SYNTHETIC METHODS ON THE HP-15C

INTRODUCTION AND DISCLAIMER

The complexity of advanced calculators often means that it is neither possible nor useful to keep the internal workings totally secret. Manufacturers may find it useful to be able to test a calculator without opening it and advanced users are always looking for ways to squeeze more out of their machines. Thus, bugs and other loopholes are often discovered that allow the user to 'see' some of the internal workings. Over the years the use of non-standard keystrokes to make a calculator perform operations not described in the owner's handbook have generally been associated with the word 'synthetic'. Thus a synthetic number is one which cannot be keyed in directly and synthetic programming generally involves synthesizing program lines using various tricks.

This article is intended to be a introduction to all synthetic methods that I have investigated on my HP-15C. So far these methods have worked on every HP-15C that I've tried, and therefore are expected to work on your HP-15C. However, I, and I'm sure HP also, do not guarantee that these methods will work on your calculator or any purchased from HP in the future. There is no reason to expect HP to support these functions.

IMPORTANT NOTICE: Most synthetic operations will not harm the calculator in any way. In fact the 'rotate' function and recalling numbers from anywhere in memory are extremely 'safe' operations. However, if you do get hooked into trying out new ideas, odds are that at some point you will do something that will cause the complete loss of continuous memory. Consider yourself warned.

If you cause the processor to crash, the display will blank and the calculator will not respond to any keystroke. For this problem HP recommends holding down the y^x key while turning the calculator on. This has always worked for me although the message 'Pr Error' often appears. The sequence of holding down the minus key while turning on, can be used to initialize memory should this be neccessary. Note; the status registers are also initialized.

SYNTHETIC NUMBERS

It will be useful to understand how a number is stored in the calculator, so let us start with a quick review. For a more detailed discussion of numbers see the book Synthetic Programming on the HP-41C by William C. Wickes. HP uses binary coded decimal (BCD) to store numbers. This means that 4 bits are used to store each decimal digit 0-9. Since 4 bits are also called a nybble, we have the following relation: 4 bits=1 nybble= 1 BCD digit. Of course, 4 bits can also store the numbers 10-15 which I will call by their hex names of A-F. When there may be some doubt as to what number system has been used, I will precede hex numbers with

a small x. It is impossible to generate the numbers xA-xF via any legal operations.

So much for digits, what about signs and the exponent? The sign is stored in a nybble to the left of the first displayed digit and the exponent occupies 3 nybbles to the right of the last displayed digit. To facilitate referring to individual nybbles I will use the following system,

(s) 1 2 3 4 5 6 7 8 9 0 (S E E)

where the parentheses mark off the digits that f-PREFIX does not display. Thus, nybble 1 will always mean the first displayed nybble, s will refer to the sign, and SEE to the exponent. All (legal) numbers are stored internally in a f-SCI 9 format, with the first non-zero digit of the mantissa being left justified. A number in this format is said to be normalized. By HP convention s and S contain 0 for positive and 9 for negative.

Throughout this article I will use the word rotate to describe the series of steps which are: 1) turn calculator off and 2) hold down y^x while turning it back on. HP mentions this sequence of steps on p. 263 of the Owner's Handbook where they made the tantalizing statement that "this will alter the contents of the X-register, so clear the X-register afterward." While playing around I discovered that the 'rotate' operation merely rotated the number in the X-register to the right by 22 bits. If you have access to an HP-15C I suggest trying the following. First, press 1 (no enter!) and then rotate. When you press f-PREFIX you should see 0000004000 in the display. To understand this remember that a BCD 1 has the following binary representation

```
BCD:   1    0    0    0    0    0    0    0    0    0
BIN:0001 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

where I have inserted a space between the BCD digits to guide the eye. Rotating right by 22 bits gives

