HP-41 ADVANCED

PROGRAMMING TECHNIQUES

HP-41C Advanced Programming Techniques

by Randy Cooper and John Nickel

P.O. Box 218648
Houston, Texas 77218

ACKNOWLEDGEMENTS

Acknowledgements are made to: Hewlett-Packard for designing the HP-41C Hand-Held Computer and all of its peripherals with the user in mind, and to the PPC.

PPC

The PPC (Personal Progamming Center), is a non-profit California corporation whose members are dedicated to promoting 'personal' computing. The PPC has thousands of members around the world and even behind the Iron Curtain. They come from all walks of life: insurance salesmen, chemists, landlords, and engineers. To the PPC member, a 'personal' computer is one that can be carried about one's person...available at any time. The first such device was the HP-65 calculator. The PPC had its origins in the HP-65 Users Group, which was formed in 1974 by Richard Nelson. The PPC publishes approximately 10 issues of the PPC Calculator Journal during a calendar year. A PPC Computer Journal has been introduced in the last year to accomodate the many HHC's and provide adequate coverage for both calculator-type and more complex HHC's. The PPC has undertaken many projects, the most astounding to date has been the PPC ROM. It is an 8K custom ROM containing programs for synthetic programming, financial calculations, data plotting and manipulation, and extended control of the 41C.

Persons interested in obtaining more information about the PPC should send a self-addressed, 9" x 12" stamped envelope (~~2~~ 3 oz first class mail) to:

PPC CALCULATOR JOURNAL
~~Dept T & R~~
2545 West Camden Place
Santa Ana, CA 92704

HP-41 ADVANCED PROGRAMMING INDEX

SECTION I

MACHINE ARCHITECTURE (ACTUAL/FUNCTION)

SECTION I

Machine Architecture (Actual/Functional)

The HP-41C Hand Held Computer is truly a computer in its own right. It has all of the functional components that make up a computer: a CPU, RAM storage, programmed ROM, off-line mass storage, ports, interfaces, and peripherals. All of this has been designed into a familiar calculator style package. With all of this sophistication, it has not abandoned the friendly nature of a calculator. A calculator is there in your hand when you need it. It performs calculations upon your command, with single keystrokes wielding powerful and complex operations. Like any computer a working knowledge of its architecture is necessary to utilize its full capabilities. The following are terms with which the architecture of the 41C will be described:

Definitions:

BIT - a binary digit representing an ON/OFF, SET/CLEAR or a 1/0 state.

BYTE - a contiguous group of 8 bits. This is the usual unit of information worked with in most computers. When used as a unit of communication information, it may be called a "character".

NIBBLE - a contiguous group of 4 bits. This is sufficient to define a single hexadecimal or binary coded decimal (BCD) digit.

REGISTER - a hardware grouping of 7 bytes used by the HP-41C to denote stack registers, data registers, and status registers. This is the same 'register' that HP refers to in their documentation.

CHARACTER - one byte of data corresponding to a single displayed or printed character.

RAM - an acronym for 'Random Access Memory'. In all references here it will mean any semiconductor memory within the 41C that is normally alterable, including extended memory.

ROM - an acronym for 'Read Only Memory', this type of semiconductor memory is not alterable and refers to that within the 41C and in external modules or peripherals.

FUNCTIONAL DESCRIPTION

The 41C contains machine language programs in ROM that tell it how to behave as a 41C. Machine language is the natural language of the CPU. Each machine language instruction is made up of one or more 10 bit words in ROM memory. This is in contrast to calculator instructions which are from one to sixteen bytes long and may reside in ROM or RAM. There are functionally three programs: an editor, a run-time interpreter, and an operating system. Any time the 41C is turned on, you are interacting with one of these programs. These programs are interlaced to make the most efficient use of memory space and they utilize many common subroutines.

The editor accepts keyboard input as instructions and enters them into RAM memory, displaying the contents of memory as a calculator instruction and adding a computed line number. This is the program that you converse with when you invoke 'PRGM' mode and key in your calculator program. It takes the function or string that you key in and converts it into the appropriate calculator instruction code for subsequent review or execution.

The run-time interpreter is initiated by a R/S, XEQ, time module control alarm, auto-execute power-on or card read, or a wand execute command. It reads instructions out of memory and performs all operations needed to execute them, giving diagnostics and interfacing with peripherals where necessary. It also compiles numeric branches and checks for auto key assignments. When this program is running, you are in 'RUN' mode.

The operating system accepts non-programmable as well as programmable commands and executes them on a one-to-one basis. Most of the housekeeping is done by the operating system. When you first turn the 41C on, this program will inspect a RAM status register for system integrity and execute a MEMORY LOST if corrupted. It also sets all of the display annunciators to match their respective internal states. The key assignment registers are checked for any deletions that can be recovered for use. If the auto execute flag is set it will transfer control to the run-time interpreter. Peripheral status is checked and the value of the X-register is displayed according to the current display mode. Finally, this program will await your command; but not patiently...if more than ten minutes elapse between keystrokes with no program running and the continuous-on flag is clear, it will turn itself off. This program has several modes: 'normal', 'USER', and 'ALPHA'.

All of these programs are contained in three internal ROM's. The ROM for the 41C differs from other computers in that it has 10 bits per word, rather than eight. Most ROM's that perform complex tasks are programmed in machine language. This should not be confused with application ROM's that are usually ROM-based versions of calculator language programs.

