit accuracy. OK?

Unexpected Evaluative Exercise*

1. What's a data register?

2. How many data registers are available in the HP-16C in Floating Point Mode?

3. What type of register is the T-register? The I-register?

4. What is the display register (also simply called "the display")?

* Pop Quiz

Pop Answers

1. A data register is a memory storage location somewhere inside the calculator. It holds one number at a time.

2. The HP-16C can have up to 29 numbered data registers in Floating Point Mode. There are also the 4 stack registers, along with the Last-X register, the I-register, and the display register.

3. The T-register is one of the stack registers: the Topmost register. The I-register is another data register (you haven't heard much about it yet).

4. The display is another register in the HP-16C. It's a special type of register that *interprets* the number in the X-register, showing the rounded-off version of the number.

How did you do? If you gave answers similar to those on the previous page, then you already have a good "mental picture" of the insides of the HP-16C.

On the other hand, if you missed some of the answers, you really should go back and re-study the material you just read. Start on page 12--and take your time; there's no hurry at all. The next section can wait for as long as you want it to....

Ready? All right, what do you know so far?

You've seen what the stack looks like and how the data registers and the stack registers are named in the HP-16C. Remember that "register" is simply a convenient word for a "storage bin" in the memory of the HP-16C.

You also learned how to adjust the display register to vary the number of digits after the decimal point (when you're operating in Floating Point Mode, that is).

What about the rest of the keyboard? What is all that stuff?...

KEYS AND THE KEYBOARD

The Prefix Keys

Up to now, you've been using the prefix keys without really being told what they do (you probably guessed anyway, right?):

The gold [f] key will let you execute any operation that appears in gold letters on the keyboard of the HP-16C. And the blue [g] key will execute the blue-labelled functions.

Notice that you have to use these keys just like shift keys on a typewriter--once for every shifted function you want to use.*

Notice also that when you press the [f] or [g] keys, their annunciators appear (those tiny little symbols beneath the digits) in the display.

You can have one or the other prefix in effect--but not both. If you want neither, just press [f] CLEAR [PREFIX]. And if you do this in Floating Point mode (the mode you're working in now), it has a secondary effect, too: It momentarily shows you all 10 digits of the number in the X-register.

*Wondering why the gold key isn't labelled [g] for gold--and the blue key [b] for Blue? When Hewlett-Packard started making calculators, they decided to make some keys do double duty. They labelled the prefix key with the letter [f] for "function." But later, when they found they needed to make some keys do triple duty, they added a second prefix key, and this one they labelled [g], because "g" follows "f" in the alphabet. (Well, you asked.)

Keying In Numbers

Of course, you'll want to use your HP-16C just as a desktop calculator--as well as a computer science tool. In fact, that's why you have this Floating Point Mode in the first place--and it's time to start doing it.

But before you start to key in any kind of number, consider: Do you know how to key in negative numbers--or very large or very small numbers?

(If so, then you can probably skip over now to page 23.)

The [CHS] Key

The [CHS] key will change the sign of the number in the X-register.

Try This: Put -44.0 into the X-register.

Solution(s): [4][4][CHS] *or* [4][CHS][4]

(But you *can't* do it by pressing the [CHS] button as the first key-stroke in the sequence).

If you now want to change from - to + again, just press the [CHS] key once more.

See how it alternates? You'll find several such on-again-off-again keys on the HP-16C; these keys are usually known as *toggle* keys.

The EEX Key and Exponential Notation

You use the EEX key for entering very large or very small numbers in exponential notation.

Try This: How would you enter 13,400,000 into the X-register--without using the 0 key?

Solution: Press 1 · 3 4 f EEX 7.

EEX stands for Enter EXponent, which means that under this format, you are using exponential (also called "scientific") notation to represent this number. And while you're keying it in, that's the form the number takes in your display.

Of course, once you press ENTER or any other key that terminates this numeric entry, the display will show this number in whatever mode you've requested.

Right now, for example, you're probably in "Float 2" or "Float 5" or something; so you see just that many decimal places.

But you could ask your display to show you exponential notation *always,* if you wanted: Just press f FLOAT ·.

Sudden Skill-Assessment Session

1. What are the prefix keys on the HP-16C, and what do they do?

2. How do you clear the prefix annunciators from the display?

3. What is the [CHS] key, and when do you use it?

4. How would you key in -0.00000789?

Inevitable Conclusions

1. The prefix keys are the [f] and [g] keys. You must press (and release) the appropriate prefix before executing one of the gold or blue functions on the keyboard.

2. You can clear the prefix annunciators by pressing [f] CLEAR [PREFIX], (which is the gold version of the [BSP] key).

3. The [CHS] key changes the sign (+ or -) on the number in the X-register. You can use it while you're keying a number in, or after it's already fully entered--but not before you begin; there has to be at least one digit before you can change its sign.

4. Press [7] [.] [8] [9] [CHS] [f] [EEX] [CHS] [8].

   (Do you see how you change the sign of an exponent?)

OK, so now you've seen a bit more of the keyboard--and how to enter floating-point numbers (i.e. those with decimal points and fractions) into the X-register.

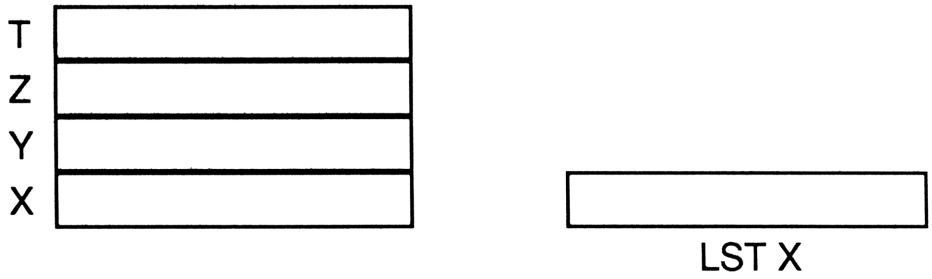
Wunderbar.

Now what do you do with them once you've got them there?...

GETTING TO KNOW THE STACK

Getting Acquainted With the Stack

Whenever you do arithmetic with the HP-16C, you'll be using the set of five data registers called the stack. The registers are labelled X, Y, Z, T, and LSTX, and as you saw a few pages back, they're usually shown as a set, like this:



(If you're sure you know all about the stack, you may skip ahead to page 35.)

As you may know--if you've ever worked with an RPN calculator before--the HP-16C has no [=] key--nothing you press to get your answer--or so it seems.

So how can you possibly do arithmetic? And what is this "RPN," anyway?

As with all of HP's RPN calculators, the HP-16C's method for performing arithmetic is based loosely on the work of a Polish logician and mathematician, Jan Lukasiewicz* (1878-1956).

His writings in symbolic logic are full of abstract statements such as "AND A B," which most other logicians would have written as "A AND B." But Professor Lukasiewicz had adopted a shorthand of his own, where he placed the logical operator (AND, OR, IF, etc.) in front of the quantities on which it would operate.

Well, HP borrowed this "Polish Notation" but chose to place the operators *after* the quantities on which they would operate. Thus we get the name, "Reverse Polish Notation," or RPN for short.

HP borrowed RPN for its simplicity and logical efficiency. For example, you might normally expect to press:

25 + 65 =

and get the result upon pressing the =;

but an "RPNer" would instead press

25 ENTER 65 +

and get the result upon pressing the +.

*pronounced "Voo-ka-szee-vich"

The real beauty of this is that since all operators need only one or two "arguments" (numbers to operate upon), there is no need for any parenthesis keys with RPN--unlike the algebraic calculators (those with [=] keys).

In this sense, the RPN system is much more "natural."

After all, if you'll think back for a minute to when you first learned how to do addition, you'll remember how you wrote the problems out like this:

$$\begin{array}{r} 25 \\ +\ 65 \\ \hline 90 \end{array}$$

See? You *stacked* the 25 and 65 and *then* added them. That's exactly what the HP-16C stack does, too.

And later, when you were practicing with a big, hairy problem like this, how were you taught to simplify and solve it?

$$\frac{(129 / ((63.5 + (27 - 49))) - 11}{((93 + 42) \times 76) - 80}$$

"Work from the inside parentheses outward."

Again, that's exactly what you do here when you're doing arithmetic with the stack!

OK, knowing just that much, then, it's time to start exploring this stack--and the keys you use to fill it, adjust it, and crunch it:

Try This: First, press the f FLOAT 3 keys to adjust the display to three decimal places.

Next, key in 23.456 (of course, you would do this by pressing: 23.456).

Then press ENTER.

Next, key in 9.6 ...and press ENTER.

Then key in 3.758

...but now pretend you made a mistake: change this number to 3.5 , instead (by pressing: BSP BSP BSP 5).

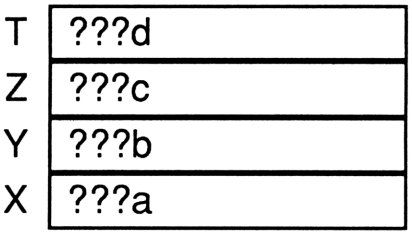
Now press the × key, to multiply.

Now press the BSP key one more time. Your number is gone, right? OK, so key in, say, 1.4 , and press +.

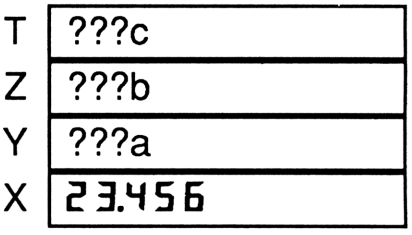
What's going on here? To find out, do it over again--with pictures this time--and watch the stack (but if you already know, go ahead and skip over to page 33).

(You'll notice that there's no LSTX register shown in any of the following diagrams. That'll come a little later on. For right now, just use this example as a "warmer-upper"--just to get used to how the four main stack registers generally work together.)

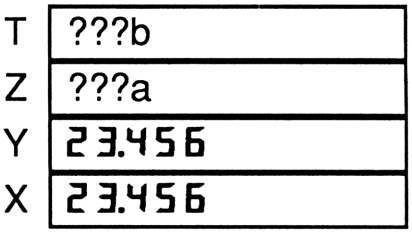
Here's how the stack starts out (the ???x means that you don't know what numbers are in these four registers--and you don't care--but you want somehow to identify each number and follow its movements):



Now key in your **23.456** ...and the stack now looks like this, because stack lift was "enabled." This means that whatever the machine had just done previously (⊞ in this case) is an operation that finishes by leaving the stack "enabled" to lift and make room for the next number coming into the X-register.



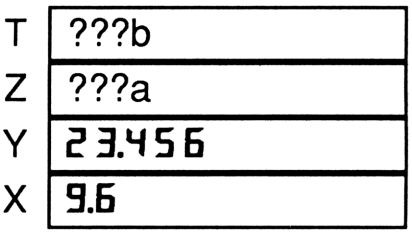
Next, you press the [ENTER] key, terminating digit entry:



Two other things happen here also, but one is more obvious than the other: First, [ENTER] performs a stack lift, bumping every number up one register (this means that the T-register's previous contents are popped off the top--lost for good--and the X-register is duplicated). You can see this, all right, just by looking at the numbers in the stack.

But secondly, [ENTER] *disables* the stack, so that if the very next operation brings a new number to the stack, then this number will *not* bump everything up one notch; rather, it will overwrite (replace) the **23.46** in the X-register.

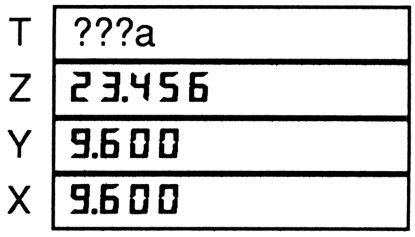
Prove this: Key in your **9.6** (which is the next step in this little repeat performance) and look what happens in the stack:



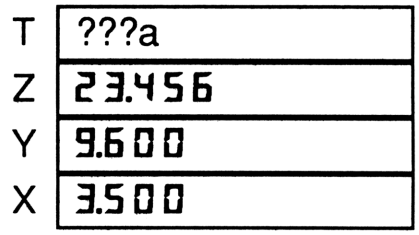
The **9.6** has overwritten the **23.46** in the X-register--because the stack was disabled. Get it? When the stack is enabled, a new number bumps everything up; but when the stack is disabled, the new number simply replaces what's in the X-register.

Now watch as the exercise continues:

To save the **9.6** farther up in the stack and to prepare for the **3.5**, press [ENTER]. This is the result:

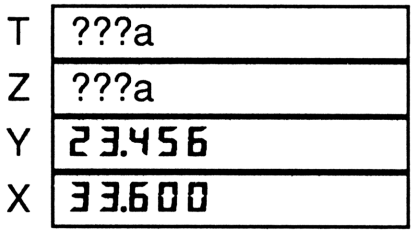


Next, key in **3.758**, but then correct your "mistake" with the [BSP] (backspace) key (i.e. press: [3] [.] [7] [5] [8] [BSP] [BSP] [BSP] [5]). Here's what you did:



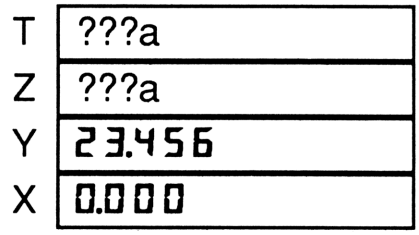
Now press [×], to multiply the **3.500** and the **9.600**.

Notice how the stack drops when the bottom two numbers combine (and almost all your arithmetic behaves this way--acting upon and between the X- and Y- registers); when this happens, the T-register is duplicated.



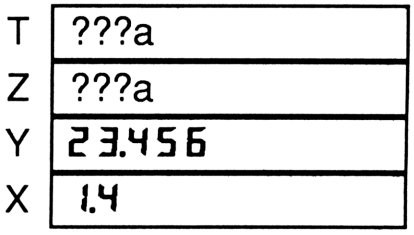
Now press the [BSP] once more. See what happens?

So when you're not in the process of keying in a number, the backspace key will clear the *entire* number out of the X-register--all at once (not just one digit at a time). Under this circumstance, this key behaves as a "Clear-X" key, right?



Next, key in your **1.4** ....

Whoa! Shouldn't the **0.000** in the X-register have been "bumped up" in a normal stack lift? Instead, that **1.4** just overwrote the **0.000**! What gives?



Well, what operation just happened previously? Whatever it was must have disabled the stack.

"Hmmm....Aha! When used as a CLear-X key ([CLX]), the [BSP] key disables the stack!"

Exactly: [ENTER] and [CLX] are the two major functions that *disable* the stack (i.e. make it "not ready to lift"). Most other functions--such as +, -, x, etc.--will leave the stack *enabled* ("ready to lift").

Now, press [+]--your final result:

T	???a
Z	???a
Y	???a
X	24.856

All these rules and manipulations may seem a bit much to remember if you're just now learning about them, but don't worry--with a little practice, these become as automatic as, say, shifting gears on a car.

Remember: Nobody comes out of the womb knowing these things, but they're not hard to learn and make into habits--so hang in there if you're feeling swamped.

All it takes is a little practice!

The R↓ Key

Here are some other good things to know about the stack (and more good ways to get some more practice in the process):

For example, how can you *look at* each of the four stack registers--without messing them up.

Fortunately, you don't need to use these little diagrams all the time; the HP-16C will show you the contents of its stack registers any time you want to see them.

Try This: Fill up your stack so it looks like this:

T	4.000
Z	3.000
Y	2.000
X	1.000

(Press 4 ENTER 3 ENTER 2 ENTER 1)

But now, press R↓ ("Roll down").

What happens?

The contents of the stack literally roll down one notch, and the stack winds up looking something like this:

T	1.000
Z	4.000
Y	3.000
X	2.000

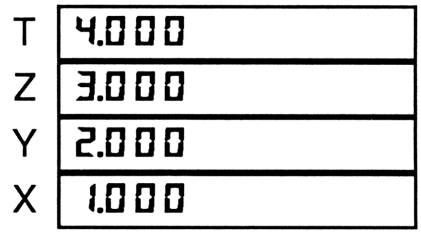
That's a handy key, R↓. You can use it to view the contents of each of the stack registers at any time. And of course, four consecutive R↓'s will return the stack to its original state.

Make sense?

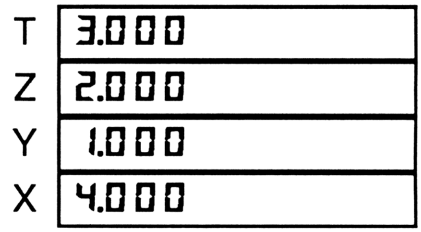
The R↑ Key

The contents of the stack can also be Rolled *up*. The usual symbol for "Roll-up" is R↑, and you can do this "Rolling up" by pressing the g R↑ keys.

Try It: If the stack starts out like this:



Then after you do a R↑, it's like this:

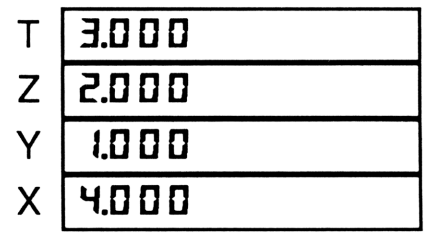


And again, just as with Roll Down, four successive R↑'s will bring you back where you started.

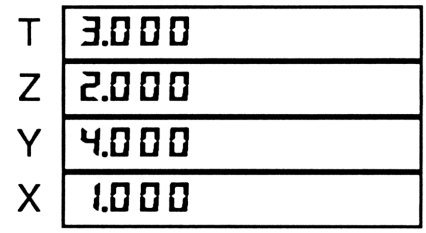
The x≷y Key

Another nice stack feature to know about is x≷y ("X exchange with Y").

For Example: Here's what the stack looks like now:



And after you press x≷y:



One-Number (X-Register) Operations

You've already seen how the stack does your basic two-number arithmetic. But notice that there are three mathematical operations that "crunch" only the contents of the X-register? These operations are: [√x], [1/x], and [CHS]. You know all about [CHS], but now, look at the other two.

Start with the stack looking like this:

T	3.000
Z	2.000
Y	1.000
X	4.000

Now Try This: Press [g] [√x] ("the square root of X"):

What happens to the stack? See? You just took the square-root of what was in the X-register. Simple enough, right?

T	3.000
Z	2.000
Y	1.000
X	2.000

And now that your stack looks like that,

Try This: Use the [1/x] ("reciprocal of X") function, by pressing [g] [√x]).

Here's what happens:

It's all just as you'd expect, right?

T	3.000
Z	2.000
Y	1.000
X	0.500

Well, that's about all you need to know about the stack to do most of your basic Floating-Point arithmetic.

Look at all the stuff you know now:

-- You know how to key in large, small, and negative numbers--using [CHS] and [EEX];

-- You know what "stack lift" means and how it may be either "enabled" or "disabled," depending upon the operation you've just completed.

-- You know that [ENTER] and [CLX] are the two operations that leave the stack *dis*abled; [ENTER] does this after performing a stack lift (copying the X-register into the Y-register); and [CLX] does this after clearing the X-register;

-- You know how [R↑], [R↓], and [x≷y] all manipulate the stack registers;

-- You know how the stack performs two-number and one-number arithmetic;

-- You know there's probably going to be a little quiz on this--just to make sure you have it all down cold.

As you go through this quiz, keep in mind that there are always more ways than one to solve any such arithmetic problems on your HP-16C. Also, set your display format to show you as many decimal places as you need in order to see all the significant digits.

Stack Quiz

1. What is 2 x 11.943?
   Calculate the answer twice, and the second time, don't use the ⊠ key.

2. Find 22.11 x [13.56 - (19.98 + 20.22)]

3. How much is 1024^4 ?

4. Load up your stack so that it looks like this:

T	6.000
Z	3.500
Y	4.700
X	2.200

   Now, without keying in any more numbers, find

$$\frac{(3.5 - 2.2)^2 + 4.7}{6}$$

Stack Answers

(and there are others which are just as valid)

T	???c	T	???b	T	???b
Z	???b	Z	???a	Z	???a
Y	???a	Y	2.000	Y	2.000
X	2	X	2.000	X	11.943

1. [2]↗ ➥ [ENTER]↗ ➥ [11][.][9][4][3]↗

T	???b	T	???b	T	???b
Z	???b	Z	???a	Z	???b
Y	???a	Y	11.943	Y	???a
X	23.886	X	11.943	X	23.886

[×]↗ (result) ***or*** [11][.][9][4][3] [ENTER]↗ ➥ [+]↗ (result)

T	???c	T	???b	T	???b
Z	???b	Z	???a	Z	???a
Y	???a	Y	19.9800	Y	19.9800
X	19.98	X	19.9800	X	20.22

2. [19][.][9][8]↗ ➥ [ENTER]↗ ➥ [20][.][22]↗

T	???b	T	???b	T	???b
Z	???b	Z	???a	Z	???a
Y	???a	Y	40.2000	Y	13.5600
X	40.2000	X	13.56	X	40.2000

[+]↗ ➥ [13][.][56]↗ ➥ [x≷y]↗

T	???b	T	???b	T	???b
Z	???b	Z	???a	Z	???b
Y	???a	Y	-26.6400	Y	???a
X	-26.6400	X	22.11	X	-589.0104

[−]↗ ➥ [22][.][11]↗ ➥ [×]↗ (result)

Stack Answers (cont.)

(and there are others which are just as valid)

T	???c	T	???b	T	???b
Z	???b	Z	???a	Z	???b
Y	???a	Y	1024.000000	Y	???a
X	1024	X	1024.000000	X	1,048,576.000

3. 1024 ↗ ➥ ENTER ↗ ➥ × ↗

T	???b	T	???b
Z	???a	Z	???b
Y	1,048,576.000	Y	???a
X	1,048,576.000	X	1.099511 12

ENTER ↗ ➥ × ↗ (result)

T	6.000	T	4.700	T	4.700
Z	3.500	Z	6.000	Z	4.700
Y	2.200	Y	3.500	Y	6.000
X	4.700	X	2.200	X	1.300

4. x≷y ↗ ➥ R↓ ↗ ➥ − ↗

T	4.700	T	4.700	T	4.700
Z	6.000	Z	4.700	Z	6.000
Y	1.300	Y	6.000	Y	1.690
X	1.300	X	1.690	X	4.700

ENTER ↗ ➥ × ↗ ➥ g R↑ ↗

T	4.700	T	4.700	T	4.700
Z	4.700	Z	4.700	Z	4.700
Y	6.000	Y	6.390	Y	4.700
X	6.390	X	6.000	X	1.065

+ ↗ ➥ x≷y ↗ ➥ ÷ ↗ (result)

Notes

USING THE DATA REGISTERS

Using the Data Registers

Now, the next question you're probably asking is: "OK, those are the stack registers. But what about all the other data registers in my HP-16C? After all, there are 29 of them, right?"

Right. And, as you know, these registers store numbers (one number per register). But [you're asking] how do you actually do this storing and retrieving?

(If you already know, then leap ahead to page 45).

Storing Numbers

The easiest way to learn this is just to do it a few times:

Try This: Store the number 199 in data register 2.

Solution: Key in 199 and press the keys STO 2.

Simple, right? The calculator has just copied the number 199 into the register labelled "2" (and so there are now two places in your calculator that contain the number 199--the X-register and register 2).

The STO key always *copies* the number in the X-register into the register you specify--never actually moving the original.

Storing does something else, also: It leaves the stack enabled, so that if your next operation brings a number to the X-register, the stack will lift, right? (Hereafter, you'll notice that all of the new operations you see will behave just like this.)

Try Some More: Try storing 100 into the A-register, 33 into the .0-register, and 45 into the I-register.

Solutions: Press:

[1][0][0] [STO] [A]

[3][3] [STO] [.][0]

[4][5] [STO] [f] [I] (or just [4][5] [STO] [I]--either way will work).

(Again, don't worry too much about the I-register right now. Just think of it as another data register you can store into and retrieve from--which it certainly is. You're going to discover its real usefulness a little later.)

See how simple the STOring procedure is?

If you can say it, you can do it.

Recalling Numbers

And now that you've stored (copied) numbers from the X-register into other data registers, just how do you propose to get them back?

Again, no harder done than said:

Press the [RCL] key and then the number of the register.

Try It: Recall those four numbers that you just stored in registers 2, A, .0, and I.

Solution: [RCL] [2] [RCL] [A] [RCL] [.] [0] [RCL] [f] [I] (or just [RCL] [I]).

And here's how your stack will look when you've finished (check it out with the [R↓] key):

T	199.000
Z	100.000
Y	33.000
X	45.000

Note that the [RCL] key enables the stack-lift. See how handy this is? It lets you recall several numbers in a row, and each time, the stack lifts up, preserving the previous numbers above.

And keep in mind: The [RCL] operation looks in the given data register and *copies* it into the X-register, still leaving the number in the data register--so you can recall it as often as you like, just in case you lose it or "crunch" it.

Unscheduled Retention Analysis

1. Which two keys will let you move numbers from one register to another?

2. Which of the stack registers does the calculator always use when it stores or recalls numbers?

3. Is storing and recalling more like "moving" or "copying" numbers?

4. How does the stack "enabling" process work with these two keys?

Piece of Cake, Right?

1. The two keys are [STO] and [RCL].

2. The STO and RCL functions always use the X-register.

3. The STO and RCL functions copy the numbers.

4. Both [STO] and [RCL] leave the stacked enabled--ready to lift when accepting the next entry into the X-register.

- -

OK, now what have you learned so far? You know that your HP-16C has 2 operating modes: Floating Point and Integer. But so far, you've been learning only about the Floating Point Mode:

You now know how to: visualize the registers, adjust the display, enter numbers, manipulate the stack, perform simple RPN arithmetic, store and recall numbers from other data registers, etc. (that's quite a lot, really).

And there's plenty more to learn about Floating Point Mode. For example, you can *program* your HP-16C to run routines in Floating Point Mode (and if you're curious about how to do this, check page 248).

But Floating Point isn't really why you bought the HP-16C "Computer Scientist," is it? You want help with your computer math (i.e., integer math), right?

Well, then, it's time for the main event....

INTEGER MODE: THE HP-16C AND ITS DISPLAY

The HP-16C in Integer Mode

So now you're ready for Integer Mode, eh? The *real* talent of this HP-16C thing-amabob.

OK, then what better way to do this than simply to take a "test drive?" After all, when you buy a new car it's pretty tough just to park it in the garage for a couple of weeks while you read the manual (do *you* know anyone like that)?

So go on--take a spin. Just follow along here and find the right keys to press.

Don't worry about where you're going right now. This is just an exercise in getting the feel of the HP-16C. The exercise will twist and turn and wind up going nowhere in particular--but that's all right. It's just a test drive.

Ready?

GO! ---->

What You Press	What Appears in the Display
BSP	0.000
DEC	0 d
1 6	16 d
f WSIZE	0 d
f SET COMPL UNSGN	0 d
HEX	0 h
B C D E	bCdE h
f SHOW DEC	48350 d
f SET COMPL 2'S	bCdE h
f SHOW DEC	-17186 d
OCT	136336 o
BIN	1101111 0.b
f WINDOW 1	10111100 b.
DEC	-17186 d
f SR	24175 d
f SL	-17186 d
CHS	17186 d
CHS	-17186 d
g ASR	-8593 d
2 f RRn	-2149 d
2 f RLn	-8593 d
3 g RRCn	-1075 d
3 g RLCn	-8593 d
f NOT	8592 d

What You Press	What You See in the Display
f SHOW HEX	2190 h
8 f MASKR	255 d
f SHOW BIN	11111111 b
f AND	144 d
f SHOW HEX	90 h
8 f MASKL	-256 d
f SHOW HEX	FF00 h
f OR	-112 d
f SHOW HEX	FF90 h
f SET COMPL 1'S	-111 d
2 CHS	-2 d
×	222 d
f SHOW HEX	dE h
g LJ	8 d
x≷y	-8703 d
f SHOW HEX	dE00 h
3 2 7 6 8	-32767 d
−	24064 d
f SHOW OCT	57000 o
g LSTX	-32767 d
+	-8703 d
1 5 f CB	24064 d
1 5 f SB	-8703 d

Here's the checkered flag.

Did you lose your way or crash along the way? Well, even so, you probably figured out at least a few things anyway:

Did you guess, for example, that in Integer Mode, the stack is the same arrangement of data registers as in Floating Point Mode? And the stack manipulation properties are the same, too-- you can roll the stack up or down and swap the X- and Y- registers, etc.

You can also perform arithmetic operations just as you do in Floating Point Mode, *but* there are no decimal points, fractions or exponential notation.

But with all the other, unfamiliar operations, this "test drive" may not have taught you much else--except what it feels like to have Button Pushing Syndrome (yep, that was it, all right).

Pushing buttons without knowing what you're doing isn't a very comfortable feeling, is it?

OK, then, it's time to slow down, look at a map, and find out about all these different operations....

The Display in Integer Mode

The real star of the HP-16C is the display register--as you're going to discover right now.

As you already know, this display acts as an interpreter between you and the number in the X-register. Of course, in Floating Point Mode, this interpretation job wasn't much more than rounding off decimal places.

