One-Minute Marvels

(A collection of short HP48/49 Programs)

By Wlodek Mier-Jedrzejowicz and Richard Nelson

INTRODUCTION

Early HP high end calculators were very limited in the amount of available user program memory. Small memory machines could only run short programs and writing down and keying in short programs was not much of a problem. As memory and program complexity increased, users became less willing to key in programs. External storage in the form of magnetic cards, bar codes, and serial ports solved the problem. By the time the HP48 arrived on the scene in 1990 the use of printed programs was disappearing. Of the many books published for the HP48, very few contain program collections.

Users of the latest machine, the HP49, with its internet compatibility, probably won't even think about printed or "listed" programs. They will simply transfer programs to and from their PC connected to the internet. Still, having a program printed on a piece of paper is useful. This collection of 100 short (about 100 bytes or less) programs is called *One-Minute Marvels* because it is possible to key in a 100-byte program in about a minute. These programs were chosen because they represent the more unusual aspects of HP48/49 programming. Slightly longer programs are included if text is used for prompting or labeling. Omit the prompts and the core program qualifies as "short". We hope that new users will be encouraged to key in a few of these programs and discover the fun of programming for themselves. Each program, or algorithm in program form, has the following information as appropriate:

A. Name, Title, Classification.
B. What it does, description.
C. Inputs and outputs, (arguments).
D. Program listing.
E. Program statistics: number of commands, bytes, check sum, execution time.
F. Comments, why selected: speed, example of good programming, small size, unique solution, etc.
G. Source and author (if not Wlodek or Richard) if known.

This collection of programs is divided into eight categories: A-Mathematics, I-Input/Output, L-List Programs, M-Miscellaneous, S-String Programs, T-Time and Date, U-Utilities and V-SYSEVAL programs. The authors thank HP for printing *One-Minute Marvels* for the 1999 HP Hand Held User's Conference.

PROGRAM NOTES

1. Variables not native to the HP48/49 are **bold** and checksums are in Hexadecimal for the HP48.
2. Program commands referenced in the text are underlined for easy user reference/changes.
3. Last Argument is usually assumed active (normal, default setting).
4. When in doubt use STD display mode. Some programs may alter the machine display mode.
5. Lower case program names are used to avoid conflict with machine used names.
6. Timing values are not provided for non-repeatable situations such as prompting programs or those involving menu changes.
7. The programs are divided into categories and numbered. A category prefix letter is added for ID.
8. 48PCH is an HP48 Programming Class Handout from six years of EduCALC/Abby HP 48 weekly Programming Classes conducted by Joseph Horn and Richard Nelson. Much greater detail may be found in these handouts. Email Richard or Joseph to request copies. See table five.

CONTENTS

* - Multiple programs, *=2, **=3

CONTENTS, CONTINUED

A — MATHEMATICS PROGRAMS

Jeremy Smith heard we were putting together a collection of programs. Here is his response.

"*One-Minute Marvels* are unrequited built-in functions. The 48 has a gazillion functions, so rather than adding more, you roll your own to suit. I had Jim Donnely's first book, and annotated it with such functions, amongst other things. I bumped into him at the bookstore one day, and he wanted to include my annotations in his next version (the present version). He added lots of the features, but not the little routines. The following *(A1-A8)* consists of those routines, which still annotate my new book, listed right along with the built in function table."

"One-Minute Marvels is an excellent name for these. Previously called, variously, tricks, tips, and routines in 25 bytes or less, this catchy name is likely to become one of those online self-enhancing lists, like a FAQ."

"Some of the following are more than 100 bytes but good programmers will have a blast shoe-horning them into < 100 bytes."

"I've not timed the commands, nor measured many of them, since many of them aren't in my machine; I keyed them in from my book notes. Warning: this means that I haven't run and re-tested them since many years ago. They should really be gone over. I've added other notes in addition to programs. I've attributed programs where I know them."

A1. Next Prime (Joe Horn, Brian Walsh)

Enter any integer n (greater than 0) and press '**np'**. The original n will be raised to level 2, and the first prime factor of n will be placed in level 1. To find the next factor, you can press / and then run '**np'** again.

```
'np' << DUP √ → s << DUP 2 MOD { 3 WHILE DUP2 MOD OVER s <
        AND REPEAT 2 + END DUP s > { DROP DUP } IFT } 2 IFTE >> >>
```

31 commands, 108.5 Bytes, #A114h.

A2. Prime Factors (Joe Horn, Brian Walsh)

If you want to find all the prime factors of n, run '**pf'**. It replaces n with a list of its prime factors. (Calls '**np'** above.)

'**pfa'** << { } SWAP DO **np** ROT OVER + ROT ROT / DUP 1 SAME UNTIL END DROP >>

16 commands, 55.5 Bytes, #2E8h. Timing: 27 ⇒ { 3 3 3 } in 161_ms.

Here is another variation.

pfb' << { } SWAP DO **np** ROT OVER + ROT ROT / 1 OVER MOD NOT UNTIL END DROP >>

17 commands, 58.0 bytes, # 267h, Timing: 27 ⇒ { 3 3 3 } in 165_ms.

A3. Combinations

comb(n,r) replaces all **comb**(n,r) with n!/(n-r)! * r! according to the mathematical definition.

'**comb'** << {'**comb**(&A,&B)' '&A!/(&A-&B)!*&B!'} ↑MATCH >>

7 commands, 93.0 Bytes, #F798h.

A4. Greatest Common Divisor. Two versions

'**GCDa'** << WHILE OVER MOD DUP REPEAT SWAP END DROP >>

8 commands, 30.0 Bytes, # 15B3h, Timing: 1071, 459 ⇒ 153 in 18.4_ms.

'**GCDb'** << WHILE DUP2 REPEAT MOD SWAP END DROP2 >>

7 commands, 32.5 Bytes, # 2FA6h, Timing: 1071, 459 ⇒ 153 in 27.5_ms.

A5. Lowest common multiple (uses GCD above)

'**LCM'** << DUP2 **GCD** / * >>

4 commands, 25.0 Bytes, # 6288h. Timing: 1071, 459 ⇒ 3213 in 26.9_ms.

A6. Log to any base

LG(2,5) → $\log_2 5$ or X a lg → $\log_a X$

'LG' << → b x 'LOG(x)/LOG(b)' >>

4 commands, 43.0 Bytes, # 27F8h, Timing: 2, 15 ⇒ 0.55958024809 in 28.9_ms.

'lg' << SWAP LN SWAP LN / >>

5 commands, 22.5 Bytes, # 3147h, Timing: 2, 15 ⇒ 0.5595802481 in 17.8_ms.

A7. Permutations

perm(n,r) replaces all **perm**(n,r) with n!/(n-r)! according to the mathematical definition.

'**perm'** << { '**perm**(&A,&B)' '&A!/(&A-&B)! '} ↑MATCH >>

7 commands, 82.5 Bytes, #4211h.

A8. Weighted Average (Joe Horn)

[weights] [data] → weighted mean
e.g. [20 20 20 40] [75 80 85 90] → 84
(total=100%) scores → result

'WTAV' << OVER DOT SWAP CNRM / >>

5 commands, 22.5 Bytes, # CAC5h, Timing [20 20 20 40], [75 80 85 90] ⇒ 84 in 70.6_ms.

A9. Ulam's Conjecture (Joe Horn)
Professor Ulam says this is not his idea, but many math students have seen this conjecture. It states that given any integer if you repeatedly apply one of two operations on the number, and the result, you will eventually reach one. The operations are:

1. If odd, multiply by three and add one.
2. If even, divide by two.

Take the first interesting integer, 3. The resultant values are: 3 ⇒ 10, 5, 16, 8, 4, 2, 1. The "Ulam" process was applied seven times. Note that when a power of two is reached the sequence directly divides by two, repeating operation 2, to reach one. The two programs below provide the basic Ulam values. **'ULAM'** applies the rules and returns the next value in the series. **'ULM'** repeatedly applies **'ULAM'** until one is reached keeping count of how may times **'ULAM'** is applied.

'ULAM' << IF DUP 2 MOD THEN 3 * 1 + ELSE 2 / END >>

13 commands, 52.5 Bytes, # 2F47h. Timing: 27 ⇒ 82 in 12.7_ms.

'ULM' << 0 OVER DO **ULAM** SWAP 1 + SWAP UNTIL DUP 1 SAME END DROP SWAP →TAG >>

16 commands, 55.0 Bytes, # FBE2h. Timing: 27 ⇒ 111 in 2.78_sec.

A10. High-precision remainder (Joseph Horn)
The algorithm HP uses for their MOD function is very high precision, higher than "normal" division. This may be used to advantage when high precision remainders are required. 'RMD' returns the remainder of level two divided by level one.

'RMD' << MOD LASTARG DUP SIGN ROT SIGN ≠ * - >>

9 commands, 32.5 Bytes, # EDE2h. 987654321, 0.123456789 ⇒ 0.111111192 in 17.9_ms.