Figure 1.1 shows an example of some 41C assembly language instructions. Like the assembly language instructions on most computers, it is unreadable to most people. The assembly language instruction mnemonics shown here were created by PPC members who are deeply interested in this type of programming. Assembly language programming on the 41C requires much additional hardware and knowledge of the internal operation of the machine. This method of progamming is beyond the needs and capabilities of most users.

```
READ 4(X)
?NC XQ BINBCD
       [01AF]
READ 6(N)
WRIT 12(b)
JNC -0B
SLCT Q
R=R-1
?FSET 6
RCR 2
C<>B ALL
SETF 1
CLRF 2
WRIT 11(a)
```

Figure 1.1 — Example of HP-41C Assembly Language

Some advantages of assembly language are: adding special purpose instructions such as decimal-to-hexadecimal conversion or alpha sorting, and dramatic speed increases can be realized over similar calculator instructions. The major disadvantages are: that it requires much additional hardware, assembly language conversion must be done by hand or in another computer, and there are no diagnostic routines to aid the assembly language programmer.

NOTE: Assembly language programming is not supported by Hewlett-Packard.

ROM SPACE ORGANIZATION

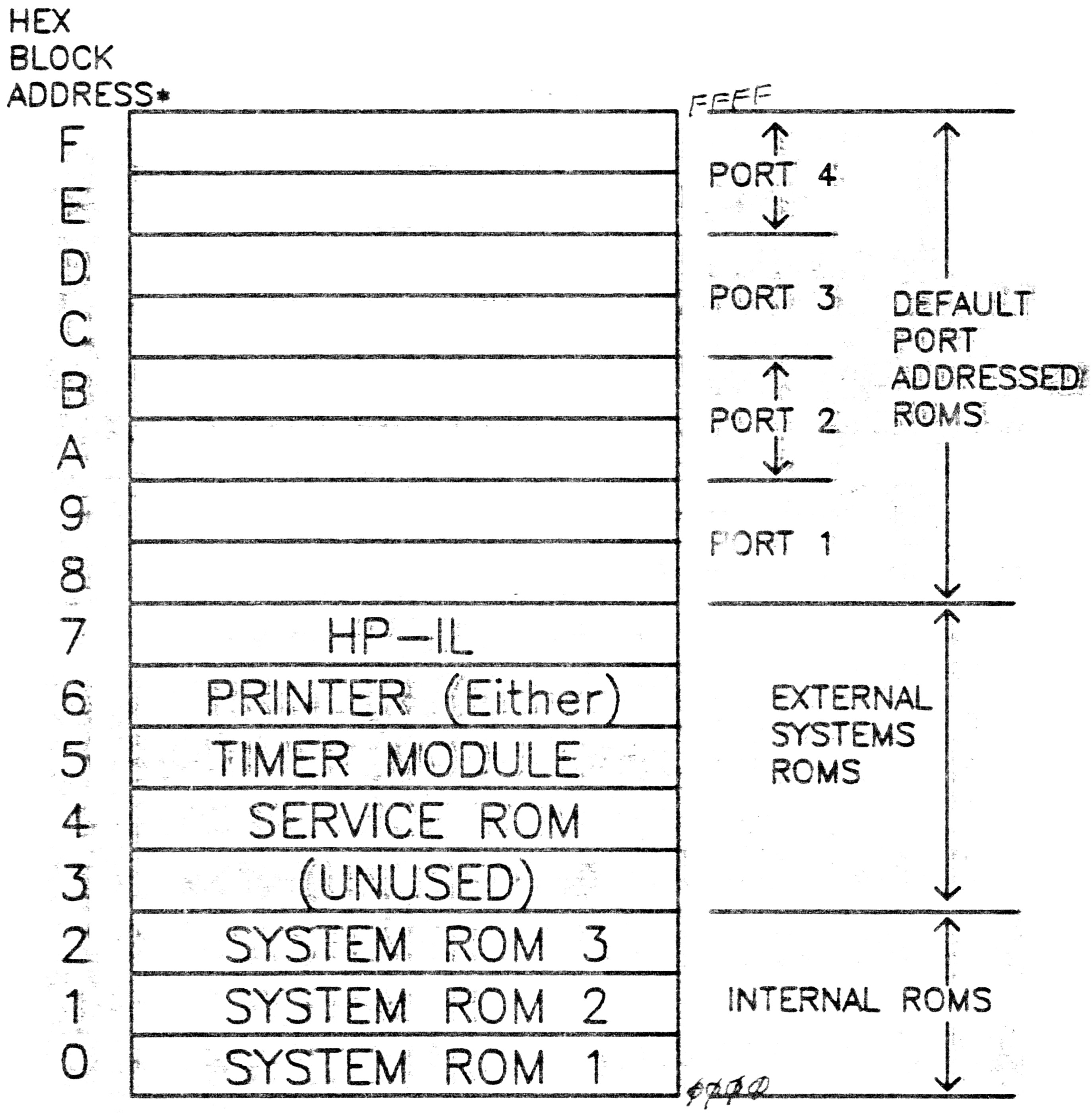
Figure 1.2 shows the layout of ROM memory space within the 41C. The 41C can address up to 64K words of ROM memory in 16 blocks of 4K words per block. The three internal ROM's mentioned earlier reside in blocks 0 through 2. All other ROM space is for external ROM's. These ROM's may be of two possible types: system ROM's or application ROM's.

As previously mentioned, the application ROM's are simply calculator instructions stored in a ROM memory. An exception to this rule is the Petroleum Fluids Pac ROM which contains both calculator instructions and machine code. It uses machine code in its "all-base" conversion factor program to increase execution speed. When plugged in, these ROM's take on an address defined by the port in which they reside. The mechanism that creates these addresses is built into the port structure itself.