```
BCD:   0    0    0    0    0    0    4    0    0    0
BIN:0000 0000 0000 0000 0000 0000 0100 0000 0000 0000
```

which is what was displayed. Since both the sign and the exponent contained zero, zero was rotated in from the left.

The above example illustrates another important fact; the number is not normalized. Thus f-PREFIX displays the mantissa with nybble 1 going into the first place of the display, nybble 2 into the second place, etc. If you rotate a 3 and press f-PREFIX you should see a small o in the display. Since a 3 rotates to a xC the small o is the display code for xC. The display codes for the other hex digits are listed in figure 1. A xD will generally display as a P unless it is the fourth digit, when it becomes an small (upper) u. The fourth digit is the location a u appears if you are in program mode and have just entered a 'User mode' instruction.

Esoteric facts concerning rotate and Complex mode: If you do a rotation with stack lift disabled, then the C annunciator will turn on, but no registers for a complex stack will be allocated. A Re<>Im returns an overflow to the X-register. Rotation with the

stack lift enabled turns off Complex mode.

SYNTHETIC MATRICES

If the sign nybble of a register contains a 1 then the register is treated as containing a matrix descriptor. Nybble 1, of a matrix descriptor, contains a digit in the range xA to xE to logically denote matrices A to E. All other nybbles appear to be ignored. To see this, rotate the number 1.000000070 and the descriptor for matrix C will appear in the display. The 7 (0111) rotated to become xC (0001 1100) with the 1 going into the sign nybble and xC going to nybble 1. The initial '1.' is needed since the calculator will normalize any number you enter.

It is clear that nybble 1 of a descriptor can also contain the digits 0 to 9 and xF. Matrices with these numbers are interesting and will be useful to us. To construct matrix 1, rotate the number 1.000000044, and the matrix name (which for this case looks like the Greek letter Xi) with the dimensions of (0,0) should appear in the display. Store the descriptor in I and dimension A to be a (2,2) matrix. If you recall I you will find that matrix 1 now has the dimensions of (2,2). You can quickly discover that matrix 1 does not point at the same memory locations as matrix A. Remember that to recall elements from an arbitary matrix, store the descriptor in the index register I and then use either RCL (i) which uses R0 and R1 as pointers or RCL g (i) which uses X and Y as pointers. Notice that when you recall an element from matrix 1 the correct descriptor (the Greek Xi) appears in the display when you hold down the (i) key.

Where does matrix 1 point? Experimentation has shown that the base of matrix 1 points at the first register in the unassigned pool area. The area allocated for matrix 1 equals the area allocated for matrix A. If we dimension A to be great enough, matrix 1 will point into the program area. In figure 2 I have listed the basics for all synthetic matrices.

The descriptor for synthetic matrices appears to light up random segments in the left half of the display. Thus column 2 in figure 2 shows what the standard display for the matrix descriptors are. Matrix F lights up no segments, while matrix 5 lights the User mode annunciator but does not turn on User mode.

THE PROGRAM AREA

Using the rotate function of the HP-15C, we have constructed synthetic matrices which are dimensioned with one standard matrix but point at some other area of memory. This inconsistent property of synthetic matrices allows us almost complete access to all internal calculator registers. In this section we will examine program memory and discover the op-codes for one- and two-byte instructions.

Construct matrix 1 (1.000000044, rotate) and then type STO I. Zero all matrices, zero program memory, dimension (i) to be 19 and A to be (1,23). At this point there should be 23 registers in the pool and so the top of matrix 1 should point at the top of the data area which is the first program register. Store 1 in R0 and 23 in R1. If you push RCL (i) at this point, element (1,23) of matrix 1 will be returned. To test that you are looking at program memory switch to program mode and enter LBL 1, LBL 2, LBL 3, LBL 4, LBL 5, LBL 6, and LBL 7. Switch back to run mode and RCL (i). If you press f-PREFIX you should see the number 7060504030. Including the sign and exponent nybbles, the true number is

(0) 7 0 6 0 5 0 4 0 3 0 (2 0 1)

Remember that each LBL instruction takes one byte or two BCD digits. It can be shown that program lines fill a data register from right to left thus the above information tells us that the op-code for the LBL n instruction is 0n. To avoid misunderstanding I refer to the two nybbles in a program byte as left and right. For the LBL n instruction the left nybble is 0 and right nybble n.

Figure 3 is a table of all one-byte instructions. All program bytes that have a right nybble containing a xF are interpreted by the processor to be the first byte of a two-byte instruction. This makes it possible to uniquely decide if any given byte is part of a two byte op-code. This was not true on the HP-41C where, in order to back step, the calculator had to go to the beginning of the program and forward step one fewer line numbers.

Figure 3 clearly shows that there are no unused one-byte op-codes. However there are many unused two-byte op-codes. All two-byte instructions require an argument, either a label, a data register or a matrix. This argument always fills the second nybble of the second byte. Therefore, the two-byte op-code table need not list all of both bytes. In figure 4, I have listed two-

