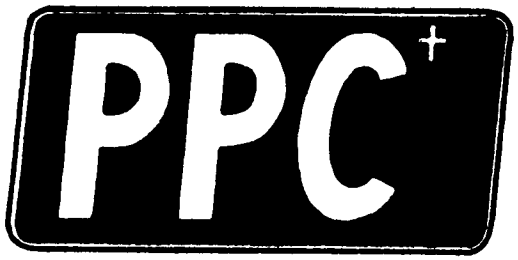


Reply To:
Richard J. Nelson
2541 W. Camden Place
Santa Ana, CA 92704
U.S.A.

Dear PPC Member,

Enclosed is a copy of "Better Programming on The HP-67/97. The history and purpose of this publication is described on pages two and three. You will find this publication helpful even if you don't have or use an HP-67 or HP-97. The format is different from PPC JOURNAL although the front cover intentionally resembles a regular monthly issue. The print is larger and there is space for notes and additions.

The index on page 31 has a black page edge 'TAB'. For convenience in leafing from the back you may want to make a similiar 'tab' on the opposite side on page 30.

PPC is a club, we share our experiences and mutually help each other to improve our programs and uses of our machines. We do not engage in the business of selling services or products. We must support ourselves, however, and if members wish to purchase additional copies of "Better Programming on The HP-67/97 they may order them at the rates given below.

Sent To:	Single copy	Two copies
U.S.,Canada,Mexico	$3.00	$5.00
Europe	$4.30	$6.90
Asia	$4.70	$7.50
South/Central America	$4.00	$6.25

Your comments and suggestions are welcome and will be applied to future PPC publications.

Happy Programming
Richard Nelson

Errata: Page 2, 3rd paragraph blank space after "all HP" should be machines

† PPC, formerly the HP-65 Users Club, is the worlds first and largest organization devoted to personal programmable calculators. The Club is a volunteer, non-profit, loosely organized, independent, world-wide group of Hewlett-Packard personal programmable calculator users. The official Club publication, PPC JOURNAL, formerly 65 NOTES, disseminates user information related to applications, programs, programming techniques, problems, hardware inovations, - any information related to the selection, care, use, and application of Hewlett-Packard Personal Programmable Calculators.

A PPC PUBLICATION

BETTER PROGRAMMING ON THE HP-67/97

EDITED BY: William Kolb (265)
John Kennedy (918)
Richard Nelson (1)

This Publication is available to members only.

PREFACE

This book is a publication of PPC. It is written by, and for PPC members. It is not available to non-PPC members. PPC exists because even though programming is an individual, personal activity, programmers need to share their experiences. Additionally, the tremendous amount of time required to assemble even a modest library of programs requires that programmers exchange their work to avoid continual "re-inventing of the wheel." This book is a small attempt to assemble in one reference a few of the many programming techniques developed for the HP-67/97. While the supply lasts, and it should last until long after the HP-67/97 are retired, each new member of PPC will automatically receive a copy.

This "better Programming" book started out as a formalized collection of notes prepared by Bill Kolb, co-ordinator of the Washington Chapter. Many of the new chapter members requested help in mastering the various functions of the HP-67. After many exchanges between friends and fellow PPC members, Bill produced-in camera ready form- the first 28 pages of techniques he called Better Programming on the HP-67/97. The material following page 32 was added to Bill's basic work.

For you non-67/97 users, I highly recommend that you also read and study the material even though you may use an HP-25,29,33,38,55, or 65. Resist the temptation to ignore material written for another machine. There are many valuable tips that can be applied to all HP Following the index on pages 30 thru 32 is "RPN and the Stack" also by Bill Kolb. This article is applicable to all HP PPC users and it should provide greater insight into the tremendously powerful automatic "HP STACK."

The stack concept is used in all PPC's, although those machines that use a form of algebriac logic may not use the term stack. I attempt to contrast RPN with TI's advanced AOS^{tm} from 'our' point of view in "AOS^{tm} vs RPN" Do you understand the difference? The basic idea is simple, although most users don't compare the systems by the architectural differences. Between "RPN and the STACK" and "AOS vs RPN", you should have a better understanding of why the HP system of RPN, the automatic stack, and LAST x are so powerful, expecially the difficult to explain ease of solving every problem the "same" way.

John Kennedy prepared the article "Data Packing" as well as some of the routines that finish up the book. John also contributed his time in preparing the book for printing.

This book is a small, beginning effort, towards what might be called a PPC Programming Techniques Handbook. Ideally this handbook should be written by PPC members, and published by PPC. The topics to be included should cover the whole range of tips, techniques, formulas, tables, routines, and tutorial articles. It should be a 'readable reference' well indexed and organized. Adding the articles and routines to Bill's material was a small attempt to illustrate the different types of material that a future PPC Programming Techniques Handbook could contain. One of the goals of this publication is to outline content and encourage members to committ to producing such a work. A few topic ideas are outlined below:

a. Indirect Addressing Techniques - Articles and routines to illustrate how this powerful instruction works and is applied to such applications as data input, data output, multiple indicies, and computed GO TO's.

b. Searching Techniques - Articles and routines to search registers for information, Table look-up, and compression of data into a form that is encoded for storage and decoded for output.

c. Sorting - Even with limited data storage, sorting is a real problem that should be discussed in PPC terms. Methods should be flow charted and plotted with run time vs number and type of data given.

d. Curve Fitting - All types of curves, the learning curve, the five 'standards', etc. should be discussed with equations and application provided.

e. Program Segment Sequence - Techniques with illustrated examples should be covered. Almost every 'average' program can be 'resequenced' to save steps or execution time.

f. Register Rotation - is a useful idea that should be explained. Often program steps can be saved by processing data in a set of registers, rotating the registers and repeating the process loop as required.

g. Convenience practices is a good topic for experienced programmers to write about. Use of ENTER, and R/S in data entry/correction, and the selection of logical key sequences for speed, convenience, standardization, and easy remembering.

h. Multifunction keys for getting more user definable keys for data entry. Having 30 or more inputs is possible with the right use of the user definable keys. Other techniques' include the use of the function keys themselves as input/prefix keys.

i. Routines - Basic calculator functions not implemented by firmware, clever use of firmware functions for other uses, etc. could be accomplished with an efficient routine. Examples appear in this book. The Handbook, however, should have a version for all machines.

j. Extended Precision - Techniques for reducing/

predicting errors in extended precision programs. Equations and methods of computing all mathematical functions to greater accuracy or for extended input ranges.

k. Constants - The common constants used in all fields. Tables of extended precision values for use in checking programs. Conversions, and other useful 'numbers' given for reference.

l. Machine Conversions - Program conversion from one machine to another can be full of putfalls for the novice. Precedures, or guidelines for converting programs from one machine to another, Considerations for converting programs from other sources should also be included.

m. Timing - The execution times for all instructions for all machines should be tabulated as Bill has done for the HP-67.

n. References - Books, and other publications that provide information usefull to programmers. Books on numerical methods, handbooks, mathematical dictionaries, text books, journal articles, etc. are all candidates for the reference section. Reviews, summaries, and recomendations for their use by members is most desirable.

o. Error Traps - Calculator useage that results in errors can be subtle enough to trap the user into thinking his answers are accurate. Methods and techniques that discuss the proper use of calculator form an accuracy viewpoint should be a part of the PPC Programming Techniques Handbook.

p. Machine Anomalies - Unusual machine behavior or unsupported features that enhance the use of the machine. Examples are NNN's, use of program steps to control display appearance, etc.

q. Hardware Modifications - Changes to HP PPC's that improve performance or increase user convenience. Adding a card write defeat switch or phase one interrupt switch are examples.

r. Accessories - Home made or modified commerical items that aid the programmer in accomplishing his programming task or making his using the calculator easier. Proven, quality commerical accessories available could also be included.

s. Software Hardware - Mechanical aids such as marking pens, flow charting templets, etc. are "hardware" items that make documentation of programs easier.

These topics are only a few examples of subject areas that should be in the Handbook. The PPC Programming Techniques Handbook should save the programmer time. Equations should be solved for all variables and given in "calculator solution" format. Many topics are often obvious and trivial to the proficient programmer, but the form of the information can make a difference. An example is the missing conditionals that Bill gives at the top of page 12. All the information is there, but the tables below present the same information in a "different" form. The proficient programmer recognizes that the so called inverse logic (the f^{-1} HP-65 function) is the logical negation of the do if true conditional. Adding this information to the tables makes them complete.

Logic	HOW IMPLEMENTED			
	Do If True	Remarks	(Inverse Logic) Do If Not True	Remarks
$x=y$ $x\neq y$	$x=y$ $x\neq y$	Keyboard Keyboard	$x\neq y$ $x=y$	Keyboard Keyboard
$x>y$ $x\geq y$	$x>y$ $x\neq y$ $x>y$	Keyboard Two steps required	$x\leq y$ $x\leq y$ $x=y$ f 2?	Keyboard Three steps required Flag must be clear when tested
$x<y$	$x\leq y$ $x=y$ f 2?*	Three steps required Flag must be clear when tested	$x\neq y$ $x>y$	Two steps required
$x\leq y$	$x\leq y$	Keyboard	$x>y$	Keyboard

* Use any flag as long as it is clear when tested.