A system ROM is programmed entirely in machine code and becomes a part of the operating system once it is attached. These ROM's have a predetermined address regardless of which port into which they are plugged. As shown in Figure 1.2, the timer module (Part no. 82182) has an address at block 5 no matter where it is plugged in. Most peripherals have dedicated addresses in the first half of ROM memory. This insures that there are no address conflicts whenever any combination of devices are plugged in.

The space allocated for the 'SERVICE ROM' is important in that when the 41C is powered up, a check is made for the existence of a ROM at this address. If one does exist, all control is transfered to it. It contains programs for trouble-shooting the display, keyboard, and memory. The 'SERVICE ROM' is used by Hewlett-Packard for service check-out of a 41C and is not available to the public.

This transfer of control to a ROM at block 4 is used by some assembly language programmers to create their own operating system, independent of the regular operating system. It may be speculated that someone may use this feature to create other languages for the 41C such as BASIC, FORTRAN, or some application specific language.



*1 BLOCK = 4,096 (4K) 10 BIT WORDS

Figure 1.2 — ROM Space Addressing

RAM SPACE ORGANIZATION

The RAM space of the 41C is fairly complex in that it is dynamically allocated by the operating system for particular tasks. Figure 1.3 shows a layout of the RAM space of the 41C, including that added by the Extended Functions Module (XFM) and the Extended Memory Modules (XM). The RAM memory is organized into 'registers' that are 7 bytes wide. The way that registers are utilized for storage will be covered later in this section.

The first sixteen registers within RAM space are reserved for the RPN stack and status registers. These registers contain important information as to how the rest of RAM memory is to be interpreted. These registers and their uses will be covered in detail in the section entitled 'Synthetic Programming'.

There is a 'void' between the top of the status registers and the bottom of the XFM memory. Hewlett-Packard may use this area in future revisions or peripherals for the 41C. The XFM memory is added by plugging its module into a port. It adds 127 registers for use as on-line files. Each XM module added will increase this file capability by 238 registers, up to a maximum of 603 registers. The organization and use of the extended memory will be examined in Section IV.

The rest of RAM memory varies in size, depending upon whether you have a 41CV or a 41C and a complement of RAM modules. In any case, Figure 1.4 shows the relative layout of what will be called 'User RAM'. The data registers are organized in descending numerical order from the top of user RAM. The first program stored will begin at the first register below register R00 and continue on downward through memory. Other programs are stored below this one as they are entered or cleared. Key assignments are stored from the bottom of user RAM and continue on up as required, with two key assignments per register used. The space just above the key assignments is used up by timer module alarms. The remaining space between the uppermost key assignment or alarm and the permanent '.END.' is what is displayed as free register space in the display '00 REG nnn', where nnn is the number of free registers. This number will vary based upon the amount of key assignments made and deleted, alarms set, program files (packed and unpacked), and the current SIZE.