A11. Rounding three ways Ten-Second Marvels (Joseph Horn)

1. Round obj2 "up" to a multiple of obj1: **'UP'** << SWAP OVER / CEIL * >>
   Examples: 153 25 **'UP'** ⇒ 175
   167 25 **'UP'** ⇒ 175

2. Round obj2 "down" to a multiple of obj1. **'DOWN'** << OVER SWAP MOD - >>
   Examples: 153 25 **'DOWN'** ⇒ 150
   167 25 **'DOWN'** ⇒ 150

3. Round obj2 to the "closest" multiple of obj1. **'NEAR'** << SWAP OVER / 0 RND * >>
   Examples: 153 25 **'NEAR'** ⇒ 150
   167 25 **'NEAR'** ⇒ 175

A12. Hamming weight of binary number (# of 1 BITS) (Jurjen NE Bos)

'BITS' << # 777777777777777h OVER SR OVER AND DUP2 SR AND ROT OVER SR AND + + - DUP SR SR SR SR + #F0F0F0F0F0F0F0Fh AND #FFh DUP2 / * - >>

28 commands, 111.5 Bytes, # 4169h

A13. Extracting first 24 digits of π (Joseph Horn)
'π' returns the correct ***rounded*** 12 significant digits (ending in 9) of π. The HP48 uses 31 digits internally for its floating point trigonometry functions. **'π24a'** uses a trigonometry function found by Joseph Horn to return π to a ***truncated*** 12 digits (ending in 8) by using 0 ACOS 2 * (in radian mode) as an input to SIN, to essentially extract 24 (limit of keyboard input) of the 31 internal digits. **'π24a'** is the basic program. **'π24b'** is the "fancy" version. The name is created using **'SMENU'** found in the SYSEVAL Programs category. Store π related programs in a directory named, π, created with **'SMENU'** as well.

'π24a' << RAD 0 ACOS 2 * DUP SIN MANT DEG >>

9 commands, 32.5 Bytes, # 6449h. Timing: 3.14159265358, 9.79323846264 is returned in 20.3_ms.

'π24b' << RCLF 0 ACOS 2 * DUP SIN MANT 100000000000 * →STR SWAP STOF >>

13 commands, 55.5 Bytes, # 1F95h. Timing: "3.1415926535879323846264" is returned in 47.1_ms.

A14. Benford's Law
Given enough data without artificial restrictions the percentage distribution of leading digits 1 to 9 is predicted by Benford's Law. **'BEN'** leaves the machine in STD mode. Only digits 1 to 9 input is allowed. This program is an example of making a simple program "fancy". The output is rounded to three significant figures. Underlined commands convert decimal to percent (faster, fewer bytes than 100 /).

'BEN' << STD "Digit " DUP "?" + "" INPUT OBJ→ MANT IP DUP NOT + DUP INV
1 + LOG 1 SWAP %T -3 RND "%" + " is " SWAP + + + >>

30 commands, 109.0 Bytes, #1EF4h

A15. Quadratic equation (Eric Lane)
Solving $Ax^2 + Bx + C = 0$, $A \neq 0$, is fast with **'quad'**. Input A, ENTER, B, ENTER, C, ENTER, **'quad'**

'quad' << 3 PICK / SWAP ROT -2 * / DUP SQ ROT - √ + LASTARG - >>

16 commands, 50.0 Bytes, # 94B5h, timing: 12, 36, -48 ⇒ x=1, x=-4 in 24.3_ms.

A16. Number of BITS for decimal number (Joseph Horn)
This in an unusual (looping) approach.

'NBIT' << 0 1 ROT FOR **c** 1 + **c** STEP >>

9 commands, 36.5 Bytes, # 17BCh. 1024 ⇒ 11 in 66.8_ms.

A17. How many random numbers are required to be ≥ to n? (Detlef Müller)
This is a unusual application of START…STEP.

'NRN' << 0 0 ROT START 1 + RAND STEP >>

8 commands, 30.0 Bytes, # 43DCh.

I — INPUT/OUTPUT, I/O

I1. Keyboard keycode to digit
When **'K→D'** runs nothing seems to happen. The machine is waiting for a key to be pressed. If any key other than a digit key is pressed "nothing happens". You will know the machine is responding because

the busy annunciator will turn on briefly. When you do press a digit key that digit will be returned. This may be used with a display "menu" type prompt for a zero to nine choice.

'K→D' << **-7** DO DROP 0 WAIT UNTIL "\RSTHIJ>?@" SWAP IP CHR POS DUP
END 1 - >>

15 commands, 60 Bytes, # 82DEh.

The –7 is a dummy variable for the DO loop to avoid cluttering up the stack with the keycodes of non-digit keys. The text string is the ASCII characters of the digit keycodes. See **'S1'**.

I2. Keyboard keycode to sequential number (Joseph Horn)

The ZERO WAIT sequence is very powerful for use with screen (DISP) prompts because the machine waits for a key press in a low battery drain state. This is unlike KEY which leaves the machine running. It is useful to convert the A through X key keycodes into sequential numbers 1 through 24 with **'K→S'**.

'K→S' << 0 WAIT 10 MOD SWAP 10 / IP 1 - 6 * + IP >>

14 commands, 63.5 Bytes, # D8D7h.

TABLE I2 — "A" through "X" HP48 Keycodes

Key	In	Out	Key	In	Out	Key	In	Out	Key	In	Out	Key	In	Out	Key	In	Out
A	11.1	1	E	15.1	5	I	23.1	9	M	31.1	13	Q	35.1	17	U	43.1	21
B	12.1	2	F	16.1	6	J	24.1	10	N	32.1	14	R	36.1	18	V	44.1	22
C	13.1	3	G	21.1	7	K	25.1	11	O	33.1	15	S	41.1	19	W	45.1	23
D	14.1	4	H	22.1	8	L	26.1	12	P	34.1	16	T	42.1	20	X	46.1	24

'K→S' works for keys past the "X" key except it "counts" the ENTER key twice. It also "skips" keys after the three shift keys. The + key returns 53. The sequential number may be used to GET something from a list, SUB a character from a string, etc.

I3. Maximum text on the screen

Size one font (small menu font) may be used to display text (with an upper case only restriction) if it is converted to a graphics object first. While **'DSP'** is longer than 100 bytes it is so useful for those big text notes, names, addresses, etc. that it had to be included in this collection. You only need one copy of DSP in HOME for all of your text strings.

'DSP' << STD OBJ→ # 83h # 40H BLANK SWAP 1 - 6 * 0 FOR **s** # 0h **s** R→B
2 →LIST ROT 1 2 →LIST ROT 1 →GROB REPL -6 STEP PICT STO
{ # 0h # 0h } PVIEW 0 FREEZE >>

35 commands, 149.0 Bytes, # 4476h.

'DSP' expects a list of text strings. Each string may be up to 33 characters (longer strings are truncated). Ten strings are all that will fit on the screen, the eleventh string is only partially displayed. Lower case is converted to upper case. To store text messages in your HP48 you key the text as a program using the format: << { **"**line one**"** … **"**line ten**"** } **DSP** >>

I4. Press "any" key to continue (clear)

You have seen the computer screen message "Press any key to continue". Of course they don't mean ***any*** key. Shift, etc. doesn't work. A similar input response may be programmed on the HP48. It may be used at the end of **'DSP'** above in place of 0 FREEZE. The shift keys (α, left and right shift) do not respond. The ON key does, however.

'AKEY' << IFERR 0 WAIT THEN END DROP >>

6 commands, 35.0 Bytes, # 31C3h.

This I/O option takes advantage of the "half" keystroke. If the search and press times are approximately equal, pressing the same key twice in succession saves the second key's search time. This gives a value of 1-1/2 keystrokes to clear the screen and start the next operation.

L — LIST PROGRAMS

Many of these list programs are from an HP48PCH. See Table 5 for details.

L1. Remove duplicates. (Joseph Horn)

'RDUPS' << LIST→ { } 1 ROT START SWAP IF DUP2 POS THEN DROP
ELSE + END NEXT >>

14 commands, 50.0 Bytes, #4A55h. Timing: 64 elements with 16 duplicates removed in 0.904_sec.

L2. Randomize a List

'RANL' << LIST→ → **t** << 1 **t** FOR **n n** RAND * CEIL ROLLD NEXT **t** →LIST >> >>

16 commands, 62.5 Bytes, #1F8Ch. Timing: 100 elements in 1.52_sec.

L3. Remove an element

Given a list on level two and a position number on level one, **'REL'** will remove an element.

'REL' << SWAP LIST→ 2 + DUP ROLL OVER SWAP - ROLL DROP 3 - →LIST >>

14 commands, 45.0 Bytes, #849Eh. Timing: #13, A to Z in 38.4_ms.

L4. Remove a series of elements

Given a list on level three and two position numbers on levels two and one, **'RSL'** will remove the elements between the two position numbers.

'RSL' << 3 DUPN DROP 1 - 1 SWAP SUB 4 ROLLD SWAP DROP 1 + OVER
SIZE SUB + >>

18 commands, 55.0 Bytes, #ABEDh. Timing: { A…E }, 2, 4 in 31.5_ms.

L5. Exchange two positions

Given a list on level three and two position numbers on levels two and one, **'ETP'** will exchange the elements of the two position numbers.