But now comes the part where that display really earns its pay:

When you switch to Integer Mode by pressing the BIN, OCT, DEC, or HEX key, you are, in effect, telling the display register to *display* whatever is in the X-register as either a binary, octal, decimal, or hexadecimal integer number.

This display register is actually a very smart set of electronic circuits that take the integer in the X-register and convert it into what appears to be a number--which is what you see in your HP-16C.

So, what's really in the X-register?

Would you believe...

Binary Digits (Bits).

"Ah...OK, so how many binary digits are we talking about here, anyway?"

Well, actually, you can *choose* how many...but before getting into all that, notice a few other things first:

The display register has a **b** , **d** , **o** , or **h** at the right side of the display.

This is to remind you that it's translating numbers into binary, decimal, octal, or haxadecimal format for you (awfully considerate of it, don't you think?).

And you can switch from one display format to another--for just a moment or as long as you like:

Try This: Press [DEC] to see the decimal display format.

Now press [f] SHOW [OCT] to glimpse momentarily what this number looks like in octal format. Of course, you could also press the [OCT] key itself--to *keep* the display in octal format until you're ready to go back to decimal.

"Fine. But what *are* these different formats, anyway?" (you might ask).

As a matter of fact, if you're new to computer science, all these new words and phrases must indeed have left you scratching your head--notions such as "bits, word size, binary, octal, hexadecimal," etc.

So this is probably a good time to take an important detour and review what all these terms mean.

(If you already know everything mankind has ever discovered about number base conversion and word sizes, then you may speed ahead to page 65;)

(But if you're brand new--or even just a little rusty--stick around.)

(Anyway, it never hurts to review something, right?)

The Decimal Number System

Since you're going to be dealing with different number systems--and learning how to translate from one system to another--this is probably where to begin the story:

A long time ago--before there were hand-held computers--most people did their computing or reckoning on their hand-held fingers.

They might even have written down their answers--if they were among the few and fortunate who learned how to read and write.

Well, the ancient Romans developed one such written numbering system--one that some people still use, in fact (e.g. movie producers still show the date of a film using Roman numerals).

But just offhand, can you figure out

MDCLXVII x CDXLIV ?

As you can see, the Roman system isn't very convenient for doing computations.

Then, about 800 years ago, along came the Arabic numbering system--the one widely used today.

Back then, most people had ten fingers (i.e., ten digits)., so this system used ten symbols (0 - 9).

Well, western Europe eventually adopted this system, but Latin was the predominant written language there. And so, because the word "ten" is "decem" in Latin, this numbering system became known as the decimal system.

And what makes the decimal system so much easier to use? Think about it:

When counting in the decimal system, you start with the digit having the lowest value--zero. Counting upward, you reach a limit at 9--where you run out of symbols for each successive number; there's no single symbol for "ten."

Instead, you use a *combination* of two of the ten existing symbols, thus signifying that you've completely exhausted that set of symbols. You have to mark down a 1 and then a 0 beside it, meaning, "I've run through my set of symbols 1 time, plus I've counted 0 positions farther than that."

This forms the composite symbol for the number 10.

Similarly, when you exhaust your symbol set a second time, you mark down a 2 and a 0 ("2 complete sets, plus 0 extra counts"), and so on.

Now, what happens when you reach 99--and you want to count farther? Just keep the same pattern: "10 complete *sets* of my symbols, plus 0 extra counts."

So you write it: 1 0 0

And notice that you can think of this as a 10 next to a 0 ('ten sets of ten, plus zero extra counts"), or as a 1 next to a 0 next to a 0.

This 1 is the running tally of a new group (called a "hundred"). Then the 0's represent the extra tens and extra counts (called "ones," naturally).

So each successive decimal place (going right to left) represents a kind of running count--keeping track of the number of times that particularly-sized grouping has been completed. This is called a *positional* number system, because a digit in the second (10's) position represents ten times the *value* of the same digit in the 1's position--and so on, for each successively greater position.

Take a look at a particular number as it's written in this *positional, decimal* number system:

547

This number means that you've gone through 5 complete sets of "ten tens," plus 4 extra sets of ten, plus 7 extra counts ("ones").

So that 5 really stands for 500; the 4 really stands for 40; only the 7 is really representing just itself, and thus it "weighs" the least; it's the Least Significant Digit (LSD), because it's in the ones position.

On the other end, the 5 in the hundreds position really "weighs" 500; it's the Most Significant Digit (MSD) here. So by writing 547, you're saying:

"(5x100) + (4x10) + (7x1)"

See? Each digit has 10 times more "weight" than its right-hand neighbor, and thus this decimal system is also called the "base 10" number system.

Now, there's probably nothing really new to you about all of this. It's probably the way you learned to count when you were little. But here's the key:

It all comes from the fact that you use only ten different symbols (0 - 9).

Well, what would happen if you had, say, 41 different symbols available?

Or 603?

Or just 8...?

The Octal Number System

The octal system has only 8 digits (symbols) to use, instead of 10 (sure--you do know of two other numerals, but they're not allowed now).

Thus, when counting in octal (also called "base 8," since it's based on the use of 8 symbols), you would start with 0 and work your way up to 7. When you reach this limit, you carry a one into the next position and continue counting with 10.

But this 10 is *not* "ten." "Ten" is a word from the decimal number system. The number 10 in base 8 is written as 10(o) and is read as "one-zero, octal."

You can't say "ten," because when you *see* 10, this really means you've only counted to "eight"--not "ten!" Each successive digit in an octal number is "weighted" (speaking in decimal now) like this:

. . . 512 64 8 1

And see? Your first grouping on the right may still be called "ones," but your next group has to be called "eights," because that's when you run out of symbols and must start over. Then the next group is "eight eights" ("sixty-fours"?), and so on.

So a number such as

314(o)

would mean there are 3 sets of 64, 1 set of 8, and 4 sets of 1. So it's easy to convert an octal number to a decimal number--just do that arithmetic. Here are the keystrokes you would use (in Floating Point mode):

3 ENTER 6 4 ×	192.00
1 ENTER 8 × +	200.00
4 +	204.00

So 314(o) = 204(d). OK, fine. But now how do you go the other way--from decimal to octal [i.e. convert 204(d) back to 314(o)]?

You repeatedly divide the decimal number by 8 until the quotient becomes 0:

204 / 8 = **25**	with 4 remainder
25 / 8 = **3**	with 1 remainder
3 / 8 = **0**	with 3 remainder

You'll see the octal number by reading the *remainders* from bottom to top: 314

But of course, the whole idea here is that, with the HP-16C in hand, you don't have to do this conversion process so laboriously. Just let your super-smart display register work for you:

Try It: Convert 314(o) to decimal--and then back to octal again.

Solution: Begin by pressing the [OCT] key, to get into octal Integer Mode.

Now key in the number, [3][1][4] and press the [DEC] key.

You'll see: 204 d

Finally, press [OCT] to return to your beginning format.

Ain't this grand? No more tedious conversions between bases!

(But remember: Always know how your HP-16C does these chores--in case you're caught without its help some fateful day.)

The Hexadecimal Number System

Besides base 8, base 16 is another important number system in computer science and engineering.

In base 16, of course, there must be 16 symbols available. But we have only 10 standard "numerals," so we "invent" six more: A, B, C, D, E, F.

Thus, when counting in hexadecimal ("hex" for short), you start with 0 and count up to 9--and then continue with A-F. When you reach that limit, you have to carry a 1 into the next position and start over with 10.

But again, this is "one zero"--not ten--because you've actually counted up to "sixteen" already. And here are the relative (decimal-notation) "weights" of the successive digits in the hexadecimal number system:

. . . 4096 256 16 1

Of course, you convert between decimal and hexadecimal similarly to your decimal-octal conversions--either the hard way or with your HP-16C!

The Binary Number System

Of course, the binary number system (base 2) is the most critical number system for any computer. After all, it's the *only* number system a computer can understand.

And of course, since it's base 2, there are only two digits in the binary numbering system: 0 and 1. You'll often hear Binary digITS called by their shortened name: BITS. Here are the first four positional values ("bit" values) in the binary system:

. . . 8 4 2 1

(....in base 10 equivalent notation, of course).

Thus, the largest binary number that you could write, using only these first four positions would be

1111(b)

which is the same as 15(d).

And there is an important idea: The number of binary digits (bits) you allow when writing a binary integer is called the Word Size. Thus, in this example above, the Word Size is 4.

Binary-to-decimal conversion works similarly to conversions from octal or hexadecimal--but it's a bit simpler (no pun intended)--because you only need multiply each positional value by 1 or 0, right?

For example, 1011(b) would be (1 x 8) + (0 x 4) + (1 x 2) + (1 x 1) = 11(d)

And--again, as with octal and hexadecimal, you can go the other way (do decimal-to-binary conversion) by repeated division (but this time by 2, of course).

Thus, to convert 100(d) to binary...

100 / 2 = **50**	with 0 remainder
50 / 2 = **25**	with 0 remainder
25 / 2 = **12**	with 1 remainder
12 / 2 = **6**	with 0 remainder
6 / 2 = **3**	with 0 remainder
3 / 2 = **1**	with 1 remainder
1 / 2 = **0**	with 1 remainder

Result (reading the remainders in reverse order): 110 0100(b)

So it would take a Word Size of 7 bits to represent 100(d) as 110 0100(b).

You'll probably find it easier to convert between decimal and binary than between decimal and octal or hex. After all, it's easier to divide by 2 than by 8 or 16. Before the advent of devices such as the HP-16C, if you wanted to convert a number from decimal to hex or octal, you'd most likely do it in a two-step process: decimal to binary and then binary to octal or hex.

As an example of this two-step process, convert 42876(d) to octal and hex.

First, convert to binary with repeated division:

	Remainder
42876 / 2 = **21438**	0
21438 / 2 = **10719**	0
10719 / 2 = **5359**	1
5359 / 2 = **2679**	1
2679 / 2 = **1339**	1
1339 / 2 = **669**	1
669 / 2 = **334**	1
334 / 2 = **167**	0
167 / 2 = **83**	1
83 / 2 = **41**	1
41 / 2 = **20**	1
20 / 2 = **10**	0
10 / 2 = **5**	0
5 / 2 = **2**	1
2 / 2 = **1**	0
1 / 2 = **0**	1

Result (splicing together all the remainders): 1010 0111 0111 1100(b).

Now, if you break the number up into groups of 3 digits--beginning on the right side--you get this:

001 010 011 101 111 100

Notice that you add two zeros to the group on the far left.

Now you can write

001	010	011	101	111	100
1	2	3	5	7	4

assigning in this way the equivalent octal digits (which, hopefully, you have memorized just for such emergencies). So 42876(d) = 123574(o).

But even better, if you break the binary number into groups of *four* digits each, then you can match each group with its corresponding *hex* digit:

1010	0111	0111	1100
A	7	7	C

So 42876(d) = 123574(o) = A77C(h). You see? Once you've converted a number into binary, it's fairly easy to go to either hex or octal.

It's also easy to go back to binary from hexadecimal or octal--and then on to decimal format. For example, suppose you wanted to convert 18AD(h) to decimal. To do this, you first write the hexadecimal number itself, followed beneath by the binary representation of each digit (which you should also memorize):

1	8	A	D
0001	1000	1010	1101

Then separate the bits and write their positional weights underneath them:

0	0	0	1	1	0	0	0	1	0	1	0	1	1	0	0
0	+0	+0	+4096	+2048	+0	+0	+0	+128	+0	+32	+0	+8	+4	+0	+1

Then if you add this resulting line of values, you get 6317.

So 18AD(h) = 6317(d).

Now, for practice, make up your own decimal and hex numbers and try several paper-and-pencil conversions--in both directions. Then check your results on your HP-16C--to see if you have the hang of it....

...See why computer scientists prefer hex numbers to any other?

Although binary digits (bits) are really what you're dealing with, it's far easier to use hex numbers to represent them--because hex numbers take fewer digits to write: one hex digit corresponds to 4 bits.

Of course, after having done a few of these number base conversions with pencil and paper, you'll certainly appreciate the speed and capability of your HP-16C.

You might even be tempted to forget how to do such conversions by hand.

Don't!

Don't ***ever*** forget the fundamentals of your science! People who can perform number base conversions *without* a calculator tend to do much better work in computer science and engineering.

Now that you've refreshed yourself on a bit of number theory (or the number theory of bits) are you ready for that closer look at how the HP-16C works in Integer Mode?

Good, because it's time now for a detailed explanation of your keystrokes--and how the HP-16C display interprets them.

(If you already know how to key in and interpret negative integers--in any of the four number bases--*and* how to change the word size of the calculator, then feel free to skip to page 83.)

Hewlett-Packard designed this "Computer Scientist" calculator so that casual users wouldn't even be aware of the complexity and speed of the calculations going on inside it. And for the most part, you really and truly don't need to think about it.

But just so you appreciate the wonder of things a little bit, here's what's actually happening:

Smoke and Mirrors: The Display's Bag of Tricks

Yes, fans, that's right--it's all window dressing. As you heard a little while ago, the display is the real star of the show. In fact, while in Integer Mode, most of what goes on in the HP-16C happens right there--in the display register.

Remember what that display register is all about? It's not merely another memory location in the calculator. Rather, it's a tiny translating engine--a rather complex set of electronic circuits.

And recall back to when you were using the calculator in Floating Point Mode.

Remember how you could adjust the display register to show more or fewer digits after the decimal point? Well, the whole point to all that was this:

Although a number may appear in a rather limited form in the display, the number itself is retained in its full precision in the X-register.

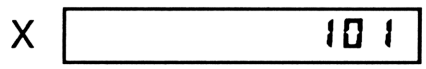
And that precision was a 10-digit real number when you were working in Floating Point mode.

Well, here in Integer Mode, the rule is much the same:

The display register will show you integer numbers in several different digital formats. But all the while, the *integer actually residing* in the X-register (and in each of the other registers also) *is a binary number,* (just 1's and 0's) because that--and only that--is the language a computer speaks.

Demonstrate this to yourself: Put your HP-16C into integer mode--decimal mode (of course, to do this, you would press the [DEC] key). Now key in some digit, say, a five: [5]

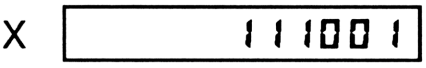
What *actually* happens? The calculator figures that since you're keying this in while in decimal mode, you want to place a binary number in the X-register whose decimal value is 5. So it places a *binary* five into the X-register:



And since you're in DEC format right now, in the blink of an eye, the display register converts this binary string and puts the corresponding decimal number (5) *in the display*.

Of course, once the calculator has performed this operation, it will let you key in another digit. So suppose you do key in another digit (try [7]).

This time, the calculator realizes that you've now keyed in the decimal number 57. So again, the *binary* version of this number actually goes into the X-register:



And again, since you're in decimal format, the display shows you the decimal value, 57, what you intended to key in.

So you can add more digits, just as you please (and back them out by using [BSP] -- just as in Floating Point Mode), until one of two things happens:

You finish keying in this number (by pressing [ENTER] or [+] or some other operation);

OR you run out of room in the X-register... ("say what?")....

Defining the Word Size

Just what does it mean to "run out of room" in the X-register (or any other data register, for that matter) when you're working in Integer Mode? Does it mean you're allowed only 10 digits (ten 1's and 0's)--similar to the limit in Float Mode?

Not at all. With the HP-16C you can specify that you want to work with integer numbers that have anywhere from 1 to 64 bits(!)

As you know, this limit is called the Word Size, and when you set this limit, it becomes the limit not only for the X-register but for all data registers.

Try This: Key 16 into the X-register and then press f WSIZE.

You have just set the Word Size to be 16 bits (remember what a bit is?). This means that each data register in the HP-16C will now hold a maximum of 16 binary digits (1's and 0's).

Of course, you can reset this limit anytime you wish, if it becomes too restrictive for you.

But make no mistake: You *will* be restricted to the Word Size you have set: The HP-16C will actually *prevent* you from entering numbers that call for more bits than your specified Word Size.

For example,

Try This: Set your Word Size to 4: 4 f WSIZE and go to binary notation by pressing BIN.

Now try to key in the binary number 1 1 1 1 1.

You can't do it, can you? The greatest number of bits you will be allowed to enter is the Word Size, which is 4 right now.

What's more, if you try to enter a number *in any notation* which calls for more bits than the Word Size, the HP-16C simply won't respond to your keystrokes. For example, watch what happens when you

---

Try This: First, go to decimal mode (press DEC).

Now try to key in any number *more than 15.*

No way, José. The keys just won't respond (or they'll produce some negative number, in which case you should press f SET COMPL UNSGN to get positive numbers only--more about that in a minute).

So what's so magic about the number 15?

Well, it's the largest integer you can represent with 4 bits (it's represented by 1 1 1 1), and since 4 bits is all you get right now, with your Word Size set at 4, the calculator won't let you do any more.

Pretty smart machine, eh?

By the way, speaking of smarts, there's something even more basic (and convenient) your HP-16C is doing for you, too--something you may have already noticed:

No matter what the Word Size is, your machine won't ever let you use digits that are meaningless to the format you've chosen.

When you're in DECimal mode, you can get the digit keys [0]-[9] to work, all right.

But when you're in BINary mode, only the [1] and the [0] will work for you.

Likewise, in OCTal, you get [0]-[7]; in HEXadecimal, you get [0]-[9] and [A]-[F].*

* The HP-16C uses the lower case letters b and d in place of B and D since you might otherwise confuse B with 8 and D with 0. Ah, those engineers at HP...they think of everything, don't they?

"Hmmm...," (you say), "what if I change the Word Size *after* I have already stored some numbers in some data registers? Will I effectively change the values of those numbers?"

Durn right, y'will.

Try This: Set your Word Size to 8: [DEC] [8] [f] [WSIZE]

Now go into BINary mode (press [BIN]), and key in the largest number you can:

```
11111111 b
```

Press [f] SET COMPL [UNSGN] [DEC] to get a look at this number's decimal value:

```
256 d
```

Now reduce the Word Size to 4: [4] [f] [WSIZE]

Your number changed!

What happened? To find out, switch back to BINary (press [BIN], or, for a temporary look, press [f] SHOW [BIN]).

By reducing the allowed Word Size, you just "lopped off" the lefthandmost 4 bits of every integer you had stored in the machine. Now those bits aren't part of the integer (so set the Word Size back to 8: [8] [f] [WSIZE])

Beware! Such an error can be disastrous if you alter the Word Size during calculations, especially since those lefthand bits are the most significant in value.

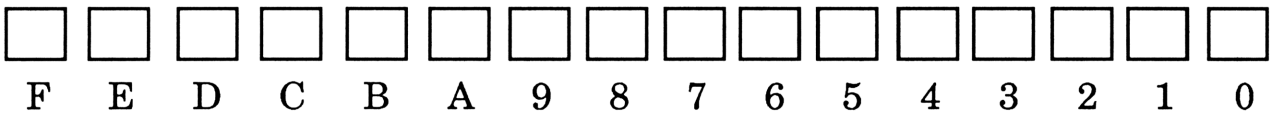
In fact (as you'll recall from page 56), *the most* significant bit (MSB) of any binary word is the first digit on the left, true?

Or, to put it another way, the MSB for a Word Size of 8 would be the 8th bit, counting from the right to the left (i.e. going from least to most significant).

But this does become rather tedious to say as you work with binary words so much: "the 8th bit, counting from right to left."

So here's a convention that computer scientists have adopted for referring to the first 16 bits of any word:

(Pretend this ☐ is a bit)



Thus, for example, you can refer to "the 11th bit from the right" as "bit A" and the "7th bit from the right" as "bit 6." It saves a little breath.

Signed Numbers

So far, you've been using your HP-16C in Integer Mode and with the UNSigned number display format.

Simply put, this means that you haven't had to mess around with negative numbers yet. But of course, there has to be a way to do this--because it's very common for a computer to need to work with positive *and* negative integers.

Well then, consider that for a minute: How do you use bits--just 0's and 1's--to represent a negative number?

You now know all about how the bits represent the value of the number itself: Each bit represents a certain power of two, and the sum of the powers of the "1 bits" gives you the number.

So what do you use for a minus sign?

If, for instance, you wanted to express 15(d) as a binary, you would need every bit in a 4-bit number to be a 1:

1111 b

There *is no bit* left to act as the minus sign.

Because this kind of binary number doesn't allow you to express negative numbers, it's called an "unsigned number." And as you know, you've already been using this kind of number (recall earlier--page 69--when you might have had to press f SET COMPL UNSGN to get rid of any negative numbers you were seeing).

This is what you were assuming--that *all* bits of each binary word be used to represent the value of the number itself--without allowing for a minus sign. You said, essentially: "I want only unsigned numbers."

OK, that's all good and fine for certain occasions. But you still haven't answered the question: What about when you *do* want negative numbers?...

Well, you get what you pay for: If you want one bit to represent the sign of your integer, you have to give up one of the bits you would have otherwise used as part of the number's value.

This "sign bit" is the *leftmost bit* in the number--i.e. it's the Most Significant Bit in any given word.

Take an example: It's quite common to work with a 16-bit Word Size, so suppose you're doing just that, and you find that you need a sign bit.

So you dutifully sacrifice the 16th bit (bit F), the MSB, to become a sign bit, thus limiting yourself to only 15 out of the 16 bits to form the number's value itself.

Now, how does this constrain you? Well, without the need for a sign bit, you were able to represent a positive integer as large as 65,535--which is the decimal equivalent to 1111 1111 1111 1111 (b).

But with only 15 bits to use, you can go up to only 32,767:

111 1111 1111 1111 b

That's the bad news. The good news is: now you can also go *down* (in a negative direction) as far as -32,768.

"All right, fine. but how does this work? How does that 16th bit act as a negative sign?"

(OK, you asked...)

Well, it *seems* simple enough: If the Most Significant Bit (i.e. the sign bit) is a 1, then the number is considered to be negative.

So you *might* think that all you need to do is change the MSB to a 1 and the rest of the number would become the same decimal number with a negative sign in front of it. (Nooo.....sorry.)

The simplest way to understand how the HP-16C --and how most computers-- interpret negative numbers is to imagine the tape counter on a cassette recorder. Of course, most of these tape counters have only 3 digits to work with, so for the sake of this explanation, imagine that you're working with a Word Size of 3. The argument works similarly for any Word Size, though:

Suppose you reset the tape counter to 000 and then press the rewind key so that the counter moves backwards by 1 unit. You'd see the number 999--and you'd know this is really a -1, right?

Naturally, to get the same number on paper, you would have had to perform (1000 - 1), to get 999. And in that subtraction process, you actually "borrow" a 1 from an imaginary 4th position, don't you?

Now, imagine that you have this three-place (a three-wheel) counter, but that instead of 10 symbols, each wheel has only two: 0 and 1.

Fine. So you reset this unique counter to 000 and then move the tape back by 1 unit. What will appear? It would show 111, wouldn't it? Each wheel shows the last symbol in the set, just as the 999 did in your 10-symbol set.

So at least in the world of binary tape counters, 111 is equivalent to -1.

And again, on paper, this 111 would have had to come about just as the 999 did with your 10-symbol counter: *You would have had to borrow a 1 from an imaginary fourth counter wheel.*

Thus you can see a pattern developing. Notice how you can use your binary counter idea to represent either an unsigned integer or a signed integer--and how that leftmost bit (the MSB) changes its role in each case:

Binary Representation	Unsigned Value	Signed Value
111	7	-1
110	6	-2
101	5	-3
100	4	-4
011	3	3
010	2	2
001	1	1
000	0	0

Now, why have computer scientists made this so complicated? Why not just use the MSB as a negative sign and call it good?

The reason for this is that by using the above "tape-counter" logic, a computer can use the same process for subtraction as for addition, thus allowing for speed and simplicity in computer design.

To see how this works, take a simple example:

$$\begin{array}{r} 2 \\ -\ 1 \\ \hline \end{array}$$

Here's the key to the whole thing: Subtracting one from two is the same as *adding* -1 to (+)2. So by using your 3-bit "tape counter" logic, you would write this *addition* problem this way:

$$\begin{array}{rcl} 010\ (b) & & 2\ d \\ +\ 111\ (b) & = & +\ -1\ d \\ \hline \end{array}$$

Now, if you performed this addition on the left side, you'd get 1001, since (as in all arithmetic) you have to carry an extra 1 to the next column on the left whenever the total of any column exceeds 10 (remember: 1 + 1 = 10 in base 2--and this result is *not* pronounced "ten," right?).

But...hmmm...your addition has forced you to carry a 1 into a *fourth* bit. Problem is, there's no fourth bit in a 3-bit counter. So that carried 1 is just dropped and thus through this "planned error" you wind up with 001, i.e., 1 d --the correct answer!

So how do you tell your HP-16C how to do all this? How do you tell it that you *do* want to work with negative numbers?

1's or 2's?

There are actually *two* different ways that your calculator can represent and work with subtraction and negative numbers.

The first thing you should realize about these representations is that they are *interpretations of the values* of internal binary integers--representations for your eyes and for the arithmetic logic of the HP-16C.

And because they're for your eyes, these display formats may fool you into thinking that you're not doing your arithmetic correctly. But this isn't necessarily the case (although it's wise to assume this is at least possible).

The whole thing boils down to this: Your HP-16C calculator is a tool designed to help you design other computing tools. So part of the time, you want it to "think" like a computer--so it can simulate the computer program you might be designing. But the other part of the time, you'd like it to "think" about numbers the way you do--"common sense."

Well, the display is caught in the middle of all this. It doesn't really know when you want what. So if it can't read your mind, the next best thing is to try to be entirely consistent--and that's why you really need to have a good grasp of different binary formats (1's and 2's complements).

2's Complement Format

Try This: Set your HP-16C for a Word Size of 16 bits, and then put it into the 2's Complement format.

Then key in F F F F h --which is just a quick way to key in sixteen ones, since F h is the same as 1 1 1 1 b , right?

Solution: Press [DEC] (if you're not already in Decimal Integer Mode).

Then press: [1][6] [f] [WSIZE] and [f] SET COMPL [2'S] and [HEX] [F][F][F][F].

So, in the X-register there are now 16 binary digits: all of them having a value of 1--even the Most Significant Bit (MSB). And since you're *not* in UNSigned mode, this MSB = 1 means that this integer must be negative, right?

Check it out; press the [DEC] key to go into decimal format:

- 1 d

What gives? This number is negative, all right, but it's sure a lot smaller than that F F F F you keyed in...isn't it? Or is it?

Well, it's true that the X-register contains 16 ones. And since your Word Size is 16, your imaginary tape counter has 16 wheels, each of which has just 0 and 1 on it. So if you had set this big counter to all zeros and then had rewound the tape by 1 unit, wouldn't you now be seeing 16 ones?

Aha! So -1 (d) (in 2's complement format) *is indeed the same as*

1111 1111 1111 1111 b

Now Try This: Change the sign on this -1.

Solution: Press the [CHS] key and you'll see: 1 d

This shouldn't surprise you much, if you're used to the idea that changing the sign of -1 will get you a +1.

But a lot goes on here inside the calculator. It has changed into 0's *all 16* of those 1's in the X-register. And then it has mathematically added 1 to the 16 0's.

This process of changing ones to zeros--and vice versa--then adding 1, *is the mathematical definition of taking the 2's complement of a number.*

1's Complement Format

So you've seen one way that the HP-16C shows you negative numbers. Here's the second way: 1's complement format.

Try This: Set your calculator to this format now by pressing f SET COMPL 1'S. And while you're at it, press the HEX key.

Now, as before, key in F F F F. Then press the DEC key to see what this is in 1's complement format.

Lo! A rare sighting of the elusive Negative Zero (in its native habitat, no less).

Of course, if you press the CHS key now, you will certainly change the sign of negative zero to positive zero.

But what's happening this time when you do that? Under the 2's Complement format, a lot was going on backstage--inverting bits and adding 1.

Well, here in 1's Complement format, the calculator also inverts all the bits in the X-register, but it doesn't add 1 after the inversion process.

This is called taking the 1's complement of a number.

Here's a summary of the relationship between unsigned, 2's complement and 1's complement formats for a three-bit word (and of course, this generalizes for larger Word Sizes):

Binary Representation	Unsigned Value	2's Complement Value	1's Complement Value
000	0	0	0
001	1	1	1
010	2	2	2
011	3	3	3
100	4	-4	-3
101	5	-3	-2
110	6	-2	-1
111	7	-1	-0

Stop for a moment now--to see if you have a rough idea of what this juggling of numbers is all about. Yep, that's right--here comes another...

...wait for it...

Spontaneous Comprehension Examination

1. How many binary digits are available in the registers of the HP-16C?

2. If you change the Word Size to 15, how many HEX digits could you key in? What is the largest hex number that you could key into the X-register under those circumstances?

3. What are the two different formats that you may use to view negative integers on the HP-16C?

4. If you've initially set the Word Size to 15 and have keyed in the maximum number, and then you reduce the word size to 8 bits, what is the decimal equivalent of the number in the X-register (assuming that you're using 2's complement format)?

Answers to S.C.E.