Footnotes for figure 4

Note	argument	right nybble
a	0-E	0-E
b	2-E	2-E
c	.0-.9,(i),I	0-9,D,E
d	0-9,I	0-9,E
e	0-9	0-9
f	.0-.9	0-9
g	A-E	A-E
h	user A-E	A-E
i	user (i)	6
j	g(i)	D

byte instructions. The small letters a to j inform the reader of the legal choices for the right nybble according to the footnote table. An example should help. Suppose you are examining program memory and find a byte containing xDF. The xF tells you that this is a two byte instruction, so you must examine the second byte. Assume the second byte contains a x93. The xD in the first byte tells us to examine row D of figure 4 and the x9 in the second

byte tells us to go to column 9. At this point we find that the instruction is STO+ (argument). Footnote c informs us that a right nybble of 3 is a legal argument corresponding to .3 and so STO+.3 is the complete instruction.

SYNTHETIC NUMBER TRANSFER

Using program memory, we can now construct numbers with arbitary bit patterns but, if we try to transfer a synthetic number from where it was created to a new location, we will find that sometimes the number will be changed. In this section I list the conditions that I've found that cause a number to be changed. If the copy is from memory to the X-register then, the number in memory will never be altered. However, if the copy is from the X-register then some operations will alter the original number. I have not investigated memory-to-memory type moves that occur when the STO-MATRIX command is used. I suspect the memory-to-memory copy will cause less damage to the numbers moved. Conditions that cause changes are:

1) The only non-zero nybble is the sign s.
From memory: Zero appears in the X-register.
To memory: The X-register is set to zero and then the copy takes place.

2) The mantissa is zero but the exponent is not.
From memory: Zero appears in the X-register.
To memory: The correct number is stored and then the X-register is set to zero.

3) EE=00 but the exponent sign S is greater than 7.
From memory: Zero appears in the X-register.
To memory: The correct number is stored and then the X-register is set to zero.

4) Storing into a matrix.
From memory: No problem.
To memory: If s=1, the number is interpreted to be a matrix and the HP-15C will not allow a matrix to be stored as an element of a matrix. If s>9 then s will be changed and the number stored. The change in s, is as follows,

was	becomes	was	becomes
A	0	D	7
B	5	E	8
C	6	F	9

Note: This does not happen if you store into a numbered data register.

5) If sign of exponent S is not 0 or 9.
From memory: S will be changed to 0 if S was <8 but will be changed to 9 if S>=8.
To memory: The correct number is stored. However, the X-register

will then be loaded with the flashing 9 display, signaling an overflow.
Note: The HP-15C uses the exponent sign nybble of element (2,2) of a square matrix to mark if the matrix is in LU form. If S is 1 or 8 (for + or -) then the matrix is in LU form. Since S is automatically changed to the correct value when placed in the X-register, all calculations based on this number will give the correct result.

THE STATUS REGISTERS

To examine the status registers, clear program memory and then hit f-MATRIX 0. Use g-MEM to find out the number of pool registers, say 46, and then allocate all registers to matrix A, in this case type 1, ENTER, 46, f-DIM A. If you refer to figure 2 you will see that matrix 1 points into the pool area but by construction there is no pool area. So where does matrix 1 point? It can be shown that the top of the data area occurs at the largest allowed physical address of xFF. When you add one to the largest address you wrap around back to zero. Therefore, the base of matrix 1 now points at address x00 which is the base of the status area.

In figure 5 I have listed all registers from address x00 to x1F and for the useful ones assigned a simple name based on some function that the register is used for. Since different parts of one register are often used for different functions, the name assigned is not intended to describe the entire register, but rather to serve as a memory aid. In the following I will discuss each status register and what I've discovered it is used for. There are several places where my knowledge is clearly incomplete.

If I have never seen a non-zero number in a register then I have put a '0' for the register name in figure 5. Notice that if the register at address x0n is zero then the register at x1n is also zero. Because of this symmetry I like to say that all registers below x20 are potential status registers. Registers x05 and x06 appear to contain zero but careful examination reveals the exponents are non-zero.

Registers y,z and t comprise the user stack. Notice that the X-register is missing. Registers R0, R1, I and last x are also standard user registers. It was interesting to discover that the lowest numbered data register in the data area was R2. This is one reason why registers R0 and R1 cannot be assigned to the pool.

The ran register (x14) is interesting in that only the mantissa is used to store the random number seed. Are nybbles s and SEE used for anything? The answer is yes. The exponent (SEE) holds the current program line number. The sign s generally returns zero but when a program is running all bits are set. This does not appear to be the 'program running' flag since changing the value does not alter the calculator state.

The map register (x15) is quite useful for synthetic programmers. This gives the base address of the numbered data registers, matrices A to E, and the pool in the following manner,