Table 1. Conditionals for HP-67/97 - X and Y

LOGIC	HOW IMPLEMENTED			
	Do If True	Remarks	(Inverse Logic) Do If Not True	Remarks
$x=0$	$x=0$	Keyboard	$x\neq 0$	Keyboard
$x\neq 0$	$x\neq 0$	Keyboard	$x=0$	Keyboard
$x>0$	$x>0$	Keyboard	$x\neq 0$ $x<0$	Two steps required
$x\geq 0$	$x\neq 0$ $x>0$	Two steps required	$x<0$	Keyboard
$x<0$	$x<0$	Keyboard	$x\neq 0$ $x>0$	Two steps required
$x\leq 0$	$x\neq 0$ $x<0$	Two steps required	$x>0$	Keyboard

Table 2. Conditionals for HP-67/97 - X and 0

All references made herein of the form VnNnPn with out a specific source applies to "65 NOTES" if V4 or earlier and "PPC JOURNAL" if V5.

I hope you will find this book useful and that your programming time is reduced with its use.

Happy Programming
Richard Nelson

CONTENTS

INTRODUCTION

"Better Programming On The HP-67/97" is a collection of tips and techniques from 65 NOTES, KEYNOTES and "Old Timers" that save steps and speed up your programs. It is intended to help fill the gap between the Owner's Handbook and good programming. The kind of information you can expect to find belongs in the following categories:

1. *Time and step saving ideas*
2. *Commonly used routines*
3. *General information not covered in the owner's manual*

The ideas presented are collected from many sources. Although many are original, credit belongs largely to all of the ex-beginners who have contributed to HP KEY NOTES, HP-65 KEY NOTE, 65 NOTES, the PPC JOURNAL, and the HP Users Club. I have elected not to include the source of each item since most have been independently discovered by several people. If you see your favorite tip, I'm thankful for your sharing it; if not, it's because I haven't heard about it. Your comments and corrections are most welcome.

It is hoped that these pages will help you get more out of your programmable calculator and provide the right inspiration the next time you find yourself needing just a few more program steps.

William M. Kolb (265)
34 Laughton Street
Upper Marlboro, Maryland 20870
U.S.A.

Standard HP notation has been used as much as possible throughout these pages. There are two exceptions to be noted:

1. X∿Y *represents the command exchange X and Y*
2. *Inverse functions are written without the full superscript, e.g.,* TAN⁻ *represents* TAN^{-1}

STACK OPERATIONS

XYZT	Initial Order
XYTZ	R↑, R↑, X∿Y, R↑, R↑
XZYT	R↓, X∿Y, R↑
XZTY	X∿Y, R↓
XTYZ	R↑, X∿Y
XTZY	X∿Y, R↑, R↑, X∿Y, R↑
YXZT	X∿Y
YXTZ	X∿Y, R↑, R↑, X∿Y, R↑, R↑
YZXT	R↑, X∿Y, R↑, R↑
YZTX	R↓
YTXZ	R↓, X∿Y, R↓
YTZX	R↑, R↑, X∿Y, R↑
ZXYT	R↓, R↓, X∿Y, R↓
ZXTY	R↑, X∿Y, R↑
ZYXT	X∿Y, R↓, R↓, X∿Y, R↓
ZYTX	R↓, X∿Y
ZTXY	R↑, R↑
ZTYX	X∿Y, R↑, R↑
TXYZ	R↑
TXZY	R↓, X∿Y, R↓, R↓
TYXZ	X∿Y, R↑
TYZX	R↑, X∿Y, R↓
TZXY	R↑, R↑, X∿Y
TZYX	X∿Y, R↑, R↑, X∿Y

CONSTANTS IN PROGRAMS

Usual	*Improved*	*Remarks*
		Frequently used constants should be stored and recalled as they are needed. This is true for short constants such as 2, 5, 0.1, 0.5, etc., as well as long constants.
1	EEX	Faster; must not be followed by CHS.
1 0	EEX 1	Faster.
1 0 0	EEX 2	Faster and saves steps when exponent is greater than one.
0	CLX	Faster; X is lost.
1 8 0	π D←R	Fewer steps.
π 1 8 0 ÷	EEX D→R	
4 π x 3 ÷	2 4 0 D→R	$4\pi/3$
LBL A STO A RTN LBL B STO B RTN LBL C STO C RTN	LBL A STO A R/S STO B R/S STO C RTN	Use one label to store several constants.

Usual	*Improved*	*Remarks*
LBL A STO A RTN LBL B STO B RTN LBL C STO C RTN	LBL A STO A R/S STO B STO C	Use SST to store constants. If the main program follows the last STO, press R/S after the last constant is entered.
1 0 0 STO A 2 0 0 STO B	EEX 2 STO A ENTER + STO B	Use previous constants to compute new constant.
		Some constants can be computed in fewer steps than it takes to enter them. For example, 0.111111111 = 1/9. 0.7777777777 = 1/9 x 7. Note that in the second example if the computation is shortened to 7/9, the result will be 0.7777777778.
		The digits 0 through 9 in reverse order can be obtained by: 80 ÷ 81.
	5 LN	Conversion factor for miles to kilometers; error is less than one in ten thousand.
0	.	Use the DECIMAL to enter zero; it's faster.
LBL A STO A STO B STO C [main] [prgm]	LBL A RTN STO A STO B GSB n LBL n STO C [main] [prgm]	Use SST to store data and begin execution automatically. The main program must not contain subroutines or RTN's.

BUILT IN FUNCTIONS

Usual	*Improved*	*Remarks*
	X∿Y + LSTx R→P LSTx R→P x^2	In general, the built-in functions save steps but not time. The example shows how the following equation can be solved in seven steps assuming X is in the X-register and Y is in the Y-register: $X^2 + 2XY + 3Y^2$
1 0 0 ÷	EEX %	
1 . 0 1 x	EEX % +	
. 9 9 x	EEX % -	
ENTER x	x^2	Slightly faster and saves a step; the stack is not lifted, however.
	R→P	$\sqrt{x^2 + y^2}$; built-in conversion is slower.
2 STO-n RCL n X ≠ 0 GTO m	DSZ DSZ GTO m	Decrement by two and skip on zero.
1 STO+n X∿Y	ISZ	Iteration or loop count.

BUILT IN FUNCTIONS (CONT.)

Usual	*Improved*	*Remarks*
STO a X∿Y STO b . . RCL b RCL a	Σ+ . . . RCL Σ	Store and recall two numbers. R_{14} through R_{19} must be clear. If the technique is used more than once in a program, follow each RCLΣ with Σ-.
RCL 6 RCL 9 ÷ RCL 4 RCL 9 ÷	P∿S $\bar{x}$	Divide two numbers by the same value; R_4 and R_6 are divided by R_9 in both methods.
STO+4 1 STO+9	Σ+	Also sums contents of Y-register.
SIN LSTx COS	EEX R↔P	SIN and COS; X-register contains COS and Y-register contains SIN. TAN can be obtained by adding ÷ .
	RCL Σ Σ-	Clear R_{14} and R_{16} .
2 $\sqrt{x}$ x	ENTER R↑P	Multiply by $\sqrt{2}$; improved method is slower.

REGISTER ARITHMETIC		
Usual	*Improved*	*Remarks*
RCL n + STO n	STO+n RCL n	Faster and saves a step.
RCL n 0 STO n X∿Y	RCL n STO-n	Clear register after use.
RCL n 1 STO n X∿Y	RCL n STO÷n	Set register contents to one after use. Register must not contain a zero.

INDIRECT REGISTER OPERATIONS		
	LBL A RCL n X∿I R↓ STO (i) R↑ X∿I R↓ RTN	Using registers other than I for indirect storage. The number is stored in the register pointed to by n. I is restored.
	LBL A RCL n X∿I RCL(i) X∿Y X∿I R↓ RTN	Using registers other than I for indirect recall. The number is recalled from the register pointed to by n. I is restored.

CONDITIONAL BRANCHING	
X ≠ 0 X < 0	X ≤ 0
X ≠ 0 X > 0	X ≥ 0
X ≠ Y X > Y	X ≥ Y
X ≤ Y X = Y F2?	X < Y; Any flag may be used as long as it is clear when tested.
X ? Y CHS +	Add or subract depending on flag or test.
X ? Y 1/X x	Multiply or divide depending on flag or test.
X ? Y 1/X y×	Power or root depending on flag or test.
X = Y I(x) X ≠ Y F(x)	$I(x)^1$ or $F(x)^2$ depending on test. I(x) must leave X unchanged. Opposite tests are required.
FO? F(x) G(x) FO? LSTx	$F(x)^2$ or $G(x)^3$ depending on flag. G(x) must generate LSTx .
FO? F(x) H(x) FO? $H^{-}(x)$	$F(x)^2$ or $H(x)^4$ depending on flag. H(x) must have an inverse, $H^{-}(x)$.

CONDITIONAL BRANCHING (CONT.)

[1] *I(x) is any keycode that occupies one memory step and does not change the X-register:* STO, STO+, DSP, SCI, RAD, P↔S, ENTER, *etc.*

[2] *F(x) is any keycode that occupies one memory step.*

[3] *G(x) is any keycode that occupies one memory step and produces a LSTx:* LOG. SIN, 1/X, INT, ABS, *etc.*

[4] *H(x) is any keycode that occupies one memory step and has an inverse:* R↑/R↓, Σ+/Σ-, LN/e^x, COS/COS^{-1}, R↔P/R→P, DSZ/ISZ, 1/X, X↔Y

F3? F3?	Skip a step if flag is set. F2 or F3 may be used.
	Additional self-clearing flags can be created using DSZ, DSZ(i), ISZ, ISZ(i). Set these "flags" by storing ±1 in appropriate registers.
X?Y 1 .	Enter 0 or 1 depending on flag or test.
LBL A SF1 X=0 CF1 RTN	Set or clear flag by entering 0 or 1.
X?Y m . +	Add m if true; otherwise leave unchanged. m is a single digit entry; RCL will not work.
X?Y . 1 x	Divide by 10 if true; otherwise leave unchanged.
X?Y . 0 1 x	Divide by 100 if true; otherwise leave unchanged.