1. This depends on your Word Size setting, which you can set for anywhere from 1 to 64 bits (and by the way, when you first enter Integer Mode from Floating Point Mode, the Word Size is 56--a wee tidbit to know and tell).

2. You can key in 4 hex digits, but the first digit must be 7 or less. Therefore the largest hex number you can key into a 15-bit word is 7FFF h.

3. The two formats are 1's and 2's Complements.

4. If you've keyed in 7FFF h with the Word Size set at 15, and then you reduce the Word Size to 8, you'll be left with only the last two hex digits: FF. Of course, this is -1 d in 2's complement format.

- -

Well, how did you do?

If you missed more than one of the answers, you might want to go back now and take another run at it. Start on page 65--and don't worry: Doing arithmetic like a computer is not all that trivial and intuitive (that's why you let the computer do it, right?).

So don't be discouraged if this takes a little while, OK?

Doing Windows

Well, isn't this great? You can now work on positive *and* negative integers with up to 64 bits each (yup--it just don't get any better'n this).

Fine. But how do you see bits 9-64 (or octal digits 9-22, or hex digits 9-16, or decimal digits 9-20)? It looks as if the HP-16C display shows you only the first 8 digits of any integer.

Not to worry: The HP-16C will let you view even the 64th bit if you want to.

But you may not even know in advance whether the results of your arithmetic will have more than 8 digits. How can you tell--by looking at the display--whether or not this is the case?

Simple: Your calculator will tell you if there are more than eight digits in your number by placing a dot to the left of the **h**, **d**, **o** or **b** symbol in the display.

And once you see this cue, you have two different choices as to how to view those other digits:

First of all, you can "scroll" the number, rolling it by your display, one place at a time.

Try This: Press the [HEX] key and key in [F][0][F][0]. Now press the [BIN] key to see

1111 0000 .b

The dot to the left of the **b** tells you that there are more digits to the left of those you see now. So scroll on out there and take a look at them: Press the [g][➤] keys, and you'll see

0111 1000 .b.

Now there are dots on *both* the left and right of the **b** to indicate there are more digits to the right *and* to the left of those you see.

If you scroll right seven more times ([g][➤]), the dot on the left of the **b** will vanish.

This tells you there are no more digits out of sight to the left; but the dot on the right of the **b** remains, indicating there are still such digits to the right (no prizes for guessing how to scroll back to the left).

OK, that's the one-digit-at-a-time method for viewing the extra digits in a long integer.

In the second method, the display "does windows."

That is, it shifts between adjacent sets (called "windows") of eight digits in your long integer.

Try This: First, set the word size to 64, by pressing [6][4][f][WSIZE].

(Or you could press [0][f][WSIZE]--a handy shortcut sometimes--since how would you key in 64 if you had previously set the word size to 3? Keyboard lockout would prevent you from keying in anything greater than 7. So the engineers who designed the HP-16C even thought of that eventuality: You just key in [0] to get the maximum word size.)

Now press [HEX] and enter this number:

[8][0][C][0][E][0][F][0][F][8][F][C][F][E][F][F]

When you're finished, you'll see: F8FCFEFF h

As you'd expect, these are the rightmost 8 digits; in other words, you're looking at window #0.

To get to the next window (#1), just press [f][WINDOW][1], and you'll see:

80C0E0F0 h.

As you can see, windowing--just like scrolling--will work when the HP-16C is in any of the four integer number-base formats. But you're most likely to use it with the BINary format (because you use the most digits, of course).

So practice in Binary:

You Press	You See	
BIN	1 1 1 1 1 1 1 1 .b	(window #0)
f WINDOW 1	1 1 1 1 1 1 1 0 .b.	(window #1)
f WINDOW 2	1 1 1 1 1 1 0 0 .b.	(window #2)
f WINDOW 3	1 1 1 1 1 0 0 0 .b.	(window #3)
f WINDOW 4	1 1 1 1 0 0 0 0 .b.	(window #4)
f WINDOW 5	1 1 1 0 0 0 0 0 .b.	(window #5)
f WINDOW 6	1 1 0 0 0 0 0 0 .b.	(window #6)
f WINDOW 7	1 0 0 0 0 0 0 0 b.	(window #7)

Notice that the HP-16C has 8 possible windows (numbered 0-7), which makes sense, since the maximum number of digits is 64.

And notice that when you shift it from one format to another, it automatically resets its window to #0 (far right)!

Flags and Machine Status

Well, you've been learning a lot about the display's everyday behavior. It's time to talk about those special occasions when HP-16C's display their flags.

"Flag" is another word for a switch (or a bit, if you like--something that is either on or off--and can be set one way or the other).

In the HP-16C, there are six such flags, and it turns out that three of them are for "special occasions." These are flags 3, 4, and 5 (the others are for everyday use, and you'll see those later).

You can "set" ("switch on" or "set to 1") any of those flags with the keystroke sequence [g] [SF] n, where n is one of the keys [0], [1], [2], [3], [4], or [5].

Similarly, you can "clear" ("switch off" or "set to 0") any of the flags simply by pressing [g] [CF] n.

Also, you can test a given flag's status ("set or clear," "on or off," "1 or 0") by pressing [g] [F?] n. You'll get a 1 or a 0 as an answer to your query.

So what do these "special occasion" flags mean?

Flag 3: Show the Leading Zeroes

Whenever flag 3 is set, for example, the display will show all the leading zeros in an integer. But if you clear flag 3, these leading zeros will be suppressed.

Try This: Press f SET COMPL 2'S and then DEC 16 f WSIZE to set the Complement, the Word Size and the format.

Next, set flag 3: g SF 3.

Key in 20 and see:	20 d
Press HEX and see:	00 14 h
Press OCT and see:	000024 o
Press BIN and see:	000 10 100 .b
And press f WINDOW 1 to see:	00000000 b.

The display shows all leading zeroes in these numbers (even those in other windows), except in decimal mode.

Now, if you clear flag 3 (press g CF 3), these leading zeros will wink out. But the **b.** will still be there, since there are digits in the "zeroth" window off to the right.

So press f WINDOW 0 to see them: 10 100 b

Notice that the **b** no longer has a dot in front of it; there are no bits to look at beyond those in the zeroth window when the leading zeros have been suppressed.

OK? That's flag 3, the Show-the-Leading-Zeroes Flag.

Flag 4: Carry/Borrow

Flag 4 is the CARRY/BORROW bit flag. The HP-16C will set and clear this flag as it performs certain arithmetic operations. You'll see this in more detail later. For now, just note that whenever flag 4 is set, the C ("Carry") annunciator will appear in the display.

Flag 5: Out of Range

Flag 5 is the Overflow or "Out-of-Range" flag, which the HP-16C will set whenever a calculation result extends beyond the range of the current Word Size.

Again, you'll see more of this flag when doing arithmetic operations in Integer mode. For now, just remember that, whenever the G annunciator in the display is on, then flag 5 is set and something is out of range.

- -

Well, those are a few more things to add to your growing storehouse of knowledge on this HP-16C:

You now know how your calculator does windows by showing you successive chunks of large integers.

You now know what a flag is--and you know that three of them (3, 4, and 5) are reserved by the machine to signify certain things about the status of your integers numbers and calculations.

But did you know that the HP-16C can actually give you a summary of its current formats and flag settings?

The Status of the Machine

It's a fact: To get the HP-16C to show this information, just press the f STATUS keys and hold down the STATUS key for as long as you want to view this.

Try It: Press f STATUS immediately after you press g SF 3, and see:

2- 16- 1000

Here, the 2 indicates that the display register is in 2's Complement format, (and this would instead be 1 for 1's Complement and 0 for UNSigned format.)

The 16 stands for the current word size. This value always appears as a base 10 number.

The 1000 stands for the status of flags 3, 2, 1, 0 respectively; that is, it's a 4-bit "word" that tells the states of the four flags 0-3 (numbered from the right, of course--as usual).

In this case, for instance, 1000 says that flag 3 is set (1) and that flags 2, 1, 0 are all clear (0), respectively.

Unforeseen Regurgitative Incident

1. How many windows can the HP-16C show for a Word Size of 32?

2. What are the keystrokes to use to scroll right?

3. What would the status display 0-8-0000 mean?

4. You'll notice, in the lower right corner of the front cover of your HP-16C Owner's Handbook, there's a picture of something that could be a "dump" of a computer's memory. All those numbers are in hexadecimal. And there are 8 hex digits in each grouping. What do you suspect is the Word Size of this computer?

   What are the decimal equivalents of the hex numbers in the group beginning with AB643106?

U. R. I. Answers

1. With a Word Size of 32 bits, BIN format could use up to 4 windows, depending on whether leading zeros are shown or not (flag 3). In OCT and DEC formats, there could be up to two windows, but just one window with HEX.

2. To scroll right, press [g] [➤].

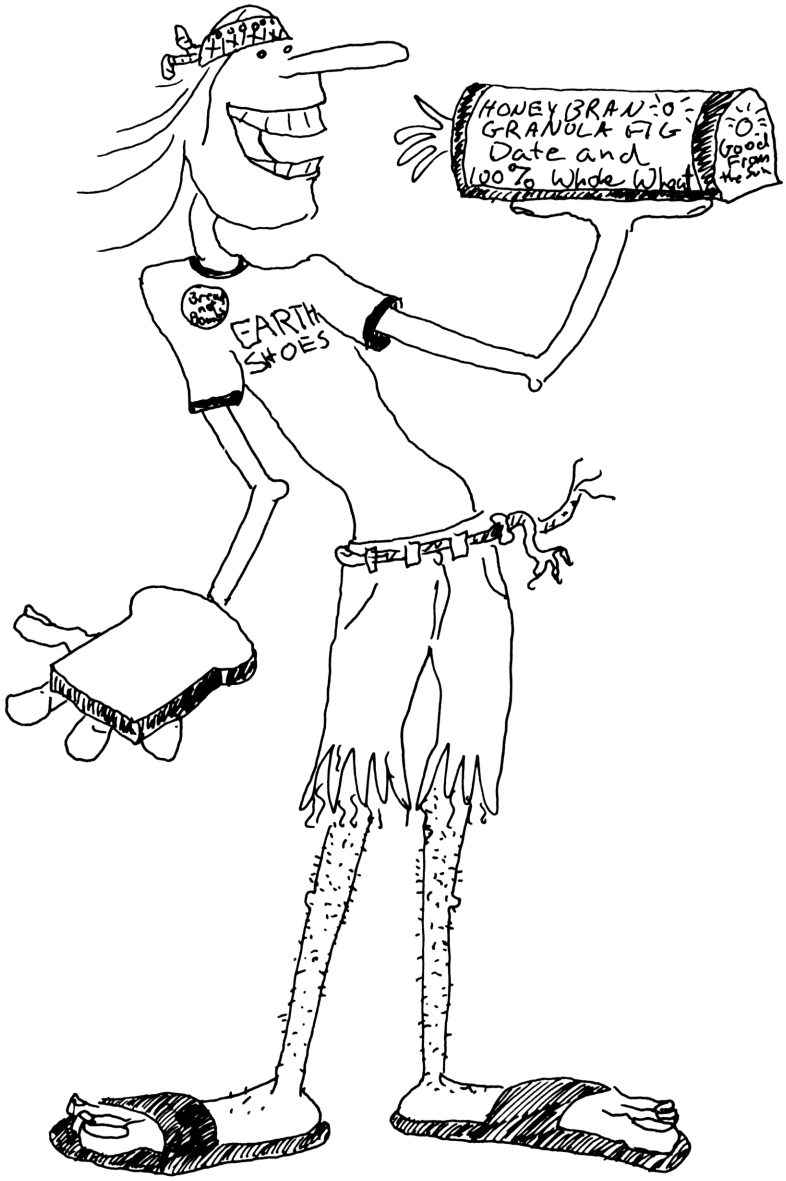
3. `0-8-0000` would indicate that the display register is in UNSigned format. The word size is currently 8 bits and flags 3, 2, 1, 0 are all clear.

4. The computer apparently works with a 32-bit word size.

You Press	You See	Why
[DEC]	??? d	Get into decimal format.
[3][2] [f] [WSIZE]	??? d	
[HEX] [A][B][6][4][3][1][0][6]	Ab643106 h	Key in the number in hex format.
[DEC]	-1949721 9 .d	The dot left of the d says that there's more to this number than meets the eye. Note also that the negative sign shows up even in the zeroth window.
[f] [WINDOW] [1]	- 14 d.	(Left part of - 14 1949721 9 d)

Results for all four numbers:

AB643106 (h)	=	-1419497219 (d)
4966041B (h)	=	1231422491 (d)
6E0F3184 (h)	=	1846489476 (d)
6ECE31E3 (h)	=	1859006947 (d)



INTEGER ARITHMETIC

Now that you've successfully explored most of the display properties of the HP-16C in Integer Mode, it's time to look more closely at how the calculator does arithmetic and logic operations with all these bits and words.

First of all, exactly how does the HP-16C do integer arithmetic?

(This may seem obvious after having all that practice with arithmetic and the stack in Floating Point Mode, but this isn't quite as straightforward. So even if you've skipped other parts of this Easy Course, you probably shouldn't skip this part.)

Operations That Need Two Numbers

There are two kinds of arithmetic operators in the HP-16C: those that use only one number (the one in the X-register) and those that require two numbers (usually the X- and Y-registers). Remember? You got a taste of this when warming up with Floating Point Mode.

And as you'll recall, the most prevalent kind of operator on the HP-16C is the two-number kind: addition, subtraction, multiplication and division, etc. Such mathematical operators are called **dyadic** operators, since they require two numbers (each called a dyad) to get a result.

Of course, after having seen how to use the HP-16C to add two numbers in Floating Point Mode, you might think that the only difference here in Integer Mode is that you don't have to worry about the decimal point. Well, it's a little more involved than that.

Why? Because you have to know how the calculator responds when the results it calculates *are too large for a given Word Size.*

This means you need to keep one good eye on the display, where little flag annunciators (G and C--representing flags 4 and 5) may alert you to carrying, borrowing, and overflow problems.

Bottom Line: You gotta keep your wits (and bits) about you when doing integer arithmetic.

RPN Integer Arithmetic

Start from a familiar point:

As you know from your earlier practice with the stack, the basic keystroke sequence you always use to perform one simple dyadic operation is this:

1. You key in a number;
2. You press the [ENTER] key;
3. You key in a second number;
4. You press the operator key ([+], [−] etc.);
5. You look at the display to see the result;
6. You nod sagely.

OK, now here's what's really going on when you do this in Integer Mode. Remeber: You may know some of this--but probably not all of it:

1. As you key in the digits of any number, the calculator is placing a string of binary digits in the X-register--forming a binary number equivalent in value to the number you see in the display--regardless of which display format you are using. Each time you add another digit, the bits in the X-register are recombined to represent the resulting integer (but you knew this, of course -- from page 67).

2. The HP-16C is constantly watching your keystrokes so that you don't key in a digit that's illegal for a given display format. For instance, if you're operating in ◘ display format, the calculator will prevent you from keying in the digits 8 and 9, because those digits aren't available in base 8 (and you saw this back on page 70).

3. As you continue to key in the number, the calculator also checks the Word Size, disallowing any number that's too large for that Word Size (page 69).

4. Then, when you press the [ENTER] key, you do several things:

   First, you terminate the entry of the first number--telling the HP-16C, "I'm done--that's the whole dyad you've got there now." At that point, the display register will show you the final version of the number in the X-register. For example, if you're working in HEX, OCT, or BIN display format, and with flag 3 set, the display will show you your dyad with any leading zeros it might need in order to fill out the current Word Size (see page 90 if this sounds only vaguely familiar.)

   Next, the number in the X-register is duplicated into the Y-register. The rest of the numbers in the stack are pushed up by one register, and what was formerly in the T-register is blown off the top (as on page 30)

   Then the stack-lift feature is *disabled.* meaning that if your very next move is to introduce some new number into the X-register, in doing so, you will not push the current contents up a register; instead you will simply overwrite (replace) what's now in the X-register--which, at the moment, is one copy of your dyad (and all this is painfully obvious to you after that Stack Quiz back on page 37, no?)

5. Now, with the stack-lift disabled, you begin to key in the second number (the second dyad). And sure enough--it simply overwrites that bottom copy of the first dyad. Of course, while you're keying in this second integer, the X-register and the display are going through their usual routines, accumulating binary bits in the X-register--but showing you the number, as it forms, in whatever format you've requested.

6. Then, when you press the operator key, the HP-16C really gets busy:

   It stores the current value of the X-register in the LSTX register (and this is something you may not have realized yet; you'll see it in coming examples).

   It then removes the values from the X- and Y- registers, lowering each remaining value in the stack by one register, duplicating the contents of the T-register down into the Z-register (you've seen this, right?--page 31).

   The calculator then does its actual arithmetic, computing the result of the mathematical operation and placing this result back into the X-register (again, old hat, from page 31).

   The stack lift feature is *enabled*--so that if the next thing you do brings a number to the X-register, the contents of the stack *will* bump up one notch (recall from page 32 that most operations leave the stack enabled; in fact, the only two common ones that don't act this way are [ENTER] and [CLX]).

   If the result is too large for the given Word Size--or if some unusual carrying or borrowing must be done, the HP-16C will set the appropriate flag (thus turning on the little G or C annunciator in the display), *and trim the result* to fit the given Word Size (this is certainly *not* something you saw in Floating Point Mode).

   The display register interprets the result--in your chosen display format.

   The calculator pauses and waits for your next command.

Whew! That's a lot of work for one little calculator to do (and a lot of words just to describe it). So here are...

Some Examples:

First, to make sure you're going to be working with the same calculator you'll see here, be sure to do the following keystrokes:

f SET COMPL 2'S	(sets the display to 2's complement format).
DEC	(sets the display to show decimal integers).
9 CF 3	(suppresses the display of leading zeros).
16 f WSIZE	(allows each register to handle numbers up to 16 bits wide).

Ready? All right...

Try This: Evaluate $\dfrac{13 \times 32 + 6}{12}$

Keystrokes	What You Should See
13	13 d
ENTER	13 d
32	32 d
×	416 d
6	6 d
+	422 d
12	12 d
÷	35 d

Notice that you press the [ENTER] key but once–right after the original entry of the first dyad. Why? Because the [ENTER] key is only necessary when you want to key in more than one dyad in succession; here, after keying in the first two dyads, you then *alternate dyad and operator.* Thus, you don't need [ENTER] anymore.

The other thing to note is that the last operation, division, gave you only an integer--35-- as a result(!) After all, doesn't 422/12 = 35.166666....?

Yes--and the HP-16C knows this, too. In fact, it has left you a display message, saying, in effect, "Be careful! This arithmetic wasn't as clean as it looks!"

That's right: the C annunciator is turned on in the display.

When doing division, the C annunciator is the HP-16C's way of telling you that the result had non-zero remainder.

So how would you tell how much that remainder was? Is there any way to do this?

"But of course!"

Do It: First, shut off the C annunciator by pressing [9] [CF] [4]. Then...

Press	And See
[4] [2] [2]	422 d
[ENTER]	422 d
[1] [2]	12 d
[f] [RMD]	2 d

The [RMD] key will let you determine the remainder of any integer division.

A New Problem: Evaluate the following expression:

$$\frac{C(h) \times 40(o) + 110(b)}{13(d)}$$

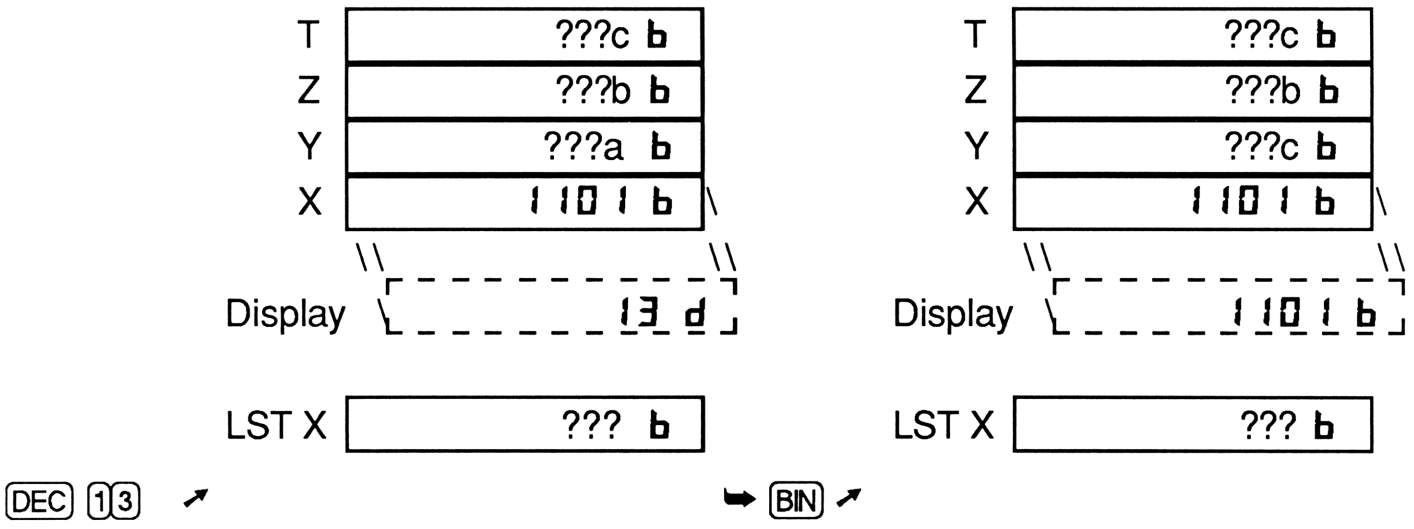
AND give the final result in decimal (base-10) format.

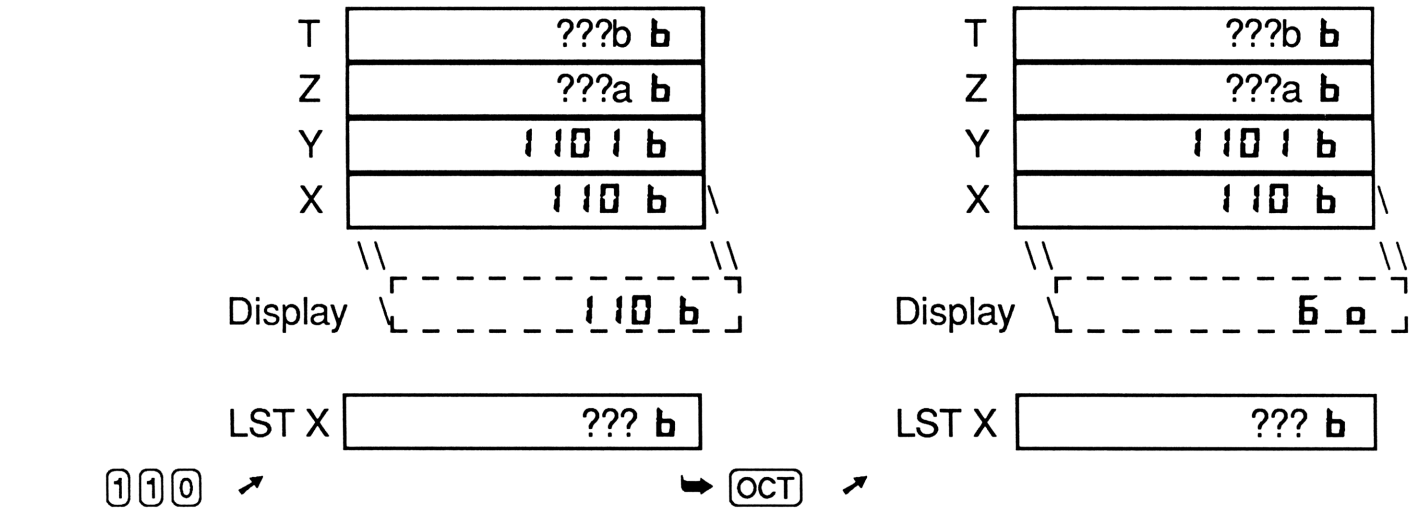
Solution: This time, put *all* the numbers in the stack *before* performing any mathematical operations. This will let you see another property of the [BIN], [OCT], [DEC], and [HEX] keys (and it will also give you some more practice in using the stack manipulation properties of the calculator)

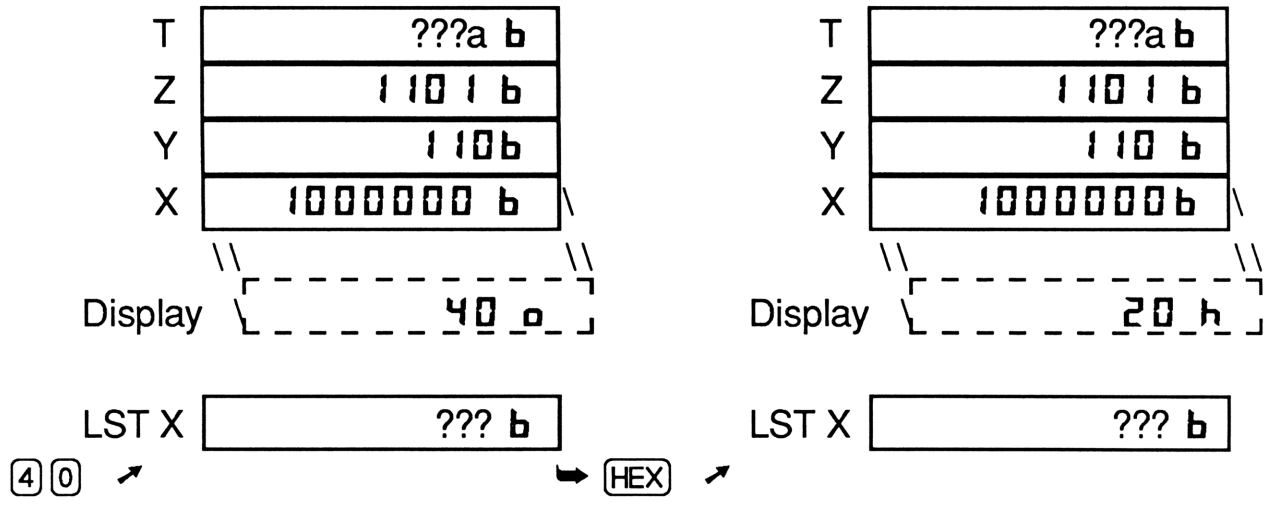
Keystrokes	What You Should See
[DEC] [1] [3]	13 d
[BIN]	1101 b
[1] [1] [0]	110 b
[OCT]	6 o
[4] [0]	40 o
[HEX]	20 h
[C]	C h
[DEC]	12 d
[×]	384 d
[+]	390 d
[x≷y]	13 d
[÷]	30 d

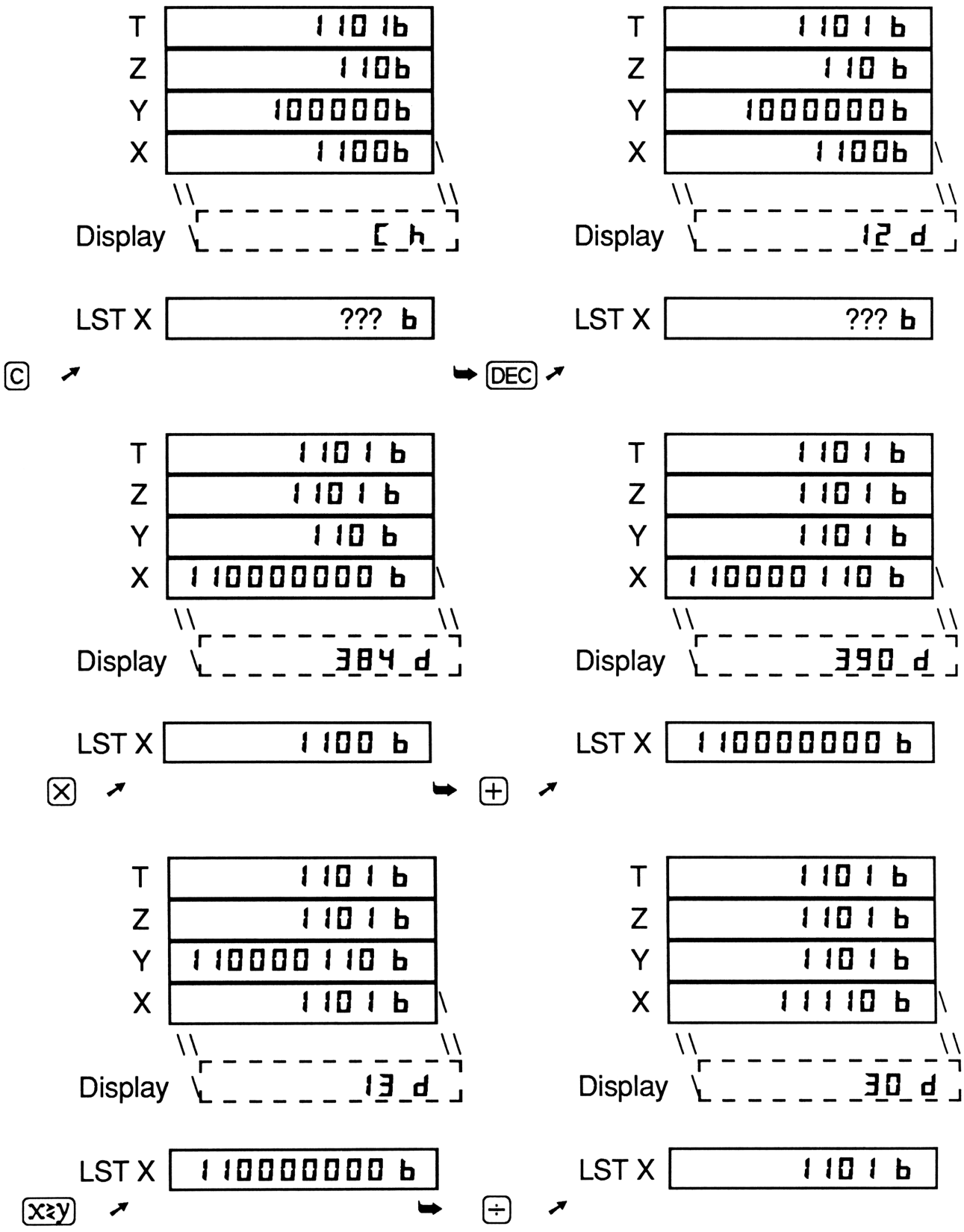
Notice: When you shift from one number base to another, this terminates digit entry and enables stack lift.