'ETP' << ROT DUP 4 DUPN DROP ROT GET PUT 4 ROLLD SWAP GET PUT >>

13 commands, 42.5 Bytes, #1BA9h. Timing: { A…E }, 1, 4 in 61.7_ms.

L6. Rotate elements right one element

'RR1' << LIST→ ROLLD LASTARG →LIST >>

4 commands, 20.0 Bytes, #9D6Dh. Timing: { A…Z } in 29.4_ms.

L7. Rotate elements left one element

'RL1' << LIST→ ROLL LASTARG →LIST >>

4 commands, 20.0 Bytes, #A32'1h. Timing: { A…Z } in 29.3_ms.

L8. Rotate elements right N elements

Given a list on level two and the number of elements to rotate right on level one, **'RRN'** will rotate the elements in the list.

'RRN' << SWAP LIST→ DUP 2 + ROLL 1 - OVER MOD 1 + 1 SWAP START ROLLD LASTARG NEXT →LIST >>

19 commands, 57.5 Bytes, #E51Ch. Timing: { A…Z }, n=5 in 0.689 sec.

L9. Rotate elements left N elements

Given a list on level two and the number of elements to rotate left on level one, **'RLN'** will rotate the elements in the list.

'RLN' << 1 + OVER SIZE SUB LASTARG DROP 1 - 1 SWAP SUB + >>

13 commands, 42.5 Bytes, #FC43h. Timing: { A…Z }, n=5 in 0.0569 sec.

L10. Fill a list with zeros (or digit 1-9)

Given a real number of zeros on level one, **'LN0'** will create a list of N zeros. Use short form digits ±1 to ±9 in place of 0 in the program if desired.

'LN0' << 1 →LIST 0 CON ARRY→ EVAL →LIST >>

7 commands, 27.5 Bytes, #6535h. Timing: 100 in 0.203_sec.

L11. Calculate average of list elements

Given a list of reals, **'LA'** returns a real that is the average of the values in the list.

'LA' << ΣLIST LASTARG SIZE / >>

4 commands, 23.0 Bytes, #2CF0h. Timing: 1 to 100 in 0.406_sec.

L12. Calculate % of total of list elements

Given a list of reals, **'L%'** returns a list that is the element for element percent of the total of the elements in the list.

'L%' << ΣLIST LASTARG %T >>

3 commands, 20.5 Bytes, #CE45h. Timing: { 5 10 15 20 } in 112_ms.

L13. Calculate median of list elements

Given a list of reals, **'LM'** returns a real that is the median of the elements in the list.

'LM' << SORT DUP SIZE 1 + 2 / GET LASTARG FLOOR GET + 2 / >>

14 commands, 48.0 Bytes, #71CCh. Timing: 1 to 9 in 0.227_sec., 1 to 10 in 0.360_sec.

L14. Test a list for reals

Test a list for all elements being reals. Return 1 if true, 0 if not.

'TLR' << 0 IFERR MOD THEN SWAP DROP ELSE DROP 1 END >>

10 commands, 45.0 Bytes, #1F8Ch. Timing: 100 elements in 0.505_sec.

L15. Replace a list object with another (or others)

'RPL' << → **p r** << LIST→ → **t** << **t** 1 + **p** - ROLL LASTARG **r** SWAP ROT DROP ROLLD **t** →LIST >> >> >>

22 commands, 84.0 Bytes, #4B76h. Timing: 104 elements in 0.115_sec.

Timing is made with a list created by **'MLA'** with the 2 / FLOOR element replaced with a two element list: { "XXX" # AFh }.

L16. Return lowest (or highest) value of a list

'RLV' << LIST→ 2 SWAP START MIN NEXT >>

6 commands, 25.0 Bytes, #D57Ch. Timing: 1 to 100 in 0.457_sec.

Replace MIN with MAX for the highest value in the list.

L17. Insert an element into a list

'IEL' << → **p r** << LIST→ → **t** << **r t** 2 + **p** - ROLLD **t** 1 + →LIST >> >> >>

19 commands, 76.5 Bytes, #F0BDh. Timing: "X" is inserted in the "middle" of 100 elements in 0.103_sec.

L18. Make a test list 1 to N

'MLN'. << 1 SWAP FOR **n n** NEXT DUP →LIST >>

8 commands, 34.0 Bytes, #5DDh. Timing: 100 elements in 0.394_sec.

L19. Make a test list A to Z

'MLA' << → **n** << { A…Z } 1 **n** 26 / CEIL LN 2 LN / CEIL START DUP + NEXT 1 **n** SUB >>

21 commands, 198.5 Bytes, #62B9h. Timing: 100 elements in 0.120_sec., 500 in 0.408_sec.

If the input exceeds 26, A to Z is repeated as required. This program is very fast, especially for large lists.

L20. Tag elements in a list

Run this program with a list on the stack and it returns a list with the elements tagged with the element position number.

'TAGL' << 1 OVER SIZE FOR **n** DUP **n** GET **n** →TAG **n** SWAP PUT NEXT >>

14 commands, 53.0 Bytes, 8927h. Timing: { A B C D E F G H } ⇒ { :1:A :2:B :3:C :4:D :5:E :6:F :7:G :8:H } in 378_ms

M — MISCELLANEOUS PROGRAMS

M1. Roll a pair of dice, m.n

'DICE' << 1 2 START RAND 6 * CEIL NEXT 10 / + >>

11 commands, 45.5 Bytes, # Adl. Timing: 26.5_ms.

M2. Move menu to end (Ten-second Marvel)

Pressing VAR displays the current user variable menu. Frequently menu variables at the far left of the menu line are not the most used and you want them "out of the way". One solution is to move them to the far right end. 'M2E' moves the first menu variable to the right end. Keep in HOME, put in custom if desired.

'M2E' << VARS TAIL ORDER >> *3 commands, 26.5 Bytes, # B5Bh.*

A more versatile version below allows the variable (one only) to be in a list when the program is run. This variation also returns to the menu where the program is run.

'M2Eb' << RCLMENU → **m** << IF DEPTH THEN IF DUP TYPE 5 SAME THEN ORDER END END VARS TAIL ORDER **m** MENU >>

21 commands, 86.0 Bytes, # 971Fh.

M3. Sort directory (Ten-second Marvel)

Sorts the current directory into ASCII order. Called 'ZD' "Zort Directory" to put itself at the far right end so you will know where it is.

'ZD' << VARS SORT ORDER >> *3 commands, 23.5 Bytes, # C67Ah.*

M4. Sort directory (directories first)

'ZDF' << 15 TVARS SORT VARS SORT + ORDER >>

7 commands, 44.5 Bytes, #A64Dh.

M5. Clear solver variables (Ten-second Marvel)

'**CSV'** clears the variables created when the solver is run. Keep in HOME and include in your solver list for fast easy "clean up" when you are done solving. The program clears ***all*** reals type 0, complex numbers type 1, and Unit objects type 13.

'CSV' << { 0 1 13 } TVARS PURGE >> *3 commands, 35.5 Bytes, # E836h.*

M6. Random selection without replacement (Joseph Horn)

This program is typically used to deal hands from a deck of cards. It returns a random list of n items from a total of **t** items. Input is **t**, ENTER, **n**, **'DEAL'**.

'DEAL' << → **t n** << 1 **t** FOR **x x** DUP RAND * CEIL ROLLD NEXT **t n** - DROPN **n** →LIST >> >>

22 commands, 81.0 Bytes, # 7B14h. Timing: t=52, n=30 in appros. 0.75_sec.

M7. See junk in display (Detlef Müller, clears PICT, suggested by Jeremy Smith)

'dis' << { #0h #0h } PVIEW PICT PURGE 0 WAIT DROP >>

8 commands, 63.5 bytes, #A344h.

M8. Telephone memory aid text to number

The telephone keypad has three letters associated with each digit. Using these letters it is possible to create a word or phrase to serve as a memory aid to remember a seven digit telephone number. Examples are EduCALC, HOTmama, and MyLawyer. Decoding the text and producing the telephone number was the HHC98 programming contest problem. James Unterburger came up with the winning program that was both the shortest and the fastest as determined by multiplying the program bytes and the execution time. For purposes of the contest Q & Z were assigned to zero. In practice they may also be assigned to the one digit.

TABLE M1 — Telephone digit and corresponding letters for HHC98 Programming Contest

Letters		ABC	DEF	GHI	JKL	MNO	PRS	TUV	WYX	QZ
Digit	1	2	3	4	5	6	7	8	9	0

'PHONE' << 1 7 START "22233344455566670778889990" OVER NUM 64 - DUP SUB
SWAP 2 8 SUB NEXT DROP + + + "-" + + + + >>

24 commands, 112.5 Bytes, # 34Fah. EDUCALC ⇒ "338-2252" in 181_ms.

M9. Body Mass Index, BMI

Body Mass Index or BMI is a more realistic ***statistical*** measure of a healthy weight. This program calculates your BMI and returns a statement of your condition. Key in your weight in pounds, ENTER, key in your height in inches, **'BMI'**.