FLAG LOGIC

The Flag Logic Table can be used whenever it is desirable to have a test which skips (or does not skip) a particular program step depending on the status of two flags.

The various tests possible are listed at the top of the table where A and B represent the two flags. A is interpreted as Flag A SET and $\overline{A}$ is interpreted as Flag A NOT SET.

Several solutions are presented for each test; insert one of these keycode sequences immediately ahead of the program step which is to be skipped as a result of the test. A and B are shown in parentheses beside the flags they represent in the actual program.

F2 and F3 are self-clearing flags and may be used interchangeably in all tests. F0 and F1 are also interchangeable wherever they are used. Any comparison such as X<0, X≤Y, etc., can be substituted directly for F0 or F1 in a test. This technique is useful since it avoids the requirement for setting and clearing a flag.

Testing Flag Status

The following procedure will allow the testing of a flag(s) to determine if it (they) are set or cleared.

Press h, RTN*. Test flag from keyboard, i.e. press f, F? n. Switch to W/PRGM. If display shows step 001 instead of 000 flag tested was clear. Remember that if flags 2 or 3 were set, i.e. display showed 000 when switching to W/PRGM, they were cleared by testing and should be reset if desired. Flag 3 is set when any digit key is pressed or a number entered by a program.

* This procedure is recommended to avoid switching back and forth to W/PRGM. The program pointer will move one step anywhere in program memory if the tested flag is clear. Starting at the top of memory saves one switch movement.

NON-SKIP CASE SKIP CASE	A $\overline{A}$	$\overline{A}$ A	$\overline{A}$ or B A and $\overline{B}$	A and $\overline{B}$ $\overline{A}$ or B	$\overline{A}$ and B A or $\overline{B}$	A or $\overline{B}$ $\overline{A}$ and B	A and B $\overline{A}$ or $\overline{B}$	$\overline{A}$ or $\overline{B}$ A and B	$\overline{A}$ and $\overline{B}$ A or B	A or B $\overline{A}$ and $\overline{B}$	($\overline{A}$&B) or (A&$\overline{B}$) (A&B) or ($\overline{A}$&$\overline{B}$)	(A&B) or ($\overline{A}$&$\overline{B}$) (A&$\overline{B}$) or ($\overline{A}$&B)
	F0 (A) F0 (A) F3 F0	F3 (A) F3 F0 (A) F3 F0 F3 CF0 F1 (A) F0	F0 (A) F1 (B) F3 (A) F0 (B) F3 F0 F3 (A) F3 F0 (B) F0 F2 (A) F3 (B) F2 F3 F0 (A) F3 (B) SF3 F3	F0 (A) F1 (B) CF0 F0 F3 (A) F0 (B) F3 CF3 F0 (A) F1 (B) F3	F3 (A) SF3 F2 (B) F3 F2 F0 (A) CF1 F1 (B)	F0 (A) F3 (B) F3 F0 F0 (A) F1 (B) F1 F0 F0 (A) CF1 F1 (B) F0	F3 (A) F3 CF0 F0 (B)	F3 (A) F3 SF3 F0 (B) F3	F3 (A) F0 (B) F0 F3 F3 (A) F3 F0 (B) F3 CF3 F0 (A) F1 (B) F1 F3 F0 (A) SF3 F3 (B) F3	F3 (A) F3 F0 (B) F0 (A) SF1 F1 (B) F0 (A) F1 (B) F1 F0 F0 F0 (A) F1 (B) F0 F1 F1 F1 (A) F0 (B) F0 F1 F1	F0 (A) F3 (B) F3	F2 (A) F2 F3 (B) F3 F2 (A) F3 (B) F3 F2 F0 (A) F3 (B) F3 F3
	NORMAL	INVERTED					AND	NAND	NOR	OR	EXCLUSIVE OR	EQUIVALENCE

Note: F2 and F3 are test cleared flags

FLAG LOGIC

COMMON PROBLEMS

```
RCL θ
RCL a
R↔P
RCL b
-
R→P
```

Law of Cosines: $c = \sqrt{a^2 + b^2 - 2ab\cos\theta}$

```
RCL x
ENTER
ENTER
ENTER
RCL a
x
RCL b
±
x
RCL c
±
x
RCL d
±
```

Polynomials: $ax^3 \pm bx^2 \pm cx \pm d$

```
RCL b
RCL a
ENTER
+
÷
CHS
ENTER
x²
RCL c
RCL a
÷
-
√x
+
```

Quadratic Equation: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

The second root may be found by adding the following steps: X↷Y, LSTx, +

```
RCL a
RCL b
R→P
RCL c
R→P
RCL d
R→P
```

Square-root of Sum of Squares:

$$\sqrt{a^2 + b^2 + c^2 + d^2}$$

COMMON PROBLEMS (CONT.)	
COS^{-} SIN	$\sqrt{1 - x^2}$; $-1 < X < +1$
SIN^{-} TAN	$X / \sqrt{1 - x^2}$; $-1 < X < +1$
COS^{-} TAN	$\sqrt{1 - x^2} / x$; $-1 < X < +1$
TAN^{-} COS	$1 / \sqrt{1 + x^2}$; $-1 < X < +1$
TAN^{-} SIN	$X / \sqrt{1 + x^2}$; $-1 < X < +1$
RCL y RCL x R→P R↓	ARCTAN; avoids division by zero and distinguishes between (-y)/x and y/(-x).
x^2 LSTx + 2 ÷	$1 + 2 + 3 + \ldots + X$
x^2 LSTx + LSTx ENTER + EEX + x 6 ÷	$1^2 + 2^2 + 3^2 + \ldots + X^2$

COMMON PROBLEMS (CONT.)

	x^2 LSTx + 2 ÷ x^2	$1^3 + 2^3 + 3^3 + \ldots + X^3$
X∿Y ENTER ENTER R↑ ÷ LSTx X∿Y INT x -	÷ LSTx X∿Y FRAC x RND	Remainder of Y divided by X. Improved method is not as accurate when Y is large compared to X. DSP setting will also affect results.
ENTER ENTER RCL a ÷ INT RCL a x -	RCL a ÷ FRAC RCL a x RND	Remainder of X divided by a constant, "a".
	RCL A RCL B R→P LSTx R→P	$\sqrt{A^2 + 2B^2}$
	RCL a RCL b X>Y X∿Y RCL c X>Y X∿Y	Find the largest of three numbers; the routine is easily extended to four or more numbers. You can find the smallest of three numbers by substituting X≤Y for X>Y.

MISCELLANEOUS

Usual	*Improved*	*Remarks*
	SIN SIN⁻	Keeps angles less than 90 degrees.
	COS COS⁻	Keeps angles less than 180 degrees.
	RCL θ RCL r R↔P R→P	Keeps vector (r) positive.
2 x	ENTER +	Doubles the X-register; improved method is faster. LSTx will be different.
RCL I STO b RCL a STO I	RCL a X∿I STO b	Keeping two indexes.
SPACE	LBL x	NOP; x is any unused label. Program will be compatible with the HP-67 as well as the HP-97. If labels can't be spared, use DEG or DSP corresponding to the mode the calculator is in.
EEX 9 CHS +	$\sqrt{x}$ x^2	Introduce a small error in X; does not work if X is a perfect square. See TAN on p 20 for typical application
RCL a RCL b 2 ÷ y^x	RCL a RCL b y^x $\sqrt{x}$	$a^{b/2}$
y^x 1/X	CHS y^x	Faster.
EEX 6 CHS x	EEX 6 ÷	Saves a step.

MISCELLANEOUS (CONT.)		
Usual	*Improved*	*Remarks*
	h RAD D→R TAN h DEG	TAN ±90 degrees. Prevents overflow at multiples of ±90.
	$\sqrt{x}$ x^2 TAN	TAN +90 degrees. Prevents overflow at multiples of +90 .
0 ÷	GTO n	Generates an error message in the program; n is any unused label.
h BST SST	RUN W/PRGM	Cancels prefix key when pressed by mistake in the program mode .
h DEL	W/PRGM RUN	Cancels prefix key when pressed by mistake in the run mode .
	a + b ENTER c ENTER d ENTER e	Stores five numbers in the stack and LSTx. Assumes y=0, or x+y = desired number for "a".
	9 9 EEX 9 9	Generates largest number. A somewhat slower method which saves two steps is: 7 , 0 , N! 70! ≃ 1 sec 99! ≃ 1.4 sec
	NOP	You can create a NOP keycode on the HP-67 by switching to the W/PRGM mode and pressing GTO, DECIMAL, 2, then A and D simultaneously. A 32 24 should appear in the display. This instruction executes in 7 msec. making it the fastest NOP available.

MISCELLANEOUS (CONT.)		
1 ENTER ENTER RCLn	EEX EEX RCL n	Faster and saves a step.
	RCL θ 1 CHS R↔P R→P X∿Y CHS	Supplement of an angle (θ + Supplement θ = 180) $-\infty < \theta < +\infty$.
	X?Y . 1 0 x	Divide by 10 if true; multiply by 10 if false.