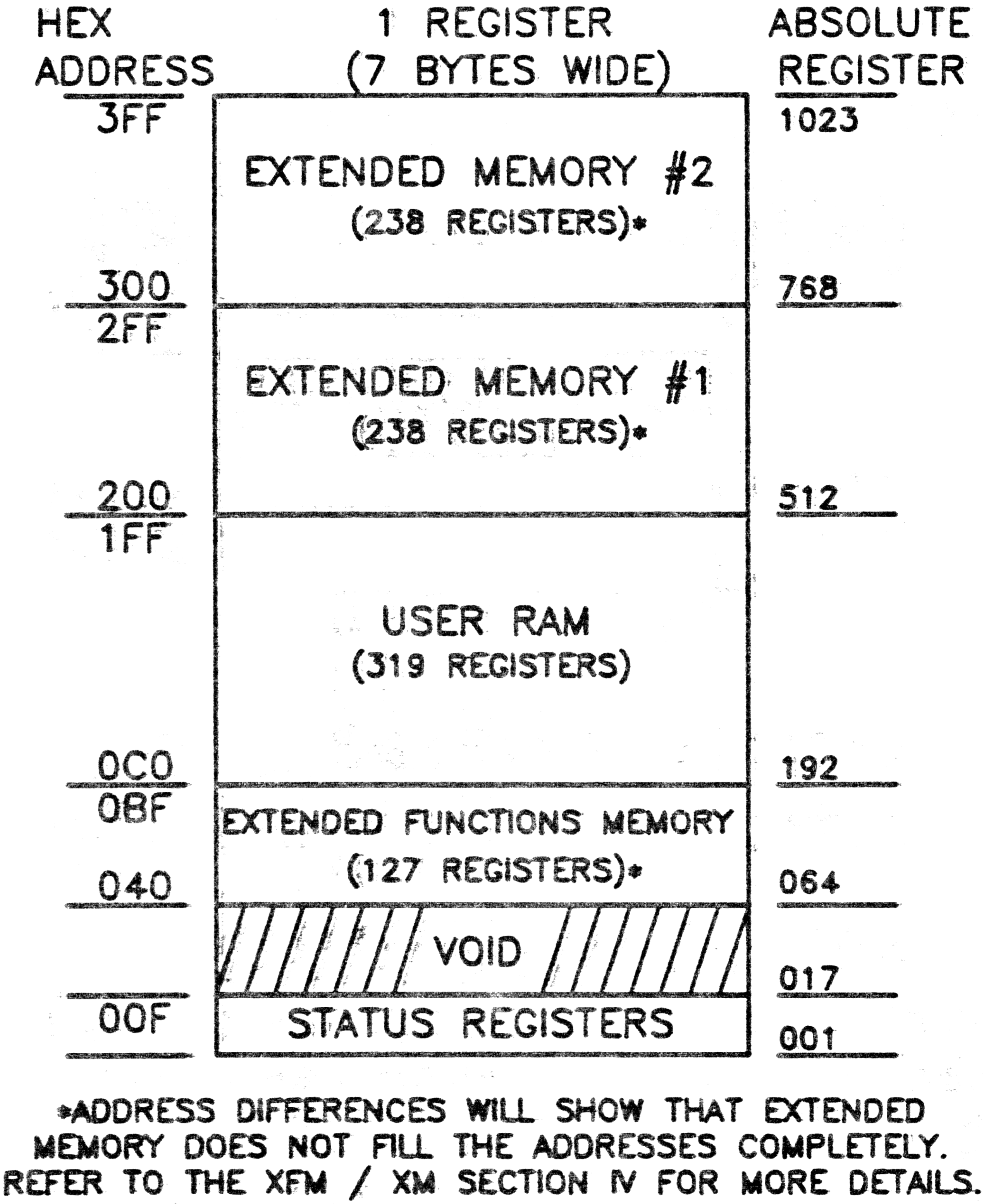


Figure 1.3 — RAM Space Addressing

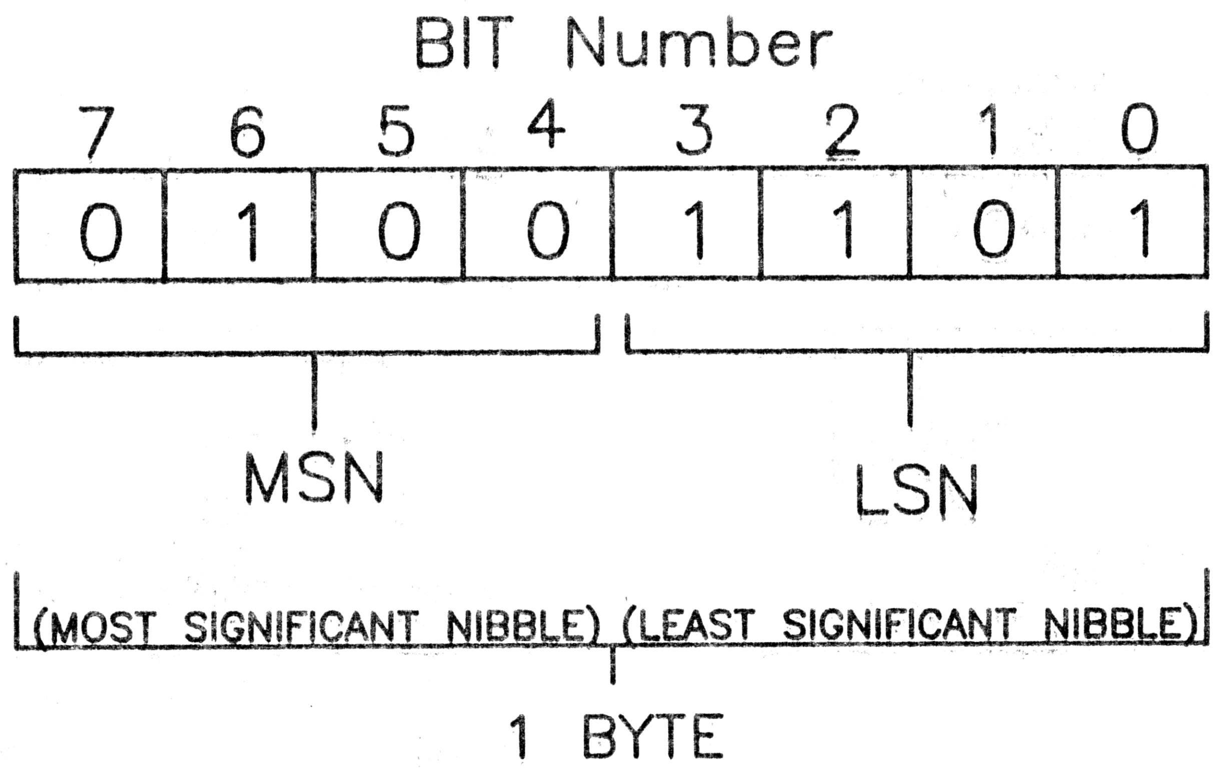


Figure 1.5 — Data Nomenclature

DATA REPRESENTATION AND NOMENCLATURE

As in most microcomputers the 41C's memory and most of its communication with peripherals have to do with bytes. A byte is a block of eight contiguous bits and is usually represented by a number in decimal notation from 0 to 255, or in hexadecimal notation (base 16) by two characters '00' to 'FF'. Figure 1.5 shows how we will reference the parts of the byte in further discussions. The eight bits will be numbered from 0 to 7 from the right to the left. Further, bits 7 through 4 will be designated as the 'Most Significant Nibble' (MSN) and bits 3 through 0 will be the 'Least Significant Nibble' (LSN).

Figure 1.5 - Data Nomenclature

BCD NOTATION

The 41C uses the nibble in numeric data to represent a single decimal digit internally in what is termed 'Binary Coded Decimal' notation (BCD). That is, two consecutive nibbles having the decimal notation of '9' would be interpreted as being 99 in base 10 rather than base sixteen. Figure 1.6 gives some examples of BCD versus hexadecimal (HEX) notation.