```
          data| A | B | C | D |  E |POOL
        (s) 1|2 3|4 5|6 7|8 9|0 (S|E E)
```

With no complex stack, nybbles s1 will contain xC0 which is the address of the base of the data area. Unknown to the user the numbered data registers move about in memory. Thus the instruction STO 2, means store the X-register in the first location relative to the numbered data area. Whereas, the instruction STO 1 means store the X-register in the absolute address x11. Thus STO 1 should be faster than STO 2 and my test indicated that it is slightly (4 msec or 6%) faster.

Nybbles EE of the map register point at the first register in the pool. If EE contains 00 then there is no program area or pool. Nybbles 2 through 0 contain the addresses of the bases of matrices A through E. Matrices A to E always exist in order in memory. Thus if you are using matrix B and decide to allocate some registers to A, the calculator will physically move all elements of B upward to make room for A. The complex stack, and any registers used by solve or integrate are located below the numbered data registers. Thus when you allocate a complex stack the calculator will physically move all data in the numbered data registers and matrices upward five registers to make room for the complex stack at address xC0. I have not tried solve, integrate, and Complex mode together to see what happens.

When solve finishes, the data is not immediately moved back five registers. The move does not take place until the next memory allocation or when g-MEM is pressed. This allows solve and integrate to share registers. Warning; g-MEM does not just compute the differences between two addresses but physically moves data around. If you have altered the map register so that the matrices point at unusual places in memory then, pressing g-MEM could zero the dimensions of all matrices and the pool allocation. With the calculator in this state, it is impossible to use the data area or to reallocate memory. The solution is to initialize the calculator. Moral--when doing fancy memory mapping, keep your cotton-pickin' fingers off g-MEM.

The dimension register (x19) contains the row-column dimensions of matrices A through C as follows,