'BMI' << SQ 703 / / DUP 5 / IP 1 - 7 MIN 1 MAX { "Underweight" "Lean"
"Healthiest" "Overweight" "Obese" "Clinically Obese" "Morbidly Obese" }
SWAP GET SWAP 1 RND →TAG >>

21 commands, 178.0 Bytes, # B4AEh, Timing: 120, 64 ⇒ 20.6: "Healthiest" in 44.4_ms.

S — STRING PROGRAMS

S1. Build a text string (Five-second marvel)

I (rjn) once looked at Joseph Horns's machine during a class and saw **'BS'** in his menu. Sure enough he had done the same thing I had in response to a problem of entering a text string. Suppose you have the list of keycodes for the digit keys on your HP48. They are 0 to 9: 92, 82, 83, 84, 72, 73, 74, 62, 63, and 64 (without the *shift plane* decimal part). If you need a text string of these ASCII characters you start with an empty string, key the number, run **'BS'**, and repeat as required. Programs that use unusual text strings should provide the ASCII character numbers. This is faster than using CHARS. Of course the HP49 CHARS is nicer.

'BS' << CHR + >> *2 commands, 15 bytes, # 430Ah*

S2. Generate a space text string

Input n and **'GS'** returns a string of spaces n characters long. The method used is to double the initial ***two*** spaces until enough are generated and using SUB for the desired number. **'GS'** is very fast. This technique is used in **L19**. The leading string may be any characters. Space strings are used for XORing with other strings to toggle case.

'GS' << " " OVER LN 2 LN / 1 SWAP START DUP + NEXT 1 ROT SUB >>

15 commands, 52.0 Bytes, # 2936h, Timing: 10 in 55.8_ms., 30 in 67.0_ms., 100 in 89.5_ms., 500 spaces in 115_ms.

S3. Replace Nulls with "▶" (ASCII 134)

ASCII character zero is called a Null. The HP48 will not allow you to edit strings, lists, programs, etc. if they contain a Null character. If you try you will get an "Error: Can't Edit Null Char." message. 'REPL▶" accepts a string and replaces all Null character occurrences with ASCII character 134, "▶". This character was chosen to be highly visible. It is seldom used and it makes a good marker. The Null character, like all ASCII characters 0 to 30 display as a small black square.

'REPL▶' << WHILE DUP 0 CHR POS DUP REPEAT "▶" REPL END DROP >>

11 Commands, 56.0 Bytes, # 2E09h. Timing: "A■B■...■Z"⇒ "A▶B▶...▶Z" in 673_ms.

S4. Replace "▶" (ASCII 134) with Nulls

S4 is the inverse of **S3**. Note that in both programs the Null in the program is generated (so it may be edited) with the sequence 0 CHR. If you want to make a test string use spaces where you want Nulls and change 0 CHR to " " (one space).

'REPLN' << WHILE DUP "▶" POS DUP REPEAT 0 CHR REPL END DROP >>

11 Commands, 46.0 Bytes, # 66CBh. Timing: "A▶B▶...▶Z" ⇒ "A■B■...■Z" in 671_ms.

S5. Decode string with ASCII number list

With a string on the stack **'C→N'** will return a list of the corresponding ASCII numbers.

'C→N' << { } SWAP 1 OVER SIZE FOR **n** DUP **n** DUP SUB NUM ROT SWAP + SWAP NEXT DROP >>

18 commands, 61.5 Bytes, # C449h. Timing: "ABCDEFGHIJ" ⇒ { 65 66 67 68 69 70 71 72 73 74 } in 284_ms.

S6. Reverse a string

'REV$' reverses the order of characters in a string on level one of the stack.

'REV$' << " " OVER SIZE 1 FOR n OVER n DUP SUB + -1 STEP SWAP DROP >>

15 commands, 54.0 Bytes, #DAFBh. Timing: "A…Z´⇒ "Z…A´in 433_ms.

This is a very slow program reversing about 60 characters per second. An HP48 machine code version will reverse 20,000+ characters per second. The HP49 SREV will reverse about 30,000 characters in a second. It is left as a challenge to the reader to improve the user code performance of this most useful program.

S7. Split a string into two parts

'SPLT' accepts a string on level two and a character position on level one and returns two strings. The first string on level two is the characters up to but not including the character of the specified position. Level one contains the remainder of the characters of the input string. Pressing plus after the program is finished will concatenate the two strings to produce the original input string.

'SPLT' << DUP2 1 SWAP 1 - SUB ROT ROT OVER SIZE SUB >>

11 commands, 37.5 Bytes, # 6F12h. Timing: "A…Z", 13 ⇒ "A…L", "M…Z" in 16.2_ms.

S8. Insert character(s) into a split string

'SPLT' may be used to insert a character or characters using **'INS$**' below. Three inputs are required. The string on level three. The first character that is to follow the character or string that is to be inserted is on level two. Level one is the character or string that is to be inserted.

'INS$' << → **a b** << DUP **a** POS **SPLT b** SWAP + + >> >>

12 commands, 55.5 Bytes, # 2C13h. timing: "A…Z", "I", "III" ⇒ "A…HIIIIJ…Z" in 47.1_ms.

S9. Converting a string to a "name" (Ten-second Marvel)

There are times when a program must "create" the name of a program to recall and/or run. Timing several programs called 'P1', 'P2', to 'Pn' may be done using a loop and creating the names as a string. The string is converted to the name with **'S→N'** below.

'S→N' << "'" SWAP + OBJ→ >>

4 commands, 23.5 Bytes, # 6029h. Timing: "NAME" ⇒ 'NAME' in 167_ms.

The "trick" is to preface the string with a single tic and let the HP48 smart operating system decide that it is a name object (type 6) and not a string (type 2) using the OBJ→ command. The inverse operation is performed by **'N→S**.

'N→S' << →STR IF "'" POS 1 == THEN DUP SIZE 1 - 2 SWAP SUB END >>

15 commands, 58.5 Bytes, # 88AEh. Timing: 'NAME' ⇒ "NAME" IN 26.6_ms.

T — TIME AND DATE PROGRAMS

T1. Friday the 13th (Joseph Horn)
Key in a year and **'FRI13'** returns the dates of Fridays that occur on the 13th of the month.

'FRI13' << 1000000 / 1.13 + 13 FOR **d** 6.133 **d** DDAYS 7 MOD NOT **d** IFT NEXT >>

16 commands, 88.0 Bytes, # 670Bh. Timing: 1987⇒ 2.131987, 3.131987, 11.131987 in 0.148_sec.

Every year has at least one Friday the 13th. What is the maximum number possible in one year? The following years from 1900 to 2000 have three Friday the 13ths: 1903, 1914, 1925, 1928, 1931, 1942, 1953, 1956, 1959, 1970, 1981, 1984, 1987, 1998.

T2. Test for leap year (Christian Meland)
Input a year and **'LY?'** returns a 1 if the input is a leap year, a 0 if not. This program is a good example of the powerful date commands of the HP48 and HP49.

'LY?' << 2280000 + MANT 1 DATE+ 3 < >>

7 commands, 35.5 Bytes, # 580Dh. Timing: Avg. For 1900, 1996, 1997, 2000 is 12.3_ms.

T3. Day of week
Input a standard date in mm.ddyyyy format and **'dow1'** (system flag –42 clear) returns a three letter day.

'dow1' << 0 TSTR 1 3 SUB >>

5 commands, 22.5 Bytes, # A8D0h. Timing: 8.211999 ⇒ "SAT" in 26.2_ms.

Joseph Horn suggests using DDAYS and a known date to calculate the day of week. The known date is a Sunday (year 3,000) and is selected to have the day and month the same so system flag –42 setting doesn't matter. He had to "hunt" for a date that met these requirements. Given a date in mm.ddyyyy format, **'dow2'** returns a number between 0 (Sun) and 6 (Sat). Example: HHUC date 8.211999, **'dow2'** returns 6.

'dow2' << 2.023 SWAP DDAYS 7 MOD >>

5 commands, 30.5 bytes, #B181h. Timing: 8.211999 ⇒ 0 in 7.17_ms.

T4. Nth day of week from date (Brian Walsh)
A company that builds large engines always ships on Friday when it has the required truck available. Delivery is quoted in weeks. The XJ engine requires nine weeks to build. An order is received on Monday, August 23, 1999. What is the shipping date? (date N D ⇒ date) 8.231999, 9, 5 **'ND'** ⇒ 10.221999.

'ND' << ROT DUP 1.022 SWAP DDAYS 7 MOD ROT DUP2 ≤ 5 ROLL - 7 * - -
NEG DATE+ >>

19 commands, 65.5 bytes, #C52Eh, Timing: 8.231999, 9, 5 'ND' ⇒ 10.221999 in 28.8_ms.

Table T4 — Numeric code for the days of the week used by **T3**, **T4** & **T5**

Day	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Day Code	0	1	2	3	4	5	6

T5. Nth day of week of month (Brian Walsh)
On what date does Thanksgiving fall this year? Thanksgiving is always the fourth Thursday in November. Input: date N D ⇒ date. 11, 4, 4, **'NDM'** returns 11.251999 (November 25). D is the same day code as in Table T4. Note: this program calls **T4**, **'ND'**.