Take a look at what's going on in the stack during this last problem:








T
Z
Y
X
1101b
110b
1000000b
1100b
Display
C h
LST X
??? b
C
DEC
1101 b
110 b
10000000b
1100b
12 d
??? b
1101 b
110 b
1100 b
384 d
LST X
1100 b
×
1101 b
1101 b
110 b
1100000000 b
390 d
LST X
1100000000 b
+
1101 b
1101 b
1100001101 b
1101 b
13 d
LST X
1100000000 b
x≷y
1101 b
1101 b
1101 b
11110 b
30 d
LST X
1101 b
÷

Notice a couple of things:

1. In this example, you've taken advantage of the fact that the T-register replicates itself when the stack drops:

   You began the problem by keying in 13 d (i.e. 1101 b), but then you let it just "float" up above in the stack, replicating itself until you needed it in the X-register to perform the integer division. By the time you actually needed it, there were copies of it from the T-register all the way down to the Y-register; so all you had to do to get it to the X-register was to use the [x≷y] key.

2. The other thing to notice is the LST X ("LaST X")-register--something you've not yet looked at very closely:

   The whole idea of the LST X-register is to preserve a record of what was in the X-register *prior to any arithmetic operation*. And this makes some sense, don't you think?

   For example, if you make a mistake and press [+] where you should have pressed [−], what do you do? The number now sitting in the X-register isn't what you wanted. How can you recover--without starting your calculation all over again?

   Simple--just press [9] [LSTX]. (which really means [RCL] [LSTX]).

   Appearing out of nowhere is the number you mistakenly added--so now subtract it. Bingo! You're back where you were before you made your mistake (and now you would go ahead and subtract the number again--as you had originally intended, right?). That's the beauty of the LST X-register.

Some More Problems

So far, all the arithmetic operations that you've performed have stayed within the bounds of your 16-bit Word Size. There haven't been many surprises, really.

So keep on going--walk through several more problems like the previous one -- but don't be surprised if things get a bit more complicated. You may get into some tall weeds before it's over, so be ready to do some clear thinking and smooth talking with your HP-16C.

Try This: Evaluate the following, and express the result as a hex number:

45(h) + 25(d)

Keystrokes	What You'll See
DEC 25	25 d
HEX 45	45 h
+	5E h

See how the right kind of rearranging can save you a few keystrokes?

You keyed in the hex number last so that you didn't need to change formats again before getting your final answer.

Also, you didn't ever need to use the ENTER key because your change of formats midway through the problem effectively terminated your first entry (25) and enabled the stack--thus configuring the stack exactly as you would want it to be prior to your next entry.

How About This: Evaluate it, and again, show the result in hexadecimal:

$$[204(o) + 130(o)] / A(h)$$

Keystrokes	What You'll See
OCT 204 ENTER	204 o
130 +	334 o
HEX	dC h
A	A h
÷	16 h

(And there's no remainder--as you can tell from the fact that the C annunciator is now off.)

Now Then: Find the quotient *and* the remainder of $\frac{7(o)}{5(o)}$

Keystrokes	What You'll See
OCT 7 ENTER	7 o
5 ÷	1 o
(with the C annunciator on, indicating a remainder)	
g CF 4	1 o
(turns the C annunciator off)	
7	7 o
g LSTX	5 o
(this is what was in the X-register the last time you performed any arithmetic--when you pressed ÷ above.)	
f RMD	2 o

And Of Course: Express this in hex and also in 2's complement decimal format:

10(d) - 3F(h)

Keystrokes	What To Expect
f SET COMPL 2'S DEC	??? d
10	10 d
HEX	A h
3F	3F h
−	FFC6 h
DEC	-53 d

Notice that the C annunciator is on (and of course, you can clear it by pressing g CF 4).

But why exactly *does* that C annunciator appear in the display here?

After all, this isn't even a division problem; there couldn't be any remainder. And anyway, how could you get such a large hex number, FFCB, as the result of subtracting two relatively small numbers?

Well, it's time to look very closely at integer arithmetic in general--and specifically what that C annunciator tells you. All you know about it so far is this:

It comes on whenever the result of integer *division* has a non-zero *remainder.*

But what about plain old addition and subtraction?....

Understanding Integer Arithmetic

To understand why the HP-16C comes up with some of its answers, you must realize that it *doesn't necessarily do arithmetic in the way that mortals would.*

Addition

To work your way up to computer math, you need to begin with grade school math--the kind that Ms. McBride taught you back in second grade.

Surely you remember doing an arithmetic problem like this: 87 + 45

Your paper would probably end up looking something like this:

```
   1  1
      8  7
+     4  5
  ---------
   1  3  2
```

You would put the two numbers in a column and add up one column at a time, starting on the right (the ones column).

But you want only the extra ones in the ones column, and since 12 is really made up of 1 ten and 2 ones, you carry that 1 ten over to be included when you add up how many tens you have (and as you know, it's traditional to put this carried 1 as a small mark up at the top of its proper column).

So off you go, moving right to left, doing this same thing for every column. Of course, there are no other hundreds from the original numbers to add to the final carried 1, so it begins a new column--the hundreds column--and becomes the 1 in your final answer: 132.

Now, do a similar addition problem in base 2:

Suppose you were to add (in binary) the equivalents of 13 (d) and 9 (d):

$$\begin{array}{rr} & 1\,1\,0\,1 \quad \text{(b)} \\ + & 1\,0\,0\,1 \quad \text{(b)} \\ \hline \end{array}$$

As usual, you would start on the right, adding and carrying as before. But just remember: You carry over to the next column whenever the result of your current column reaches 10 or more (and that 's not "ten;" it's "one zero").

(And keep in mind also that $1 + 1 = 10$ and $1 + 1 + 1 = 11$)

Here's the result of your addition, again, with the carried digits appearing in smaller size at the top:

$$\begin{array}{rr} & {\scriptstyle 1 \quad 1} \\ & 1\,1\,0\,1 \quad \text{(b)} \\ + & 1\,0\,0\,1 \quad \text{(b)} \\ \hline & 1\,0\,1\,1\,0 \end{array}$$

So the total would be 10110(b) or 22(d).

Well, this is basically what your HP-16C does, too--as do most computers when they add binary numbers. It's all very straightforward, right?

Right, but here's the rub:

Suppose you were like a computer, limited to, say, a Word Size of four bits. In that case, when it came time to carry the 1 into the fifth column, you would have had no place to put the Carry.

So you would have to hide it or forget it.

And this is what the HP-16C does in this particular situation: it hides the last carry and shows only a four-bit answer, 0110 b (6 d--nowhere near the correct answer of 22 d).

But it doesn't totally ignore the final carry. It holds the number in an "imaginary" bit, called (strangely enough) a Carry bit.

So instead of showing you the full number, 10110 b, the calculator will show you the truncated number, 0110 b, and turn on the C annunciator *and* the G annunciator.

This is the calculator's way of telling you two things:

-- The G annunciator simply means (as it always does) that the correct answer (10110 b) cannot be represented with the current Word Size (four bits).

-- The C annunciator tells you something more specific: In doing this addition, there was a 1 carried over from the MSB (the fourth bit here) into the Carry bit.

<u>So:</u>

For doing *addition,* this C annunciator is the HP-16C's way of telling you *there's a 1 in the Carry bit* .

By comparison, in doing *division,* it meant there was a non-zero *remainder* .

Subtraction*

Now, what about subtraction? Suppose you similarly examine the process of subtraction in bases 10 and 2, respectively.

What if you wanted to do this subtraction problem?

```
  3 4 2
- 1 7 3
-------
  1 6 9
```

In second grade, your teacher may have taught you that if you were asked to subtract more from any column than what you had to begin with, you had to go next door (to the left, of course) to "borrow" 10 extra (and that meant taking away 1 from that neighboring column, since it had ten times the "weight" of your current column). It was a much more complicated set of instructions for second graders to grasp:

"You can't take 3 from 2; 2 is less than 3, so you look at the 4 in the tens place. Now, that's really 4 tens, so you make it 3 tens, regroup (and you change a ten to 10 ones), and you add it to the 2 to get 12, and you take away 3. That's 9.

"Is that clear? Now, instead of 4 in the tens place, you've got 3 (because you added 1--that is to say, 10--to the 2), but you can't take 7 from 3, so you look in the hundreds place..."

So you borrow similarly from the hundreds place, change it to 10 tens, add it to the 3...."So you have 13 tens and you take away 7 and that leaves"...well, you get the idea.

It's not nearly as simple as addition, is it?

*with apologies to Tom Lehrer

Can you imagine doing a similar operation in binary, using, say, a four-bit word?

```
    1 1 1 0
-   1 0 1 1
-----------
```

All you second graders would have heard this:

"You can't take 1 from 0; 0's less than 1, so you look at the 1 in the twos place. Now, that's really 1 two, so you make it 0 twos, regroup (and you change the two to two (10) ones) and you add it to the 0 to get 10, and you take away 1. That's 1.

"OK? Now, instead of 1 in the twos place, you've got none (because you added 1 -- that is to say, two (10)--to the 0), but you can't take 1 from none, so you look in the fours...."

Etc.

Now, that's probably not the simplest way to get this idea of subtraction across to anybody. And if a second grader might have trouble with it, you *know* a computer would. After all, smart as it may seem, a computer is just a simplistic tool, incomparably less complex than a human being (it doesn't even *like* mud puddles--let alone go stomping in them on purpose).

So it's a cinch that a computer wouldn't do its subtraction with the old "go-borrow a-cup-of-10's-from-the neighbor-to-the-left" method.

What about your HP-16C? It's supposed to think like a computer--but communicate with you--a human being. So how should it do its subtraction?

To see how it happens, its best first to understand how a computer would do it. So go back and repeat the subtraction--but this time, the idea is to *add* a -173 to 342.

And why's that? Well, remember how you first learned about negative numbers and why computer scientists chose to represent them with 1's and 2's complement formats--page 77? It was so subtraction could happen in the computer with exactly the same logic as addition: Adding the negative is the same as subtracting the positive.

That's how a computer would do subtraction. To do that yourself, then, you would need to somehow represent -173 in a way that allows you to stack it under the 342 and actually go through your normal addition process (i.e. line up columns to add-and-carry)--just as you do when both numbers are positive. (And of course, you want the right answer when you're through.)

So try it--in base 10, of course: You need to represent -173 not with a minus sign slapped in front of it, but rather by taking the *10's complement* of 173.

Right.

What's that?

Remember back on page 76, you went through the tape-counter argument for representing a negative binary number? In this case here, suppose you had a (normal) tape counter with ten places (0-9) on each wheel. Then--as you'll recall--to represent a negative number, you would begin at zero and move backwards--in this case, 173 notches. Then, whatever number the counter showed would be the 10's complement of 173.

And what would the counter show? Hmm...if you moved back 1 notch, it would show 999; if you moved back 2, it would show 998....It looks as if this 10's complementing business is really just counting backwards from 1000, doesn't it?

That's exactly right:

The 10's complement of a decimal number is the difference between that number and the next higher power of 10. That's the mathematical definition.

Therefore, for 173, the next higher power of 10 would be 1000.

And 1000 - 173 = 827

So 827 is the 10's complement of 173

Now you perform the subtraction by *adding* the 10's complement:

$$\begin{array}{rr} & 342 \\ + & 827 \\ \hline & {\scriptstyle 1}\ 169 \end{array}$$

And--as a computer would--you should ignore any digit that appears in a column not appearing in the original numbers: There were only three columns in the two numbers 342 and 173; so ignore any digit past the third place.

Voila! The Answer!

But wait a minute (you object)! How does this save you from having to subtract anyway? You had to figure out 1000 - 173 , didn't you?

It's true--in base 10, it looks as if you still need to do "real" subtraction--using the borrow method and all that rot.

Nope....

An easier way to find the 10's complement is to find the 9's complement of a number and then add 1. The mathmatical definition of the 9's complement of a decimal number is *what you get when you subtract it from "all 9's:"*

```
  9 9 9
- 1 7 3
-------
```

Thus, the 9's complement of 173 would be 826.

You can see how the 9's and 10's complements differ only by a value of 1, can't you (after all, 1000 - 173 is the same as 999 - 173 + 1, right)?

So, to go from 9's to 10's complement, all you do is add 1: 826 + 1 = 827.

Presto! There again is the 10's complement of 173.

You see? Although this is still conventional subtraction, you're guaranteed not to need to borrow, since each column begins with a 9!

A devious trick, admittedly.

But: It's a very *useful* trick when you start dealing with binary numbers. To wit: As you know, most computers do their binary subtraction by adding the 2's complement (i.e. the "negative") of the number to be subtracted.

Well, the 2's complement is exactly analogous to the 10's complement: It's the difference between the number itself and the next higher power of 2 (if you think about it for a minute, that's merely another way of stating the definition of 2's complement that you saw on page 80).

AND, just as you've discovered with base 10, an easy way to get at the 10's complement was to take the 9's complement first--then add 1. Well, shucky darn, will you look at this: Remember back on pages 80-81? You saw that an easy way to get the 2's complement is to take the 1's complement--then add 1!

Next non-coincidence: Do you remember what it means to take the 9's complement of a decimal number? That's right--it meant simply to subtract that number from "all 9's." And this eliminated any need to borrow, didn't it?...

Well then, no prizes for guessing how to take the 1's complement of a binary number. Do it right now--take the 1's complement of 1 0 1 0:

```
   1111
-  1010
------
   0101
```

It's easy! It involves no borrowing and can be done practically by inspection, right? Right. In fact, if you'll notice, the whole process simply reverses the bits: Every 1 turns into a 0, and vice versa--again, as you saw on page 80.

Now *that's* the kind of logic a computer can handle--and it does so--extremely well. This is the way it comes up with the 1's complement of a binary number--inverting the bits.

OK, fine. But remember that the aim is to get at the 2's complement of a binary number--so that your computer can easily use it for easy binary subtraction.

Just add 1 to the 1's complement, right?*

$$0101 + 1 = 0110$$

So the 2's complement of 1010 is 0110. Does this truly let you subtract 1010 from any other 4-bit number--simply by adding this 2's complement form of the number? It should--if all this haranguing was worth anything:

$$\begin{array}{rr} & 1011 \\ - & 1010 \\ \hline \end{array}$$

Find the 2's complement of the bottom number (by taking the 1's complement and then adding 1--you already know the answer to this one, having just done it). After doing so, the problem is reduced to this:

$$\begin{array}{rr} & 1011 \\ + & 0110 \\ \hline \end{array}$$

Adding these two numbers gives 10001. And you know enough to ignore any number in the carry bit (i.e. the 5th bit in this 4-bit problem).

$$\text{So} \quad 1011 - 1010 = 1.$$

Sure enough.

*By the way, in case you're interested, there's another easy way for humans and other life forms to find the 2's complement of a binary number: Start with the Least Significant Bit and start copying down the digits from right to left--just as they appear. Continue to *copy* the digits until you *pass* the first 1. From there on *invert* the digits. You'll wind up with the 2's complement. Try it!

More To-Do With 1's and 2's Complement

OK, you've now seen how a *computer* might use 1's and 2's complements to perform subtraction. But is this how the HP-16C actually does it? Not really.

With subtraction, the HP-16C actually does borrowing and carrying, just like you did in second grade--whereas a computer would use the trick of adding the complement.

Now why on earth (or anywhere else) would the HP-16C go to all the trouble of borrowing during subtraction, when every computer it's supposed to help you design will be "cheating" with this 1's and 2's complements business?

Because--as you read on page 115--the HP-16C must somehow talk to you in *your* arithmetic "language" and then turn around and behave and "think" like a computer. It must *emulate* a computer without actually *being* one; otherwise, it wouldn't be much of an interpretive tool to use in following the computer logic. After all, an interpreter must speak both languages.

That's why the annunciators, C and G, are there--to help the interpretation. They try to give you consistent reports on what's happening to the bits as your HP-16C plays its role and obeys the computer's logic of 1's or 2's complement arithmetic. And they seem so intertwined with the idea of 1's and 2's complement that it's easy to confuse the issues. But fear not--more help is on the way!

Because of the interrelatedness of these ideas, there's more than one way to express their relationships, and therefore you're going to see it explained from more than one angle here.

So if you're still confused at this point, don't worry, OK? One of these upcoming explanations *will* work for you--and you're in good company meanwhile!

Here is a puzzle:

How can you add two apparently identical binary numbers and arrive at two different answers that are both correct? To see how this can happen,

---

Try This: Press f SET COMPL 1'S to set 1's complement display format.

Establish a Word Size of 4 (DEC 4 f WSIZE).

Then press the BIN key.

Check the status of the calculator by pressing the f STATUS keys. You should see

1-04-0000,

As you know, this means that you're in 1's complement format, the Word Size is 4 bits, and flags 3, 2, 1, and 0 are all set to zero (i.e. "clear," not "set").

Now add

```
  1101
+ 1101
------
```

by pressing 1101 ENTER + (remember this shortcut?)

The answer: 1011b

Notice that the C annunciator is on.

Now do the same addition problem on paper or in your head. If you didn't limit your brain to a 4-bit Word Size, what answer would you get?

It would be 1 1 0 1 0 , right?

So, how did the HP-16C arrive at this answer? How did it get 1011 b when you got 1 1 0 1 0 ?

To arrive at its answer, the HP-16C performed its binary addition in the same way that you did.

However, it also detected that there was an overflow beyond its 4-bit word size. That's why the Carry (C) annunciator turned on: No matter what format you're using, the C will appear anytime a 1 is carried into the Carry bit.

And then, *under the rules of 1's complement*, this Carry bit is *not* ignored (unlike 2's complement). Instead, the display register takes that Carry bit and adds it to the sum, thus effectively adding it back to the right end of the number. This is what's called an End-Around Carry (Jerry Kramer and the old Green Bay Packers would be proud):

```
  1 0 1 0
+       1  (from the Carry Bit)
---------
  1 0 1 1
```

OK, that's how the calculator would figure the binary answer *under the 1's complement addition rules it simulates from a computer.*

Now, do the same addition in 2's complement format and "computer rules" and compare the results....

Go: Begin by pressing [f] SET COMPL [2'S].

Then press [g] [CF] [4] to clear the C annunciator (remember that the C signals that the Carry flag, #4, is set).

If you now press the [f] [STATUS] keys, you should see:

2-04-0000

Now perform the same addition as before: [1][1][0][1] [ENTER] [+]

You'll see: 10 10 b

And the C annunciator will be on again.

But the answer here is 1 less than in the 1's complement case! Here, with 2's complement rules, the calculator does NOT perform an End-Around Carry; rather, it ignores the Carry bit altogether, as you saw earlier (page 77).

At this point, if you're like most card-carrying Homo sapiens, you might say "Aww, c'mon, gimme a break! Which answer is right--this one or the one on the previous page?"

And, of course, the reply is "Both!"

Puzzled by this puzzle? Well, just perform both additions again, but this time, use [f] SHOW [DEC] to *check the decimal values* of the numbers--both before and after you double them (i.e. add them to themselves)....

Guess what.

1101 b in 1's complement display format *is not the same number* as 1101 b in 2's complement format!

You weren't starting from the same point. It *looked* like it, but you weren't.

In the first case (1's complement), you were starting with -2 d, then doubling it to get -4 d. In the other case (2's complement), you started with -3 d and wound up doubling it to -6 d.

So the answer to the puzzle is this reminder: The format setting of your HP-16C affects *both* how the machine's display *shows* you the interpreted value of the bits it contains *and* how it actually *crunches* those bits during its arithmetic. In essence, it's following two different counterbalancing sets of translation rules (*displaying* the number's interpreted value and *crunching* the arithmetic) in order to maintain mathematical accuracy; and if you forget to use *both* of these translations when checking your math, things are going to look all bollixed up.

In this sense, the word format is probably a little misleading, since you usually think of format in terms of the calculator display and how it edits things for you. But here you can begin to see that it means not only how the HP-16C interprets bits to *you* but also how it interprets bits to *itself* during its simulated arithmetic.

That's what you mean when you say that it's "acting" like a computer: it's following the computer's arithmetic rules. And if you don't pay close attention to its counterbalancing interpretation as to how a binary number is *assigned a decimal value* within those rules, you're going to wind up with a massive headache from all the *apparent* contradictions.

Starting to make some sense out of those two formats, 1's and 2's complements? Try another illustration--this time with subtraction....

Try This: While you're still in 2's complement format, press [g] [CF] [4] to clear the C annunciator from the display.

Then perform this subtraction:

```
  0100
- 0110
------
```

The display shows the answer to be: 1110 b

The C annunciator is also turned on.

Is it because the HP-16C is using the computer rules in its subtraction--adding the 2's complement of the bottom number? Here's how that might look:

```
  0100                  0100
- 0110    becomes    + 1010    which results in  1110
------               ------
```

Well, that arrives at the correct answer, all right.

But then *why* is the C annunciator turned on? The final addition didn't force you to Carry into a fifth bit! Have you done something wrong?

Check yourself, by doing the same subtraction using 1's complement format (and rules).

Strategy: First, be sure to clear the C annunciator from the display. Then press the [f] SET COMPL [1'S] keys.

Now key in [1][0][0] [ENTER] [1][1][0] [−].

In the display is `1101 b`, with the C annunciator on again!

Drat! No help there. Do the 1's complement "computer rules subtraction" on paper, instead:

```
    0100                    0100
-   0110    becomes     +   1001
                        ----------
                            1101
```

"(Sigh) That's the right answer, but it still doesn't explain why the C annunciator was set. There wasn't any carrying into the fifth bit necessary here, either!"

That's right. But there was some borrowing(!)

You haven't done any of the math wrongly. You just wrongly assumed that the HP-16C does its subtraction as a computer would--and it doesn't. It *arrives* at the same result a computer would if that computer were using 1's or 2's complement arithmetic rules. But in actually subtracting bits from bits, the HP-16C doesn't use the complementing trick at all. It actually borrows.

No matter which complement format you're asking your HP-16C to simulate, if in doing so it must subtract 1 from 0 in the MSB (far left), it won't be able to, of course. So it looks to an imaginary Borrow bit from *beyond* the MSB (one place more to the left), borrowing that 1 to finish its MSB subtraction. The C appears during subtraction to signal this forced Borrowing *beyond* the MSB--just as it appears during addition to signal a forced Carry beyond the MSB.

Now, forget for a minute what the HP-16C is actually *doing*. What is it trying to *simulate?* Can you tell what the difference in logic is between 1's and 2's complements for subtraction? That is, what does a big, dumb computer have to do to arrive at a correct answer when "cheating" at subtraction (i.e. adding the complement)?

You already saw that with addition, a computer using 1's complement would add the Carry bit (if any) around to the righthand end--in a left-to right, End-Around Carry.

In 2's complement, it would ignore this Carry bit altogether.

With subtraction, it's just the opposite: A computer using 2's complement simply adds the 2's complement of the number, and there's the answer.

A computer using 1's complement would subtract by adding the 1's complement, but then it would *subtract* 1 from the righthand end (the LSB). This is therefore a right-to-left, End-Around Borrow.

Look at one last summary now--of what you've discovered about 1's and 2's complements--and UNSigned formats. Here are both the rules for representation (four-bit Word Size) and for arithmetic (addition and subtraction). These two parts for each format are the counterbalancing set of translations the HP-16C does for you to ensure the proper representation and mathematical accuracy of its computations. Note the interrelations:

Binary Pattern	Format		
	UNSigned (no neg. values)	1's Complement (- = inversion)	2's Complement (- = inversion + 1)
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-7	-8
1001	9	-6	-7
1010	10	-5	-6
1011	11	-4	-5
1100	12	-3	-4
1101	13	-2	-3
1110	14	-1	-2
1111	15	-0	-1
Addition Rules	Ignore Carry bit	Left-to Right, End-Around Carry	Ignore Carry bit
Subtraction Rules	Ignore Borrow bit	Right-to Left, End-Around Borrow	Ignore Borrow bit

Now take a last summary look at the logic of the C annunciator (Flag 4):

The appearance of the C annunciator is always for the same reasons--regardless whether you're working in UNSigned, 1's, or 2's complement format:

With addition: The C appears when*ever* an addition forces the calculator to *carry beyond* the MSB (the Most Significant Bit), *regardless* whether the current "computer rules" would subsequently add that Carry bit back onto the other end.

With subtraction: The C appears when*ever* a subtraction forces the calculator to *borrow* from *beyond* the MSB, *regardless* whether the current "computer rules" would subsequently subtract that Borrow bit from the other end.

With division: The C appears when*ever* an integer division results in a non-zero remainder. This is the easiest to remember; the only caution is that--as always--you must be sure that the interpreted value of the numbers you key in are what you intend them to be (remember the apparent paradox you saw on page 124?). The same keystrokes will key in different integers under different complement formats!

And don't forget the G annunciator (Flag 5):

The G appears anytime the correct answer to a calculation simply cannot be expressed within the constraints of the Word Size and the complement format.

If your result is too big (or if it's negative in the case of UNSigned mode), that G will pop out and yell at you (but you can shut it up again by pressing [g] [CF] [5]).

Just remember: If the G shows up, you can't trust the answer you're seeing.

The Other Dyadic (Two-Number) Math Operations

Besides [+], [−], and [÷], there are five other math keys on the HP-16C: [×], [RMD], [DBL×], [DBL÷], and [DBLR].

Those last three keys involve a discussion of "double sized numbers," and that looked like a good candidate for an Appendix--so check in the back of the book here for more about those.

You've already seen an example of the use of the [RMD] key to find the remainder of a division process. Not much more to tell there.

That leaves the [×] key, then.

Not much has been said about it up to this point--probably because it behaves just as you would expect it to from your practice with it in Floating Point Mode, except for one thing:

Whenever the result of multiplication (or division) is too great for the Word Size, the X-register will retain only the *least* significant (righthand) bits *and* the sign bit (if any) of the full answer (and of course, the G annunciator will announce the overflow).

This may help to explain that weird answer you get when you try to square 1100(b) (which is 12(d)) with a Word Size of 8, in 2's complement format.

(Yep, that's a direct challenge: Try it on your own and see if you can follow the logic behind the bits that finally show up. Then try 12(d) x -12(d)).

Single-Number Integer Operations

Similar to those you saw in Floating Point Mode, there are a few mathematical Integer Mode functions on the HP-16C that work only with the number in the X-register. These are the [CHS], [ABS] and [√x] functions. And because they work with only one number at a time, they are called Monadic operators.

Of course, you've already seen how to change the sign of an integer with the [CHS] key--and how this process differs in 1's and 2's complement modes (check back on pages 80-81 for a quick review, if you want). But what about the other two functions?

The [ABS] Key

You can find the absolute value of any integer by keying it into the X-register and then pressing the [g] [ABS] keys.

As you might expect, if the number is positive, or if you're in UNSigned mode, there'll be no change in the display. The only thing that will happen inside the calculator is that the number will be stored in the LST X-register.

(And this makes sense, right? The LST X-register is duty-bound to record the value in the X-register anytime that value is about to undergo a mathematical operation which could change its value. And although a positive number's value wouldn't change with the [ABS] function, a negative number's would; hence the LST X-register doesn't ask any questions--it just records that X value--positive or not)

The only time something happens when you press the [g] [ABS] keys is if you are in 1's or 2's complement mode AND you have a negative number in the X-register. Then the HP-16C will perform a [CHS] operation--taking the 1's or 2's complement of the number as appropriate. No surprises here, right?

The [√x] Key

The only other single-number function that works in Integer Mode is the square root key. To see how this key works,

---

Try This: Press the [DEC] and [f] SET COMPL [UNSGN] keys.