	Bit Pattern in byte (MSN) (LSN)	HEX NOTATION	BCD NOTATION	BASE 10 VALUE IF INTERPRETED AS HEX	BCD
(A)	0001 0010	12	12	18	12
(B)	0100 0001	41	41	65	41
(C)	1001 0111	97	97	151	97
(D)	0110 0111	67	67	103	67
(E)	1111 1010	FA	NOT VALID	250	---
(F)	1001 1001	99	99	153	99
(G)	0010 1011	2B	NOT VALID	43	---
(H)	0100 0101	45	45	69	45

Figure 1.6 - HEX vs. BCD Notation

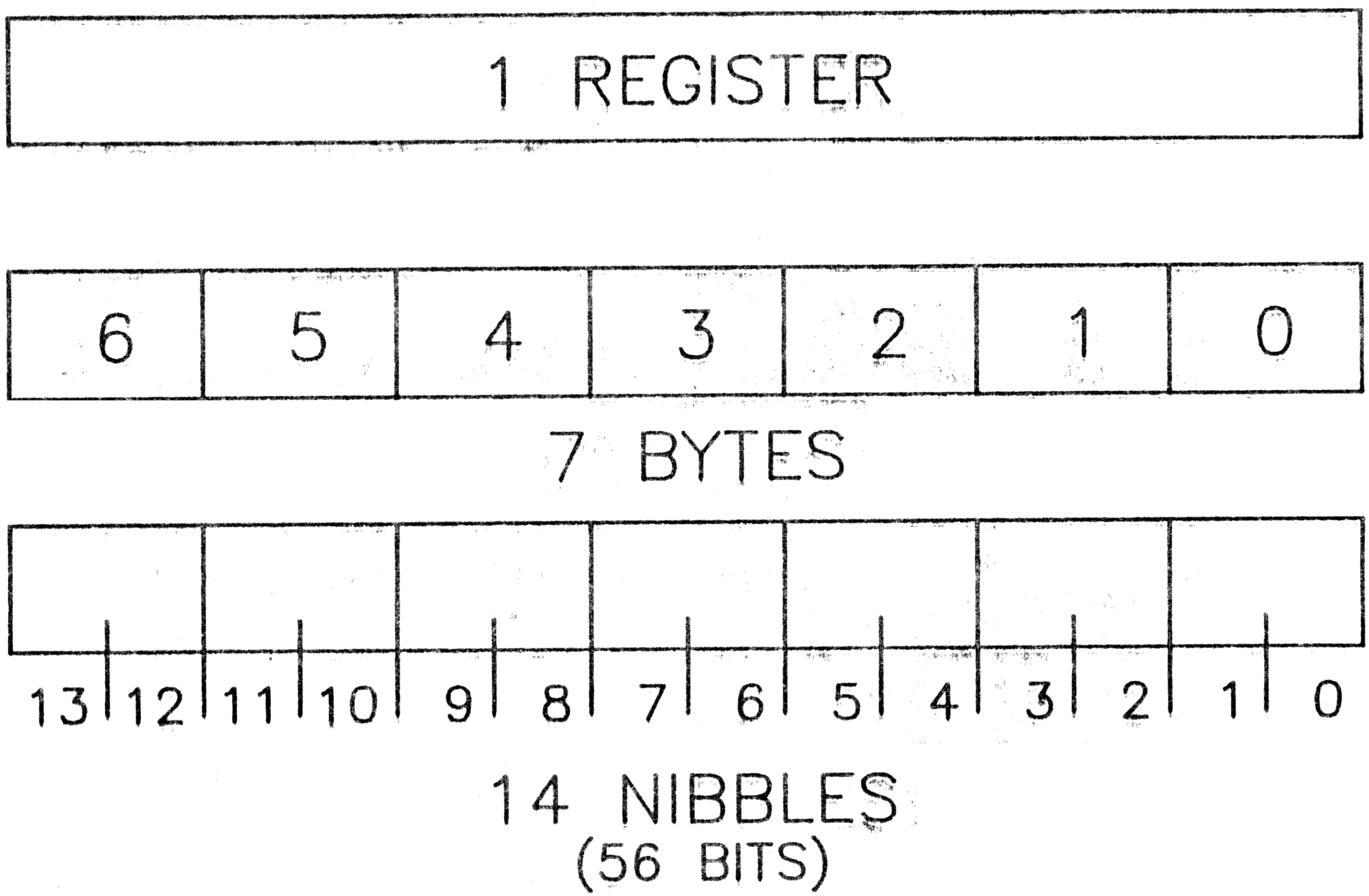


Figure 1.7 – Register Organization

As is evident, BCD notation is the same as HEX notation except that the nibbles are never allowed to be greater than '9'. The 41C uses BCD notation to store numeric data. To a computer, BCD is not the best notation to use for computations, but it is the easiest notation to convert into human readable form. When the 41C is operated as a manual calculator, the result of every operation must be put into the display. BCD notation makes this a quick and easy task. On large computers where calculations and throughput are important, others notations are used; however, the programming to convert these notations to a readable form for output consumes thousands of bytes. It would be uneconomical to use that much memory in a hand-held unit just to display a number.

THE REGISTER

The register is the gross unit of RAM memory in the 41C. A register may be said to be 7 bytes long, 14 nibbles long, or 56 bits long. A unit of 7 bytes is unusual in computers, but some unit of storage is needed when talking about or managing memory. The unit used must have a physical meaning for the user. Since a single register is required to store a number in a numbered storage register, the unit of storage was chosen as the register. Figure 1.7 show the convention used when referring to parts of a register. HP also refers to this unit as a register in their documentation.