'NDM' << ROT DUP 100 * FP 1 + 100 / SWAP IP + ROT ROT **ND** >>

15 commands, 66.5 bytes, #4F2Dh, 11, 4, 4, 'NDM' ⇒ 11.251999 in 53.3_ms.

T6. Chinese New Year (Prompting & labeling)

'CHINyear' << "Enter Year" "" INPUT OBJ→ { "Rooster" "Dog" "Pig" "Rat" "Ox" "Tiger"
"Hare" "Dragon" "Snake" "Horse" "Sheep" "Monkey" } OVER 1 - 12
MOD 1 + GET SWAP →TAG >>

27 commands, 187.0 Bytes, # 3998h. Timing: 1999 ⇒ "Hare" in 28.9_ms.

T7. Electronic stopwatch
Single key starts/stops. The machine does not run while "timing" and is usable for other tasks. The start time is stored in user menu **'t'**. Move this out of the way to the far right of your menu. **'SW'** uses Flag annunciator 5 (looks like "S") to indicate stopwatch is "running". Time is tagged with "Sec". Note: The fastest time possible is 0.2 seconds due to system (debounce) response. See 48PCH for additional details.

'SW' << 1 FIX TICKS IF 5 FS?C THEN **'t'** RCL - B→R 8192 / "Sec" →TAG ELSE
't' STO 5 SF END >>

21 Commands, 100.0 Bytes, # 69F7h.

T8. Electronic Stopwatch time units
'SWU' may be called in place of the two underlined commands in the Electronic Stopwatch program to add minutes and hours tagging. Alternately you may key in the commands to make one program.

'SWUa' << IF DUP 60 < THEN "Sec" ELSE 60 / IF DUP 60 < THEN "Min" ELSE
60 / "Hr" END END →TAG >>

22 commands, 122.5 Bytes, # 2DADh.

Brian Walsh suggests that **'SWUa'** can be rewritten as:

'SWUb' << DUP 60 < "Sec" { 60 / DUP 60 < "Min" { 60 / "Hr" } IFTE } IFTE →TAG >>

18 commands, 107.5 bytes, #E644h checksum

'SWUb' is 15 bytes shorter, and faster than **'SWUa'.** This is a good example of the use of the short form IF…THEN…ELSE…END structure — IFTE.

T9. Alarms

Here are two simple repeating alarm sounds programmed to start when you press the menu key and stop when you press any other key. **'ALM1'** and **'ALM2'** produce alternate low, then high, tones. **'ALM2'** is about twice as fast as **'ALM1'**. **'AM1'** repeats about once per second.

'ALM1' << DO 900 .2 BEEP .2 WAIT 1200 .2 BEEP .4 WAIT UNTIL KEY END DROP >>

15 commands, 95.5 Bytes, # C3Afh

'ALM2' << DO 1000 .1 BEEP .1 WAIT 500 .1 BEEP .2 WAIT UNTIL KEY END DROP >>

15 commands, 95.5 Bytes, # FD5h

T10. Shorter Time String, TSTR

For US users (HP48 system flag –42 clear) the time string is 22 characters long showing time in 12 hour format with an A or P at the end. This is too long for some display applications and for many applications the seconds is not needed so dropping a semicolon and two seconds digits is practical. This program makes a 19 character time string.

'tstr' << DATE TIME TSTR DUP 1 9 DUP + SUB SWAP "■" DUP SUB + >>

15 commands, 51.0 Bytes, # 1677h. Timing: 56.9_ms typically.

This is an example of entering a special character into a string and using short form numbers from table 1. 9 DUP + is used in place of 18, and "■" is ASCII character 22. This saves five bytes (8.9%) of memory and adds 6 milliseconds to the run time. Run **'tstr'** and press down cursor to view the whole string. Compare with TSTR.

U — UTILITIES

U1. Convert zero to one (Joseph Horn)

Suppose you start your program with a divide and you do not want zero as an input and one is acceptable. Joseph Horn improved the "obvious" solution of << DUP 0 SAME { NOT } IFT >> with this solution. Restricted to reals.

'0→1a' << DUP NOT + >> *3 commands, 17.5 Bytes, #F0FEh.*

To replace zero digit counts with 1 digit counts, could we assume we'll only be working with zero or positive *integers*? If so, then we can come up with a two-instruction solution:

'0→1b' << 1 MAX >> *2 commands, 15.0 Bytes, #109Bh.*

U2. Toggle flags one and two

'TF1&2' << 2 CF 1 FS?C 1 + SF >> *7 commands, 27.5 Bytes, #49F9h.*

U3. Reverse stack order (?)

This program has been in my (rjn) machine for more than five years. I normally don't use **j** as a local variable so it is not mine. We apologize to the unknown author of a great short efficient program.

'SREV' << DEPTH 2 SWAP FOR **j j** ROLL NEXT >>

8 commands, 34.0 Bytes, #3115h. Timing: 1 to 100 in 0.767 seconds.

U4. Delta percent

Calculating percent is difficult for most people. It is not difficult because of the math, it is difficult because of people. Just look at the percent function on a dozen calculators of various manufacturers and

you will see this. Consider a pie that cost $4 last year and costs $5 this year. What is the percent change, delta percent?

Percent problems of this type always have two answers. This either confuses people or inspires them to use it to their own advantage. A sales commission may be 15%. Is that 15% of the sale or 15% of the cost of the product sold? The sales person wants the sale price as the reference, the employer wants the cost as the reference. The pie is 25% more expensive this year. The pie was also 20% less expensive last year. Same numbers, different reference.

This program solves this "problem". **'D%'** accepts two positive numbers in either order and returns ***both*** percentage answers. The reduction is negative and ***always*** on level two.

'D%' << %CH LASTARG SWAP %CH MAX LASTARG MIN >>

7 commands, 27.5 Bytes, #DDBCh. Timing: 4 ENTER 5 → -20, 25 in 13.2_ms.

U5. Horizon distance

If you are 5'-6" tall and you are at the sea shore looking at an object of the same height on the seashore on an island on a clear day, how far away in miles can the object be visible? **'HORZ'** prompts and labels the answer to this question. Use zero as object height for distance to the horizon. (2.71 miles).

'HORZ' << "Object" "Observer" 1 2 START " Height↵in feet?" "" INPUT OBJ→ 4 3 /
√ SWAP NEXT + -3 RND "Visible Miles" →TAG >>

Note: ↵ is the new line character, ASCII 10. The sequence 4 3 / is 1.33. This factor ranges from 1.23 to 1.53 depending on the reference you use. The "trigonometry" solution is closer to 1.50.

22 commands, 120.5 Bytes, # 703Ah, 5.5 ENTER, 5.5 ENTER → Visible Miles: 5.42.

U6. Rename (Joe Horn)

Input the old name, ENTER, new name and run **'RENAME'.** The new name will be moved to the far left end of the menu line. Note that the HP49 has a rename command that leaves the menu name "in place". It also uses the more conventional input order of old name before the new name.

'RENAME' << OVER RCL SWAP STO PURGE >>

6 commands, 22.5 Bytes, # 543Bh.

U7. Temperature Conversion

This programming example is not much of a marvel. It uses straightforward programming techniques to illustrate the following concepts.

1. The input is a number to convert degrees Fahrenheit to degrees Celsius ***and*** vise versa.
2. The program may optionally prompt. Set user flag one to prompt for Temperature if desired.
3. The output is labeled and formatted.
4. Two temperature conversions are made for every input. °F→°C & °C→°F.
5. Because of 3 above, the display mode is altered as needed with 1 FIX and returned to pre-program run status when finished with STOF at the end (underlined).

'DEG?' << IF 1 FS? THEN "TEMPERATURE?" "" INPUT OBJ→ END RCLF SWAP DUP
9 * 1 FIX 5 / 32 + OVER "°C = " + SWAP + "°F" + SWAP DUP 32 - 9 / 5
* SWAP "°F = " + SWAP + "°C" + ROT STOF >>

*40 commands, 182.0 Bytes, # 3460h. timing: 55, **'DEG?'** ⇒ "55.0℃ = 131.0℉", "55.0℉ = 12.8℃"*

U8. Simple numeric ID, "register", data base

These three programs were written in 1990 to emulate sequentially numbered registers á la the HP-41 and most other early calculators. The HP-41 had a SIZE "command" which allocated memory to the number of data registers desired. Memory limited SIZE to 319. The STO key was pressed when you wanted to store something in a register and you responded to a prompt for a register number. This was not pure RPN , but we really didn't mind. Does this demonstrate the importance of prompting?

The HP48 and HP49 are true RPN and so the syntax for storing changes. HP49 users will take their machine out of the box in algebraic mode and will have set up a Start variable to insure their machine will always turn on in RPN mode if they desire (especially after a warm start). The "registers" are stored in a list called **'reg'** when SIZE is run. They are numbered from 0 to n.

'size' << 1 →LIST 0 CON OBJ→ DROP →LIST **'reg'** STO >>

9 commands, 44.0 Bytes, # E0Fah. 1000 registers created in 2.22 seconds.

Zero is used as a place holder for each register. Register counting is from zero so if you want register 100 use 101 as the input for **'size'**.