```
              A   |   B   |   C   | ?
        (s) 1 2 3|4 5 6 7|8 9 0 (S|E E)
```

I do not know what EE is used for in this register. For each matrix the left byte contains the row length and the right byte contains the column dimension. The dimensions are stored as binary numbers. Thus if B is dimensioned as (1,2), (1,8) or (1,20) then nybbles 4567 will contain x0102, x0108, or x0114 respectively.

The flags register (x1A) contains the dimensions of matrices D and E, the name of the result matrix, and the state of the user

flags as follows,

```
              D   |   E   |R|F|flag| ?
        (s) 1 2 3|4 5 6 7|8|9|0 (S|E E)
```

I've never seen nybbles EE to be non-zero. The dimensions D and E logically complete the set in the dimension register. Nybble 8 contains a hex digit (A to E) which denotes the current result matrix. If any bit in nybble 9 is set the display will flash. However, flag 9 will test true only if bit 2 is set. The 8 bits of nybbles 0S hold the current state of user flags 0 to 7, with flag 0 being the rightmost bit and increases to the left. Logic says that the right bit (bit 1) or nybble 9 should be flag 8. However, testing has shown that this is not so.

Since programs often use memory at the byte level it will be necessary to refer to a particular byte of a data register. To do this HP numbers the bytes in a register from right to left with the numbers 1 to 7. I understand 0 to 6 was used on the HP-41C. Thus every byte has a unique 3 nybble address, two nybbles give the register address and the third nybble identifies the correct byte.

The return registers are best understood with an example. Consider the following program; LBL A, 1, -, x>=0?, GSB A, R/S, RTN. If you run the program with 7 in the X-register then the program will execute GSB A, 7 times. With 7 outstanding returns the return registers will contain,

```
rtn 4|rtn 3|rtn 2|rtn 1|current
(E) F|6 E F|6 E F|6 E F|(7 F F)   reg x17
(6)|F E 6|F E 6|F E 6|0|(7 F F)   reg x16
   |rtn 5|rtn 6|rtn 7|?| last
```

The exponent of register x16 points at the last legal program line, which in this case is byte 7 of register FF. The exponent of register x17 points at the current program byte. Switch to program mode and you see that you are indeed at the last line, byte 7 of register FF. The return addresses fill register x17 from right to left but, continue into return 2 going from left to right. Notice that not only the order of the returns is reversed but also the nybbles in the return addresses. This is strange! Also, for some reason x10 has been subtracted from all the addresses. The address loaded when either solve or integrate calls a routine is of the form x00n where n is a digit that indicates what routine is calling and changes as a solution is found.

The pointer register (x03) is used as follows,