Figure 1.7 - Register Organization

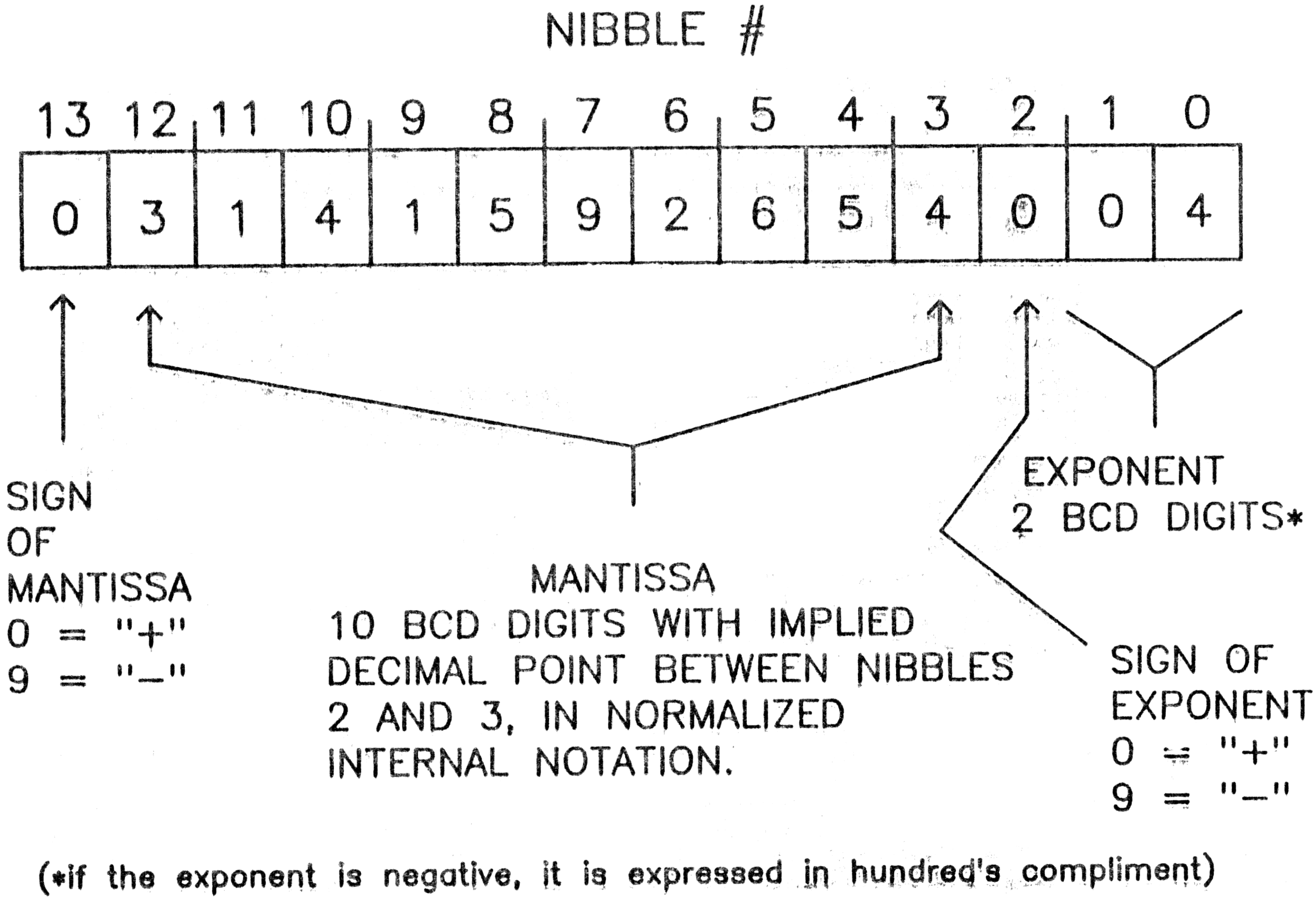


Figure 1.8q – Numeric Data Storage

NUMERIC DATA

The 'register' is used in many ways inside the 41C. Figure 1.8a shows the organization of the register when it contains numeric data.

Exponent is store as 100's complement.
ie: 10^{-04} 100
− 04
996 → Hex of Exponent

Figure 1.8a - Numeric Data Storage

Nibble #13 contains the sign of the mantissa. A '0' here denotes a positive number and a '9' denotes a negative number. Nibbles 12 through 3 contain the ten digits of the mantissa in BCD notation. Nibble #2 is the sign of the exponent using the same convention as in nibble #13. The exponent (base 10) is stored in the last two nibbles, 1 and 0. The exponent, if positive, is stored exactly as it is written in BCD. When negative, the exponent is stored in 'hundreds complement' form. This means that the exponent is subtracted from 100 and the resulting two digits will be stored in its place.

	NUMBER AS IT IS ENTERED INTO THE 41C	NORMALIZED STORAGE NIBBLE NUMBER 13	12	11	10	9	8	7	6	5	4	3	2	1	0
(A)	6.082 E-26	0	6	0	8	2	0	0	0	0	0	0	9	7	4
(B)	-.0078	9	7	8	0	0	0	0	0	0	0	0	9	9	7
(C)	1492.1983	0	1	4	9	2	1	9	8	3	0	0	0	0	3
(D)	299.6 E-6	0	2	9	9	6	0	0	0	0	0	0	9	9	6
(E)	PI	0	3	1	4	1	5	9	2	6	5	4	0	0	0
(F)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(G)	987654 E-5	0	9	8	7	6	5	4	0	0	0	0	0	0	0
(H)	-10.101981	9	1	0	1	0	1	9	8	1	0	0	0	0	1
(I)	-9.999999999E-99	9	9	9	9	9	9	9	9	9	9	9	9	0	1