ROUNDING		
	ENTER FRAC + INT	Rounds both positive and negative numbers.
	RND H← H.MS←	Eliminates display of 60 minutes and 60 seconds. Also reduces 60+ minutes and seconds to a value less than 60. Mostly used in DSP2 and DSP4 mode.

GENERAL INFORMATION
Use the stack to simplify computations (cf. Polynomials and Quadratic Equation under COMMON PROBLEMS).
Use LSTx to save registers whenever possible.
Conserve registers by reusing them when the data is perishable .
More than one number can be stored in a single register. Two five-digit numbers, for example, can be stored as the INT and FRAC parts of a register.
In programs with a large number of iterations, look for ways to use the previous calculations in order to save steps. Refer to the TIMING tables and pick keycodes that will save time.
Avoid the use of $X \leq Y$ and $X > Y$ whenever possible to reduce execution time.
When using comparisons, the most probable event should test true to reduce execution time.
Speed up programs by avoiding the use of subroutines as much as possible.
Keep labels as close to the call as possible.
Labels can be used more than once in a program. The first occurrence of the label after its call will be used .
Figure out a logical output sequence and structure the program to print in this order. It will help the next time you use the program.
When writing a family of programs or programs that use identical routines, keep the routines exactly the same and use the same labels. Use the same registers for the same purpose in each program.

GENERAL INFORMATION (CONT.)

Debugging programs is easier if R/S is used throughout at logical checkpoints. Once debugged, the R/S keycodes are easily deleted.

Document any tricks or exotic routines or no one, including yourself, will understand the program six months later.

When inserting or deleting steps from a program, begin with the highest step number and work backwards.

Use an AM radio near the calculator to tell when a long program has finished.

Magnetic cards can be marked with a PENTEL P335 film marking pencil. It is non-smearing, erasable and convenient to carry.

Cards can be permanently marked with the SANFORD No. 3000 SHARPIE soft-tipped pen.

Cards which have been protected by clipping the corner can be re-recorded. Prepare a magnetic card as shown using a 45 degree cut. Insert this card in the left side of the calculator as far as it will go. Switch the calculator to W/PRGM or W/DATA and enter the "protected" card in the normal fashion.

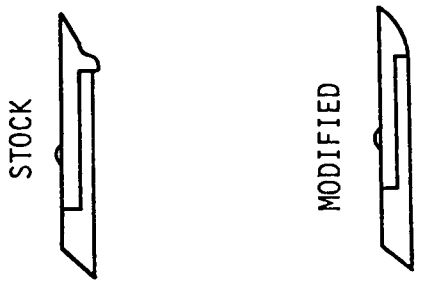
Uneven printing on the HP-97 is usually due to deposits on the print-bar which holds the paper against the print head. Slide the plastic window straight up and out. Use a wooden toothpick to gently rub any build-up from the print-bar. Solvents are not necessary.

GENERAL INFORMATION (CONT.)

Darker printing can be obtained by removing the plastic window on the printer. Slide it straight up and out.

Rockwell and Texas Instruments thermal paper can be substituted for HP Thermal Paper. Be sure to check the width since it comes in different sizes. TI paper usually gives better contrast than the HP paper.

The plastic window on the HP-97 printer obscures one line of print when listing the stack. This can be fixed by sliding the window out and filing off the lip. Polish the edge smooth using polishing compound and emery cloth. Toothpaste can be substituted for polishing compound.



Extraneous steps in memory after the end of your program will slow execution. An easy way to delete these steps is to prepare a special "blank" card. Clear memory and enter a zero in step 224. Record this program. To delete unused steps in your program, go to the last program step, switch to the RUN mode, press MERGE, and enter the blank card.

f LBL x is preferred over LBL x to reduce execution time. Use LBL A, LBL B, etc., only when you have used all LBL a, LBL b, LBL c, etc.

When print commands are used within an iterative loop, keep as many program steps as possible between print statements. This will help reduce execution time since the program must halt for the printer whenever more than one output is queued up.

Always handle magnetic cards by the edges to avoid finger-prints on the magnetic surface.

GENERAL INFORMATION (CONT.)

The prefix key can be canceled on the HP-97 by simply depressing DSZ. This must not be followed by (i) however.

1	.	Always use multiplication instead of division whenever possible to save execution time
0	1	
÷	x	

X⇆Y	X⇆Y	Use the stack as much as possible to save execution time. The seven step solution requires 2204 msec. while the nine step solution requires only 465 msec. (almost five times faster. The example is from p.9.
+	+	
LSTx	LSTx	
R→P	x^2	
LSTx	ENTER	
R→P	+	
x^2	X⇆Y	
	x^2	
	+	

$$X^2 + 2XY + 3Y^2$$

CARE AND MAINTENANCE
Preserve battery power when the calculator is in use by pressing the decimal key. Use CLX to resume.
The charger may be connected or disconnected from the calculator while it is still running. Be sure that the charger is first plugged into an AC outlet and that the power is on.
Obtain maximum battery life by operating the calculator until the low-battery indicator comes on before re-charging.
Calculators should not be left on charge for more than fourteen to sixteen hours.
Batteries should not be discharged further once the low battery indicator comes on.
When batteries are not used for extended periods, they should be recharged every 30 to 45 days.
You can avoid accidentally running the batteries down by placing the battery pack upside down in the calculator with the metal contacts facing out.
When using the HP-67 (or HP-65) for extended periods on AC, a transient spike protector will provide extra insurance against logic damage due to fluctuations in the line voltage. Use GENERAL ELECTRIC GESP-752.
When replacing the HP-67 in its case, the display should face away from the inside pocket.
Never touch the display with your fingers. Fingerprints can be removed with a Q-TIP using a light rubbing action in one direction only. Moisten the plastic first by breathing on it.
Remove minor scratches from the display using a soft cloth or Q-TIP and MIRROR GLAZE PLASTIC CLEANER MGH 17. For deeper scratches, use MICRO MESH SCRATCH REMOVER KIT.

TIMING FOR THE HP-67/97

Instruction	*Typical Time(msec)*	*Instruction*	*Typical Time (msec)*
ENTER	32	STO n	35
EEX	48	RCL n	37
CHS	37	STO (i)	90[1]
CLX	30	RCL (i)	100[1]
Digit Entry	72 per digit		
		STO + n	70[2]
R↑	33	STO - n	70[2]
R↓	33	STO x n	100[2]
X ∿ Y	34	STO ÷ n	130[2]
X ∿ i	35	STO + (i)	115[1]
P ∿ S	91	STO - (i)	128[1]
LSTx	35	STO x (i)	130[1]
		STO ÷ (i)	178[1]
+	50[2]		
-	50[2]	DSZ	60
x	107[2]	ISZ	60
÷	122[2]	DSZ (i)	117[1]
		ISZ (i)	117[1]
SF	40		
CF	40	INT	45
F?	36 (set)	FRAC	50
F?	43 (clear)	RND	65 (FIX) 64 (SCI) 131 (ENG)
X = 0	39 (F) 33 (T)		
X ≠ 0	39 (F) 33 (T)	LBL A	28
X > 0	39 (F) 33 (T)	LBL n	28
X < 0	40 (F) 34 (T)	LBL a	32

Instruction	*Typical Time (msec)*	*Instruction*	*Typical Time (msec)*
X = Y	39 (F) 34 (T)	GTO (i)	110[1]
X ≠ Y	39 (F) 34 (T)	GTO A LBL A	160[3]
X > Y	68 (F) 61 (T)		
X ≤ Y	69 (F) 62 (T)	GSB A LBL A RTN	240[3]
Σ+	350		
Σ-	290	LN	530[4]
RCL Σ	32	e^x	340[4]
DSP n	40	LOG	620[4]
DEG	30	10^x	320[4]
RAD	30	x^2	90
		$\sqrt{x}$	120[4]
1/x	130		
Y^x	650[4]	PAUSE	1250
ABS	32	π	42
		REG	28000
D←R	180		
D→R	240	%	100
H←	80	Δ%	150
H.MS←	50	H.MS+	150
		N!	320
SIN	930 (D) 770 (R)[4]		
SIN^-	850 (D) 760 (R)[4]	DECIMAL	42
COS	960 (D) 780 (R)[4]		
COS^-	850 (D) 760 (R)[4]		
TAN	670 (D) 500 (R)[4]		
TAN^-	590 (D) 500 (R)[4]		
R←P	1050 (D) 920 (R)[4]		
R→P	980 (D) 920 (R)[4]		

[1] *Execution time increases as the value of (i) increases. The typical variation was plus 40% to minus 35% from the value given in the table. An exception was GTO (i) where the variation was only plus 15 msec. to minus 3 msec. about the mean value given.*

[2] *Operations with +, -, x, and ÷ varied plus or minus 16% about the mean value given depending on the sign and magnitude of the arguments.*

[3] *Execution times for these instructions were dependent on the location in memory as well as the number of steps between the label and the call. The minumum and maximum times measured for GTO A . . . LBL A were 101 msec. and 218 msec.*

[4] *Transcendental functions had the greatest variation. Depending on the size of the argument(s), execution times ranged between ±50% of the typical value given.*

Typical times presented in this table are used to compare relative speed advantages of alternative routines. The actual speed will be dependent on the particular arguments used. The times shown will also vary depending on the speed of your calculator. You can calibrate your calculator by loading the program memory with (+). Fill the stack with 1's and press R/S. The count at the end of one minute should be 1175. Divide the count you actually obtained by 1175 to obtain a correction factor for the times given.