```
        IC| res |integ|S|solve|   ?
        (s)|1 2 3|4 5 6|7|8 9 0|(S E E)
```

Again I don't know what the exponent nybbles are used for. When the solve is called, the nybbles 890 record the byte address of the subroutine being solved. Nybble 7 is used by solve as a counter. A 3 stored here indicates that a root is being found. If the number is 2, 1 or 0 then solve is searching for the root by

looking outside your initial limits. When integrate is used nybbles 456 store the subroutine address and nybble s is used as a counter. Matrix operations use nybbles 23 to point into the result matrix. I suspect that nybbles 23 point at every element in turn but, when the operation finishes the pointer is generally pointing near the top.

The exponent register has the following structure,

```
R/G|U| exponent stack  |  ?
(s)|1|2 3 4 5 6 7 8 9 0|(S E E)
```

In run mode the number is SEE is xEAE. If, however, you examine this area in Program mode then SEE will contain x000. If you are a fast reader, then hold down SST and pass over the region. This time you should find x10C in the exponent. These nybbles could be related to the current calculator state, to what key is being pressed or to something else. Nybble s of the exponent register contains a 4 if the calculator is in rad mode, 5 if it is in grad and 0 otherwise. Nybble 1 is F when in User mode and 0 otherwise. Interesting tidbit--if some (but not all) bits are set in nybble 1 then the calculator will be in User mode but f-USER will not change it out of this mode.

The most interesting part of the exponent register is the exponent stack. This area is logically grouped into three groups of three nybbles each. When, for example, Px,y is executed the numbers in 567 are pushed into nybbles 234; the numbers in 890 are pushed into nybbles 567; and the exponent of the result appears in nybbles 890. Trig functions, Py,x, and Cy,x cause one number to be pushed into the exponent stack. Complex trig functions push two numbers into the stack. However, the two functions ISG and DSE overwrite nybbles 890 with the address of the register being modified and anytime EEX is pressed nybbles 890 are overwritten with xFFF. The ability of the exponent stack to shift by three nybbles 'must' be useful for synthetic programmers but I cannot think of what that use would be.

The last three registers are used by the calculator to control what is seen in the display. To demonstrate the display register, construct matrix 1 and store it in the I register. Dimension A to be a (1,36) matrix. Now press g-MEM and find out the total number of pool and program registers (which will be 10 if you started with the default dimensions). The display register is at address 4 which means that it is 5 registers above program memory. Since 15=10+5, then the display register will be elememt 15 of matrix 1. So type 1, STO 0, 15, STO 1 and now RCL (i) should return the contents of the display register. To see this register in action, write a program with RCL (i) on the first line. If you run this program, the number -0.0225... should appear. When you hold down f-PREFIX you should see 001- 45 24 which is what is displayed when you pressed R/S. The sign nybble in this case contains a xF which means that a program line is being displayed. With a number in the display, nybble s will contain a 0 to 9 meaning shift the displayed decimal point right 0 to 9 places. If the display contains a matrix pointer then s will contain a xB. For this case

the name of the matrix is added to the display at some later time.

Writing numbers to the display register does not alter the display, since the calculator almost always rewrites the display after the operation performed. The one exception occurs while a program is running. The calculator considers a program to be a (sometimes long) single operation. Therefore, it does not use the display register until the program stops. The bad news is that changing the display register does not alter the display. Thus the display register can be used for scratch by a program.

The two control registers allow a greater degree of control over what is seen in the display. I call these registers control L and control R since they control what segments are on in the left and right half of the display respectively. These registers work via a bit map, meaning that when a bit is set in the register the corresponding segment will light in the display. In figure 6 I have illustrated the control registers in a manner that I hope will be useful to others. I have assigned every segment a unique two digit code. The first digit is the nybble number I've used throughout this article. The second digit is the segment number as shown in the top half of figure 6. The segments are numbered to aid finding in the bottom half of figure 6.

To use figure 6 photocopy the page. On the copy write the message you want to appear in the 'display' provided near the top of the figure (see V5N4P23d for a seven segment alphabet). Next x-out all the boxes in the bottom half of figure 6 that correspond to segments you want to light. Finally, using the bit numbers, add up the columns to get the number to be stored in the corresponding nybble. An example should help understanding. Suppose we want to write a 1 into the display. The top half of figure 6 tells us we want segments 2 and 6 of digit 1 to be on, which correspond to the code numbers of 12 and 16. In the bottom half of figure 6 we quickly find 16 as bit 8 of nybble 8 and 12 as bit 8 nybble 9. Thus, if we write the number (0)0000000880(000) to control L, a 1 will appear in the display. To light all segments in the first digit, including the comma and decimal point, one must write (0)080C000FC0(000) to control L. This obviously must be constructed via synthetic means.

We can now see a pattern in how the divide self test works. At the start of the test 2 bits are set in control L and the same 2 bits in control R. Pushing the keys in order causes the number to be shifted left one bit per keystroke. The display goes from four segments to two when two of the bits are shifted out of the register.

A DEMONSTRATION PROGRAM

I have written a program which writes the message 'HEllo PPC ' to the display while running. I have carefully selected the message (and program) so that no synthetic program lines need to be constructed. It does require the use of synthetic matrix 1 but

this is easily constructed.