Then set the Word Size to 16 bits. Notice that if you're currently operating with a Word Size of 4, you'll have trouble switching to 16, because 4 bits won't represent the `16 d` you need to key in to switch.

So remember the shortcut to get a 64-bit Word Size: [0] [f] [WSIZE]

Then you can key in [1][6] [f] [WSIZE]

Now, key in the number `65535 d`, the largest integer possible for a Word Size of 16 bits. Then press [9] [√x].

You'll see: `255d`

But, as in the case with division, the C annunciator will come on to indicate that the real answer is *non*-integer.

So just as in division, the [√x] operation will show you the integer portion of the answer and turn on the C annunciator to alert you that this answer is not exact, because the real answer is non-integer.

Summary

Well, that's about all you need to see of integer arithmetic to get you started. Come to think of it, you *have* covered quite a bit of territory:

You know how to manipulate the stack and its arithmetic operators--just the mechanics and keystrokes of crunching numbers;

You know how a computer would do arithmetic (addition and subtraction, at least) and how your HP-16C simulates this for you--following the computer's rules for UNSigned, 1's complement or 2's complement formats;

You also know how the computer would evaluate binary integers expressed in either 1's or 2's complement--and that the HP-16C will do the same for you;

You know that the C and G annunciators appear in the display to tell you what the HP-16C has had to do in following the computer rules for arithmetic;

You now know, after glancing at the opposite page here, that you've managed to get away without a final quiz for this section, but you have a sneaking suspicion that it's because you'll get plenty of practice anyway--anytime you do any arithmetic in Integer Mode on your HP-16C.

LOGIC OPERATIONS ON YOUR HP-16C

Well here you are, about halfway through the book. And you know all about integer arithmetic now--all about how those 1's and 0's can represent numbers and can be combined arithmetically. But did you ever stop to wonder just how any machine (computer or calculator) can actually "obey" any set of rules, do any addtion or any other complicated operation at all.

How does a computer even add 1 and 1?

Your HP-16C is equipped with even more basic, low-level functions than arithmetic; indeed, these are the functions that make machine arithmetic possible. These functions are called "logical operators:" The AND, OR, XOR, and NOT operators.

In a sense, then, you've actually been working backwards--learning about machine integer arithmetic before learning about the logic that makes it possible. Well, now's the time to find that horse and put him in front of your cart....

(Of course, if you're familiar with these four logical operators--and you're on a first-name basis with the Queen--then by all means, jump ahead to page 145).

Logic: The Queen of Science

The word "logic" comes from the ancient Greek word for knowledge. The study of "logic" as a branch of philosophy has been around since the days of the ancient Greek philosopher, Aristotle.

It wasn't until the last century that Logic began to look more like a branch of mathematics. The work of people such as George Boole, Jan Lukaciewicz, and Lewis Carroll, among others, made logic into a set of precise mathematical operations. Before that, logic was more nearly a set of rules that people were supposed to follow when thinking and writing.

George Boole began by developing a system of logic in which everything spoken or written could be considered as either true or false. Then he went on to define different ways in which to combine statements that are true and false. He developed what is called the first "two-valued" system of logic.

Similarly, Lewis Carroll developed a three-valued logic system, and Jan Lukaciewicz (of RPN fame) even went so far as to invent a four-valued system.

But all of these early forms of mathematical logic were mostly in the realm of paper and pencil exercises in thinking. They didn't have much of an impact on the way that people thought, reasoned, or conducted business.

In fact, it wasn't until 100 years after George Boole invented his two-valued logic that the first primitive electronic computers were made. The person who really made the connection between logic and computer science was Claude Shannon. He developed the idea of "switching circuit theory," the notion that electronic switches being ON or OFF could exactly represent Boole's system, where every assertion is either TRUE or FALSE. ("Aha!")

From that simple analogy came the idea of binary math--done by machines....

As you know, in binary math (also called Boolean algebra), there are only two values, 1 and 0. And--more importantly--there are only a finite number of ways in which to combine these values.

To get an idea of all the possible combinations of 1 and 0 (ON and OFF), consider the following "thought experiment:"

Suppose you have a neighbor who is a "hardware hacker." He loves to put together little electronic gadgets (that sometimes actually work). One day, he appears on your doorstep with a box full of his latest inventions. He mumbles something about having put these gadgets together just for fun, but now he has a problem, and he's hoping you could help.

Just then his phone rings, and it's his boss, telling him to be on the next plane to Mesopotamia.

So you're left with this box, and naturally, your curiosity gets the better of you. Inside the box, you find 17 smaller, plastic boxes. There's also is a sheet of paper on which your friend has scribbled something to the effect that each plastic box contains a simple digital circuit. Each has terminals for a 5-volt power source and all but one have contacts for two inputs and one output, the other box having just a single input and a single output contact.

Your friend's note also says that each circuit is different from the others, and his problem is that he can't remember which one does what--because that morning he mixed them all up--by mistakenly dumping them into a bowl (of course, the absence of raisins alerted him immediately).

He wants you to help him re-determine what each of the circuits does.

Well, it's a weekend, and pro wrestling doesn't come on for awhile yet, so you set out to solve this guy's problem for him.

You start by connecting up a 5-volt power supply (a few flashlight batteries connected in series do the job very nicely). Then you rig up a small light bulb on a bypass switch--just something to cause a voltage drop in the circuit--at your option.

The idea is to use two parallel leads to connect this circuit--at either higher or lower voltage--to the mysterious boxes, one box at a time. You can then use your trusty-dusty voltmeter, connected to the output contact, to determine the result of each combination of connections.

After some initial messing about, you come up with the following shorthand scheme for recording your findings (after all, wrestling is only an hour away--no sense wasting time):

You decide to represent a high voltage output reading (+5V) with a 1 and a low voltage (less than +1 volt) as a 0.

And here's what your tally sheet looks like after you test each of the 17 different circuits (being a good digital logician, you have labelled each unknown two-input circuit with a number, starting your count at zero):

Experimental Results for 17 Unknown Digital Circuits

(Four Possible Input Voltage Combinations)

	0 0	0 1	1 0	1 1	
Box #	(Circuit's	Corresp.	Output	Voltage)	Circuit's Apparent I.D.
0	0	0	0	0	???
1	0	0	0	1	AND **
2	0	0	1	0	Inverse of #13 +
3	0	0	1	1	???
4	0	1	0	0	Inverse of #11 +
5	0	1	0	1	???
6	0	1	1	0	eXclusive OR (XOR)**
7	0	1	1	1	inclusive OR (OR)**
8	1	0	0	0	Inverse of OR (NOR)*
9	1	0	0	1	Inverse of #6 (IFF)*
10	1	0	1	0	???
11	1	0	1	1	Converse of #13 +
12	1	1	0	0	???
13	1	1	0	1	IF...THEN +
14	1	1	1	0	Inverse AND (NAND)*
15	1	1	1	1	???

Those are the 16 double-input circuits. Your results for the only single-input circuit are these:

(Two Possible Input Voltages)

	1	0	
Box #	(Circuit's	Output)	Circuit's Apparent I.D.
16	0	1	NOT**

Hmmm...some of these look an awful lot like some of the more common logical operators for binary arithmetic.

So you whip out your HP-16C and start checking its keyboard against the names you've given to some of these circuits. And you place a double asterisk** out in front of each such circuit: AND, OR, and XOR.

Of course, you recognize some others, too--circuits you can buy ready made, shrink-wrapped and everything, from off an electronics store shelf. You mark these with a name and a single asterisk*, or--if you don't know a good name for them--just a +.

Then there are some circuits that seem to do nothing useful at all. You give these the old ??? treatment. After all, they also have no known logical equivalent.

And as for the one single-input circuit, that's an easy one, right? 1 in gives 0 out--and vice versa. It's an inverter circuit, commonly called NOT, and, as you note, it's also on the HP-16C keyboard (must be important, eh?).

So there you are: Ten useful dual-input circuits and your one-input inverter circuit.

You decide to have some more fun--using the four operators on the HP-16C keyboard to emulate the others in your chart....

The first thing to do, then, is to create a set of tables that show the relationship between input and output for these four HP-16C keys.

As you probably know, the HP-16C would combine dual inputs into a single output by starting with the inputs (1 or 0) in the X- and Y- registers.

These tables below show the contents of the Y-register (a 1 or a 0) and the contents of the X-register (another 1 or 0) and then the results that would appear in the X-register (a 1 or 0) upon pressing the different logic operator keys ([OR], [NOT], [AND], and [XOR]).

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

X	NOT(X)
0	1
1	0

OK, this is a nice little warmer-upper, but after all, the idea here is to emulate other logical operators. The HP-16C can certainly do much more with these Logic Functions than just help you to remember their own "truth tables."

Try This: Press f SET COMPL 2'S and DEC 16 f WSIZE, to set the complement mode and Word Size. Also, press 9 SF 3, to display leading zeros (remember that from page 90?).

Now press the BIN key, and key in this binary number:

1000 1000 1000 1000

Press f NOT. You'll see: 0 1 1 1 0 1 1 1.b
Since there's a dot to the left of the b, press f WINDOW 1 to see

the rest of the number: 0 1 1 1 0 1 1 1 b.

So the whole result is: 0111 0111 0111 0111

The calculator has just "NOTted" your number--inverted all its 1's and 0's--bit by bit (as you'll recall, this is the same thing as taking the 1's complement of a number, isn't it? Hmmmm)!

And guess what will happen if you now

Do This: Key in 1010 1010 1010 1010 And press f AND.

Sure enough--the display shows: 0 0 1 0 0 0 1 0.b

And you can press f WINDOW 1 to see 0 0 1 0 0 0 1 0 b.

Entire result: 0010 0010 0010 0010

And realize what's going on here: The result amounts to 16 separate applications of the truth table definition of the AND operator. That is, the first bit of each (dyad) is used as one of the two inputs into your imaginary plastic circuit called "AND." The result is prescribed by the table you prepared on page 140, and this resulting bit becomes the first bit of the result:

```
                 0 1 1 1 0 1 1 1  0 1 1 1 0 1 1 1 b
AND              1 0 1 0 1 0 1 0  1 0 1 0 1 0 1 0 b
-------------------------------------------------
                                                0 b
```

Then the next bit from each dyad is accepted into the AND circuit, and the result is placed alongside the first result bit:

```
                 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 b
AND              1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 b
-------------------------------------------------
                                              1 0 b
```

And so on--for all 16 bits--and all this happens from one press of the [AND] key!

Well, as you might expect, you can perform the same type of operation with the OR and the XOR operators.

And of course, you're not limited to 16 bits for a Word Size; you can give the HP-16C free rein and open it up to its full-64 bit Word Size. If you do that, you'll have 64 separate applications of the logical functions for *each time* you press one of those operator keys!

Testing DeMorgan's Theorems

To get some practice at using and combining the HP-16C's logical operators, consider this:

There are a couple of theorems in digital circuit theory called DeMorgan's Theorems. One of these theorems says that if you put a pair of inputs through an AND "gate" (a circuit that simulates the AND operator electronically) and then invert its output (change 0's to 1's and vice versa), you'll get the same result as if you had inverted the two inputs first and then put the signals through an OR gate.

You can see how this might be a handy theorem to know when you're ransacking your parts bin in search of an OR circuit and all you can find is a NAND ("NOT AND"--a circuit that does an AND operation and then inverts the result): You could apply DeMorgan's theorem and thus save yourself a trip to Radio Shack.

A logician would write out DeMorgan's theorem, using the contents of the HP-16C's X and Y registers in the following way.

NOT(X AND Y) = NOT(X) OR NOT(Y)

...but, if the logician knew something about RPN , he would write this as...

X [ENTER] Y [AND] [NOT] = X [NOT] Y [NOT] [OR]

The task, for now, is to verify DeMorgan's theorem using the HP-16C.

To do this of course, you'd have to come up with all possible dual combinations of 1 and 0, and test each 2-bit combination all the way through the proposition on the previous page--keying all this into the calculator, thus pressing the [1], [0], [ENTER], [AND], [NOT], and [OR] keys several zillion times each.

That's too much work--even on a weekend.

So before you condemn your fingers to slavery, think: "Let's see... all possible combinations of 0 and 1 would look like this:"

0 0
0 1
1 0
1 1

This would imply four separate test situations--one for each pair of inputs. But notice that you could get the HP-16C to test all four pairs at once, if you just re-wrote the pairs on their sides, like this:

0 1 0 1
0 0 1 1

Those are just two ordinary binary numbers, right? And when the HP-16C operates logically on such pairs of numbers, doesn't it actually perform operations on each individual pair of bits? And that's exactly what you want, isn't it?

Yep: To run your test of DeMorgan's theorem, just reduce the Word Size of the HP-16C to 4 bits and operate in binary format (press 4 f WSIZE and BIN).

Next, key in 0 1 0 1 ENTER 0 0 1 1 (of course, you may omit the leading zeros if you wish).

Now "NAND" these two numbers by pressing f AND and f NOT.

The display will show: 1 1 1 0 b

That's the result from doing the operations on the left side of the equals sign in your expression on page 145.

Now do the right side and see if the results match.

Go: Key in 1 0 1 and f NOT, and you'll see: 1 0 1 0 b

Key in 1 1 f NOT. You'll get: 1 1 0 0 b

Finally, press f OR, and see: 1 1 1 0 b

At this point, your result from the left side of the equation is in the Y-register and the right side result is in the X-register. Now, you could either remember what you got earlier or simply press the x≷y key to see it.

But there's actually a more convenient and logical way to determine if two numbers in the X- and Y- registers are the same.

To see how this works, look back at that truth table on page 142.

Look for the [XOR] key, and notice its pattern: If the contents of X and Y are the same, the result is 0; if the contents of X and Y are different, the result is 1.

What does this suggest to you as an easy way to ask the machine to test the equality of what's in the X- and Y- registers?

(Hint: Use the XOR function)

The answer will be yes (i.e. the two binary numbers are equal) only if the result of the [XOR] is 0000 b

---

So do it: Press: [f][XOR]. Result: 0000 b

This proves the theorem.

Taa-daa!

Here's another theorem-proving problem. This one comes from the field of symbolic logic (as opposed to digital logic).

In symbolic logic there's an operator called the Material Implication operator. Its English counterpart is a statement that contains the words IF...THEN...

(For all you programmers reading here, this is *not* the same IF...THEN... with which you are familiar. The Material Implication operator in Logic is just that: an operator. It's a procedure in which two input quantities, called truth values, may be combined to produce a third truth value. Sound familiar?)

The truth table definition for the Material Implication operator is

X	Y	MI
0	0	1
1	0	1
0	1	0
1	1	1

There's no key on the HP-16C for MI, but you can simulate such an operator by an artful combination of the OR and NOT keys:

Here is the symbolic logic way of writing the relationship:

$$Y \rightarrow X \quad \Leftrightarrow \quad \text{NOT(Y) OR X}$$

where the $\rightarrow$ means "implies" and $\Leftrightarrow$ means "is equivalent to".

Of course, an RPNer would write this as $Y \rightarrow X \quad \Leftrightarrow$ Y NOT X OR

and prove it with a final XOR.

You're an RPNer: Key in the truth-table summary of the MI operator (i.e. the left side of the theorem as stated in equivalence notation): 1 1 0 1 ENTER.

Then key in the right side of the statement, 0 0 1 1 f NOT 0 1 0 1 f OR.

Finally, test the equivalence of the two sides. Press f XOR and see: 0.0000 b

---

The test says "yes," the two sides are indeed equivalent; each bit in the X-register is the same as the corresponding bit in the Y-register.

There you have it: an "IF...THEN..." operator--built from the NOT and OR keys!

There's another similar operator in symbolic logic, one called the "biconditional." In saying it out loud, you tend to use the characteristic phrase "IF AND ONLY IF..."

The truth table definition of this biconditional operator is

X	Y	IFF
0	0	1
1	0	0
0	1	0
1	1	1

Notice that the results of the IFF operator are just the inverse of the XOR operator (check page 142 if you're skeptical). So if you ever need an IFF operator on the HP-16C, you can generate it by using f XOR f NOT! How about that?

Actually, you can do a whole lot more with the HP-16C than just verify such trivial little theorems--more than you might think is possible.

You may have noticed, for example, that when you work with just two variables, X and Y, then the resulting truth table has four rows in it.

If you were working with three variables (X, Y, and...Z?), you'd come up with a truth table eight rows deep--because there are eight different possible combinations of three bits (and how do you know this? Because that's what it means to count up to 111 in binary: run through all possible combinations representable in just 3 bits), right?

So you'd have to set the Word Size of the HP-16C to 8 bits.

Similarly, with 4 variables, you would need a Word Size of 16; with 5 variables, it would be 32; with 6 variables, you would reach the limit of the HP-16C--a Word Size of 64 bits.

Now granted, proofs of logical equivalences usually don't involve more than 6 variables. Nevertheless, isn't it reassuring to know that you could help yourself in a symbolic logic course with your handy dandy HP-16C?

Here's one just for the heck of it. Prove (or disprove) the following five-variable logical equivalence:

$$(P \text{ AND } Q) \text{ AND } [R \rightarrow (S \rightarrow T)] \Leftrightarrow (Q \text{ AND } S) \rightarrow T$$

Hmmm...how are you going to do *that?*

First off, you should rewrite this question, making use of some things you've already proven--the facts that Y → X is equivalent to NOT(Y) OR X, and that any equivalency should produce identically zero whenever the two halves of the proposition are XORed together:

[(P AND Q) AND (NOT(R AND S) OR T)] XOR [NOT(Q AND S) OR T] = 0

So set up the following list of all possible combinations of 0 and 1 for the five different variables involved in the argument:

P	Q	R	S	T
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	0	1	0	0
0	0	1	0	1
0	0	1	1	0
0	0	1	1	1
0	1	0	0	0
0	1	0	0	1
0	1	0	1	0
0	1	0	1	1
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	1

(cont.)

P	Q	R	S	T
1	0	0	0	0
1	0	0	0	1
1	0	0	1	0
1	0	0	1	1
1	0	1	0	0
1	0	1	0	1
1	0	1	1	0
1	0	1	1	1
1	1	0	0	0
1	1	0	0	1
1	1	0	1	0
1	1	0	1	1
1	1	1	0	0
1	1	1	0	1
1	1	1	1	0
1	1	1	1	1

(Again, notice that thinking of every possible combination of digits just means counting up to 31--in binary. Remember? That's what the positional numbering system is all about!)

Now turn that list of combinations on its side--just as you did earlier with that simpler, two-valued case:

```
P =  0000 0000 0000 0000 1111 1111 1111 1111
Q =  0000 0000 1111 1111 0000 0000 1111 1111
R =  0000 1111 0000 1111 0000 1111 0000 1111
S =  0011 0011 0011 0011 0011 0011 0011 0011
T =  0101 0101 0101 0101 0101 0101 0101 0101
```

Now you can use the HP-16C's logic operators to run through the logic of the theorem--bit by bit--except that with a Word Size of 32, it does 32 parallel tests at once!

Of course, nobody really wants to key in all those 0's and 1's--even on the HP-16C. So why don't you recall one convenient little fact about hexadecimal notation--that you can simply look at a long binary number and quickly read off the equivalent hex digits (remember that from page 63)?

Thus, you would transform the problem into this:

```
P =  0000 FFFF
Q =  00FF 00FF
R =  0F0F 0F0F
S =  3333 3333
T =  5555 5555
```

Now those numbers are cut down to size, and you can conveniently test the equation at the top of page 152.

Here Goes Nothing: Set the Word Size to 32 bits and press [g] [CF] [3], to suppress leading zeros. Then perform the following:

Keystrokes	Display	Significance
[HEX] [F] [F] [F] [F] [ENTER]	FFFF h	This is P.
[F] [F] [0] [0] [F] [F]	FF00FF h	This is Q.
[f] [AND]	FF h	P AND Q
[F] [0] [F] [0] [F] [0] [F] [ENTER]	F0F0F0F h	This is R.
[3] [3] [3] [3] [3] [3] [3] [3]	33333333 h	This is S.
[f] [AND]	30303033 h	R AND S
[f] [NOT]	FCFCFCFC h	NOT(R AND S)
[5] [5] [5] [5] [5] [5] [5] [5] [ENTER]	55555555 h	This is T.
[f] [OR]	FdFdFdFd h	NOT(R AND S) OR T
[f] [AND]	Fd h	(P AND Q) AND (NOT(R AND S) OR T)

(This is the entire left side of the theorem. Leave it to float in the stack while you build the right side. Then XOR them together to see whether or not you get 0.)

[F] [F] [0] [0] [F] [F] [ENTER]	FF00FF h	This is Q again.
[3] [3] [3] [3] [3] [3] [3] [3]	33333333 h	This is S again.
[f] [AND]	330033 h	Q AND S
[f] [NOT]	FFCCFFCC h	NOT(Q AND S)
[5] [5] [5] [5] [5] [5] [5] [5]	55555555 h	This is T again.
[f] [OR]	FFddFFdd h	NOT(Q AND S) OR T

(This is the entire right side of the theorem, now in the X-register. Since the left side is in the Y-register, just do the final XOR to confirm or disprove the theorem:)

[f] [XOR]	FFddFF20 h

Since the final result is *not* zero, the two sides of the statement are *not* equivalent.

Whew! After that, this shouldn't be any trouble for you at all:

Pop Quiz

1. What is the decimal result of 712(d) AND 444(d)?

2. What is the octal result of 712(o) OR 444(o)?

3. What is the hexadecimal result of 712(h) XOR 444(h)?

4. Harriet "the Pro" Grammar is working on some code for her CrayFish-1 PC. She wants to input 4 octal numbers (with a maximum of 16 bits each) and get out a hexadecimal number. Her intended logic is

   (P AND Q) XOR (R OR S)

   When she uses the numbers P = 67271(o), Q = 73333(o), R = 44505(o), and S = 106120(o), the answer returned by the computer is BAD1 h.

   Is the program performing correctly?

Pop Answers

Keystrokes	Display
1. DEC 712 ENTER	712 d
444	444 d
f AND	136 d
2. OCT 712 ENTER	712 o
444	444 o
f OR	756 o
3. HEX 712 ENTER	712 h
444	444 h
f XOR	356 h
4. OCT 67271 ENTER	67271 o
73333	73333 o
f AND	63231 o
44505 ENTER	44505 o
106120	106120 o
f OR	146525 o
f XOR	125714 o
HEX	AbCC h

(Thus, the PC has *not* been programmed correctly.)

Creating Masks

OK, so those are some of the fundamental logic operators you can create and use on your HP-16C.

But so what?

What good are these operators, anyway?

Well, as you've heard before, you can build all of the arithmetic and information processing of any computer from these simple logic "gates." But how?

A good illustration of this might be the use of logic functions such as AND and OR in the creation and manipulations of "masks."

Masking is a technique for isolating portions of a binary number--often useful when you're using a single binary number to represent multiple pieces of information or instructions.

For example, suppose you were using a printer with your microcomputer--a printer that uses a "parallel interface."

That word, "parallel," means that the computer sends information to the printer 8 bits at a time; and these binary signals travel from the computer to the printer through eight separate, parallel wires.

And suppose, for some strange reason, you wanted to prevent the four high-order (i.e. most significant, or leftmost) bits from going to the printer; you want it to receive only the four low-order bits.

How would you do this?

Of course, you could just cut the wires that carry the high order bits. That would certainly do the job. But those parallel cables aren't exactly cheap and besides, you can't always tell which wires are carrying the high order signals.

Not a very practical solution.

So rather than perform a "hardware fix," suppose you try to accomplish something similar at the logical or "software" level. And sure enough, you can.

All you need do is invent an operation that will "filter" each 8-bit set of signals *before* they go to the printer--i.e. send it through a mask that will "filter out" the high-order bits.

The nice part is, this operation has already been invented; it's just a simple AND! Here's how the result of such an operation might look:

Original Signals	Mask	Processed Signals = (Original AND Mask)
10100000	00001111	00000000
10010001	00001111	00000001
10110010	00001111	00000010
11000011	00001111	00000011
01010100	00001111	00000100
01010101	00001111	00000101
01010110	00001111	00000110
11100111	00001111	00000111
01101000	00001111	00001000
01101001	00001111	00001001
01011010	00001111	00001010
11011011	00001111	00001011
10101100	00001111	00001100
01001101	00001111	00001101
00011110	00001111	00001110
10111111	00001111	00001111

As you can see, everywhere there's a 1 in the Mask, the signals (either 1's or 0's) will get through. But wherever there's a 0 in the mask, the signals will be "blocked" (i.e. only 0's will get through).

Well, it's no coincidence that there are masking functions on the HP-16C. These keys, [MASKR] and [MASKL], will help you to quickly create such patterns of 1's and 0's, so that you can "filter out" the lefthand or righthand parts of binary numbers.

You tell the HP-16C how many 1's you want in the mask by keying in a number. For example, to create the mask in the above example (00001111), you would

---

Do This: Press [DEC] [8] [f] [WSIZE] and [f] SET COMPL [UNSGN].

Next, key in the number 4(d), for the four 1's.

Then, by choosing one of the two different masking functions, you would tell the calculator on which end of the mask you want to put these four ones--right or left.

So press [f] [MASKR].

This will "justify" the 1's (i.e. push them all the way to the edge) on the right of the word (and fill the rest of the word with 0's), thus giving you the mask you used in the previous example (press [f] SHOW [BIN] to see it briefly): 0000 1111 b

---

See how this works? If instead you had pressed [f] [MASKL], this would have justified the pattern of 1's on the *left* end of the word (and filled in the rest of the word with 0's). Thus, you'd have had this mask: 1111 0000 b

And of course, this would have filtered out the low-order bits and kept only the higher ones, if you had ANDed it with any 8-bit word, right?

Try Another: Suppose you're using a Word Size of 16 bits and an unsigned display format. And you've just entered the hexadecimal number EC96, and you want to separate this into two 8-bit numbers. How can you do this?

Keystrokes	Display	Comments
[DEC] [1][6] [f][WSIZE]	??? d	
[f] SET COMPL [UNSGN]	??? d	
[HEX] [E][C][9][6] [ENTER]	EC96 h	
[STO] [0]	EC96 h	Store this in register 0
[8] [f] [MASKR]	FF h	The mask: 0000 0000 1111 1111(b)
[f] [AND]	96 h	
[f] SHOW [BIN]	100 10 110 b	There's your first 8-bit number; you can ignore the 8 0's on the left.
[RCL] [0]	EC96 h	
[8] [f] [MASKL]	FF00 h	
[f] [AND]	EC00 h	
[BIN]	00000000 .b	
[f] [WINDOW] [1]	1110 1100 b.	

And there's your second number. The only problem is, it's still forming the left side of a 16-bit number (of course, the righthand 8 bits are all 0's). So it's not quite an 8-bit number all on its own, yet. You still need to somehow shift or rotate it 8 bits to the right to create the binary number 0000000011101100 (again, forgetting about the high-order 0's).

Well, you can do that with the HP-16C; in fact, it's coming up very shortly. But before you get there, try another masking example to explore some other options you have--and to be sure you have the hang of this idea....

First of all, keep in mind that you can create masks of any size you wish--up to the current Word Size. If you try to create a mask larger than that, the HP-16C won't catch your mistake; it will simply create a mask of all 1's--a mask the same size as the current Word Size. OK?

Your Next Mission: Use the same hexadecimal number from the previous example, EC96, but this time, filter out the eight *middle* bits. Hmmm...

Keystrokes	Display	Comments
[HEX] [E][C][9][6] [ENTER]	EC96 h	
[4] [f] [MASKR]	F h	Create a mask of 4 bits on the right.
[4] [f] [MASKL]	F000 h	Create a mask of 4 bits on the left.
[f] [OR]	F00F h	Combine them (see how useful OR can be, too?).
[f] [NOT]	FF0 h	This is the mask you want.
[f] [AND]	C90 h	And this is the final result you want --the middle eight bits of EC96 h.

The trick here was to create two masks: one on the right of the word and the other on the left. Then you can use the [OR] key to combine these two masks into one. Finally you use the [NOT] key to invert all the bits in the mask--so you get the 1's in the middle instead of the 0's.

Of course, if you had stopped to think about it for a minute, you could have figured out what that "middle mask" must look like--and you could have simply keyed it in: [F][F][0]. But this method is a good one to know when it's not so easy to envision how a mask should look.

Innocent-Looking Little Quiz Questions
(Cleverly Masked)

1. Using a Word Size of 32 bits, and the hex number FC998E22, separate out the Most Significant Bit and the 15 bits on the right of the number.

2. With the same Word Size, filter out the bits numbered 29 through 21, inclusive, from the hex number ABCDEF12 (reminder: bits are numbered from 0, starting with the least significant bit on the right. So with a 32-bit word, the MSB would be numbered 31).