'sto' << 1 + **reg** SWAP ROT PUT **'reg'** STO >>

8 commands, 43.0 Bytes, # 4E76h.

Place the object to be stored on level two and the register number on level one and run **'sto'.**

'rcl' << 1 + **reg** SWAP GET >>

5 commands, 26.5 Bytes, # 9F49h.

Key the register number (you have to remember what is in which register) and run **'rcl'.**

Lower case names are used to avoid machine name conflicts. The one plus starting **'sto'** and **'rcl'** may be omitted if register "zero" is not needed. This system may be used to store programs, names and addresses, etc. as well. You may have a bunch of routines that you want to "hide" and then call in one or more programs. Simply store the program in a "register" and use the sequence n **rcl** EVAL in the calling program. A series of alarms used by different programs is one example.

U9. US letter stamp values

When the US Postal service increases the postage rates they issue a stamp that has no value on it until they are able to print "regular" stamps. These "mystery" stamps are printed with Letters, A, B, etc. Soon everyone has forgotten what the letter values mean and this program was written to provide letter stamp values because these old stamps turn up in the most unusual places. The program is included here as an example of ASCII encoding and the use of byte saving techniques. Run **'SLTR'** and respond to the prompt with a letter followed by ENTER. The output is the input letter followed by the dollar value of the stamp.

'SLTR' << "Stamp Letter?" { α } INPUT "■■■■■■ !" OVER NUM 8 SQ - DUP SUB NUM 1 % SWAP →TAG >>

15 commands, 83.0 Bytes, # 8B37h.

The string following INPUT is ASCII characters 15, 18, 20, 22, 25, 29, 32, 33 for stamps A – H. Use Build String, **S1**, to assemble this string. Beginners will create the string first and type the program around the string. Of course you could use CHARS. If you enter a letter for which there is no value in the coded string the input is tagged with zero as an output. The sequence 8 SQ is used in place of 64 in

accordance with Table 1. The sequence 1 % is used in place of 100 / and is a byte saving technique (saves 8 bytes) of dividing by 100. See Table 4.

U10. Pre tax cost of product

Merchants often advertise "We will pay your sales tax". What is the cost of the item if the sales tax is included in the quoted price? The more mathematically minded person has no difficulty with this problem and doesn't need a program. For those who have a need for this calculation this utility doesn't require any thinking. The program assumes that the tax value is less than the cost of the item so the order of the inputs doesn't matter. If you can remember to input the cost first, followed by the tax rate (in percent) you can remove the first three commands. In some countries the VAT tax may be high enough that this feature is not practical. The answer is rounded to the nearest cent. The sequence 1 % is a shorter and faster 100 /.

'BTAX' << MAX LASTARG MIN 1 % 1 + / 2 RND >>

10 commands, 35.0 Bytes, # 579Ch

U11. More meaningful random passwords

Many people like to have a random password generator rather than make up their own. The best advice is to combine upper case, lower case, digits and special symbols, but that can make for very unmemorable passwords. It is often enough to use a string of 7 or more lower case letters - at least those make up something that can be related as a foreign word! The following program generates a string of 7 lower case characters - change 7 to another number if you wish. It uses RAND 26 * 96.5 to generate the random letter between a and z. 96.5 is used instead of 97 because CHR rounds to the nearest number. Totally random letter combinations contain too few vowels, and too many letters from the end of the alphabet, so I (wmj) add SQ after RAND to increase the likelihood of the early letters, which contain a higher proportion of vowels. This makes for a higher proportion of readable words, though with too many "a"s in them. Run the program repeatedly until you find a password you like!

'RPAS' << "" 1 7 START RAND SQ 26 * 96. 5 + CHR + NEXT >>

13 commands, 61 bytes, F0B2h. Timing: with πRDZ, ⇒ "htabbst" in 176ms. Following: "aaekkga", "sjatauz".

U12. File purge protection technique

To protect variables from accidental deletion, put in their names the decimal separator that is not the one you usually use. For you or me (US & UK), that's the comma:

```
   level 2       level 1  →  level 1
obj to store     'name'
```

'PSTO' << -51 → **f** << IF **f** FS? THEN **f** CF "." + OBJ→ STO **f** SF ELSE **f** SF
"," + OBJ→ STO **f** CF END >>

System flag -51 is the decimal separator flag, change it if you need to. To purge the variable you now have to toggle the decimal flag - you can write a short program to do this again. This is left as an exercise for the reader.

U13. Invert a flag

Input a flag number and **'if'** will invert (toggle) its status. Set becomes clear, and vise versa.

'if' << DUP FC?C { SF } { DROP } IFTE >>

5 commands, 32.5 Bytes, # 6458h

V — SYSEVAL PROGRAMS

SYSEVAL is a unique command that causes program execution to start at a supplied address. If the entry point is wrong the system will most likely crash, lock up your machine, and frequently clear all of memory (very bad!). Using the SYSEVAL command is both powerful and dangerous. Another reason SYSEVAL is so powerful is additional speed. This comes, in part, from not checking inputs. User RPL always checks the input for each command to protect the user from mistakes.

The most practical use of SYSEVAL is doing things you normally can't do. The HP49 has a powerful set of "hackers" commands built-in so this category of programs have no use on the HP49. If you key in one of the programs in this category be sure to ***double check*** the SYSEVAL address (have your machine in HEX mode) ***before*** you run the program. These programs are not recommended for beginners, but are included in this collection to provide the flavor of advanced HP48 (System RPL) programming.

V1. Making an illegal name

There are times when you want to include illegal characters in a program or directory name. Perhaps you want to start a name with a number, etc. This "problem" is solved with **'SMENU'**. Be sure to double check the SYSEVAL address before running the program to avoid crashing your machine. The DUP DROP sequence insures that the SYSEVAL is not executed with an empty (safely errors) stack for a little extra anti-crash security.

'SMENU' << "Type Special Menu Text" { α } INPUT DUP DROP **# 5B15h** SYSEVAL >>

7 commands, 69.5 Bytes, # 79D3h.

Program names **'π24a'** and **'π24b'** were created using **'SMENU'**. If you try to edit an **'SMENU'** created name (Type 6) it will most likely be converted to an algebraic (Type 9) by the HP48 operating system.

V2. Generate a blank text string

An internal pair of SYSEVALs may be used to generate a string of n spaces when n is on level one. This code is used internally for spacing input forms, etc.

'GS' << # 18CEAh SYSEVEL # 45676h SYSEVAL >>

4 commands, 42.0 Bytes, 6052h. Timing: 10 in 28.0_ms, 30 in 57.4_ms, 100 in 182_ms., 500 in 1.79_sec.

This is a looping solution and is fast enough for a few spaces. Compare with **S2**. Not all system RPL operations are faster than (some) slower, but more efficient, user code programs.

V3. Position, POS, starting from the end

The normal POS command starts searching from the beginning of a string. This system program starts at the end. **'POSE'** expects a target string on level two and a find string on level one and returns the position value where the level one string begins in the level two string. If there are multiple occurrences only the first one from the end will be found.

'POSE' << OVER SIZE # 18CEAh SYSEVAL # 645BDh SYSEVAL # 18DBFh SYSEVAL >>

8 commands, 61.5 Bytes, # 7155h.

The internal command used, #645BDh, requires three inputs and OVER SIZE is used to provide the third one – the start position of the search. If exploring SYSEVALS appeals to you be sure to see James Donnelly's excellent book, *The HP 48 Handbook,*, second edition, published by Armstrong Publishing

Company, 1050 Springhill Drive, Albany OR 97321, USA. Especially see **POS$** and **POS$REV** for two very powerful commands for searching strings — page 154.

V4. Exploring the HP48 Message Table

The HP48 has a large built-in text message table. Jim Donnelly lists 429 internal text messages in *The HP 48 Handbook* and gives 10 groupings of addresses for 555 more. The message numbers listed by Jim range from 1 to 59,144. What is in between the values given? What is beyond them?

The following program accepts a single message number or a range of message numbers, start and end, and returns the text message(s) tagged with the message number. Null strings are dropped. There are lots of messages to explore and discover with these programs. Example messages are: 47505 ⇒ "Enter var name or directory path", 47390 ⇒ "Enter decimal places to display", or 47489 ⇒ "63 Custom ENTER on". The first program, **'GETM'** uses two SYSEVALS to convert the input value to a binary integer, BINT, and get the message. The second program, **'EXPM'** accepts a range of message numbers. The messages are placed on the stack for viewing or further processing.

'GETM' << # 18CEAh SYSEVAL # 4D87h SYSEVAL >>

4 commands, 41.0 Bytes, # 8DA2h. 47505 ⇒ "Enter var name or directory path" in 35.3_ms.

'expm' << FOR **n n** DUP **GETM** IF DUP "" SAME THEN DROP2 ELSE SWAP →TAG END NEXT >>

16 commands, 66.5 Bytes, # 690h. Timing: 47361 to 47515 ⇒ 155 messages in 7.13_sec.