INDEX

INDEX

INDEX

RPN AND THE STACK

OR

HOW TO TALK TO YOUR CALCULATOR

We are accustomed to writing mathematical expressions such as the sum of A and B with the aid of general symbols (usually an English or Greek letter) and operators such as +, -, x and ÷. The notation A+B, which represents the sum of A and B, is called infix notation because the operator + is in-between the two operands. This convention has been in popular use since the development of algebraic symbolism in the 17th century.

The Polish mathematician, Lukasiewicz, developed another type of notation around 1951 for sequential calculus called prefix notation. In prefix notation, the operator precedes the two operands and our sum is represented as +AB. This is often referred to as Polish notation in honor of Lukasiewicz.

Reverse Polish Notation (RPN), as you might expect, is written with the operator following its operands. In RPN, or suffix notation, the sum becomes AB+. It is worth pointing out that this notation is parenthesis free, since parentheses are not required to indicate the order in which the operations are to be performed. For example, Ax(B+C) becomes ABC+x while (AxB)+C becomes ABxC+.

RPN is an ideal convention for computers because it presents the entire mathematical expression in a way which is easily compiled. RPN is therefore almost universally used to compile statements presented in infix or algebraic notation.

The method of compiling is to scan the expression from left to right, placing the operators and operands in a "push-down" list depending on the precedence of the operators and the parentheses. An operation which must be delayed because of imbedded parentheses causes the stack to be pushed down. A close-parenthesis, on the other hand, causes the list to "pop-up." This last-in-first-out philosophy is why you may have heard that computers execute a statement from right to left. Notice that the parentheses themselves are not stored in the list. They merely signal the next action to be taken.

With the HP calculator, you are the "compiler" and you decide when the list is to be "pushed-down" or "popped-up." Fortunately the simplicity of RPN and the sophistication of the calculator require you to learn only four easy-to-remember steps. The only bad thing is that the terminology is somewhat reversed from what we've been using. Instead of a push-down list, we have a stack which is raised or lifted when data is entered. And instead of popping-up when an operation is performed, the stack contents drop. A flow diagram nicely illustrates the rules that are followed in order to evaluate most expressions with this system.

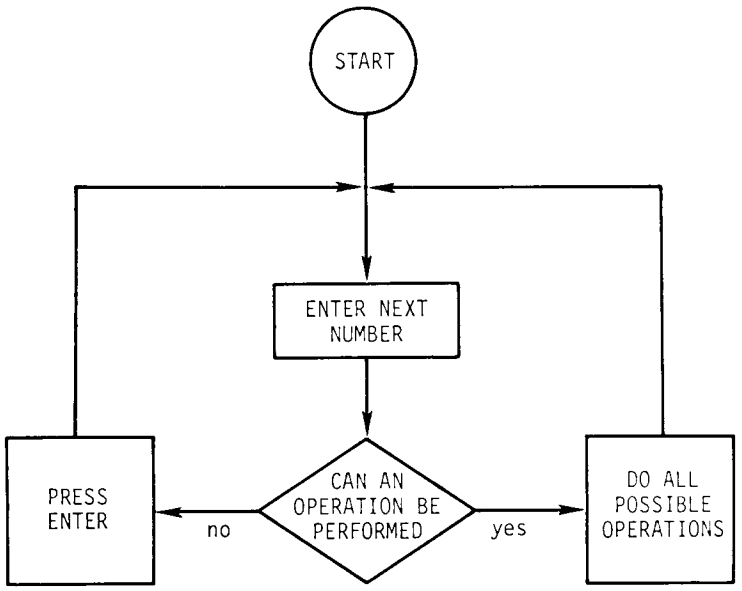


Figure 1. Procedure for Evaluating Expressions in RPN.

The advantages of this system are several fold:

1. Each problem is consistently solved the same way, without regard to complex hierarchy rules or parentheses.
2. You see every intermediate answer so that you can check your progress.
3. You can easily recover from an error since no more than one operation is performed at a time.
4. Intermediate results need not be stored or written down since the stack automatically lifts or drops as required.

Of course there is a limit to the complexity of problems which can be solved entirely within the stack. The limitation is that the stack holds no more than four numbers at one time. Thus, there can be no more than three pending operations which would cause the stack to drop. Such operations, which involve two operands, are called "diadic" and consist of +, -, x, ÷ and certain special functions. An unlimited number of single operand or "monadic" operators such as LOG, COS, ABS and FRAC can be pending since these do not cause the stack to drop.

It is interesting to note that even calculators with parentheses treat monadic functions the same way. In order to evaluate COS(A+B), for example, you first add A and B, then take the COS. If this sounds like RPN, you can see why it might be an advantage to solve every problem the same way. With RPN, the operation is executed as soon as the function key is pressed. There are no exceptions.

The stack in HP calculators is organized as four tandem registers lettered X, Y, Z and T (see Figure 2). When you key in a number, it goes into the X-register which is the only one displayed. When you press ENTER, a copy of the number in X is transferred to Y. Simultaneously, the number in Y is transferred to Z, and Z into T. The original number in T is lost. At this point, the "stack-lift" feature of the calculator is disabled so that the next number keyed in writes over X without causing another sequence of transfers.

Diadic operators always use the operands in the X and Y registers. The result of the operation is placed in X, while Z drops down to Y, and T drops to Z. The T-register is not cleared as you might expect. Instead, T replicates itself any time the stack drops.

Three commands are programmed into the calculator which permit the stack to be reorganized. These are:

1. R↑ (roll-up) which causes the contents of each register to be moved up one, with the number in T rotated to X.
2. R↓ (roll-down) which causes the contents of each register to move down one, with the number in X rotated into T.
3. X⇆Y (exchange) which causes the number in X to be switched with the number in Y.

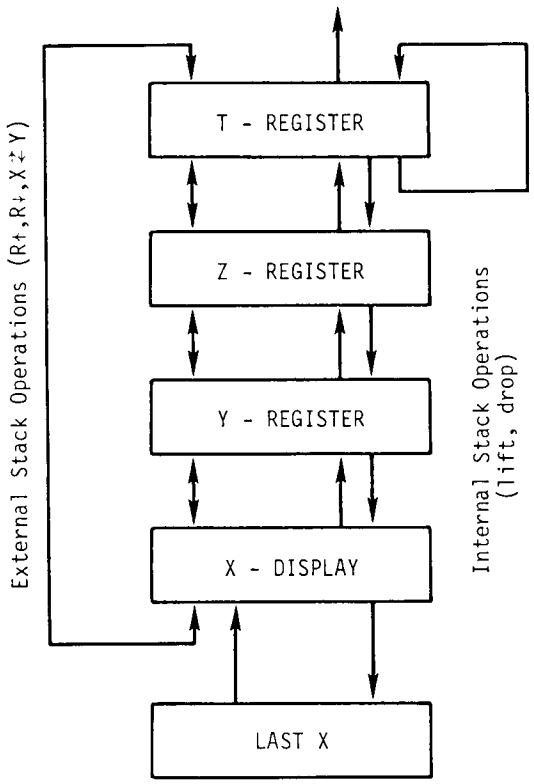


Figure 2. Stack Concept for RPN

These features are useful for reviewing the stack contents and for rearranging the operands into any desired order. Exchange is particularly useful in this regard since the order of the operands in the stack at any given time may be different from what is required for subtraction, division, exponentiation and certain special functions such as R→P, Σ+, and %. Rearranging the stack is not always easy, however, and sometimes requires as many as six separate commands, e.g. changing XYZT to YXTZ

The CLX and LSTx keys are also connected with stack operations. CLX is used to clear an error by replacing the contents of X, the display register, with zero and disabling the stack-lift so that the next entry will write over the zero. It is also a useful way to enter a zero into the stack.

LSTx is a special register that temporarily preserves the number in X when any monadic or diadic operation is executed in the stack. A command such as STO+5, however, does not operate on X and Y in the stack and therefore does not affect the LAST X register. A simple way to remember when LSTx is affected is that any arithmetic operation which changes the stack contents, also changes LSTx.

Since the calculator automatically raises and drops the stack, we must anticipate the correct response every time if we are to use it efficiently. There are only three cases to consider: 1) operations that enable the stack-lift, 2) operations that disable the stack-lift, and 3) operations that have no affect on the stack. Since the first category comprises most operations on the calculator, it is easier to remenber the operations in the other two categories and recall the first by exception. In order to determine what effect a given key will have on the stack, we must know the status of the stack-lift mechanism. Whenever the lift is disabled, a data entry will write over the current contents of the display.

There are only four operations to remember which disable the stack-lift: CLX, ENTER, Σ+ and Σ-. A number keyed in or recalled immediately after one of these operations writes over the number in X and does not lift the stack. Note that the stack-lift is only disabled for a single step and the next key will enable the lift unless it's also one of the four.

It is generally safe to assume that a number keyed in after any other operation will lift the stack. There are nine instructions to remember, however, that do not affect the stack contents even though a number is keyed in. These are FIX, SCI, ENG, DSP, STO, EEX, 0 through 9, DECIMAL, and CHS (when followed by a decimal, EEX or 0-9). Each of these instrucitons are used for inputting or displaying numbers in the X-register. Even though the stack is not lifted by these instructions, the stack-lift remains enabled.

An interesting feature of the calculator worth mentioning is the use of EEX to enter a number. EEX is normally used to format the exponent field and, like the digits 0-9, does not raise the stack when preceded by 0-9. When it is not preceded by 0-9, however, a one is assumed in the mantissa and the stack is lifted. This is the only key besides ENTER and RCL which can be used to fill the stack directly.