The Awful Truths Revealed

Keystrokes	Display	Comments
1. [DEC] [3][2] [f] [WSIZE] [HEX]		
[F][C][9][9][8][E][2][2] [ENTER] [ENTER]	FC998E22 h	
[1] [f] [MASKL]	80000000 h	
[f] [AND]	80000000 h	
[x≷y]	FC998E22 h	
[F] [f] [MASKR]	7FFF h	[F] is used because F(h)=15(d)
[f] [AND]	E22 h	
2. [A][B][C][D][E][F][1][2] [ENTER]	AbCdEF 12 h	
[2] [f] [MASKL]	C0000000 h	
[DEC] [2][0] [HEX]	14 h	
[f] [MASKR]	FFFFF h	
[f] [OR]	C0FFFFF h	
[f] [NOT]	3FF00000 h	
[f] [AND]	2bC00000 h	
[BIN]	00000000 .b	
[f] [WINDOW] [1]	00000000 .b.	
[f] [WINDOW] [2]	11000000 .b.	
[f] [WINDOW] [3]	10 10 11 b.	
[HEX]	2bC00000 h	

Bit-Twiddling Functions on the HP-16C

As you saw when masking out parts of binary numbers, in order to arrive at a complete answer, you need to find some way to rotate or otherwise shift a pattern of bits--from where it now sits--all the way to the right.

On other occasions, you'll find that you'd like to go the other way--to the left. And so on. In fact, "bit twiddling," along with "byte pushing," are two favorite pastimes of computer scientists.

Well, as you heard in that rumor back on page 161, the HP-16C does indeed have a whole slew of handy functions sufficient to let you pursue those pastimes to your heart's content. With the HP-16C in hand, you can shift bits right and left, rotate bits clockwise and counter-clockwise, and even left-justify a pattern of bits within a number.

Here's a sampling of the operations you can perform:

Arithmetic Shift Right

A good place to start in all this bit-twiddling is with an Arithmetic Shift to the Right (ASR for short).

To illustrate, pick a number, say 13, and write it in (8-bit) binary: 00001101 (b)

Now shift every digit one place to the right--and put a zero in the vacant place of the MSB. The result would be 00000110 (b)

which is the binary representation of 6 (and this is more or less one-half the original number, isn't it?).

Right away, you should notice this: A Shift is *not* a Rotate; the 1 sitting in the LSB prior to the Shift did *not* "wrap around" to become the MSB afterwards. On the contrary: It's just plain gone--bumped off the right-hand end, just as the T-value is blown off the top of the stack when you press ENTER.

Notice also that the process shifting all bits one to the right--and putting a 0 into the MSB--is merely halving the original number (and this makes sense: In base ten, you shift all digits to the right when you divide by ten. So in base 2, when you shift all digits to the right, this must be what you're doing--dividing by 2)! Of course, in halving the original number you would get only the integer part of the result. After all, this is Integer Mode.

Something else to keep in mind: You weren't messing with a negative number here, but in an Arithmetic Shift Right, the sign bit is *copied* to the right,*not* shifted. This means that a positive number would remain positive and a negative number negative (by contrast, when the MSB *is* shifted along with the other bits, the process is called a "logical shift").

Here are a couple of examples you can try on the HP-16C to catch the flavor of the ASR function:

First Taste: Begin by pressing the 8 f WSIZE keys to set a Word Size of 8 bits.

Then press DEC and f SET COMPL 2'S.

Now key in a negative number, say -127 (1 2 7 CHS).

To remind yourself how this looks in binary, press f BIN and see:

10000001 b

Now press the g ASR keys, and you'll see: -64 d

You'll also note (with mounting panic) that the C is on.

Use f BIN to view the binary equivalent of this number:

11000000 b

Sure enough, all of the binary digits have been shifted to the right--except the sign bit (the MSB), which has been copied to the right. And the original LSB has been shoved off the right side of the word.

But why did the C annunciator rear its ugly head? Thought you were all through with it, didn't you?

Actually, it's behaving quite properly and consistently. Read on....

Imagine a small "bit-bucket" to the right of the X-register inside the calculator. Here's how it looks before [ASR]:

Whenever you shift a digit off the end of the number in the X-register, it falls into this bit-bucket. If this discarded bit is a 1, this turns on the C annunciator (which is the same as saying that it sets flag 4); if it's a 0, this turns the C off. Here's the picture after you press [g] [ASR]:

This makes some kind of sense, right? That C appears during arithmetic when a 1 is being Carried or Borrowed at the end of the number. Here, it comes on when a 1 is being *bumped off* the end of the number.

Press the [g] [ASR] keys again, and see: -32 d. Here's the picture:

Notice that the C annunciator has now been shut off. Now press [f] [BIN] to see 1 1 100000 b. Notice that the bits have been shifted to the right once more--and, as usual, the sign bit remains the same; it was copied rather than shifted.

If you continue to press [g] [ASR], to keep on halving the number in the X-register, the calculator will let you do this until you work your way down to -1 d. From there on, any pressing of the [g] [ASR] keys will just leave -1 d in the display (with the C annunciator on).

Do you see why? If you're not sure, watch the display in [BIN] mode.

Logical Shifts

As you heard, the other type of bit shifting available on the HP-16C is the logical shift. In this case, the calculator doesn't preserve the MSB; rather this sign bit is shifted right along there with the rest of the bits--no special treatment.

Logical Shift Right: The [SR] Key

Learn By Doing: Assuming you still have your HP-16C set to an 8-bit Word Size and 2's complement mode, key in the number -127(d).

Press the [f] SHOW [BIN] keys to see that the binary equivalent is

10000001 b

Now press the [f] [SR] and see 64 d.

The C annunciator will also turn on.

Once again, press the [f] SHOW [BIN] to see 1000000 b

The original number has been shifted to the right; the LSB, a 1, has fallen into the bit bucket on the right; the MSB has been replaced by a 0; and since the sign bit is now a 0, the number in the decimal display is now positive.

Even so, the Shift Right has an easy-to-remember significance: It's the same as taking one half the *absolute value* of the original number.

Logical Shift Left: The [SL] Key

Hmmm....If a logical shift to the right causes the original number to be halved, then perhaps a shift to the left will double the original number.

Check It Out: Key in [6][3] and press [f][SL]. Sure enough, the result is 126 d --double the original decimal number.

Try it again: [f][SL] You'll see -4 d
Now, this may not seem to be twice the number, but if you press [f] SET COMPL [UNSGN] to set the format to UNSigned, you'll then see 252 d in the display (then shift back to 2's complement).

In the 2's complement case, the original number, 126(d) or 01111110(b), has been shifted to the left, becoming 11111100(b).

Now there's a 1 in the sign bit, so the display register interprets this to mean a negative decimal number.

And you can see that the empty Least Significant Bit holds a zero--which is what's always used to fill in the blank on the end of a shifted word.

Press [f][SL] again. You'll see -8 d , *and* the C annunciator will be on.

That bit bucket has swung around to the left side of the display register now--to catch any 1's or 0's that are shifted out of bounds. In this case, a 1 was bumped off of that left end and into the bucket--thus triggering the C.

Left Justification: An Oddball Function

The left justification function (g LJ) takes a binary number in the X-register and repeatedly shifts it to the left until there is a 1 in the Most Significant Bit.

Not too tough to comprehend, right? But what's it for?

Try It: Key a 1 into the X-register.

Place a finger over the display, so that you can't see it.

Now press the g LJ keys. What do you think you should see?

You'd expect that by starting out with `0000001 b`

you'd wind up with `10000000 b` i.e. `-128 d`

Now lift your finger and behold the result: `8 d`

This `8` represents the number of left shifts that it took to justify the number. *The actual result of the left justification is in the Y-register.*

So press x≷y, and see `-128 d`.

The utility of this function may not be obvious, but consider this: How would you go about converting any large integer into scientific notation (i.e. with a mantissa and an exponent)? You would need to left-justify the significant digits (they become your mantissa) and then record how many of those leading zeroes you stripped off. That number would then become your exponent!

Rotating Bits

Now here's the way you can finally isolate all those pieces of binary numbers you were cutting out back on page 161: You rotate the bits.

On the HP-16C there are eight different functions that will let you do this. That is, instead of pushing the end bit off into oblivion, this bit will be recycled back to the other end of the word. Sometimes it has to take a turn in the bit bucket first; sometimes it doesn't.

Four of these functions are Rotate Right and Rotate Left, Rotate Right through the Carry Bit and Rotate Left through the Carry Bit. Each of those causes a rotation of only one bit position. The other four functions are similar, except that they allow for any number ("n") of rotations at once.

Before getting underway in this discussion, take a moment to prepare your calculator so that you'll get the results you see here.

Synchronize Watches: Press [DEC] [8] [f] [WSIZE] to get an 8-bit Word Size.

Then press [BIN] and [9] [SF] [3], to set flag 3, thus showing the leading zeros in binary display format.

Now, to picture what's happening, you'll need to start thinking in circles (if you're not already doing that by now).

For instance, here is a picture of eight bits in a byte. The bits are arranged above the numbers denoting their respective positions in the word.

0 0 0 0 1 1 1 1 b
7 6 5 4 3 2 1 0

Of course, you would key this arrangement into the calculator as: 1111.

Go ahead and do that now.

Rotating to the Right...

Now press the f RR keys; the pattern becomes this:

1 0 0 0 0 1 1 1 b
7 6 5 4 3 2 1 0

It's simply as if the bit pattern had taken one step to the right in your display--and the bit on the far right had then run around to the left end to become the Most Significant Bit.

And, as you'll also see in the display, the C annunciator is on--because this "recycled" bit bumped off the righthand end was a 1. The rules here are similar to those for shifting: Anytime it's a 1 that's bumped off the end, the C turns on; anytime this bit is a 0, the C turns off. Easy, right?

...And Rotating To The Left

Rotating Left is just the reverse process of Rotating Right.

Try It: First, press g LSTX to restore the original pattern of bits (once again, that LSTX key shows its usefulness): 0000 1111 b

Now press the f RL keys.

Any surprises here? Not really: 0001 1110 b

Notice that, in this case, a 0 was the bit pushed off the (left) end of the word. It then ran around to righthand end, thus suffering the ultimate demotion--going from Most Significant Bit to Least Significant Bit. And because it was a *zero,* its bumping off was noted by turning *off* the C annunciator (how utterly humiliating).

Rotating a Number of Bits at Once

As you heard, there are keys that will let you rotate a binary number by "n" bits at a time. Well, it's true: These keys are [RRn] and [RLn].

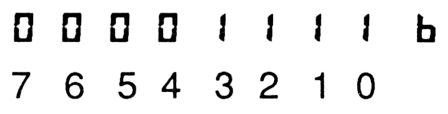
Give It A Spin: Rotate the number 00001111(b) to the left by 3 bits.

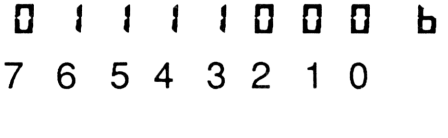
Here's How: First, key in [1][1][1][1] [ENTER] [1][1].

What are you doing here? You're keying in the number you wish to rotate (ignoring the superfluous leading zeroes, of course), then pressing [ENTER].

Next, you're keying in the number of bit positions you want to rotate it (remember: 11(b) is 3(d), right?). Of course, you also could have done this by using [1][1][1][1] [DEC] [3].

Now it's just a matter of choosing whether you want to rotate right or left ([f] [RRn] for right; [f] [RLn] for left). You choose left (since that was the assignment). Here is the display as it looks Before and After you press [f] [RLn]:





The C annunciator should be *off*, since the *last* bit to pass the boundary is a 0 (but if this bit had been a 1, then the C annunciator would now be turned on).

Rotating Through the Carry Bit

This is another kind of rotation--not too tough to grasp, either:

First of all, let's all admit it: The bit bucket is really the Carry bit in disguise; after all, it turns on the C annunciator when a 1 pops off the end--just as the C is supposed to do when you carry a 1 "off the end" in binary arithmetic.

So the idea of rotating through the carry bit is this: It's just plain old rotation, except that there's one extra position in this game of musical chairs--the bit bucket. When a bit pops off the end, it doesn't run around to the other end; it goes into the bit bucket. Then, on the next rotation, it "recycles" to the other end, as usual.

And of course, the usual rules apply as to when the C turns on: when the new arrival into the bit bucket is a 1, on comes the C; if it's a 0, the C goes off.

To Wit: Press [g] [CF] [4] to get rid of the C annunciator, if necessary. Then key in [1][1][1][1].

Then press [g] [RRC] (Rotate Right through the Carry Bit).

Before:	0	0	0	0	1	1	1	1	0
	7	6	5	4	3	2	1	0	C
After:	0	0	0	0	0	1	1	1	1
	7	6	5	4	3	2	1	0	C

(As you might guess, if you want to Rotate through the Carry Bit by a number of bits at once, just key in the original binary number, press the [ENTER] key, then key in the number of bits you want to rotate, and press either [g] [RRCn] or [g] [RLCn].)

Startling Pedagogical Device

1. What is the octal result of *shifting* (*not* rotating) 96(d) to the left by 6 bits? Use a Word Size of 16 and the 2's complement format.

2. What is the decimal result of rotating 219(d) by 15 places to the right? Try regular rotation and then rotation through the carry bit. Then show that you can obtain the same result by two logical shifts to the left.

3. What is the result of left justifying 219(d)? View the number, before and after left justification, in both binary and hexadecimal format.

4. What is the binary representation of 11110000(b) rotated left by 5 bits?

5. A rare astrophysical phenomenon occurred last night: Its only apparent effect was to limit every HP-16C on earth to an 8-bit Word Size.

   The problem is, first thing you need to do this morning (after brushing your teeth) is to rotate the binary number 0001 0101 1110 1001 to the left.

   Can you still do this?

   (This problem is similar to the one in the HP-16C Owner's Handbook, p.49.)

Dramatic Conclusions

Keystrokes	Display	Comments
1. DEC 16 f WSIZE		
f SET COMPL 2'S		
96	96 d	
f SL	192 d	
f SL	384 d	
f SL	768 d	
f SL	1536 d	
f SL	3072 d	
f SL	6 144 d	
f SHOW OCT	14000 o	
2. 219 ENTER	219 d	
15	15 d	
f RRn	438 d	(C is off)
219 ENTER	219 d	
15	15 d	
9 RRCn	876 d	
219	219 d	
f SL	438 d	
f SL	876 d	

Keystrokes	Display	Comments
3. 219	219 d	
f SHOW HEX	db h	
f SHOW BIN	11011011 b	
g LJ	8 d	The number of left shifts needed to justify the number.
x≷y	-9472 d	
f SHOW HEX	db00 h	
BIN	00000000 .b	
f WINDOW 1	11011011 b.	
4. 11110000 ENTER	11110000 b	
101	101 b	(= 5(d))
f RLn	00000000 .b	
f WINDOW 1	00011110 b.	

Keystrokes	Display	Comments
5. 1000 f WSIZE	1000 b	Set the Word Size to 8 bits (1000 b)
1010 1	00010101 b	
f SL	101010 b	Key in the left half of the number
g LSTX	10101 b	Use LST X to recover the original number.
11010001	11101001 b	Key in the right half of the number, and rotate it left through the Carry bit.
g RLC	11010010 b	with the C annunciator on.
X≷y	10101 b	Swap the contents of the X- and Y-registers.
g RLC	101011 b	Rotate the number in X to the left through the carry bit.

Now you can view the right half (low-order part) of the rotated number in the Y-register: 1 10 100 10b

And the left half in the X-register: 1 10 10 1

This means, of course, that you could use this same technique to rotate a 128-bit number(!) on the HP-16C--thus circumventing its 64-bit maximum Word Size.

That last problem illustrates the difficulty of working with binary digits. Keeping all the 1's and 0's straight can be a royal pain in the drain.

That's why many people prefer to work with the hexadecimal equivalents of the binary numbers. These hex digits are much easier on the eyes and give you far fewer chances for keystroke errors.

For example, this same problem (#5), solved in Hex, would reduce down to this:

"**Prove** that, although limited to 8-bit words, you can rotate 15E9 (h) to the left, to yield 2BD2 (h)."

(Remember: Limit yourself to a Word Size of 8 bits)

Keystrokes	**Display**
HEX 1 5	15 h
f SL	2A h
g LSTX	15 h
E 9	E9 h
g RLC	d2 h
X≷y	15 h
g RLC	2b h

Your result: 2b h in the X-register is the high-order (lefthand) byte and d2 h (in the Y-register) is the low-order (righthand) byte.

A lot easier, right?

Bit Surgery: Setting, Clearing and Summing

By now, you're probably getting the idea (and rightly so) that you can do just about anything you want to the bits of a binary number as it sits there, unsuspecting, in the X-register.

Yep, that's right--here are a few more tricks to add to your list:

On the HP-16C you can test, set or clear any of the bits in a binary number--just by specifying them by number (remember: each bit's position is numbered, starting with 0 on the far right).

To set one particular bit in any given number, for example, you must specify two things (very similar, in fact, to the way you use [RLCn] or [RRCn]): Of course, you need to give the number itself (and press [ENTER] to "stack" it up above), but also you must key in the position of the bit you want to adjust.

Try It:

Keystrokes	**Observations**
[DEC] [8] [f] [WSIZE]	Set the Word Size to 8.
[BIN] [1] [0] [0] [0] [0] [0] [0] [1] [ENTER]	1000000 1 b
[1] [0] [1]	101 b
[f] [SB]	10 10000 1 b

Thus, the sixth bit from the right (bit number 5) is set equal to 1.

Similarly you may clear an individual bit in a binary number by using the same procedure as above--but you would instead use the [f] [CB] keys. Note that if a bit is already set, setting it again will have no effect. The same is true for clearing a bit.

And there's yet another function on the HP-16C, the [#B] function, which calculates the *sum of the bits* in a binary number.

Seeing Is Believing: Key in ([BIN]) [1][0][0][0][0][0][0][1]

Now press [9] [#B]. This calculates how many bits in the number are set to 1.

And the HP-16C will show you this number, #B, in the display:

000000 10 b

(Remember, that's 2 d)

You can see how this might be useful in certain kinds of computer operations, called "checksums," where, for example, the computer wants to verify the integrity of some data it just received from a terminal.

It does so by adding up the bit sums of all data bytes (words) that were transmitted. If this number matches the checksum sent at the close of the transmission by the terminal, then it's very likely that the computer has received exactly what the terminal sent.

Here's a problem that will let you practice using the [SB] and [CB] operators and manipulating bits (after all, what else *is* there in life, anyway?).

Burning Question: How do you convert 7(d) to -7(d) (in 2's complement format) the old-fashioned way--without using the [CHS] key?

Glaring Solution: Use the Set- and Clear- Bit functions.

You Press	You See	You Remark to Yourself
[DEC] [4] [f] [WSIZE]		Set the Word Size to 4 bits (you need three bits for the value and one for the sign, right?).
[f] SET COMPL [2'S]		Set 2's complement format.
[7] [ENTER]	7 d	
[f] SHOW [BIN]	111 b	
[1] [f] [CB]	5 d	
[2] [f] [CB]	1 d	
[3] [f] [SB]	-7 d	
[f] SHOW [BIN]	1001 b	

And there you have it: -7(d), the hard way.

Really, the whole trick here is in knowing your complement formats well enough to envision how -7(d) is expressed in 2's complement.

Then you start hacking away, changing one bit at a time until the result looks right--not a very elegant way to do things, but it works.

So Short That it Barely Qualifies as a Pop Quiz

1. With a Word Size of 8 bits and in 2's complement decimal format, what is the effect of setting the sign bit for 127(d)?

2. With the same configuration, what is the effect of using f ASR with - 1 d showing in the display?

Nevertheless...

1. This changes the number to -1 (d). Remember the old tape-counter analogy?

2. The display still shows - 1 d , but with the C annunciator on (diagram this for yourself, if you're not sure why).

Notes and Doodles

Borrowing and Carrying: Looking Back and Moving Forward

Take 101(b): Stop and take a look at how much you've put under your belt by now (assuming you've taken the time to properly digest all this, of course):

You started out learning the rudiments of the registers, the keyboard, the stack and the mechanics of arithmetic--all using Floating Point Mode.

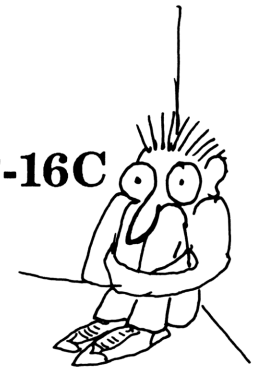
Then you shifted gears and headed straight into Integer Mode. You first saw how the display does most of the interpreting work for you; then you saw how to perform integer arithmetic--and all the clues and messages and rules being given by the display and its annunciators (especially that blasted little C).

Then you actually went within the arithmetic process to learn about logic functions (your neighbor's mixed-up box of circuits) and how they could be used to build progressively more complex logic functions. Remember how you proved and disproved logic propositions--and how to use masks?

Then you totally forgot about your TV wrestling and went down in scale even farther, to see how you could use the HP-16C to actually, surgically alter the very bits making up those masks and truth tables ("...yes, folks, that's right--it slices! It dices! It shifts, sets, clears, rotates, sums, truncates, folds, spindles, mutilates....")

"But wait! --There's more!"....

MEMORY MANAGEMENT ON YOUR HP-16C



As you've been learning more and more about bits and words--and how the HP-16C helps you make sense of them, you haven't really needed to figure out how to store very many of them for later use.

Of course, you do know how to do this--with the STO and RCL keys you saw back on pages 42-44.

But there are some fine points you haven't yet considered.

How Many Registers?

In the first part of this book, when you were using the HP-16C as a Floating Point calculator, you saw that it had 29 registers available for storing numbers.

And that's true: There's no way, in Floating Point Mode, that the HP-16C can have any more; it's always 29.

However, in Integer Mode, it's a different story: The HP-16C will allow you to divide up this memory into different-sized units, thus letting you create more or fewer than 29 registers.

If you think about that for a minute, it makes a lot of sense: For example, if you're using a 4-bit Word Size, each binary number you're storing should take up a lot less room than a 64-bit word would; so it would be very handy to be able to make sixteen little 4-bit registers out of every 64-bit register--thus letting you store a whole lot more of the smaller numbers.

To understand how you can do this kind of memory management on the HP-16C, it's best to begin by resetting the calculator.

Now, don't take that wrongly: It's *not* a good idea to reset your machine every time you want to horse around with its memory. In fact, it's a very *rare* occasion when you want to use this--because it literally wipes the slate clean of any data or programs you're currently storing in the calculator.

Nevertheless, you'd better do it here, so that you've seen it once--and so you're assured of getting the same answers as you see here. OK?

OK: Turn the calculator off.

Now press and *hold down* the [ON] key.

While holding down the [ON] key, press and *hold down* the [−] key.

Now release the [ON] key--then the [−] key. You'll see `Pr Error` in the display.

Press any key and the display will show: `0 h`

Press the [f] [STATUS] keys (remember the machine status and how to read this? See page 92 if you don't): `2- 16-0000`

What have you done? You've cleared all data registers and reset the machine status to a default or "startup" configuration. And what is that configuration?

Just read it from the status and the display: The HP-16C is in hex display format, with a Word Size of 16 bits, all flags cleared, and 2's complement format.

Now Try This: Press the f MEM keys, and you'll see: P-0 r-101

The MEM key is letting you know about the status of memory in the calculator.

The P-0 says there are currently 0 blank lines reserved for storing Programs ("program memory").

The r-101 means there are 101 Registers available for storing integer numbers ("data memory").

That's a lot more data registers than were available in Floating Point Mode. Where did they come from?

The answer to that question is that subdividing flexibility you were just considering (i.e. 4-bit vs. 64-bit "boxes"). And you have this because you're in Integer Mode--and because you're allowed to adjust that all-important Word Size.

Here are the gory details....

Data Memory Allocation

Your HP-16C has 1624 bits of memory available for you to use as numbered data registers (the stack and display registers--and the I-register--are reserved separately).

When you were operating in Floating Point Mode, the HP-16C needed 56 bits for each Floating Point Number you stored (quite a lot, really).

So the calculator simply divided its 1624 bits into parcels of 56 bits each, thus giving you 29 numbered data registers to use (0 - F and .0 - .C, remember?).

But here you are in Integer Mode. And, since you just reset the machine, you're now operating with the default settings for the calculator: You have a 16-bit Word Size.

Question: How many numbered data registers are available to you with this 16-bit Word Size?

Answer: You have 101 numbered data registers.

Says Who: Numbers don't lie--and the HP-16C does its own arithmetic:

(1624 bits) / (16 bits per register) = 101 whole registers.

Notice that it cannot use any remaining bits as any kind of "partial register." That's a no-no; after all, how could you ever confidently use such a register when it couldn't hold an entire number under your desired Word Size?

So if you want more data registers, you know how to get them, right? Just reduce your Word Size!

For example, a Word Size of 8 bits should give you 1624 / 8, or 203 whole registers for data storage.

See For Yourself: Just press [DEC] [8] [f] [WSIZE] to change the Word Size.

Then press the [f] [MEM] keys to see: P-0 r-203

Actually, there is a limit to the amount of tedium your HP-16C will put up with: It absotively will not give you any more than 406 registers (this is the number of 4-bit registers that fit into 1624 bits).

So you can specify Word Sizes as small as you like, but the machine will only give you 406 registers; the rest of those bits will be wasted. And thus, it's not very memory-efficient to specify a Word Size of less than 4 bits.

The underlying reason for that is this: The calculator won't partition bits except in even multiples of 4. So if you specify a Word Size of, say, 13 or 14, you're going to get a memory partitioned in sets of 16 bits (and thus, for a 1- to 3- bit Word Size, it partitions to 4 bits).

Program Memory Allocation

Well, that's fine--as far as it goes: You've explained away exactly half of the message you saw when you pressed f MEM. You now know what r-203 means--and how you can change it.

What about the other half of the message? What does P-0 mean?

It means you don't have any of your 1624 bits currently dedicated to storing programs. And how do you dedicate some? You do it simply by keying in a program.

In a way, that's unfortunate--because it means you have to key in a program here--without having spent any time learning what a program is or how to write one.

Alas, then, in the interests of learning about program memory allocation, you're about to have another non-fatal attack of BPS (think of it as a vaccination with a weakened strain):

This program is a set of instructions that, when executed, will configure the calculator to a Word Size of 16 bits and hexadecimal integer format.

Remember: If you make a mistake, you can press the [BSP] key to erase that step in the program; then just try it again.

Keystroke	Display	Vapid Commentary
[g] [P/R]	000-	You have just entered the program zone. The PRGM annunciator dutifully appears in the display.
[g] [LBL] [A]	001- 43,22, A	The first line of the program names it with the label "A."
[HEX]	002- 23	The second line sets hexadecimal display format.
[1]	003- 1	These two lines enter the number
[0]	004- 0	10(h), or 16(d), into the X-register.
[f] [WSIZE]	005- 42 44	The fifth line sets the Word Size: 16 bits.
[g] [P/R]	0 h	You're now back out of "Program Mode" (i.e. back into "Run Mode").

See how easy it is to program the HP-16C?

Basically, all you're doing is recording a sequence of keystrokes. The HP-16C is your recorder; the [P/R] key is the RECORD button.

OK, so much for BPS. The whole reason for keying in this little routine was to ask this question:

How many "potential program lines" are now left from the 203 you had before you keyed in the program? See if you can figure this--and predict what message you'll see when you use the MEM key.

Here's a Big Hint:

HINT: The HP-16C assumes, by default, that you want all available memory dedicated to data storage. That's why you get 0 program lines and 101 data registers configured when you reset the calculator.

However, when you demand some memory to hold a program (and you just demanded it--by keying in something while in PRGM Mode), the HP-16C *converts some of that data memory into program memory--one Floating Point register (56 bits) at a time.*

And a Small Hint:

hint: Each line of program code requires 8 bits.

"Aha!" (*Now* can you tell what you would see upon pressing f MEM?)

Here's the way you might reason it:

"When I started to write this first program, there was no memory allocated for it. But at the instant I pressed the first key in program mode, the HP-16C sensed that I now needed some program memory. So it gave me 56 bits of memory from its startup supply of data memory (1624 bits).

"Now, at 8 bits per program line, each 56-bit sacrificed data register should be enough to contain 7 entire program lines...and I used...mmm...5 lines....So, I should have room for 2 more lines in my program before I force the calculator to cannibalize another data register.

"And...let's see, here...since the HP-16C subtracted the 56 bits of program memory from the original pool of 1624 bits of data memory, I should be left with 1624 - 56, or 1568 bits of data memory. And 1568 bits divided by 8 bits should give me 196 data registers now!"

All right, that all sounds very reasonable and logical.

But is it correct?

The Acid Test: Press f MEM (and hold down the MEM key to get a longer look--just like the SHOW BIN or SHOW HEX keys)

Nice going: P-2 r- 196

So remember: P-2 says "2 lines of Program memory available;" and r- 196 says "196 data Registers available."

Now then, Mr. Holmes:

Mystery: What will happen to your calculator's staus and memory configuration when you run this little program you keyed in?

Deduction: Run the program and see what happens (press [GSB] [A], and you'll see the word "running" blinking on and off in the display).