```
          HEllo PPC program
001 LBL 0              015 LBL A
002 LBL 0              016 2
003 4                  017 0
004 4                  018 STO 1
005 LBL 2              019 1
006 x!                 020 STO 0
007 LBL 0              021 9
008 x<> 0              022 RCL g(i)
009 LBL 0              023 STO (USER) (i)
010 L.R.               024 1
011 DIM D              025 ENTER
012 LBL 8              026 1
013 GTO 3              027 0
014 GSB 2              028 RCL g(i)
                       029 STO (i)
                       030 LBL .0
                       031 GTO .0
```

To run;

```
19, DIM (i)
1, ENTER, 36, DIM A
1.000000044, rotate, STO I
GSB A
```

Lines 1 through 14 are not intended to be executed. This area contains the bit patterns that will later be transferred to the control registers. Line 22 recalls element (1,9) of matrix 1 which contains program lines 7 to 14. Line 23 stores this number in the control L register generating 'HEllo' in the left half of the display. Line 28 recalls the register containing program lines 1 to 7. When this is stored in control R, ' PPC ' appears on the right side of the display. Lines 30 and 31 loop since the message will disappear when the program is halted.

People who run this program will notice that I have used a blend of capital and small letters whereas I could have used all caps. This was done to avoid memory mapping which would have made the program more difficult and dangerous to run. It is left as an exercise for the reader to re-write the program to make the hello come out in all caps. This is not easy and I confess I've never tried it.

ADVANCED METHODS

It is possible to modify the return 2 register so that the last program line occurs anywhere in memory. If one sets EE of the return 2 register to xC0 then, all the numbered data registers can be accessed as program lines. Of course, program editing in this mode will shift around all the numbers in the number data registers. It is also possible to make the last program line lie in the status area. Remember that the quick, and sometimes only,

way to get to the last program line is to go to line 000 and then backstep once. The line numbers in the status area are generally >600 however, simple arithmetic shows the 'true' line number is >1600 meaning that the 1 has rolled off. Warning; when a program extends into the status area, it is impossible to edit a program. If you do try to insert or delete a line, the processor will crash.

If the effective exponent of the number in the X-register is greater than 99 then the figure 1 display codes can be seen directly (without using f-PREFIX). The effective exponent equals the exponent (EE) plus C, where C=1 if there is a carry out of nybble 1 when you normalize the mantissa and C=0 otherwise. To see this enter the program LBL .0, LN, LN, INT, y^x, LBL A and recall the register containing the program entered. The message roPE--- should appear.

FUTURE PROSPECTS

Currently I have only synthesized a few of the unused two-byte op-codes. I can report that often a synthetic instruction will display the keycodes for a known function and often the synthetic appears to preform the displayed function. However, I have seen the display show STO .0 but the operation acts like STO (i). If anyone out there would like to search two-byte op-codes, I suggest that that person concentrate on the column number, in figure 4, that equals his PPC number mod(16). I suspect that most two-byte instructions will be uninteresting. Therefore, until I start getting tons of mail, I will act as the central clearinghouse for negative results. If you send results to me, please include what keycodes were displayed, and what the synthetic function appears to do. If enough people respond, I will summarize the results for publication including names and numbers of contributers. I would be interested if anyone can figure out the use of any status register or part of that I've marked with a '?'. Of course, if you find anything exciting, you should also send the information to PPC directly.