A few words are also in order with regard to the ENTER key. The primary function of ENTER is to tell the calculator logic where one number begins and another ends in a string of digits being input. A secondary feature is that it can be used to raise a number in the stack without having to reenter each digit again.

It may seem inconsistent at first that ENTER lifts the stack and at the same time disables the stack-lift. This need not be confusing if you always distinguish between a key's primary function and its secondary affect on the status of the stack-lift.

Normally, you should only be concerned with the stack-lift status after using ENTER, CLX, $\Sigma+$ or $\Sigma-$ when entering or recalling a number does not produce the usual lift.

These stack-lift basics must be mastered for all but the most routine calculations. In many cases, solving problems entirely within the stack is fastest in terms of execution time. Some extra effort at organizing data entry and the order of calculation is therefore worthwhile. There are certain situations, however, which prevent a stack solution, viz. when one or more variables are used repeatedly in the calculation, and when there are more than three pending diadic operations. In either case, try factoring the expression before programming. Sometimes expanding the factors and recombining terms will lead to a shorter solution. Next, try to order the sequence of calculation to take advantage of LSTx. The self-replicating feature of T may be used for temporary storage as well as a method for having more than three pending operations. Failing this, you would resort to STO and RCL. Remember that each STO adds an extra step when compared to LSTx. Using the T-register may also require extra steps in order to rearrange the stack. Don't forget that register arithmetic is another way to reduce the number of pending diadic operations.

There are no shortcuts for minimizing the number of steps required, so use the approach that seems most comfortable. It is usually a good idea to begin with the most complicated part of the expression or the inner-most set of parentheses. Bear in mind that this is the same approach you would use when solving the problem by hand. When it is absolutely necessary to make a program shorter, practice and perseverance are your best assets. Also remember that the shortest solution is not always the fastest and speed should be your primary concern in programs that loop many times.

It has been siad that the best study of programming is programming. With this philosophy in mind, we shall explore several ways that the stack can be used to evaluate an expression. The problem selected for this purpose is:

$$x^2 + 2xy + 3y^2$$

We will assume in each case that x is in the X-register and y is in the Y-register at the start. It is a good idea for the beginner to write out the contents of the stack at every step so we will follow this procedure also. This will familiarize you with stack operations and help you remember where things are in the stack at any point in the program. The same technique is also helpful for the experienced programmer when trying to reduce the number of steps and in debugging complicated programs.

In our first solution, we will solve the problem as it is written using storage registers. This will hopefully give us some insight into other ways the problem could be solved.

KEY	ENTRY	STACK CONTENTS X	Y	Z	T	LAST X	STACK LIFT DIS.*
001	STO A	x	y	-	-	-	NO
002	X∿Y	y	x	-	-	-	NO
003	STO B	y	x	-	-	-	NO
004	x^2	y^2	x	-	-	y	NO
005	3	3	y^2	x	-	y	NO
006	x	$3y^2$	x	-	-	3	NO
007	RCL A	x	$3y^2$	x	-	3	NO
008	RCL B	y	x	$3y^2$	x	3	NO
009	x	xy	$3y^2$	x	x	y	NO
010	2	2	xy	$3y^2$	x	y	NO
011	x	2xy	$3y^2$	x	x	2	NO
012	+	$2xy+3y^2$	x	x	x	2xy	NO
013	RCL A	x	$2xy+3y^2$	x	x	2xy	NO
014	x^2	x	$2xy+3y^2$	x	x	x	NO
015	+	$x^2+2xy+3y^2$	x	x	x	x^2	NO

Execution Time: 960 msec. * DISABLED

Now we should try factoring the expression to avoid the use of storage registers. Rewriting the formula using the associative and distributive laws of algebra produces:

$$\begin{aligned} & x^2 + 2xy + 3y^2 \\ = \; & x^2 + (2x+3y)y \\ = \; & x^2 + (2x+2y+y)y \\ = \; & x^2 + (2(x+y) + y)y \\ = \; & x(x) + ((x+y) + (x+y) + Y)y \end{aligned}$$

While this may look more complicated, it suggests a way to use the T-register to hold x^2 and LSTx to hold y. A program that does this might be written as follows:

KEY	ENTRY	STACK CONTENTS X	Y	Z	T	LAST X	STACK LIFT DIS.*
001	ENTER	x	x	y	-	-	YES
002	ENTER	x	x	x	y	-	YES
003	x	x	x	y	y	x	NO
004	R↓	x	y	y	x^2	x	NO
005	+	x+y	y	x^2	x^2	x	NO
006	ENTER	x+y	x+y	y	x^2	x	YES
007	+	2(x+y)	y	x^2	x^2	x+y	NO
008	X∿Y	y	2(x+y)	x^2	x^2	x+y	NO
009	+	2x+3y	x^2	x^2	x^2	y	NO
010	LSTx	y	2x+3y	x^2	x^2	y	NO
011	x	$2xy+3y^2$	x^2	x^2	x^2	$2xy+3y^2$	NO
012	+	$x^2+2xy+3y$	x^2	x^2	x^2	$2xy+3y^2$	NO

* DISABLED

Execution Time: 612 msec.

Encouraged by saving three steps and one-third of a second, we renew our attack on the problem. We notice that another set of factors can be obtained by setting apart $2y^2$:

$$\begin{aligned} & x^2 + 2xy + 3y^2 \\ = \; & x^2 + 2xy + y^2 + 2y^2 \\ = \; & (x+y)^2 + 2y^2 \\ = \; & (x+y)^2 + (2y)y \end{aligned}$$

Now we don't have to worry about storing x and LSTx can be used as before to temporarily store y.

KEY ENTRY		STACK CONTENTS				LAST X	STACK LIFT DIS.*
		X	Y	Z	T		
001	X∿Y	y	x	-	-	-	NO
002	ENTER	y	y	x	-	-	YES
003	+	2y	x	-	-	y	NO
004	LSTx	y	2y	x	-	y	NO
005	x	$2y^2$	x	-	-	y	NO
006	X∿Y	x	$2y^2$	-	-	y	NO
007	LSTx	y	x	$2y^2$	-	y	NO
008	+	x+y	$2y^2$	-	-	y	NO
009	x^2	$(x\ y)^2$	$2y^2$	-	-	x+y	NO
010	+	$X^2+2xy+3y^2$	-	-	-	$(x+y)^2$	NO

* DISABLED

Execution Time: 517 msec.

Two more steps and one-tenth of a second are saved by this approach. Anyone expounding the law of diminishing returns will quit at this point and get on with the number crunching. A few individuals (notably those who have one step too many in their program, those who can't sleep will until its perfect, and authors of articles such as this) will be persuaded to have yet another try. At this point, the experienced programmer recognizes that he's got about all the mileage possible from factoring and it's a choice of doing things in a slightly different order or of finding a new approach altogether. Using the previous results, we try factoring $2y^2$ another way:

$$\begin{aligned} & x^2 + 2xy + 3y^2 \\ = \ & (x+y)^2 + 2y^2 \\ = \ & (x+y)^2 + y^2 + y^2 \end{aligned}$$

KEY ENTRY		STACK CONTENTS				LAST X	STACK LIFT DIS.*
		X	Y	Z	T		
001	X∿Y	y	x	-	-	-	NO
002	+	x+y	-	-	-	y	NO
003	LSTx	y	x+y	-	-	y	NO
004	x^2	y^2	x+y	-	-	y	NO
005	ENTER	y^2	y^2	x+y	-	y	YES
006	+	$2y^2$	x+y	-	-	y^2	NO
007	X∿Y	x+y	$2y^2$	-	-	y^2	NO
008	x^2	$(x+y)^2$	$2y^2$	-	-	x+y	NO
009	+	$x^2+2xy+3y^2$	-	-	-	$(x+y)^2$	NO

* DISABLED

Execution Time: 465 msec.

Our best efforts have shaved-off one more step and reduced the execution time by a mere 50 miliseconds. The number-crunchers probably feel vindicated. Meanwhile, the insomniacs have come up with a seven step solution that requires about 2,200 milliseconds to execute. But before you get carried away and attempt to duplicate this "feat," you had best make sure you have a firm grasp of stack basics. When you can complete the missing steps in the following program, you will be well on the way to mastering RPN and the stack

W.M. KOLB (265)

Now that you are familiar with RPN, try using the techniques described to solve the following problems and compare your solutions with the ones given.