V5. Hide menu

The above programs are potentially dangerous because you may accidentally use an invalid SYSEVAL address. There is another danger. Using an incorrect input or doing something the system wasn't intended to do. This last program especially illustrates both the power and the danger of using SYSEVALS. You may hide the menus in your directory with a null name. When a new variable is stored it appears at the left end of the menu line. When the name of a stored variable is a null name the system thinks that the null name is the end of the directory. A single SYSEVAL will generate the null name. Store zero in this name to hide all the other variables in the menu line. Before doing this make a list of the variables you want to be seen and execute ORDER after you have hidden the menu names.

A hidden directory will behave just as if it were not hidden. You can type a variable name and evaluate it. You can purge variables, etc. You will have to do this "blind" however. Executing VARs, however, will only list those variables added ***after*** you hide the menus. *DO NOT STORE THE NULL NAME IN THE HOME DIRECTORY!* That will cause your machine to crash. Below is **'NN'** which is a simple application of the single SYSEVAL. 'NN' uses flag six to toggle between the hide and unhide state. The program also orders itself to always be visible. Unhiding is faster than hiding. The latter is dependent on the slow ORDER command. If you have lots of variables in your machine it may take several seconds to hide a directory.

'NN' << # 15777h SYSEVAL IF 6 FS?C THEN { **NN** } 0 ROT STO ORDER ELSE PURGE 6 SF END >>

16 commands, 78.5 Bytes, #B4D8h

TABLE 1 — Saving program bytes by using short form numbers for numbers 1 to 100 (2)

Number Needed	Make With	Savings Bytes	Savings %	Number Needed	Make With	Savings Bytes	Savings %
0 - ±9	0 — ±9 (special, 2.5 bytes)	8.0	76%	55	7 →STR_NUM	3.0	29%
10	1 ALOG	5.5	52%	56	7 8 *	3.0	29%
P 11	LCD→ TYPE	5.5	52%	57	1 R→D IP	3.0	29%
12	6 DUP +	3.0	29%	58	1 R→D CEIL	3.0	29%
P 13	6 7 +	3.0	29%	P 59	"," NUM	2.0	19%
14	7 DUP +	3.0	29%	60	5 3 %T	3.0	29%
15	7 8 +	3.0	29%	P 61	"=" NUM	2.0	19%
16	4 SQ	5.5	52%	62	">" NUM	2.0	19%
P 17	8 9 +	3.0	29%	63	7 9 *	3.0	29%
18	9 DUP +	3.0	29%	64	8 SQ	5.5	52%
P 19	PICT TYPE	5.5	29%	65	"A" NUM	2.0	19%
20	5 4 *	3.0	29%	66	"B" NUM	2.0	19%
21	7 3 *	3.0	29%	P 67	"C" NUM	2.0	19%
22	"ASCII 22" NUM	2.0	19%	68	"D" NUM	2.0	19%
P 23	"ASCII 23" NUM	2.0	19%	69	"E" NUM	2.0	19%
24	4 FACT	5,5	52%	70	8 4 COMB	3.0	29%
25	5 SQ	5.5	53%	P 71	"G" NUM	2.0	19%
26	"ASCII 26" NUM	2.0	19%	72	8 9 *	3.0	29%
27	3 9 *	3.0	29%	P 73	"I" NUM	2.0	19%
28	4 7 *	3.0	29%	74	5 SINH IP	3.0	29%
P 29	"ASCII 29" NUM	2.0	19%	75	4 3 %T	3.0	29%
30	5 6 *	3.0	29%	76	"L" NUM	2.0	19%
P 31	"ASCII 31" NUM	2.0	19%	77	"M" NUM	2.0	19%
32	4 8 *	3.0	29%	78	"N" NUM	2.0	19%
33	"!" NUM	2.0	19%	P 79	"O" NUM	2.0	19%
34	C $ 1 " NUM	2.0	19%	80	5 9 %CH	3.0	29%
35	5 7 *	3.0	29%	81	"Q" NUM	2.0	29%
36	"%" NUM	5.5	52%	82	"R" NUM	2.0	19%
P 37	"ASCII 37" NUM	2.0	19%	P 83	"S" NUM	2.0	19%
38	"&" NUM	2.0	19%	84	9 3 COMB	3.0	29%
39	"'" NUM	2.0	19%	85	"U" NUM	2.0	19%
40	5 8 *	3.0	29%	86	"V" NUM	2.0	19%
P 41	")" NUM	2.0	19%	87	"W" NUM	2.0	19%
42	6 7 *	3.0	29%	88	"X" NUM	2.0	19%
P 43	"+" NUM	2.0	19%	P 89	"Y" NUM	2.0	19%
44	"," NUM	2.0	19%	90	"Z" NUM	2.0	19%
45	9 5 *	3.0	29%	91	"[" NUM	2.0	19%
46	"." NUM	2.0	19%	92	"\" NUM	2.0	19%
P 47	"/" NUM	2.0	19%	93	"]" NUM	2.0	19%
48	6 8 *	3.0	29%	94	"^" NUM	2.0	19%
49	7 SQ	5.5	52%	95	" " NUM	2.0	19%
50	2 1 %T	3.0	29%	96	"`" NUM	2.0	19%
51	3 →STR_NUM	3.0	29%	P 97	"a" NUM	2.0	19%
52	4 →STR_NUM	3.0	29%	98	"b" NUM	2.0	19%
P 53	5 →STR_NUM	3.0	29%	99	"c" NUM	2.0	19%
54	6 9 *	3.0	29%	100	2 ALOG	5.5	52%

Note: (1) 1 to 255 may be generated by using "ASCII N" to save 2 bytes or 19%. This is the recommendation in the table if a shorter method wasn't found. If two methods were found of the same bytes, the fastest is listed.

(2) Special thanks to Joseph Horn who double checked and updated the table.

Table 2 — Working Vs. Improved Stack Command Sequences, Ten-Second Marvels

		Working			Improved		
#	SL	Sequence	Bts	Tim (ms)	Sequence	Bts %	Tim %
1	1	{ } +	7.5	12.0	1 →LIST	33.3	54.8
2	1	" " +	7.5	14.2	→STR	66.7	65.3
3	1	0 SWAP 1 SWAP	10	2.59	0 1 ROT	25.0	13.9
4	1	1 PICK	5	3.88	DUP	50.0	27.1
5	1	1 SWAP OVER -	10	4.14	1 – 1 SWAP	0	9.7
6	2	2 ROLL	5	3.97	SWAP	50.0	47.0
7	2	2 ROLLD	5	3.97	SWAP	50.0	47.0
8	2	2 PICK	5	3.87	OVER	50.0	48.6
9	3	3 DROPN	5	3.96	DROP DROP2	0	36.9
10	3	3 ROLL	5	4.00	ROT	50.0	47.0
11	3	3 ROLLD	5	4.08	ROT ROT	0	38.5
12	3	3 ROLLD + SWAP	10	21.2	ROT ROT + SWAP	0	7.1
13	3	3 ROLLD 3 PICK	10	6.27	DUP 4 ROLLD	25.0	29.2
14	4	3 ROLL 4 ROLL	10	6.35	ROT 4 ROLL	25.0	29.5
15	4	4 ROLL 4 ROLL SWAP	12.5	6.70	ROT 4 ROLL	40.0	33.3
16	4	4 DROPN	5	3.85	DROP2 DROP2	0	36.4
17	3	DROP SWAP DROP	7.5	2.70	ROT DROP2	33.3	8.9
18	2	DROP DROP	5	2.38	DROP2	50.0	14.3
19	3	DROP DROP DROP	7.5	2.73	3 DROPN	33.3	29.5
20	1	DUP 1 SWAP	7.5	2.51	1 OVER	33.3	16.7
21	2	DUP 3 PICK	7.5	4.38	DUP2 SWAP	33.3	43.6
22	2	DUP 3 PICK R ROLLD	12.5	6.74	DUP2 ROT	60.0	61.3
23	2	DUP 3 PICK SWAP	10	4.81	DUP2	75.0	55.1
24	2	DUP DUP 4 ROLL	10	4.91	SWAP OVER DUP ROT	0	30.3
25	2	DUP DUP OVER	7.5	2.86	DUP DUP2	33.3	12.0
26	1	DUP DUP DUP	7.5	2.86	DUP DUP2	33.3	12.0
27	1	DUP LASTARG	5	2.70	DUP DUP	0	8.9
28	2	DUP ROT ROT	7.5	2.83	SWAP OVER	33.3	17.3
29	2	DUP ROT SWAP	7.5	2.96	SWAP OVER	33.0	15.5
30	2	DUP2 DROP	5	2.47	OVER	50.0	18.6
31	2	DUP2 ROT DROP	7.5	2.85	OVER SWAP	33.3	12.6
32	2	DUP2 ROT DROP2	7.5	2.88	DROP DUP	33.3	15.6
33	2	DUP2 * ROT ROT + /	15	9.56	* LASTARG + /	33.3	6.8
34	2	DUP2 OVER SWAP	7.5	2.95	OVER LASTARG	33.3	8.5
35	2	DUP2 SWAP 4 ROLL	10	4.83	DUP ROT DUP	25.0	41.2
36	2	OVER OVER	5	2.48	DUP2	50.0	15.7
37	3	OVER OVER 4 ROLLD 4 ROLLD	15	7.17	DUP2 4 ROLLD 4 ROLLD	16.7	5.6
38	2	OVER OVER SWAP 4 ROLL	12.5	5.31	DUP ROT DUP	40.0	46.6
39	2	OVER ROT	5	2.50	SWAP DUP	0	10.4
40	2	OVER ROT DROP	7.5	2.87	SWAP	66.7	28.2