The program will end and you'll see: 0 h

Now press [f] [STATUS], and the display will show you the current status of the calculator: 2- 16-0000

Sure enough: the HP-16C has dutifully performed the steps you recorded in the program. You can tell this immediately because some of those recorded keystrokes were [HEX] [1] [0] [f] [WSIZE], which, of course, specifies a change in Word Size to 10(h) or 16(d).

You recorded those steps earlier--and the HP-16C obeys them now when you play them back (and it will do so again and again, at your command). So now there are no more 8-bit words. And no more 8-bit registers. Everything in the world is now 16-bit.

Then you better look at what happened to your memory....

Press [f] [MEM] and see: P-2 r-098

You still have 2 lines of program memory left--as before. But you started out with 196 8-bit registers of data memory; now you have 98 16-bit data registers. See? It's the same number of bits--it's just partitioned differently.

Test this logic once more: Press 4 f WSIZE.

Next, press the f STATUS keys to confirm the change in Word Size:

2-04-0000

Now press f MEM, and see P-2 r-392

Does the math check? Sure enough: Instead of 98 registers of 16 bits each, you have 392 registers of 4 bits each.

So you're convinced.

Great. Might as well set it back to 16-bit words, though--that's much more common and more useful. And why not use your program to do this? (Rumor has it that it's very good at doing this.)

So press GSB A, to run the program, thus resetting the Word Size to 16 bits.

And double check it for good measure:

Press f STATUS, and see.... 2-01-0000 (!!)

("Mayday!...Mayday!...Mayd-...")

Something's wrong...the program didn't work....

Why not? Well, when you ran the program this time, you started out with a Word Size of 4 bits, right?

And some of the instructions in the program were the keystrokes 1 0, intended to make the machine "key in" (to itself) the number 10(h). Well, it sure tried to honor your request, but it ran into a problem: The largest number representable in 4 bits is 15(d), or F(h). But you asked for 10(h), which is 16(d).

No can do.

But, as always, the HP-16C does what it can--it *can* "key into itself" the *first* of those two digits: 1 So it does. It took the 1 in line 003 of the program, but it *ignored* the 0 in line 004--just as it would do to you if you tried to enter 10(h) directly on the keyboard under these 4-bit circumstances.

So the net result was, you didn't set the Word Size to 10(h) (16 in decimal); you set it to 1(h) (which is, of course, also 1 in decimal).

You ended up with a Word Size of 1 bit. Not so good.

But, as they say in the computer business: "No problem!" Just a little more careful thinking and planning for such contingencies will correct the program.

To be sure it will be able to reset the Word Size to 16 *every* time--no matter what the current Word Size--maybe you should start by setting the Word Size to 64 bits--and then set it to 16 bits. That should work--because you know the trick about specifying a 0-bit Word Size to get 64 bits instead, right?

Before you turn the page, see if you can visualize for yourself just how to correct the program.... Got it? OK. Check yourself....

Keystrokes	Display	Comments
g P/R	000-	Go back into Program Mode. The PRGM annunciator will appear.
SST	001- 43,22, A	Label A, remember?
SST	002- 23	Sets the calculator to hexadecimal mode.
0	003- 0	Insert a new step now. The calculator will key a 0 into itself.
f WSIZE	004- 42 44	Set the Word Size to 64 bits.
g P/R		

And that's all you need to do; the rest of the program lines are correct as is. All you did here was to insert two extra lines. The calculator has handled the chore of retaining all the original lines of the program and renumbering them. And you're now back out of Program Mode and into Run Mode.

Press the f MEM keys and you'll see: P-0 r-392

Because you added 2 additional lines, there are no more spare lines (8 bits each) left from that block of 56 bits you appropriated (but don't panic: if you want more program memory, the calculator will give it to you--in the form of *another* converted 56 bits--upon your very next entry of an additional program line).

Now test your revised program. First, note the current status, by pressing f STATUS. You'll see the ruins of your first program's run: 2-0 1-0000

Now, if your fix is correct, the program should be able to get back to a 16-bit Word Size even from this current 1-bit Size.

Go for it: Press GSB A and wait a moment for the short program to complete.

Now the moment of truth: Press f STATUS: 2- 16-0000 How about that?

What To Keep In Mind As You Go On From Here

Here are the main points to remember from this little session on memory management....

1. The HP-16C has 1624 bits of memory that you can use.

2. You can adjust the Word Size and thereby adjust the *number of data registers.*

3. When you start to record a program in the HP-16C, it begins to borrow blocks of 56 bits from data memory--taking them as you require them for program memory.

4. Writing programs in the HP-16C is easy--but getting them to run correctly is no easier than on any other computer. You gotta think it all the way through (and you'll get plenty of chances to practice here in a little while).

Addressing Memory Locations

Earlier in this course (while you were learning about Floating Point Mode), you saw how the data storage registers are "named" with numbers. The first available data register has the "address" of 0 (zero); the next register is addressed as 1, and so on.

Of course, just because the data registers are numbered consecutively doesn't mean you have to *fill* them consecutively when you store data in them. You can use them in any order that you want, but you have to keep track--on your own--as to what goes where.

But here's something you haven't run across before: Only 32 of these data registers may be referred to by number. That is, you may address only 32 data registers *directly.*

Recall that the addresses of the data registers run from 0 to 9 and from A to F. That takes care of the first 16 registers. Then, to address the last 16 registers, you use .0 to .9 and .A to .F. *

For example, suppose you wanted to store the hexadecimal number ABCD(h) (note that this is a 16 bit number) in the 31st register (i.e. Register .E).

You'd press STO · E. Remember this? The calculator would take the number in the X-register and *copy* it into the data register .E and you would now have the hex number ABCD in two memory locations, etc. etc.

*You may also recall that in Floating Point Mode, when you first saw these registers and their "number-names," you could only go up to Register .C. And you know the reason for this: Floating Point numbers take up 56 bits each; and that means you have only 29 data registers available--i.e., up to .C. But in Integer Mode, with Word Sizes smaller than 56 bits, you'll easily have room for 32 or more registers. That's the situation you're talking about now.

Indirect Memory Addressing

What if you've already filled up 32 registers with data--but you have, say, five more numbers you want to store in the calculator. Can this be done?

Yes, but it's a little tricky to understand how. The process involves *indirect addressing*. Here's an example of how the process goes:

Example: Store FFFF(h) in the 33rd register (i.e. register number 32--remember to start your mental numbering at 0!).

Solution: [HEX] [2] [0] [STO] [f] [I]

This is the initial set up which says "store the number 20(h), which is 32(d), in that special register labelled I (look back on page 14 for a quick reminder about the I-register).

[F] [F] [F] [F] [STO] [f] [(i)]

You now key in the number you want to store and tell the calculator to store it *in the register indicated by the contents of the I-register.*

What has happened?

The number in the X-register has been copied (STOred in the usual fashion) into the 33rd register--the one numbered 20(h).

Notice that there's a huge difference between [I] and [(i)]. Capital "I" is the name of a very particular data register in the HP-16C. Lower-case "i" stands for the word "indirectly."

If this sounds a bit confusing, think of the process as being analogous to sending a letter indirectly.

Suppose you want to send a letter to one of us, say, me--Ed Keefe. The problem is, you don't have my address. But you do have the address of Grapevine Publications--and you know they have my address. So you just send the letter in to Grapevine--with a cover letter that tells the good folks there to

"Please forward this letter to Ed Keefe. I don't know his address, but you do."

That's really all you're talking about with indirect storage and recall, too: You can't give the name of any register numbered above 31(d), but you can put its number in the I-register and then store or recall that register indirectly, by using STO f (i) or RCL f (i).*

A quick reminder: You needn't use hexadecimal numbers to indirectly address memory locations; you can use decimal, octal or binary numbers as well.

Some people prefer to work with decimal numbers. If you do, and you wanted to store FFFF(h) in the 33rd register, of course you'd do it this way:

DEC 32	
STO f I	(or just plain STO I)
HEX FFFF	
STO f (i)	(or just STO (i))

*Another by-the-way: You may have discovered already that you don't actually need to press f before pressing I or (i). By looking at the other choices on the I and (i) keys, you can see that when you press those keys after STO or RCL, it's obvious to the calculator that you could only mean I or (i); nothing else on those keys makes any sense.

Swapping Indirectly

If you recall your basic stack manipulation functions, you'll remember a very handy one: [x≷y]. By pressing that key, you can exchange ("swap") the contents of the X- and Y- registers.

Well, there's a similar function that you can use to swap the contents of the X-register with any other numbered register--whether or not that register's number is below 32.

That function is [x≷(i)].

You'll Try Anything Once: Use [x≷(i)] to exchange the contents of the X-register with the contents of the tenth data register (that's the one called "9").

(Assume that the starting contents of Register 9 are CC(h) and the starting contents of the X-register are AA(h)--no, don't assume--be sure of it: Press [HEX] [C] [C] [STO] [9] and then [A] [A] before you begin.)

Solution: [9] [STO] [I] and then [f] [x≷(i)]

See? Now the X-register contains CC(h)--what Register 9 held previously.

Question: What's now in register 9?

Answer: Whatever was previously in the X-register--in this case, the number AA(h).

Notes

Pop Quiz
(not so Pop anymore, is it?)

1. Suppose you started out by resetting your calculator (using that [ON] and [−] combination), and then you proceeded to write a 25-line program. Upon exiting from Program Mode, how many data registers would be left?

2. If you then alter the size of the data registers to 8 bits (and how would you do that?), how many data registers will you have? What would the HP-16C show you if you pressed [f] [STATUS]? How about [f] [MEM]?

3. You want to store the number FA(h) in the 25th register. What are the easiest keystrokes to get the job done?

4. You want to store this same number, FA(h), in the 56th data register. How would you go about doing that?

Answers

1. To start with, you would have 1624 bits of data memory. If you write a program of 25 lines, this will require 4 blocks of 56 bits each for program memory (for a total of 28 lines). That would leave you with 1624 - (56 x 4) = 1400 bits of data memory. But each data register is 16 bits wide (reset default Word Size is 16 bits, remember?), so you would wind up with 1400 / 16 = 87 whole data registers remaining.

2. If you now alter the Word Size to 8 bits, then 1400/8 = 175 registers remain.

   The results of a status review right now would be `2-08-0000`
   The results of a memory review would be `P-3 r-175` The 3 program lines remaining remind you that you've used only 25 of the 28 lines currently allocated.

3. To store FA(h) in the 25th register you could key in [HEX] [F] [A] and then press [STO] [·] [8]. (Remember: The first data register is numbered "0" and the 17th data register is numbered ".0". So the 25th would be ".8")

4. To then store FA(h) into the 56th data register you'd need to do it this way:

Keystrokes	Display	Reminders
[DEC] [5] [5]	`55 d`	You want the 56th register--which is #55.
[STO] [I]	`55 d`	Store this address, therefore, in the I-register.
[R↓] [HEX]	`FA h`	It's still in the stack from the previous problem.
[STO] [(i)]	`FA h`	All done!

PROGRAMMING YOUR HP-16C IN INTEGER MODE

OK, it's time to take a good, close look at programming itself: How do you come up with the logic and the correct keystrokes to carry it out?

Well, what do you know so far?

You've already seen how to switch into and out of program mode--by pressing the g P/R keys.

You've also seen how to begin a program with a label that names it--so you can refer to it later when you're ready to run it. Those label names are just numbers, running from 0 to F.

Theoretically, then, you could hold up to 16 (very short) programs in the HP-16C at one time. Realistically, though, you'll probably wind up with two or three programs that you'll use a lot.

Finally, recall that you already have a program in the memory of your HP-16C. And it will retain this program (and any others currently stored in program memory) until you shift into program mode (g P/R) and press f CLEAR PRGM.

This procedure will erase ALL of the programs in the calculator and release the memory for use as data storage registers. Of course, you could also reset the calculator, but this drastic move would wipe out not only your programs but all of the contents of the data registers, too--not so good.

A Second Program

Time to add a second program to the one that you already have in your HP-16C.

First Thought: Label this program with a "B" so that you'll be able to run the program later (in Run Mode, of course) by pressing GSB B.

But before you can begin to key in the second program, you must be sure to properly finish the one you wrote before.

"That first one *was* finished, wasn't it?"

Yes, you finished it; but you didn't END it.

"Ah...of course--how silly of me... ...what do you mean by ENDing a program?"

Whenever you run a program in the HP-16C, the calculator simply looks in its program memory and reads each step in sequence (beginning at the label with which you called the program), executing the code as if it were being keyed in right then and there.

And of course, the program stops running when there are no more steps left.

So if you were to tack on a second program--immediately following the first one, then the next time you ran the first program, the calculator would run it all right, but then it would continue right on into the steps of the second program.

This tends to impair the accuracy of your results, so what you need is some kind of partitioning instruction--some step that says, basically, "Don't go on past this point; stop here and check the keyboard--to see if any human fingers have poked in some new instructions."

And what is this very verbose instruction?

It's RTN.

See If It Works: Insert a RTN at the end of your first program--to prepare for the entry of a new program.

Keystrokes	Display	Meaning
g P/R	000-	To go into Program Mode--at the beginning of program memory.
GTO . 007	007- 42 44	You're now looking at line 7 of your program.
g RTN	008- 43 21	Here's the all-important step.
g P/R		Back out into Run Mode once again.

You've now put an effective partition in place at the end of your first program; you can now safely key in new steps to form a second program immediately thereafter.

Notice, also, a couple of things about the second keystroke sequence above:

GTO . 007

Any time you press GTO ., followed by a three-digit (decimal) number, the program "pointer" (that imaginary indicator of the "focus" of the calculator's "attention") will jump directly to that line number in program memory.

Keycodes: Just In Case You Were Wondering

"What are those numbers I see in the display when I'm in program mode?"

The mysterious number at the left end of the display is the line number, of course. But what follows it is called a *keycode.*

Keycodes are easy to interpret--if you have your calculator in hand.

For Example: Go back into Program Mode, and go again to line 007 (press [g] [P/R] and [GTO] [.] [0] [0] [7]).

After the line number, you'll see the key code: 42 44.

To interpret this, you would read each two digit number in the following way:

"42 means the key at row 4 and column 2 on the keyboard. That's the [f] key."

"And 44 means the key at row 4 and column 4. That's the [WSIZE] key."

"So the entire keycode means [f] [WSIZE]--and that's right--that's exactly what that recorded step is supposed to be!"

No sweat, right? And after awhile, a person gets pretty good at proofreading his/her program simply by eyeballing the keycodes.

Now for that second program (so you might as well stay in Program Mode):

This program will take two 8-bit numbers (sitting patiently in the X- and Y- registers) and put them together to form one 16-bit number in the X-register.

First, key it in, and then you can take a good look at its logic...

Keystrokes	Display
SST	008- 43 21
g LBL B	009- 43,22, 6
HEX	010- 23
1	011- 1
0	012- 0
f WSIZE	013- 42 44
x≷y	014- 34
g LSTX	015- 43 36
f SR	016- 42 6
f RLn	017- 42 E
f OR	018- 42 40
g RTN	019- 43 21
g P/R	Leave Program Mode and get back to Run Mode.

Now, here's what the program will do, line by line:

Line 009 begins the program with LBL B.

Line 010 puts the display register in HEXadecimal format.

The next three lines set the Word Size to 16 bits--which is 10(h).

Line 014 swaps the high-order bits with the low-order bits so that the high-order bits are now in the X-register.

Line 015 brings 10(h) back into the X-register from the LST X-register (and this also raises the whole stack, right?).

At line 016, the Shift Right is a quick way to divide this 10(h) by 2, leaving 8 h (also known as 8 d) in the X-register.

Line 017 rotates the number in the Y-register 8 bits to the left (and after this is completed, the calculator saves 8 in LST X, and the stack drops, putting the contents of Y into the X-register.

(Notice that you wouldn't use Shift Left here, unless you wanted to key in 8 separate f SL instructions. Nor could you use LJ. Can you figure out why not? See page 171 for some reminders.)

Line 018 OR's the contents of the X- and Y- registers and leaves the result in the X-register, while lowering the rest of the stack.

Line 019 ends this program.

Now you're ready to run the program.

Go: Begin by setting the Word Size to 8 bits (8 f WSIZE).

Then key in two hexadecimal numbers: HEX F A ENTER C E

Now press GSB B and see the word "running" in the display. When the program stops, the display will show you the resulting 16-bit number.

Of course, there's a far cleverer way to accomplish the above bit of nonsense: Take advantage of the way in which the HP-16C handles its data memory.

To see what will happen, just perform the following keystrokes right from the keyboard (i.e. don't record this as a program):

You Press	You See
HEX 8 f WSIZE	
F A STO 1	FA h
C E STO 0	CE h
1 0 f WSIZE	CE h
RCL 0	FACE h
RCL 1	0 h

"OK, what is this little sleight-of-hand, anyway?"

To understand what has happened here, you need to remember that the HP-16C's data memory is just one long string of bits (1624, to be exact, remember?). And this string is partitioned according to the selected Word Size.

The key is this: You don't actually erase anything from memory when you change the Word Size. The calculator just changes the locations of its partitions--i.e. where one register leaves off and the next one begins.*

Here's the data memory before you changed Word Size (expressed as 2 hex digits per register):

Register #	...	06	05	04	03	02	01	00
Contents:	...	???e	???d	???c	???b	???a	FA	CE

Then, when you increase the Word Size to 16 bits, you alter the boundaries between the registers so that they look like this.

Register:	...	02	01	00
Contents:	...	???dc	???ba	FACE

*As a matter of fact, even when you shift from Integer Mode to Floating Point Mode, the calculator will preserve the contents of its Continuous memory. However, when you recall the contents of a register in floating point mode, you'll get an Error message. The calculator is telling you that you can't recall a number in Floating Point Mode if you created that number in Integer mode--but when you shift back to Integer Mode and restore the appropriate Word Size, your data will still be there!

A Third Program

There's one function "missing" from the keyboard of the HP-16C. The [LJ] function will Left Justify a number for you. But there's no RJ function (can you think of a good use for one?).

As a programming exercise, why not invent one?

All right. What's the first thing to do when you set out to solve a problem with a program of some kind?

Well, that depends a lot on your training and temperament, but usually, the first thing is *not* to immediately begin to encode program steps--either on paper or in your calculator (or computer).

So think about the general strategy here for a minute: How would you right-justify a binary number on your HP-16C?

Here's one thought: You could repeatedly shift the number to the right until the least significant bit becomes a 1. OK, that sounds reasonable enough, so adopt it as your working strategy.

Next level of reasoning: No matter what binary number is in the X-register, it's going to have either a 0 or a 1 as the Least Significant Bit (obvious, right?). If the LSB is a 1, then the routine should NOT perform a shift right (and the program should end immediately. But if the LSB is a 0, then the routine should shift the number to the right and then repeat the test of the LSB. And so on.

You can see right away that this will have to be some kind of program "loop," where the calcuator executes the same section of code over and over again until some condition is met. In this case, that condition is the LSB being a 1.

Now, how does the HP-16C decide when to exit such a loop?

Conditional Operators

The HP-16C has a set of commands that are posed as questions. All of these operations--called conditional operators, by the way--allow the calculator to go one of two different directions in the program, depending upon the answer to the question.

For example, in this little loop you want to write, you need to test the LSB (bit #0) to see if it's set (a 1) or clear (a 0).

And it just so happens that there's a conditional operator to do just that--test the status of any bit you wish; you just specify that bit by its number (0-63).

The conditional operator "in question" here is the [B?] key. And it asks this question: "Look at the number now sitting in the Y-register. Specifically, look within that number at the bit whose *position-number* is now sitting in the X-register. Is that bit currently set (i.e. a 1)?"

If the answer to the question is "yes," then the calculator will execute the very next step in a program (as usual). *But,* if the answer is "no," the calculator will *skip* that very next step in the program and continue instead with the following step.

This is the general pattern for all the calculator's conditional operators--the "do if true" rule. And remember, the step you're talking about (to do or not to do) is the step immediately *after* the conditional operator.

So here's how to take advantage of this tool in your little routine to test the LSB:

First, key in the new routine (remember, this is your own invention of a Right Justify function):

Keystrokes	Display	Comments
g P/R		Get into program mode.
GTO . 019	019- 43 21	
g LBL C	020- 43,22, C	
0	021- 0	The number of the LSB.
f B?	022- 42 6	
R/S	023- 31	Stop if LSB is set.
f SR	024- 42 6	If LSB is not set, step 023 will have been skipped, so go ahead: SR, and
GTO C	025- 22 C	Repeat the test!
g P/R		Get out of program mode.

Do you see how this works?

First, at line 020, you put a name on this routine: "C" And in this case, that label both identifies the program so you can "call it up" to run it, *and* it marks the point to which the execution must return--over and over again--until the LSB is a 1.

Then you go right into the testing loop.

Notice that you've assumed that the number you're testing begins in the X-register--and it's only when the program "keys in" the 0 (at step 021) that the number goes up into the Y-register where the conditional operator expects to find it.

Notice also, that when this particular test is finished, that 0 in the X-register disappears, and the number in question drops back down to the X-register again--very convenient for repeated testing!

Now check your routine to see if it actually works...

Press: [BIN] [1] [0] [1] [0] [0] [0] [0] You see: 10 10000 b

Press: [GSB] [C] You see: 10 1 b Ta-daa! Right justification!

Of course, unlike the [LJ] key, this "function" doesn't give you a second value indicating how many places you had to move the significant digits. But that's OK.

A Fourth Program

Here is a fairly large program that comes from the world of microcomputers. It involves computing a checksum for a stream of hexadecimal numbers.

Now, as you may recall from page 183, a checksum is a way to verify the correct transmission of large groups of binary numbers. You remember how the [#B] key would sum the 1-bits in any binary number?

Well, that's not the only kind of checksum there is. Another common type is the arithmetic sum--simply adding the arithmetic sum of all the words in a transmission. Commonly, those words are 8 bits long--and we humans might express them more conveniently as pairs of hex digits, like this:

AF BD 01 00 12 83 E7 56 13 82 FC 2C 2F 98 7F 3C

As usual with a checksum, to insure that the receiving computer is getting all the right numbers, the source computer will send this stream of hex numbers and follow it with another number--the sum of the previous 16 numbers.

The receiving computer will then take the first 16 numbers and compute its own checksum and compare that with the one coming in over the telephone. If the two numbers match, the chances are good that the computer has received the whole line correctly. If not, the transmission is repeated.

Well, here's a typical application of checksums--and of how the HP-16C can help you:

"While writing my own computer telecommunications program, I ran into a glitch.

"My program wasn't accumulating the checksums correctly. The first string of numbers would be sent; the receiving computer would compute its checksums--correctly--and then declare a mismatch with my checksum. Then, of course, it would ask for the same string of numbers--over and over again.

"So I enlisted the aid of the HP-16C to 'desk-check' the portion of my program that computed the checksums. It seemed like the perfect tool for the job. I only needed to key in the hex numbers and press the [+] key sixteen times. Sounds simple.

"But my hand-eye coordination must not be what it once was: I came up with three different checksums for the same string of 16 hex numbers in as many tries.

"What I needed was some way to get 16 hex digits into the calculator and *check to make sure that I had entered them correctly*. Only after the numbers were correct would I let the calculator figure the checksum.

"All this meant that I needed three separate programs. The first program would let me enter the hex number and the number of the register where I wanted to store the hex number. This would be the **input** routine.

"The second program would be a routine that would let me view the numbers that were stored in the calculator.

"The third program would actually compute and display the checksum."

Here is the input routine (go ahead and key this in to follow the reasoning):

Keystrokes	Display	Comments
g P/R		Get into PRGM mode
f CLEAR PRGM	000-	Clear out all other programs.
g LBL A	001- 43,22, A	
STO I	002- 44 32	Store the number in the I register.
g ABS	003- 43 8	Take the absolute value of this number as one quick way to put the number in the LST X-register.
R↓	004- 33	
STO (i)	005- 44 31	Store indirectly.
R/S	006- 31	Stop the program and wait for next two numbers to be entered into the stack.
GTO A	007- 22 A	Repeat the loop.

"Most of the routine is pretty straightforward, but there are a couple of steps that need some explanation.

"For example, in line 003, I used the **absolute value** function as a way to copy the number in the X-register in the LSTX register. Why do this? Well, this number is the register number where the current hex number is stored. If I forget which was the last register I used, I can just press g LSTX and find out what that register number was.

"To use this routine, all I need to do is to key in the first hex number in the string, press the [ENTER] key, and then key in the register number, e.g. [1]. Then I press [GSB] [A]--the input routine--to accept this hex number into register 1.

"Then I key in the second hex number, then [ENTER], followed by a [2] and press [R/S]. And so on, for all 16 hex numbers in the transmission.

"Note that, with this routine, if I can't remember the hex-equivalent name for the 25th register, I just have to press the [DEC] key, key in [2][4], press the [HEX] key once again, and then [GSB] [A] or [R/S], as usual."

Play with this routine awhile--until you're sure you understand why each step is necessary.

Notice how part of the design of the program is in anticipation of the needs of you, the user--providing the option to remind you where you are.

OK? Then back to the testimonial--this time for the second routine in the program--the viewing routine, where you can double-check yourself, to be sure the numbers in the calculator are really the ones you meant to key in....

"The second routine will let me view the contents of all the registers, starting from a specified register down to register #1. Or--alternatively--it will let me view the contents of any single register."

(Here are the keystrokes, and then in the next sections, you'll see more explanations of the new and tricky parts of this logic:)

Keystrokes	Display	Comments
g P/R		
GTO . 007	007- 22 A	
g LBL B	008- 43,22, 6	
STO I	009- 44 32	The number of the starting register is in the X-register.
g LBL 0	010- 43,22, 0	The **loop** starts here.
RCL I	011- 45 32	Begin by recalling the number from the I-register.
GSB C	012- 21 C	Branch to the routine labelled C.
g PSE	013- 43 34	Come back from routine C.
g PSE	014- 43 34	View the contents of the register indirectly.
g DSZ	015- 43 23	Decrement the value in the I-register.
GTO 0	016- 22 0	and repeat the LOOP.
g RTN	017- 43 21	End this part of the routine.
g LBL C	018- 43,22, C	This is a stand-alone
STO I	019- 44 32	subroutine.
RCL (i)	020- 45 31	
g RTN	021- 43 21	This ends the routine and also returns to the calling program, if necessary.

How do these programs work? They're actually one routine inside of another:

If you want to view the contents of successive registers, just key in the number of the highest starting register, say, for example, F(h), and press GSB B. The calculator will then display the contents of all the registers from Register F down to Register 1. It will pause, momentarily, to let you see each number in the display before moving on to the next one.

If, on the other hand, you only want to view the contents of a single register, key in the number of that register and press GSB C. Those contents will appear in the display.

But notice that routine B actually uses routine C as the core of its procedure. This means that you're able to call routine C automatically (through B) or manually (through C itself).

There are probably other ways to accomplish these same tasks, but this particular method helps to introduce several different features in programming the HP-16C.

Pausing During Execution

For one thing, note that you can use one or more PSE (Pause) instructions to momentarily display the contents of the X-register--without stopping the program.

If you would rather have the program stop, then you should substitute one R/S instruction for both PSE lines (of course, if you do use a R/S instruction, then when the calculator obeys this and stops during the program, you'll have to press the R/S key to resume execution).

As another example of the use of the PSE function here, consider this:

If you wanted the calculator to show you which register it's going to review next, you could insert a PSE instruction between lines 011 and 012.

Do you see why?

Loop Counters

Here's another new feature introduced by this register-reviewing routine:

As you already know, the HP-16C will let you write programs that contain loops, but thus far, the only way to get out of a loop was by using a conditonal operator--some question whose answer must be either yes or no.

Now there's a new way out: a loop that counts--and then exits automatically when a certain number is reached. And the I-register holds the loop-count.

The ISZ (Increment and Skip if Zero) and DSZ (Decrement and Skip if Zero) instructions are both meant to be used with these counting loops. As you can see from routine B, you must set the value of the I register to some target count-value *before* entering the loop.

The next-to-the-last instruction within the loop must be either the DSZ or ISZ instruction. Then the loop usually ends with a GTO instruction that branches back to the beginning of the loop.

What's going on with this counting in the I-register?

Two things, really: When the calculator gets to the DSZ instruction in a routine, it automatically subtracts one from (decrements) the value in the I-register. Then, if the result is *not* zero, the calculator *will* execute the next instruction (and usually this instruction is the GTO). But if this Decremented result is Zero, the calculator will Skip over the next instruction (i.e. skip the GTO and thus exit the loop).

(Now go ahead on your own and speculate wildly as to what ISZ--Increment and Skip if Zero-- does.)

Notice that these loop counter functions are actually do-if-true operators, but with an extra twist: They perform a side task first (Increment or Decrement the I-register), and then they ask a question about the result of that task ("Is the I-register still non-zero?"). If the answer is "yes," they continue on as usual--just like the other conditional operators; if the answer is "no," they skip the following step and continue on below it.

Notice also that you may very well want the program to end entirely upon exiting the loop. If this program is the last one in memory, you can just leave it open-ended and the calculator will figure that you're done and stop. But if there's another program immediately after the loop, the HP-16C will execute it unless you put in an instruction to partition the programs. Usually, you would insert a RTN instruction.