Figure 1.8b - Examples of Numeric Data Storage

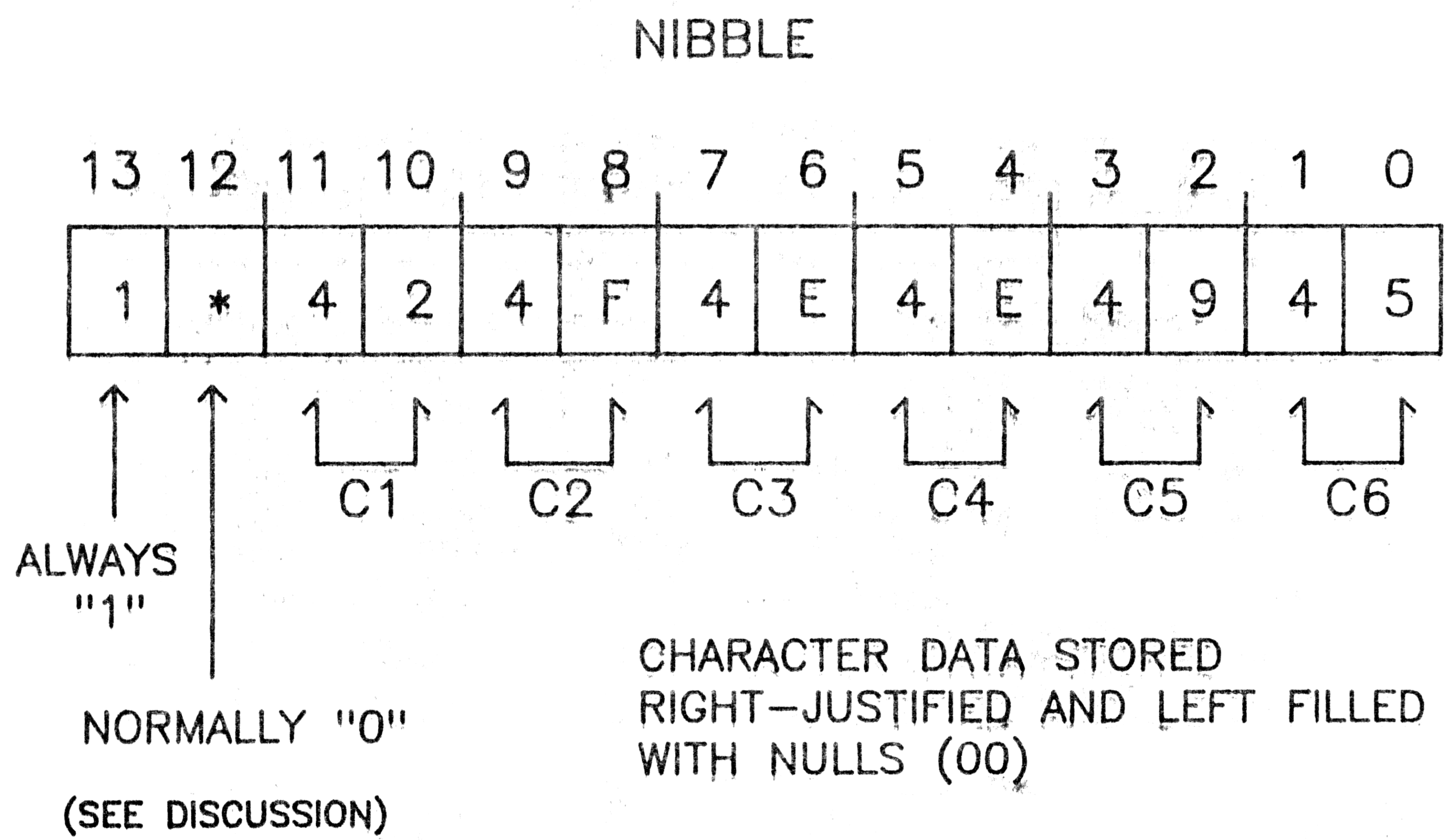


Figure 1.9a — Alpha Data Storage

Figure 1.8b shows some examples of how different numbers are stored. Note that no decimal point or 'E' is stored implicitly. The decimal point is assumed between nibbles 12 and 11. The number is normalized after data entry or numeric operations so that nibble #12 is non-zero, except if the entire register is zero.

ALPHA DATA

When an 'ASTO' operation is executed, an 'ALPHA DATA' value is stored into a status or numbered data register. This value may contain up to six characters, depending upon what was in the alpha register before the operation. Figure 1.9a shows the general layout of a register containing alpha data.

Figure 1.9a - Alpha Data Storage

The first nibble will have the value of '1'. The six bytes defined by nibbles 11 through 0 will contain the actual alpha data. This data will be right justified and filled to the left with nulls (00) if fewer than six characters are stored. The data stored for each character is the same as the printer ACCHR or XFM XTOA code in HEX format. (Note that the way a character displays and the way that it prints out are not necessarily the same. See Figure 1.9c for a table of displayed versus printable characters.)

Nibble #12 <u>may</u> contain a remnant from the seventh byte in the alpha register, depending upon how many characters are in the alpha register. It is this nibble that will cause two seemingly identical alpha data strings to compare as not equal.

To demonstrate:

1) PRGM off, ALPHA on
2) Key 'ABCDEF'
3) ASTO X
4) Append "G'
5) ASTO Y
6) ALPHA off
7) X=Y?

The display will say 'NO'. Press the backarrow key to see 'ABCDEF'. Press X<>Y to see 'ABCDEF'. Even though the strings appear equal, they are different. Now key the following:

8) ALPHA on
9) CLA
10) ARCL X
11) ASTO X
12) ALPHA off
13) X=Y?

The display should now say 'YES'. In the process of ARCL'ing the 'buggy' string, the problem was eliminated. This problem in some 41C's has been dubbed 'BUG 7' by the PPC. Whenever a bug is referred to, it will mean a deviation in calculator behavior than is to be expected from the Owner's documentation. Several 'bugs' have been cataloged by the PPC since the 41C was introduced. Many of these have been corrected in later revisions of the internal ROMs. In a complex machine there is always the chance that some features will not behave as expected.