Table 2 — Working Vs. Improved Stack Command Sequences, … Continued

		Working			Improved		
#	SL	Sequence	Bts	Tim (ms)	Sequence	Bts %	Tim %
41	3	OVER ROT ROT	7.5	2.82	OVER SWAP	33.3	12.8
42	3	ROT DUP 4 ROLLD 3 PICK 3 ROLL	20	9.45	ROT ROT DUP2 5 ROLL	37.5	42.9
43	3	ROT DUP 4 ROLLD 4 ROLLD	15	7.27	3 PICK 4 ROLLD	33.3	13.6
44	3	ROT 3 DUPN ROT ROT	12.5	5.24	ROT LASTARG	60.0	45.8
45	3	ROT 1 ROT	7.5	2.61	ROT 1 SWAP	0	1.1
46	3	ROT ROT DROP	7.5	2.96	SWAP DROP SWAP	0	2.0
47	3	ROT ROT SWAP	7.5	3.00	SWAP ROT	33.3	15.7
48	3	ROT ROT SWAP DROP	10	3.46	ROT DROP SWAP	25.0	16.8
49	3	ROT SWAP DROP	7.5	3.00	DROP SWAP	33.3	19.7
50	2	SWAP 1 SWAP	7.5	2.55	1 ROT	33.3	15.7
51	2	SWAP DUP ROT SWAP	10	3.40	OVER	75.0	42.4
52	3	SWAP DUP 4 ROLLD SWAP	12.5	5.30	OVER ROT 4 ROLLD SWAP	0	1.9
53	4	SWAP DROP SWAP DROP	10	3.32	ROT ROT DROP2	25.0	9.6
54	3	SWAP DROP SWAP DROP SWAP DROP	15	4.13	4 ROLLD 3 DROPN	33.3	34.1
55	2	SWAP OVER OVER SWAP	10	3.38	SWAP LASTARG	50.0	16.3
56	2	SWAP OVER SWAP	7.5	2.88	DUP ROT	33.3	10.1
57	3	SWAP ROT ROT	7.5	2.97	ROT SWAP	33.3	12.1

Note: SL = Stack levels used for input. Output stack levels vary.
Bts = bytes for Working command sequences, Improved sequence values are in percent.
Tim = Time to run, typically for symbolic inputs A, B, C etc. Numeric inputs may be faster.
Working sequence values are in milliseconds, Improved sequence values are in percent.

The HP49 adds six new HP48 user code stack commands. Table 3 shows how these commands may be used in place of HP48 command sequences.

TABLE 3 — New HP49 Stack Commands (1)

HP48 Command Sequence	New HP49 Stack Command
ROT ROT	UNROT
DUP 2 + ROLL DROP ROLLD	UNPICK
3 PICK	PICK3
DUP DUP	DUPDUP
SWAP DROP	NIP (1)
2 SWAP START DUP NEXT	NDUPN
SWAP DROP	; (2)

NOTES: (1) Names are based on traditional FORTH commands. The meaning is not supposed to be obvious.
(2) The algebraic semicolon works in RPN mode, why not use it?

TABLE 4 — Programming Techniques Illustrated by the Program Collection

No.	Application	Typical	Improved	See
1	Convert decimal to percent.	100 *	1 SWAP %T (1)	A1
2	Restore loop counter, n.	→ n << n 1 START … -1 STEP…	1 SWAP START … -1 STEP LASTARG (2)	
3	Clear stack except level one.	→ n << CLEAR n >>	DROP CLEAR LASTARG (3)	
4	Largest of two levels on level two. (to subtract?)	DUP2 < { SWAP } IFT	MAX LASTARG MIN (3)	U4
5	Loop n – 1 times	1 - SWAP START…NEXT	2 SWAP START…NEXT (2)	L16
6	Divide by 100	100 /	1 % (2)	U9
7				

Notes: (1) Significantly faster. (2) Significantly shorter. (3) Significantly faster ***and*** shorter.

The weekly Friday evening HP48 programming class taught by Joseph Horn and Richard Nelson stresses the values of short and fast programs. The goal of the class is to explore programming techniques to improve the basics. Problems studied have been documented in class handouts and many of the programs found here are from these handouts. The format for these class exercises is to explore as many different ways (typically 3 to 7) of solving a problem as possible. Sometimes these problems are shared on the HP48 news group and the responses are integrated into the class. Table 3 provides the information necessary to identify these informal materials. If you would like a copy you may email a request to rjnelson@aemf.org .

Table 5 — HP48 Programming Class Handouts as a source for additional program details

Ref.	Pgs.	Title	Date	Fm	Comments
A15	6	Solving single-variable quadratic Equations on the HP48G/GX.	990601	W	15 equations solved, equal, real, & imaginary
I3	5	Full screen Text on the HP48G/GX.	981023	W	
L1-L19	40	Efficient HP48 List Usage From A to Z.	990405	W	144 programs.
M9	19	Recommended Weight Program Assignment	980807	W	31 programs. (Interface)
T7,8	6	Stopwatch Program.	980501	W	
U3	40	Efficient HP48 List Usage From A to Z.	990405	W	144 programs.
U4	1	Convert Zero to One.	990709	W	
U6	20	Efficient HP48 Stack Usage From A to Z.	990102	W	60 programs.
U7	2	Delta Percent (PE9).	970613	W	

Note: Ref. - Program number. Pgs. - Number of printed pages.
Title - Name of program. Date - Six digit date of form YYMMDD.
PE# - EduCALC Programming Exercise
Fm. - Form of text; W-Word 6/95/97, T is Qedit text file with embedded HP DeskJet printer codes.

CONCLUSION

After nine years of HP48 programming the HP49 enhances what we know and love. These programs provide a program transition from the HP48 to the HP49. *One-Minute Marvels* is intended to illustrate the power of good programming to the new user. Experienced users will also find a few clever programming techniques to add to their own experience. The variety of applications from many sources and the number of programs should provide something of interest to every HP48 and HP49 user.

The authors thank Joseph Horn, Jeremy Smith, and Brian Walsh for their contributions and suggestions for better programming — not only for this collection, but for many years of contributions to the community.

August 1999 *Wlodek Mier-Jedrzejowicz* *Richard J. Nelson*

EPILOG

A few comments regarding the HP49. Some users had high hopes that the HP49 would be a whole new generation machine. These users may be quick to dismiss the 49 as "not much new'. This could not be further from the truth, and any user who makes such a statement had better look first. The HP49 represents a new approach to designing a calculator. HP has taken advantage of ten years of HP48 user experience and adapted these achievements into a machine of incredible mathematics power. Yes, we will continue to use Maple and other computer based programs for "serious" math, but it is difficult to do this while on a mountain climbing vacation at 29,000 feet or in the park doing your math homework taking a break from jogging.

All HP high end scientific calculators have had one outstanding quality. They have been so powerful that they are never fully mastered by the user community. Even when we have ten years and hundreds of thousands of users banging keys, we hear reports of new discoveries by the "collectors" of the obsolete models. The user community has always amazed HP by their accomplishments. What HP48 designer would believe the performance of real time 3D wire frame rotation that we see on the HP49 today? What HP48 designer would predict the use of gray scale photos on the HP48? These accomplishments are possible because of the quality and well thought out features HP machines provide. This is the prime reason we buy HP.

The HP49 extends the complexity and feature set of the HP48. We haven't come close to mastering the HP48 as a community. The internet has helped everybody get up to speed faster and it will continue to play a vital role for users getting the most of the HP49. Today's machines are so complex and so powerful that it takes a great deal of time to become familiar with the many features and commands. We believe that HP could not have made a better choice in adapting the achievements of the user community and enhancing these ideas to make them even better. Moving up a less steep learning curve from the familiar to the new will make all of our lives easier and better. The scary thought is: What will ***we*** do when HP makes a "quantum" leap to the next generation? Let's enjoy a smooth transition now and sharpen our skills, especially math, with the 49 to be ready for that day. There is no rush. Thanks HP.

Conference attendees are more likely to be skilled programmers. Apply your skill to the following program which solves $A^3B - AB^3$. Reduce it to seven commands. A and B are on the stack.

'IMPROVE ME' << DUP 3 Y^X ROT DUP 3 Y^X ROT ROT * ROT ROT * SWAP - >>

14 commands, 47.5 Bytes, 869Dh, Timing: 'A', 'B' inputs in 120_ms.

'BETTER' << >>

7 commands, 27.5 Bytes, # 79C6h, (rjn) Timing: 'A', 'B' inputs in 114_ms.
or 7 commands, 27.5 Bytes, # D166h, (bpw) Timing: 'A', 'B' inputs in 117_ms.
or 7 commands, 27.5 Bytes, # 72B5h , (jkh) Timing: 'A', 'B' inputs in 114_ms