KEY ENTRY		STACK CONTENTS				LAST X	STACK LIFT DIS.*
		X	Y	Z	T		
001		x	y	-	-	-	NO
002		x^2	y	-	-	x	NO
003		y	x^2	-	-	x	NO
004		y	x^2	-	-	x	NO
005		y	y	x^2	-	x	YES
006		y	y	y	x^2	x	YES
007		x+y	y	y	x^2	x	NO
008		x+2y	y	x^2	x^2	x+y	NO
009	RCL 1	x+y	x+2y	y	x^2	x+y	NO
010		2x+3y	y	x^2	x^2	2x+3y	NO
011		$2xy+3y^2$	x^2	x^2	x^2	2x+3y	NO
012		$x^2+2xy+3y^2$	x^2	x^2	x^2	$2xy+3y^2$	NO

* DISABLED

EXERCISE: Fill in the Missing Key Entries

RPN PRACTICE PROBLEMS

1. $$\sqrt{5\left[\left(\left[\left[0.2\left(\frac{400}{661.5}\right)^2+1\right]^{1.4/4}-1\right]\frac{29.96}{15}+1\right)^{0.286}-1\right]}$$

2. $$\left(\frac{1+a}{1+b}\right) - \left(\frac{1-a}{1-b}\right)$$

3. $$a\,\frac{(1-r^n)}{(1-r)}$$

4. $$\frac{i \cdot P}{1-(1/(1+i)^n)}$$

5. $$n\left[30+\left(\frac{M-1}{1.5}\right)30\right] + 4\left[30+\left(\frac{M-1}{1.5}\right)30\right] + 0.4\left[30+\left(\frac{M-1}{1.5}\right)30\right]\frac{T}{M} + 2T + \frac{22T}{nM}$$

SOLUTIONS TO RPN PRACTICE PROBLEMS

1. Indicated air speed to MACH number. Start with the inner-most parentheses and work outwards. Storage registers are not required. (29 Steps)

400	x	÷	15	1
ENTER	1	y^x	÷	-
661.5	+	1	1	5
÷	1.4	-	+	x
x^2	ENTER	29.96	0.286	$\sqrt{x}$
0.2	4	x	y^x	

2. Expand and recombine factors to obtain $2(a-b)/(1-b^2)$; use LSTx to store b temporarily. (11 Steps)

a	1	÷
ENTER	LSTx	2
b	x^2	x
-	-	

3. Sum of geometric series. Use LSTx to store r. (13 Steps)

1	1	-	x
ENTER	LSTx	X∿Y	
r	n	÷	
-	y^x	a	

4. Amount of periodic payment. Use LSTx to store the interest. Remember that EEX can be used to enter the one and does not disable the stack lift. Also note that (CHS, y^x) was used instead of (y^x, 1/X) because it's slightly faster and the HP-80 does not have the reciprocal function. (13 Steps)

P	EEX	n	÷	
ENTER	EEX	CHS		
i	LSTx	y^x		
x	+	-		

5. Cost of a satellite program. Factor out common terms to get:

$$\{30+20(M-1)\}\ \{n+4+0.4\frac{T}{M}\} + T(2+\frac{22}{nM})$$

Use storage registers and register arithmetic. (26 Steps) The Problem can be solved in 30 steps using only one register.

T	STO 3	RCL 2	+	RCL 1
STO 1	0.4	1	x	+
STO+1	x	-	RCL 3	
M	n	20	22	
STO 2	STO÷3	x	x	
÷	+	30	+	

AOS VS RPN

AOS is a Trade mark of Texas Instruments and all references to AOS refers to Texas Instruments' trade mark.

The first scientific calculator was the HP-35, announced in January 1972. The HP-35 utilized a four leval stack and Reverse Polish Notation, RPN. Much later when Texas Instruments entered the market place they advertised what they called AOS , or Algebriac Operating System. The first PPC, the HP-65 was introduced in January 1974. It, like all HP consumer calculators, used RPN. In September 1975 T.I. announced the SR-52 which used an advanced AOS system. The SR-52 AOS is what is in common use today and the term advanced has been dropped.

Prior to advanced AOS it was common practice to compare the two systems in terms of keystroke efficiency. RPN won hands down, and when advanced AOS came along, the differences shrunk to "minor differences."

In terms of keystroke efficiency, recent published work, and official statements by T.I. software managers have laid the keystroke efficiency question to rest. John Ball, in his book "Algorithms For RPN Calculators", shows the numerical differences in keystorkes between RPN and AOS for over 250 basic operations involving four variables. When the keystroke count is compared RPN uses 8% fewer keystrokes on the average. This, coupled with T.I.'s admission that RPN is more efficient in keys pressed to solve problems, should remove this aspect from system comparisons. For all practical purposes both systems are nearly equal with RPN having a slight edge.

A calculator must have adequate information to know when data entry is complete, what operations to perform, and in what sequence. To separate two numbers the machine must know when the digit entry is terminated. On RPN machines, number termination is done with the pressing a function, a stack operation or ENTER key. See your owner's handbook for details. On AOS machines it is primarily the function keys that terminate digit entry. It is the method of determining the sequence of operations that makes the primary difference between AOS and RPN.

In the TI system, the operation is appended to the number when it is placed in the stack. In the HP RPN system, only numbers are placed in the stack.

A.O.S. - Operations appended to numbers in stack.
RPN - Only numbers are stored in the stack.

Now you know the difference. There is more, but the fundamental difference is the stack and how it operates.

Before we go into more detail let's agree on terminology. "AOS vs RPN System" would have been a more accurate title for this discussion, but since most writers imply, whether they realize it or not, that when they talk about RPN they mean HP's RPN system; i.e. automatic stack, full stack control, and Last x. The question often asked is : Which is best? Ask an AOS user and he will say AOS is best, ask an RPN user and he will say RPN is best. Ask either user to switch systems, and they would rather fight than switch.

At the risk of being called what ever technically minded people call those they dispise when they are angry, I will answer the question. The best system to use will depend on the application, and the type of person who is going to use it. I will explain, but before we start classifying users and applications, let's review some more fundamentals.

This is not intended to be a calculator architecture primer, but we must mutually understand a few more terms: Parenthesis and Pending Operations. When you compare specific machines it is a lot like comparing trucks. You obviously can get more furniture in a truck with a ten foot bed than one with a four foot bed. The truck with the four foot bed may be stronger or faster, but its capacity to hold furniture is obviously less. The stack on all HP personal calculators has four registers. The stack on AOS machines typically ranges from five to ten. The numbers that can be held in the stack for later use determine the number of pending operations the machine can hold. All HP personal calculators can

hold three pending operations. Two AOS machines will be used to illustrate TI's AOS philosophy, the TI-30 and the TI-59. The table below gives the data.

MODELS	PENDING OPERATIONS	PARENTHESIS	STACK
Any HP	3	0	4
TI-30	4	15	5
TI-52/59	9	9	10

Table 1. RPN and AOS stacks compared

On HP machines the stack is automatic and because the user supplies the order of operations, parenthesis are not needed. From a mathematics point of view most users know what a parenthesis is, and why they are used. A left parenthesis following a +, -, x, ÷ raises the stack. A right parenthesis or = drops or collapses the stack. One consideration to always keep in mind when comparing machines is to compare equal stacks. Just like it is unfair to compare HP's four high stack with Nationals NOVUS machines with a three high stack, it is likewise unfair to compare TI machines with five or ten high stacks with HP's four high stack.

One of the disadvantages of TI's AOS is the need for remembering which operations are performed first, and how many parenthesis you have opened and where. There is a certain amount of memory work inherent in any complex problem, but the stack concept of locations, or compartments where data are located, is easier for most people who will spend a few minutes getting use to HP's RPN concept. The locations, or registers used in the RPN system are independent of any problem data. They are always there, and the user simply puts numbers in and performs operations on them, and takes them out. There are a few "rules" to remember which AOS proponents will argue are just as complex as the hierarchy rules, but I don't agree. It is easier for a non-mathematics person to follow the rules of raising the stack, replicating T into Z, $x \rightleftarrows y$, etc. because these can be visualized as physical operations of the four locations that comprise the stack. The most complex operation to keep straight is that the y register is acted upon by the x register. The value of x is added to y, the value of x is taken away from y, etc.

Once the operation of the stack is mastered the user no longer thinks about operations, he thinks about his data and what he wants to do with it. Th ability to dynamically move the data freely gives a freedom of problem solving that can only be appreciated when the user has had problem solving experience on an actual machine. The operations appended to the data in the AOS stack can be very restrictive.

The strong argument given for AOS is the natural "key as you read" operation. This is valid. AOS calculators solve problems very will, they are straight forward, and for routine, well written equations, they are easy for most people to pick up and use. Let the problems get complex, and changing as in many real life situations, and the rigid stack of the AOS system becomes an anchor around the users neck as he attempts to keep up with verbal and changing real world problems. The AOS proponents will still argue that the do it as you see it approach is simpler. It is obvious, however, that even TI recognizes that the user gets confused with parenthesis, otherwise why are there more parenthesis than pending operations on machines such as the TI-30? The idea appears to be, if in doubt, use parenthesis, and the machine will take care of keeping track of pending operations. Catering to this need only reinforces the confusion that users often have as to what is the best way to key up a problem. While many AOS machines warn the user when he has exceeded his limit of parenthesis or pending operations, most users really don't understand all the hierarchy logic built into their machine.

The left to right keying in of the problem is another argument, or point of discussion, that is invalid if the two machines, i.e. systems, being compared have equal stacks. Both can solve the problem keying in left to right.

With the above brief comparison of the stacks of the two systems the answer to the question of which system is best, is obvious. If the user intends on solving straight forward problems, and he doesn't have the patience, inclination, or ability to learn an easier, more flexable system, then he should be happy with a AOS machine. For all other users, RPN is so far superior, more convenient to use, and just plain easier in terms of mental "work" that there can be no question as to what is the better system.

This comparison is necessarily incomplete, and was only intended to open the door for a better understanding of the basic differences of AOS and RPN. The question of which is 'best' in programming or the real world ability of both the SR-52 and TI-59 to manipulate their stacks through unsupported "functions", or specific examples of the "dynamic" real life flexability of HP's RPN system were not covered because of space. Perhaps these topics can be continued in future PPC Journal articles.

DATA PACKING

This article will describe three methods to expand your calculator's memory capacity under software control. Since all three methods involve stuffing existing data registers with more information than usual, these techniques fall under the general heading described as data packing.