Warning; the synthetic function f-MATRIX A (which displays the keycodes for f-MATRIX-SST) is a nasty two-byte instruction. Executing this instruction with certain numbers in the X-register causes the status registers (including the map register) to be altered. I don't know what it is doing but it is definitely doing something.

In closing I would like to point out that I think the HP-15C is a very well designed scientific calculator (the only serious flaw is that it is too slow). The 'inside' view that I've had of the HP-15C has also impressed me. I can now see some of the careful and detailed planning that needed to be done at HP in order to make this calculator a success.

Esther Hu, Jack Saba, Rick Shafer and Andy Szymkowiak all contributed useful comments to this paper.

Allyn F. Tennant (10106)
2719 Curry Dr.
Adelphi, MD 20783

dec	hex	display
10	A	r
11	B	-
12	C	o
13	D	P OR u
14	E	E
15	F	

Fig 1

Display codes

(when using f-PREFIX)

Fig 2 Synthetic Matrices

MATRIX	DISPLAY	DIMENSION	POINTS AT BASE OF
0	b	C	E
1	=_	A	POOL
2	⊐	B	DATA
3	n_	C	A
4	⊔	A	B
5	= USER	B	C
6	⌝	C	D
7	∂_	A	E
8	d_	B	DATA
9	⊐	C	DATA
A	A	A	A
B	b	B	B
C	C	C	C
D	d	D	D
E	E	E	E
F		B	D

Fig 3 One-Byte Op-Codes

Right Nybble

Left Nybble	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	LBL 0	LBL 1	LBL 2	LBL 3	LBL 4	LBL 5	LBL 6	LBL 7	LBL 8	LBL 9	LBL A	LBL B	LBL C	LBL D	LBL E	
1	GTO 0	GTO 1	GTO 2	GTO 3	GTO 4	GTO 5	GTO 6	GTO 7	GTO 8	GTO 9	GTO A	GTO B	GTO C	GTO D	GTO E	
2	GSB 0	GSB 1	GSB 2	GSB 3	GSB 4	GSB 5	GSB 6	GSB 7	GSB 8	GSB 9	GSB A	GSB B	GSB C	GSB D	GSB E	
3	RCL 0	RCL 1	RCL 2	RCL 3	RCL 4	RCL 5	RCL 6	RCL 7	RCL 8	RCL 9	RCL A	RCL B	RCL C	RCL D	RCL E	
4	STO 0	STO 1	STO 2	STO 3	STO 4	STO 5	STO 6	STO 7	STO 8	STO 9	STO A	STO B	STO C	STO D	STO E	
5	RCL .0	RCL .1	RCL .2	RCL .3	RCL .4	RCL .5	RCL .6	RCL .7	RCL .8	RCL .9	RCL gA	RCL gB	RCL gC	RCL gD	RCL gE	
6	STO .0	STO .1	STO .2	STO .3	STO .4	STO .5	STO .6	STO .7	STO .8	STO .9	STO gA	STO gB	STO gC	STO gD	STO gE	
7	TEST 0	TEST 1	TEST 2	TEST 3	TEST 4	TEST 5	TEST 6	TEST 7	TEST 8	TEST 9	RCL MATA	RCL MATB	RCL MATC	RCL MATD	RCL MATE	
8	$x_{<>}$ 0	$x_{<>}$ I	DSE 0	DSE (i)	ISG 0	ISG (i)	RCL (i)	RCL I	GTO I	GSB I	RES A	RES B	RES C	RES D	RES E	
9	$x_{<>}$ I	$x_{<>}$ I	DSE 1	DSE I	ISG 1	ISG I	STO (i)	STO I	DIM (i)	DIM I	DIM A	DIM B	DIM C	DIM D	DIM E	
A	$\hat{y}$,r	RAN	CL×	FRAC	I	fREG	STO RES	RCL Σ	RCL DIMi	RCL DIMI	RCL DIMA	RCL DIMB	RCL DIMC	RCL DIMD	RCL DIME	
B	S	LSTx	RTN	ABS	R↑	RND	π	SIN^{-1}	COS^{-1}	TAN^{-1}	x^2	LN	LOG	%	Δ%	
C	.	ET	R/S	CHS	R↓	$x_{<>y}$	EEX	SIN	COS	TAN	√	e^x	10^x	y^x	1/x	
D	x!	>R	>H.MS	>RAD	HYP SIN	HYP COS	HYP TAN	RCL RES	RCL RAN	STO RAN	$P_{y,x}$	$R_{\Diamond Im}$	PSE	f Σ	LR	
E	$\bar{x}$	>P	>H	>DEG	HYP SIN^{-1}	HYP COS^{-1}	HYP TAN^{-1}	DEG	RAD	GRD	$C_{y,x}$	INT	x=0?	x<y?	Σ−	
F	0	1	2	3	4	5	6	7	8	9	+	−	×	/	Σ+	

Fig 4 Two-Byte Op-Codes

Left nybble of second byte

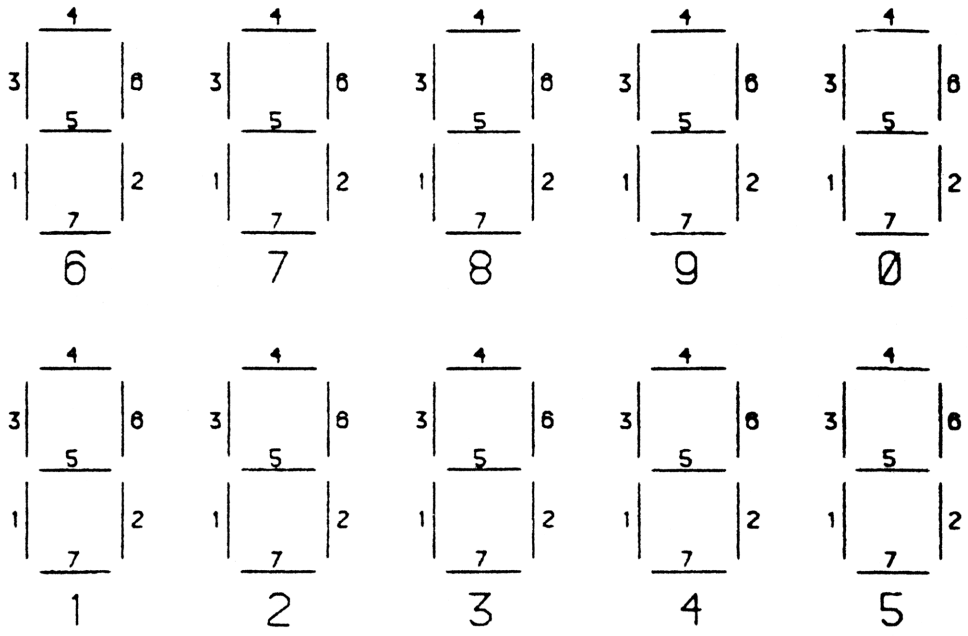
Left nybble of first byte

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A						RCL J	STO J									
B				RCL h	STO h				RCL i	STO i						
C									RCL +a	RCL +c	RCL -a	RCL -c	RCL Xa	RCL Xc	RCL /a	RCL /c
D									STO +a	STO +c	STO -a	STO -c	STO Xa	STO Xc	STO /a	STO /c
E		SOLVE a	SOLVE f	INTEG a	INTEG f	$x_{<>}$ b	$x_{<>}$ f	DSE b	DSE f	ISG b	ISG f					
F	LBL f	GTO f	GSB f	SF d	CF d	F? d	FIX d	SCI d	ENG d	MAT e	STO MATg					

Fig 5 Status Registers

REG dec	REG hex	NAME	REG dec	REG hex	NAME
31	1F	0	15	0F	0
30	1E	0	14	0E	0
29	1D	0	13	0D	0
28	1C	0	12	0C	0
27	1B	0	11	0B	0
26	1A	flags	10	0A	control R
25	19	dimension	9	09	control L
24	18	0	8	08	0
23	17	return 1	7	07	exponent
22	16	return 2	6	06	(?)
21	15	map	5	05	(?)
20	14	ran	4	04	display
19	13	last x	3	03	pointer
18	12	I	2	02	t
17	11	R1	1	01	z
16	10	R0	0	00	y

Fig 6 Control Registers



Nybble of Control R

Bit	(s)	1	2	3	4	5	6	7	8	9	Ø	(S	E	E)
1	Ø3	95	91	PRGM	8 ,	C	83	7 ,	73	6 ,	65	61		
2	Ø4	96	92	Ø7	8 .	97	84	7 .	74	6 .	66	62		
4	Ø5	Ø1	93	Ø ,	D.MY	9 ,	85	81	75	71	G	63	RAD	
8	Ø6	Ø2	94	Ø .	87	9 .	86	82	76	72	67	64	77	

Nybble of Control L

Bit	(s)	1	2	3	4	5	6	7	8	9	Ø	(S	E	E)
1	35	31	55	51	4 ,	43	25	21	13	USER	f	g		
2	36	32	56	52	4 .	44	26	22	14	27	37	47		
4	3 ,	33	—	53	1 ,	45	41	23	15	11	2 ,	5 ,	BEGIN	
8	3 .	34	17	54	1 .	46	42	24	16	12	2 .	5 .	57	