You may be wondering how many times you could execute a loop using the DSZ or ISZ instructions. The answer is "quite a few times." The I-register is 68 bits wide--and this is a permanent value, unaffected by altering the Word Size.

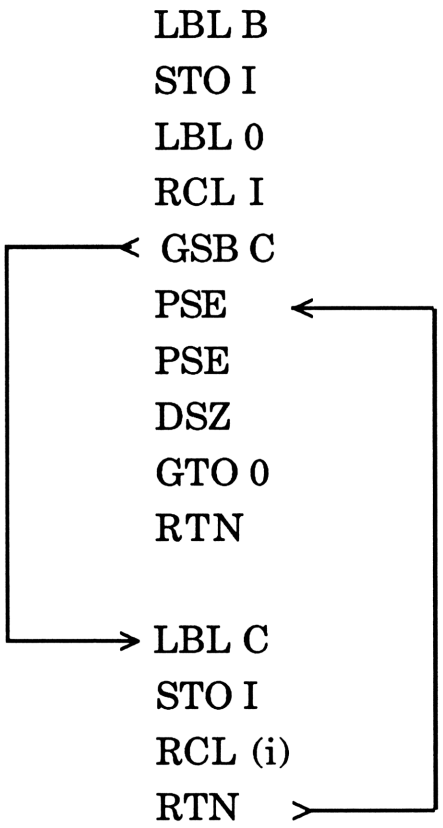
So with that many bits in the word you could, theoretically, have 2^{68} executions of a loop. Of course, because of the mechanical limitations of getting it there, you could only store a decimal number equal to 2^{64} in the I-register. That's about 2×10^{19}(d).

On most HP-16C's it takes about 1 second to execute an empty loop 4 times. So, roughly speaking, of course, you could put the calculator into a loop that would take it about 9×10^{10} centuries (give or take a millenium) to exit.

(...Thankfully, pressing any key on the keyboard will stop a program from running to completion.)

Another good lesson from that register review program: Now you've seen an example of a subroutine. A subroutine is a section of a program that's used more than once in the program--a program within a program--or a program that may be called by more than one program.

For example, in those routines labelled B and C, the B-routine is the calling program. It calls the C subroutine at line 012:



When the calculator reaches the GSB C instruction in line 012, it searches downward in memory until it finds a LBL C. Once it finds this label, the HP-16C will begin to execute the program steps that follow it, continuing until it reaches a RTN instruction. At that point, *it jumps back to the calling program and continues execution at the line following the GSB instruction.*

Now that you have a routine to let you enter data into the calculator's memory--and another one to let you review that data, it's (finally) time to develop a program to actually do the arithmetic--accumulate the checksum for all those 16 hex numbers.

This routine, then, should instruct the HP-16C to recall each of the numbers that you've stored in the data registers and accumulate a total in the X-register.

There's one slight problem to contend with, however:

Suppose you had chosen to store your 16 numbers not in registers 16 through 1, but rather, say, in registers 45 through 30?

It wouldn't be correct simply to start a loop counter at 45, and then let it go all the way to 0 with a DSZ loop--that's too far. You would want it to stop after register 30--and you need to invent a way to get your HP-16C to do this....

Hmmm...in effect, what you really need in the HP-16C is two registers that could act as indirect registers. One register could act as a pointer to the register you want to recall. The other register could act as the loop counter.

Problem is, however, you're limited to just one I-register, so you'll have to perform some sleight-of- hand tricks to get this one I-register to do double-duty.

See if you can get inside the mind of a programmer and follow this chain of reasoning about how you would write this routine on the HP-16C:

Describe to yourself exactly what is happening in the I-register and in the stack at each step of the program as it happens. Sometimes that's the only way in which you can get a program to do exactly what you want.

In this program, you want the end result to be the sum of the registers between some arbitrary, higher-numbered register and a lower-numbered register.

Take your own example: Suppose you want to compute the sum of the registers from 45(d) to 30(d), inclusive.

First consideration: You would like to be able to simply enter these two numbers into the stack, and then go ahead and run the program to get the desired check-sum as an output.

Thus, you should be able to begin by pressing the [DEC] key and then key in [4][5] [ENTER] [3][0]. Then all you'll need to do is press [GSB] [D] (for the sake of continuity from your previous routines, call this one "D"), and let the program run to completion.

So take those assumptions and begin to develop the program. Imagine the keystrokes necessary to solve this problem manually--but prepare your calculator to record them as you go:

[g][P/R]
[GTO] [.] [0][2][2]
[g] [LBL] [D]

OK, then: "Now sitting in the X-register is a (decimal) 30; the Y-register has 45."

Go.

[x≷y] [STO] [0]	First, you want to get the high register number in the X-register in order to save it.
[x≷y] [−]	Now restore the original order of the two numbers so that you can take the difference of the two numbers.
[STO] [I] [g] [ISZ]	Now put this difference in the I-register and increment it by 1, so that the counter will act on the high- to low- numbered registers, inclusively.
[RCL] [0]	Next, recall the high register number to act as your "pointer."
[0]	Since the X-register will serve as your accumulator, you need to initialize it to zero before beginning the accumulation loop.
[g] [LBL] [5]	Here you will enter the loop.
[x≷y] [f] [x≷I]	The first thing to do is swap the contents of the X- and Y- registers and then swap the X- and I- registers.
[x≷y]	Then swap the X and Y registers again.
[RCL] [(i)] [+]	Now you can recall register 45 and add it to the contents of the X register.
[g] [DSZ]	Now decrement the value in the I register so that it will "point" at the next lowest data register.
[x≷y] [f] [x≷I] [x≷y]	Then reverse the process of swapping the X-, Y- and I- registers.
[g] [DSZ] [GTO] [5]	Finally, decrement the I-Register counter, and repeat the loop.

Keep in mind that you have to do all that extra swapping just to make up for the lack of another Indirect register. Techniques like these have been common since the first programmable calculators were invented, since all are somewhat limited in their memory--and thus, their programmability.

Here's the program that you just developed, with the corresponding keycodes:

Keystrokes	Keycodes
g LBL D	023- 43,22, d
x≷y	024- 34
STO 0	025- 44 0
x≷y	026- 34
−	027- 30
STO I	028- 44 32
g ISZ	029- 43 24
RCL 0	030- 45 0
0	031- 0
g LBL 5	032- 43,22, 5
x≷y	033- 34
f x≷I	034- 42 22
x≷y	035- 34
RCL (i)	036- 45 31
+	037- 40
g DSZ	038- 43 23
x≷y	039- 34
f x≷I	040- 42 22
x≷y	041- 34
GTO 5	042- 22 5

Note that you designed the routine using decimal numbers for the data and the pointers (i.e. the numbers specifying the registers). Does it work in hex?

To test the program, begin by using the "A" routine to store AA(h) in the consecutive registers 45(d) through 30(d), i.e. register 2D through 1E.

Then use the keystrokes [2][D] [GSB] [B] to view all the data registers from 2D on down.

(Press any key to get this B routine to stop when it hits the 1E-register.)

Now, if all the data is correct, you can enter the two numbers [2][D] and [1][E] into the stack and then press [GSB] [D].

The calculator will flash "running" at you for about 25 seconds. Then it will display the result, which will depend on the Word Size.

If you've been using a Word Size of 8 bits, the result will be A0(h). But if you're currently using a Word Size of 16 bits, the result will be AA0 (h).

To those of you who are accustomed to working with desktop computers, 25 seconds may seem like an amazingly long period of time just to add 16 hexadecimal numbers.

True. But you've traded run-time speed here for the added accuracy offered by the automation of input, review, and accumulation.

"Programming is always a trade-off in this way--a trade-off between speed, power, flexibility, convenience, cost, memory, I/O, friendliness...."

Just Plain Quiz

1. What keys will turn on the PRGM annunciator in the display?

2. What keystrokes will let you jump to any existing program line?

3. Which instruction will let you momentarily see the contents of the X-register during a running program?

4. What are two purposes for the RTN instruction?

5. What two instructions will cause a running program to jump to another label in the calculator's memory?

6. What instruction is associated with the key codes 42 24?

Just Plain Answers

1. The [g] [P/R] keys will initiate the Program Mode on the HP-16C.

2. The [GTO] [.] nnn keystrokes will let you move to any line, nnn, in the calculator's memory.

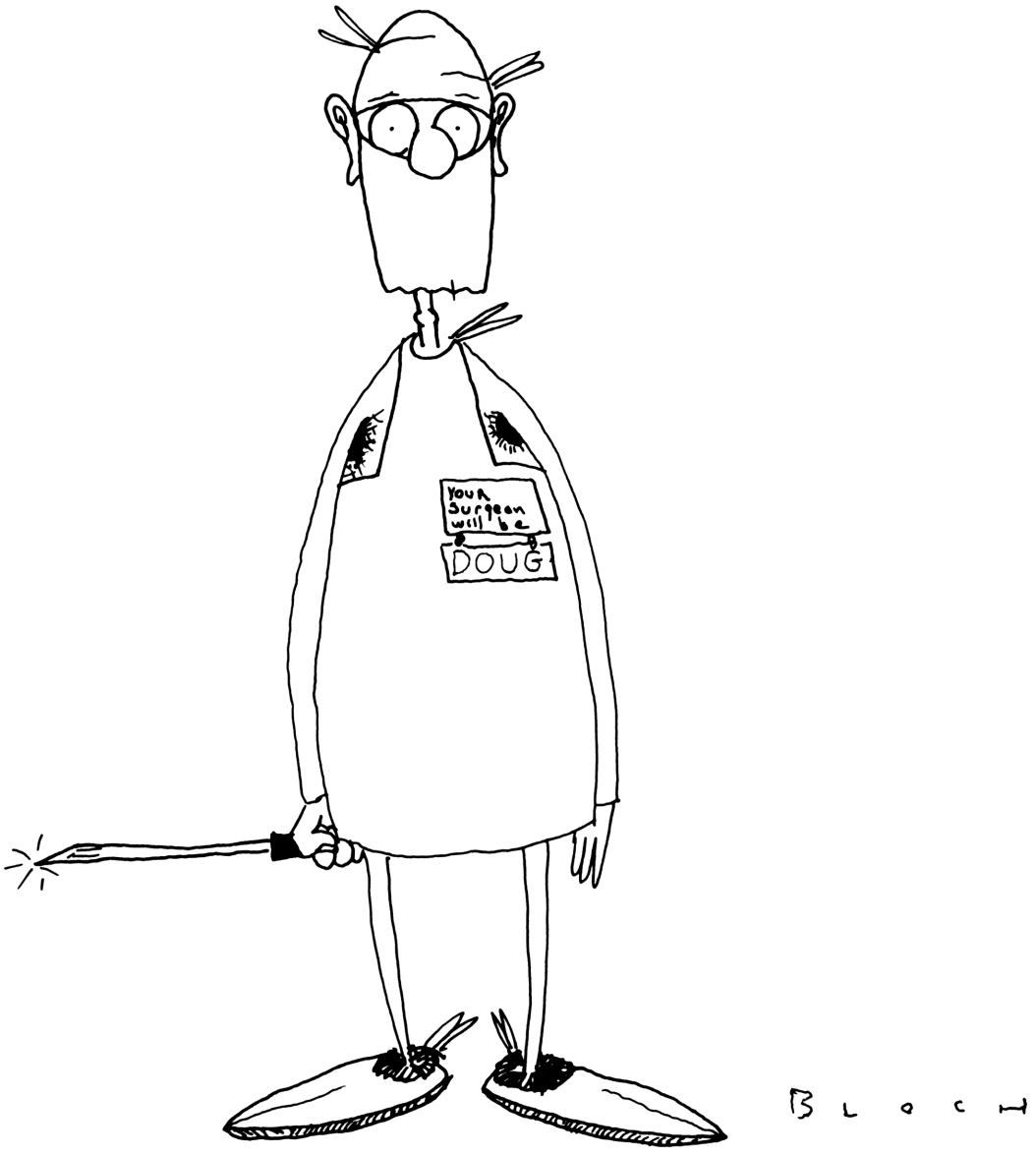
3. The PSE instruction momentarily halts a running program and shows the contents of the X-register.

4. The RTN instruction can be used to separate one program from another. In this case it acts as an End or Halt instruction.

   When the calculator comes upon a RTN instruction at the end of a subroutine called by a GSB, it will branch back to the calling program. In this case the RTN operator acts as a Return function.

5. The two unconditional branching instructions are GSB and GTO.

6. The keycode 42 24 is associated with the [f] SHOW [DEC] instruction.



APPENDICES

Double-Number Functions

The HP-16C contains three mysterious functions that work with double-length numbers: [DBL×], [DBL÷], and [DBLR].

The Double-Multiply Function

To get the basic idea, just remember that [DBL×] means this:

"Multiply two numbers of a given word size and show the result as a number with twice the original word size." If, for example, you use [DBL×] to multiply together two numbers with a 16-bit Word Size, the calculator will show the result as a 32-bit number.

But why do you need such a function, anyway?

Sometimes, when you're multiplying two numbers on the HP-16C, the G annunciator will turn on, indicating that the result has exceeded the bounds of the current Word Size. Normally you can rectify this by redoing the problem with a new Word Size twice as large as the current one.

But what do you do when you're already working the maximum Word Size (64 bits)? This is where the [DBL×] function can be quite useful.

In those cases, you could key in a full 64-bit number, press [ENTER], and key in another 64-bit number. Then, to multiply them, you would press [g] [DBL×].

Simple, right?

Yes, but how can the HP-16C display the answer? After all, each register is limited to 64 bits.

Quite frankly, you'll be forced to view the answer in two parts: The high-order bits of the number will be in the X-register (bits 127 through 64, inclusive); and the low-order bits will land in the Y-register (bits 63 through 0).

Thus, to view the whole answer as a binary number, you might need to view all 8 windows of the X-register, then use the [x≷y] key to view all 8 windows of what was in the Y-register.

Try One: Begin by setting the Word Size to the maximum, 64 bits (remember the shortcut: [0] [f] [WSIZE]). Then press [g] [SF] [3] to display leading zeroes and [f] SET COMPL [UNSGN]. Finally, check the calculator's status: [f] [STATUS]. You'll see 0-64- 1000

Now press [HEX], and enter the number ABCDEF0123456789 (h).

Press [ENTER], and then enter the number 1234567890ABCDEF(h).

Now press [g] [DBL×]. You'll see b89F98F6.h

To see the very highest-order digits, you press [f] [WINDOW] [1].

You'll get: 0C379AAA h.

To see the other (righthand, i.e. low-order) part of the number, press [x≷y] and [f] [WINDOW] [1]: 1A64331F h.

And [f] [WINDOW] [0] shows: bA375dE7.h

So the full product is 0C379AAAB89F98F6 1A64331FBA375DE7 (h).

Get the idea? Good.

There are, however, some warnings you should heed when using the [DBL×] function.

For example, when working with hex numbers, you would want to keep the Word Size to a multiple of 4; if you're working with octal numbers, the Word Size should be some multiple of 3.

Why? The very best explanations can be found on pages 52-55 and 78-80 in the HP-16C OWNER'S HANDBOOK. If you'd like to explore this further, then, there's the best place to look.

The Double-Divide Function

The [DBL÷] function will let you divide a double-length number by a single-length number and get a single-length quotient. Just as you would expect, right?

To use this function, you'll find it easiest to convert all numbers to hex format and view the quotient in hex format as well.

This is because you'll have to break the dividend into its high-order bits and low order bits--and load them appropriately in the X- and Y- registers before you can perform the division. And it's not easy at all to do this if you're working in decimal integer format.

Even in hex format, this splitting of the dividend into high- and low- order bits can be tricky, especially if the dividend's Word Size is not evenly divisible by 4.

The easiest way to deal with that situation is to pad the dividend on the left with enough leading zeros to make the Word Size evenly divisible by 4. Then you should be able to figure out where the number's "halfway point" is.

Two other details to remember:

If the quotient is non-zero, the C annunciator will turn on.

Also, if you've been too optimistic--i.e. if the quotient still can't fit into a single-length word, the calculator will display an Error 0 message.

An Example: You should still be in UNSiGNed hex format, with leading zeros shown, and a Word Size of 64 bits.

Now, suppose you try to undo the example you just saw from [DBL×]. That is, divide that huge product you obtained,

0C379AAAb89F98F61A64331FbA375dE7 (h)

by one of the original multiplicands: 1234567890ABCDEF(h)

(You should get the other multiplicand as the quotient, right?)

So first, key in the low-order bits of the dividend:

[1][A][6][4][3][3][1][F][B][A][3][7][5][D][E][7].

Then press [ENTER], and enter the high-order bits of the dividend:

[0][C][3][7][9][A][A][A][B][8][9][F][9][8][F][6]

Press [ENTER] again, and then key in the divisor:

[1][2][3][4][5][6][7][8][9][0][A][B][C][D][E][F].

Now press [9] [DBL÷], to see the quotient in the X-register:

AbCdEF0 123456789 h

(You'll have to use the [WINDOW] key to see the full quotient, of course.)

The Double-Remainder Function

The [DBLR] function is to the [DBL÷] function exactly what the [RMD] function is to the [÷] function:

You key in the dividend and the divisor exactly as you would for [DBL÷]--but instead of pressing [g] [DBL÷], you would press [g] [DBLR].

The result will be the remainder of the double division--instead of the quotient.

Two little notes:

You'll still get an error if the *quotient* is too large for the current Word Size--even though you're only interested in the remainder

The sign of the remainder will match the sign of the dividend--and again, this is just how things work with [÷] and [RMD].

Programming in Float Mode

One of the least mentioned aspects of the HP-16C is its programmability in Floating Point Mode.

First of all, just how large a program can you have?

Well now,... there are 29 data registers available--or 203 lines of program memory. Thus, if your program didn't need any data registers, you could convert all of them to program lines, and your program(s) could go as high as line 202. That's quite a bit.

So just remember: For every data register you do need to preserve, the program memory is reduced by 7 lines.

Now then, for what kinds of applications are you likely to need floating-point programmability?

"Well, what can I do? There's not exactly a lot of floating point functions to choose from!"

True: The HP-16C doesn't have the rich choice of math functions that are available on its cousins, the HP-11C and HP-15C. Nevertheless, even with its limited set of functions, you can still get it to do some useful chores for you.

For example, here is a program that will turn the HP-16C into a rather limited statistical calculator.

Keystrokes	Keycodes	Comments
g P/R		Get into program mode.
GTO . 000	000-	Get to top of program memory...
g BST	042- 22 5	...then to the bottom of program memory.
g RTN	043- 43 21	Put an End on the previous routine.
g LBL E	044- 43,22, E	Begin new program.
f FLOAT 0	045- 42,45, 0	Set Float 0 format for prompting display.
0	046- 0	Initialize accumu-
STO 1	047- 44 1	lator.
1	048- 1	Initialize counter.
STO I	049- 44 32	
R/S	050- 31	Wait for first datum.
STO 0	051- 44 0	Store in accumulator
g LBL F	052- 43,22, F	LOOP for more data.
g ISZ	053- 43 24	Increment counter.
g LBL 8	054- 43,22, 8	A dummy instruction to prevent skipping if I equals zero.
f FLOAT 0	055- 42,45 0	Reset display when entering new data after getting a mean.
RCL I	056- 45 32	Display prompt.
R/S	057- 31	Wait for input.
RCL 0	058- 45 0	Recall current mean.
−	059- 30	Subtract from input.
RCL I	060- 45 32	Recall counter value.
÷	061- 10	Divide into the difference.
ENTER	062- 36	Load the stack.
ENTER	063- 36	
ENTER	064- 36	

(cont.)

Keystrokes	Keycodes	Comments
RCL 0	065- 45 0	Add to accumulator.
+	066- 40	
STO 0	067- 44 0	
R↓	068- 33	Drop the stack.
×	069- 20	Square the difference.
g DSZ	070- 43 23	Decrement counter
RCL I	071- 45 32	and recall I and multiply it by
×	072- 20	the square of the difference.
g ISZ	073- 43 24	Increment counter.
RCL I	074- 45 32	Recall it and
×	075- 20	multiply it by the product.
RCL 1	076- 45 1	Recall the value in Register 1, and
+	077- 40	add this to the value
STO 1	078- 44 1	in Reg-1.
GTO F	079- 22 F	Repeat the loop.
g LBL 7	080- 43,22, 7	Begin Output Routine.
f FLOAT 5	081- 42,45, 5	Set Float 5 display format.
RCL 1	082- 45 1	Finish computing
RCL I	083- 45 32	the Std. Dev.
2	084- 2	
−	085- 30	
÷	086- 10	
g √x	087- 43 25	
RCL 0	088- 45 0	Recall mean.
g RTN	089- 43 21	End.
g P/R		Go back to Run Mode.

The formula for computing the mean and standard deviation are the typical ones:

$$\text{Mean (M)} = \frac{(x_1 + x_2 + x_3 + x_4 \ \ldots + x_n)}{n}$$

$$\text{SDEV} = \left[\frac{(x_1^2 + x_2^2 + x_3^2 + x_4^2 \ \ldots + x_n^2) - (nM^2)}{n - 1} \right]^{1/2}$$

The program works by first initializing the counter to 1 and setting the accumulators to 0. Then it repeats a loop--if there's more data.

The loop uses the counter value in I, incrementing and decrementing it to compute the following quantities:

$$D = (x_n - M)/n$$

$$M = M + D$$

$$SS = SS + n(n-1)D^2$$

And when you execute the output routine, the mean appears in X and the standard deviation is in Y.

OK, knowing just that much, see if you can follow the logic of each of the sections in this program--now that you're more familiar with programming on the HP-16C....

Running the Program

And of course, what does this program actually do for you?

The easiest way to see that is to run it:

Begin by pressing GSB E. The display will show you a 1, which is your prompt to key in the first datum.

After you do so, you press R/S, to tell the machine to go ahead and accept this first value.

Then it will prompt you with a 2 for the next datum, and so you can go on in this manner, keying in as many successive values as you wish.

Then, when you finish keying in data and wish to see the results, just press GSB 7, and the mean of the data will be computed and placed into the X-register; the standard deviation will be in the Y-register.

Even after that, if you want to key in still more data to this current accumulation, just press GSB F, and the calculator will prompt you for the next datum.

The End?

Well, here you are. You have just finished a full course on using the HP-16C. I hope you didn't try to take the course all in one evening. It took me months to write the book. It took me many years to acquire the knowledge to write it. How long it takes you to master the HP-16C will depend on your starting point and how far you want to go, right?

And where can you go from here? Well, I think you're ready to tackle the Owners' Manual for the HP-16C now. It's really the best, small, reference book for the calculator. From there, you can go to your text books and reference books for more examples of how to use the HP-16C. You might want to review some of the courses that you have already taken. You might even want to tackle a Symbolic Logic course at your local college or university. You could show many a Logic professor a trick or two with your calculator. Of course, the professor might teach you several important ideas that would pay off in making you a better engineer or programmer. Who knows?

Once you get into the "real world" of computer engineering and/or software engineering, you will find countless other uses for the "Computer Scientist" calculator. Your appreciation of the HP-16C will continue to grow. Hopefully, so will you.

June, 1987

We hope you have enjoyed this Easy Course book--and that you'll let us hear any comments you may have. Remember: Your response is our only way to know whether or not we have succeeded in what we work hard to do--provide books that are both informative and enjoyable. So please--your opinions (and proof-readings) are welcome--and we always read our mail!

And by the way, if you liked this book, here are some others that you or someone you know might enjoy also:

* An Easy Course in Programming the HP-41
* An Easy Course in Using the HP-12C
* An Easy Course in Programming the HP-11C and HP-15C
* The HP-12C Pocket Guide
* The HP Business Consultant (HP-18C) Training Guide
* The HP Business Consultant (HP-18C) Pocket Companion
* Computer Science on Your HP-41 (Using the HP Advantage ROM)
* An Easy Course in Using the HP-16C
* An Easy Course in Using the HP-28

You can use this handy set of order forms here --->

Or, you can contact us for further information on the books and where you can buy them locally:

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

Call: 1-800-338-4331 (in Oregon, 1-754-0583)

ORDER FORM *(Impress a Friend!)*

Yes! Please send:

_____ copies of	*An Easy Course in Programming the HP-41*	@ $20/copy	=$_____
_____ copies of	*An Easy Course in Using the HP-12C*	@ $20/copy	=$_____
_____ copies of	*An Easy Course in Programming the HP-11C and HP-15C*	@ $20/copy	=$_____
_____ copies of	*The HP-12C Pocket Guide*	@ $5/copy	=$_____
_____ copies of	*The HP Business Consultant (HP-18C) Training Guide*	@ $22/copy	=$_____
_____ copies of	*The HP Business Consultant (HP-18C) Pocket Companion*	@ $8/copy	=$_____
_____ copies of	*Computer Science on Your HP-41 (Using the Advantage ROM)*	@ $15/copy	=$_____
_____ copies of	*An Easy Course in Using the HP-16C*	@ $20/copy	=$_____
_____ copies of	*An Easy Course in Using the HP-28*	@ $22/copy	=$_____

Shipping & Handling: **$2 per order** (Just $.50 for orders under $5. Just $1 for orders under $10) = $ 2.00
For faster UPS service *add* **$1.50/order** = $_____

TOTAL AMOUNT ENCLOSED: -------> = $_____

Your VISA or MasterCard number:_____ Exp. date:_____

Your signature:_____

We will also accept your check (made out to Grapevine Publications, Inc.).

These prices might change without advance notice to you.
And please allow us 3 weeks for delivery of all books. **Thank You!** ☞

ORDER FORM *(Impress a Friend!)*

Yes! Please send:

_____ copies of	*An Easy Course in Programming the HP-41*	@ $20/copy	=$_____
_____ copies of	*An Easy Course in Using the HP-12C*	@ $20/copy	=$_____
_____ copies of	*An Easy Course in Programming the HP-11C and HP-15C*	@ $20/copy	=$_____
_____ copies of	*The HP-12C Pocket Guide*	@ $5/copy	=$_____
_____ copies of	*The HP Business Consultant (HP-18C) Training Guide*	@ $22/copy	=$_____
_____ copies of	*The HP Business Consultant (HP-18C) Pocket Companion*	@ $8/copy	=$_____
_____ copies of	*Computer Science on Your HP-41 (Using the Advantage ROM)*	@ $15/copy	=$_____
_____ copies of	*An Easy Course in Using the HP-16C*	@ $20/copy	=$_____
_____ copies of	*An Easy Course in Using the HP-28*	@ $22/copy	=$_____

Shipping & Handling: **$2 per order** (Just $.50 for orders under $5. Just $1 for orders under $10) = $ 2.00
For faster UPS service *add* **$1.50/order** = $_____

TOTAL AMOUNT ENCLOSED: -------> = $_____

Your VISA or MasterCard number:_____ Exp. date:_____

Your signature:_____

We will also accept your check (made out to Grapevine Publications, Inc.).

These prices might change without advance notice to you.
And please allow us 3 weeks for delivery of all books. **Thank You!** ☞

"Please send these books to:

_____

Name

_____

In Care Of (a company, maybe--or some other person)

_____

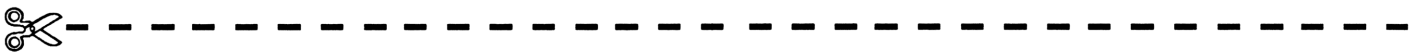
Street Address (**Note:** UPS will **not** deliver to a Post Office Box!)

_____

City State Zip

_____ ( ) -

Your Daytime Telephone Number



"Please send these books to:

_____

Name

_____

In Care Of (a company, maybe--or some other person)

_____

Street Address (**Note:** UPS will **not** deliver to a Post Office Box!)

_____

City State Zip

_____ ( ) -

Your Daytime Telephone Number

Keefe
Bloch

An Easy Course in Using The HP-16C

This cover flap is handy for several different things:

-- Tuck it just inside the front cover when you store this book on a shelf. That way, you can see the title on the spine.

-- Fold it inside the back cover--out of your way--when you're using the book.

-- Use it as a bookmark when you take a break from your reading!

GRAPEVINE

An Easy Course in Using the HP-16C

For all you computer science students and professionals, here it is--the easiest and most painless way to get up to speed on your HP-16C "Computer Scientist" calculator!

An experienced CS instructor and author, Ed Keefe leads you through a lively Easy Course on the talents and tricks of this multilingual little machine. Whether you want to work in decimal, octal, binary or hexadecimal, with 4 bits or 64, here's the class you can give to yourself--before you face your programs and their problems.

The book is filled with examples, review questions, explanations and quizzes, all designed to let you work at your own speed (and with this course, your own speed will soon amaze you)! It's always a pleasant surprise that learning both the subject and the calculator can be this much fun--but it can be, when the right explanation transforms a mysterious machine into a simple and friendly tool.

0 12841 00016 3

ISBN 0-931011-16-7

FROM THE PRESS AT
GRAPEVINE PUBLICATIONS, INC.
P.O. Box 118 • Corvallis, Oregon 97339-0118 • U.S.A. • (503) 754-0583