The first technique is decimal point encoding. The idea is a natural one; use the decimal point to separate two numbers and use the INT/FRAC functions to unpack the numbers. It is possible to store two 5-digit integers, or three 3-digit integers, or ten 1-digit integers, or other combinations. William Kolb (265) packed five 2-digit integers in a register in his program for sampling without replacement. See V5N1P5 for an excellent illustration of this technique.

Mike Louder (329) used logarithmic encoding on the HP-35. This technique allows you to store two scientific numbers (including exponents) in one register. This is accom-

plished by taking the log of each number, shifting the decimal point, then rounding the result to a 5-digit integer. Two 5-digit integers are then stored using the decimal point encoding technique described above. To recall the numbers take the antilog of each 5-digit integer and keep track of the decimal point.

The major drawback to this type of encoding is accuracy. The log and antilog functions are sensitive to the most significant digits, but the error is introduced when the log is rounded to 5 digits. Taking the antilog only gives you back an approximation of the original number. However, in many scientific applications fewer than 5 significant digits are needed and if you know ahead of time the range of numbers involved the idea of logarithmic encoding can be a useful one.

Along these same lines Mike Richter (2356) discussed using the exponent for multiple storage in V4N6P24. If $1 \leq x \leq 9$ then INT part of log(x) is the exponent. Mike also mentions using the sign bit positions as extra flags. These are also fine examples of multiple storage techniques.

Decimal Point Encoding/Decoding Routines

By substituting R(i) for R1 below, any of the registers R0-R19 may be employed. n=number of digits per integer to be stored. (n=1,2,3,4,5). The position numbers start with 1 and count up to INT(10/n).

INSTRUCTIONS:

Store 10^n in RB before using these routines.

To recall the number from position k, key in k and press A.

To store an n-digit integer j in position k, key j ENTER↑ k and press E.

Recall position k:	Store j in position k:
31 25 11 f LBL A	31 25 15 f LBL E
34 12 RCL B	31 22 11 f GSB A
35 52 h x≷y	35 52 h x≷y
35 63 h y^x	34 12 RCL B
35 64 h ABS	81 ÷
34 01 RCL 1	71 X
35 82 h LST x	33 51 01 STO - 1
81 ÷	35 82 h LST x
32 83 g FRAC	35 54 h R↑
34 12 RCL B	71 X
71 X	33 61 01 STO + 1
31 83 f INT	35 22 h RTN
35 22 h RTN	

EXAMPLES:

Store three 3-digit integers. n=3
First store 1000 in RB.
Key 589 ENTER↑ 1 and press E.
Key 637 ENTER↑ 2 and press E.
Key 41 ENTER↑ 3 and press E.
The 3 positions should now be filled.
R1=41637589. Key in any position number 1-3 and press A to check.

Store five 2-digit integers. n=2
First store 100 in RB.
Key 17 ENTER↑ 1 and press E.
Key 22 ENTER↑ 2 and press E.
Key 53 ENTER↑ 3 and press E.
Key 86 ENTER↑ 4 and press E.
Key 94 ENTER↑ 5 and press E.
The 5 positions should now be filled.
R1=9486532217. Key in any position number 1-5 and press A to check.

In V3N9P11 Jacob Jacobs (99) published his technique for obtaining 30 flags in one register. This is a superb example of data packing and Jacobs indicated he invented this technique to remember 74 game moves in his Hexpawn game. He also indicated the technique could be used for other games such as Bingo, Keno, or dealing cards.

The original idea behind Jake's Flags was to consider a 10-digit integer as a long string of 0's & 1's. This of course is the form the integer would take if written in binary (base 2) form. However, it is not required to convert the entire number into binary form. Although we may see a 10-digit integer, it is how we think of the number being represented that is important. If Jake's Flags are numbered 0-29 within a register it is not necessary to serially decode all 30 positions to find or test the particular flag we want. The only input required to either set, clear, or test a particular flag is the position number of the flag within the register. The most compact routines for this binary encoding technique are given below.

HP-67 JAKE'S FLAGS 30 FLAGS IN ONE REGISTER

The following 3 routines assume R1 contains the flag set. The 30 flags are numbered 0-29 inclusive. In testing flag k, if the result is 0 then flag k is clear, and if the result is 0.5 then flag k is set. $0 \leq k \leq 29$. For each routine below key in k and then press the appropriate user defined key.

Test Flag k:	Clear Flag k:	Set Flag k:
f LBL A	f LBL C	f LBL E
2	f GSB A	f GSB A
h x≷y	f x=0	f x≠0
h y^x	h RTN	h RTN
h ABS	h x≷y	h x≷y
RCL 1	STO - 1	STO + 1
h LST x	h RTN	h RTN
÷		
f INT		
2		
÷		
g FRAC		
h RTN		

What is significant is that the position number of a flag really acts as an address for the flag. Because a flag is just an on-off switch with limited output it is difficult to think of a flag as a memory location. But when you combine the position number idea as an address together with a base representation other than base 2 that such an artifi-

cial memory blossoms into one which is capable of holding more than simple on-off information.

For example, to deal cards choose base b= 52. Because the 5th power of 52 approaches the maximum integer storage capability in an HP-67 data register it is possible to have only 5 artificial memories per HP-67 register. Think of each HP-67 register as being made up of 5 artificial memories. The powers of 52 serve as the position numbers or the artificial addresses which are numbered 0-4. The coefficients on these powers of 52 hold the contents of each artificial memory represented by the boxes below.

$$\square\ 52^4+\ \square\ 52^3+\ \square\ 52^2+\ \square\ 52^1+\ \square\ 52^0$$

Now any integer which is stored in an HP-67 data register can be thought of as filling the boxes with 5 integers from 0-51 inclusive. Again, what is important is how we think of the number being represented, not the actual number we see. For example, when 274297969 is written in base 52 form we have:

$$\boxed{37}\ 52^4+\ \boxed{26}\ 52^3+\ \boxed{41}\ 52^2+\ \boxed{28}\ 52^1+\ \boxed{49}\ 52^0$$

In this form we can clearly see the numbers in the 5 artificial memories and it is a simple matter to extract the contents of any such memory location. To store a new number in box #2, say 17, we would simply subtract $41\cdot52^2$ from 274297969 (this places 0 in box #2) and then add $17\cdot52^2$ to that result.

BASE b ENCODING/DECODING ROUTINES

By substituting R(i) for R1 below, any of the registers R0-R19 may be employed. As with Jake's Flags, the position numbers start with 0 and count up to some maximum limit.

INSTRUCTIONS:

Store the base b in RB before using the routines below.

To recall the number from position k, key in k and press A.

To store an integer j ($0 \le j \le b-1$) in position k key k ENTER↑ j and press E.

Recall position k:			Store j in position k:		
31	25 11	f LBL A	31	25 15	f LBL E
	34 12	RCL B		33 11	STO A
	35 52	h x≷y		35 52	h x≷y
	35 63	h y^x	31	22 11	f GSB A
	35 64	h ABS		71	X
	34 01	RCL 1	33	51 01	STO - 1
	35 82	h LST x		35 52	h x≷y
	81	÷		34 11	RCL A
31	83	f INT		71	X
	35 82	h LST x	33	61 01	STO + 1
	34 12	RCL B		35 22	h RTN
	81	÷			
31	83	f INT			
	34 12	RCL B			
	71	X			
	51	-			
	35 22	h RTN			

EXAMPLE:

Key up these routines and let's check the previous example with the 5 boxed positions. First store 52 in RB.
Key 0 ENTER↑ 49 and press E.
Key 1 ENTER↑ 28 and press E.
Key 2 ENTER↑ 41 and press E.
Key 3 ENTER↑ 26 and press E.
Key 4 ENTER↑ 37 and press E.
The 5 positions should now be filled. RCL 1 to check if 274297969 is stored there. Now key in any position number 0-4 and press A. You should see the contents as you stored them. Now to store 17 in the 2nd position key 2 ENTER↑ 17 and press E. Then key 2 and press A to check that 17 was properly stored.

It should be emphasized that although the data must be in integer form, applications are not restricted to integers. The output from a card dealing routine could be of the form p.q where p is an integer between 1 & 13 which represents the card value and q is an integer between 1 & 4 which represents the card suit. Thus 11.4 would be interpreted as a Jack of Spades, but this card would be stored as an integer, say 49. A short routine breaks 49 down into the two numbers 11 & 4. (Reduce 49 mod(13)). A salesman could store a large table of prices. Each price (a decimal) would then have to be decod from some integer value.

The table below summarizes the possible configurations for expanding the HP-67 memory capacity using base b encoding.

The table represents the worst case possibilities. In many applications it will be possible to extend the values in the table.

TABLE OF HP-67 EXTENDED MEMORY CAPACITY
USING GENERALIZED JAKE'S FLAGS
BASE b ENCODING

Number of Artificial Memories Per HP-67 Register	Position Numbers	Total Memory Capacity Using R0-R19	Base b	Data Range 0-(b-1) Integers Only
30	0-29	600	2	0-1
19	0-18	380	3	0-2
15	0-14	300	4	0-3
13	0-12	260	5	0-4
11	0-10	220	7	0-6
10	0-9	200	10	0-9
8	0-7	160	14	0-13
7	0-6	140	21	0-20
6	0-5	120	37	0-36
5	0-4	100	100	0-99
4	0-3	80	215	0-214
3	0-2	60	1414	0-1413
2	0-1	40	1D5	0-99999

John Kennedy (918)