Figure 1.9b shows some examples of register contents after an ASTO operation.

	CONTENTS OF ALPHA REGISTER BEFORE 'ASTO X'	CONTENTS OF X AFTER 'ASTO X' BYTE # 6	5	4	3	2	1	0
(A)	ABCDEF	10	41	42	43	44	45	46
(B)	ABC	10	00	00	00	41	42	43
(C)	X+Y=Z	10	00	58	2B	59	3D	5A
(D)	I.T.C.	10	49	2E	54	2E	43	2E
(E)	FILENAME?	1D	46	49	4C	45	4E	41
(F)	ITEM COST?	1F	49	54	45	4D	20	43
(G)	13 LETTERS...	10	31	33	20	4C	45	54
(H)	(clear)	10	00	00	00	00	00	00
(I)	NINETEEN LETTERS....	15	4E	49	4E	45	54	45

Figure 1.9b — Examples of Alpha Data Storage

The actual byte values used in the alpha register may be interpreted differently depending upon their use. In Figure 1.9c, it can be seen that a byte with the value of 01 will display as a 'stick man' in the 41C display but will print out to the printer as a small 'x'. The basic printable characters of the 41C tend to follow standard ASCII (American Standard Code for Information Interchange) convention. When using the Video Interface, ASCII convention is followed for all printable characters, and some control codes are implemented. Figure 1.9d shows the ASCII code as implemented by the Video Interface.

LEAST SIGNIFICANT NIBBLE	MOST SIGNIFICANT NIBBLE (SEE NOTE BELOW) 0	1	2	3	4	5	6	7
0	-	-	SP	0	@	P	`	p
1	-	-	!	1	A	Q	a	Q
2	-	-	"	2	B	R	b	r
3	-	-	#	3	C	S	c	s
4	-	-	$	4	D	T	d	t
5	-	-	%	5	E	U	e	u
6	-	-	&	6	F	V	f	v
7	-	-	'	7	G	W	g	w
8	BS	-	(	8	H	X	h	x
9	-	-	)	9	I	Y	i	y
A	LF	-	*	:	J	Z	j	z
B	-	ESC	+	;	K	[	k	{
C	-	-	,	<	L	\	l	|
D	CR	-	-	=	M	]	m	}
E	-	-	.	>	N	^	n	~
F	-	-	/	?	O	_	o	

Note: the video interface will display a printable character in reverse video if the most significant nibble is greater than 7. This in effect maps the MSN for reverse video characters from 8 to F instead of 0 to 7. To reverse a character using XTOA or ACCHR, just add 128 to its decimal character value.

Figure 1.9d - Video Interface ASCII Code Table

SECTION I - QUIZ

1. The HP-41C RPN stack (registers X, Y, Z, T, and L) occupy:

   a. 5 bytes
   b. 5 registers
   c. 35 bytes
   d. b and c → correct
   e. a and c

2. When the HP-41C is turned 'ON', the first program the user interacts with is:

   a. the editor
   b. the run-time interpreter
   c. the operating system
   d. c or b (if flag 11 set) → correct
   e. none of the above

3. HP-41C assembly language is:

   a. organized in 10-bit words
   b. usually programmed into ROM's
   c. unintelligible to the average user
   d. not supported by Hewlett-Packard
   e. all of the above → correct

4. A bare HP-41CV without peripherals has:

   12K or 3 ROM
   a. 3K of ROM's built in
   b. 319 registers of ROM available
   c. 319 registers of User RAM available (exc. status)
   d. a and b
   e. a and c depending on interpitation.

5. A byte contains the bit pattern: 0100 0101
   It is:

   a. 45 coded in BCD
   b. the ASCII letter "E"
   c. 69 coded in BCD → X>Y?
   d. a and b
   e. b and c

SECTION I - QUIZ (continued)

6. The register data in Figure 1.8a is the result of which of the following keystrokes:

   a. 3.14159265, ENTER↑, 1E4, *
   b. PI, 1E4, +
   c. PI, 4, Y↑X
   d. 31415.92654 E-8, ENTER↑
   e. 1E4, PI, *

7. Which of the following nibble decoding could not have been the result of an ASTO operation?

   a. 01 41 42 43 44 45 46
   b. 1F 42 4F 4E 4E 49 45
   c. 10 00 00 00 00 00 00
   d. 13 00 00 00 20 20 20
   e. 1F F7 00 00 00 41 2D

8. A value displays in the X register as: -3.14
   Which of the following nibble patterns could produce the display?

   a. 93 14 15 92 54 00 00
   b. 10 2D 33 2E 31 34 20
   c. 93 13 50 00 00 00 00
   d. a and c
   e. all of the above

9. A decoding of the X register yields the following nibbles:

   01 23 45 66 89 09 56

   If the mode is ENG 5, the display will show:

   a. 1.23456 -56
   b. 12.3456 -44
   c. 12.3457 -57
   d. 1.23457 -44
   e. 12.3457 -44
   f. None of the above 12.3457 -45

10. If the highest numbered storage register in an HP-41CV is register 14, and it has 13 packed key assignments, no timer module, and the PRGM mode display shows: "^0 REG 256", then the program space occupies:

   a. 42 registers
   b. 41 registers
   c. 36 registers
   d. 35 registers
   e. none of the above

SECTION I - QUIZ

1. The HP-41C RPN stack (registers X, Y, Z, T, and L) occupies:

   a. 5 bytes
   b. 5 registers
   c. 35 bytes
   d. b and c
   e. a and c

2. When the HP-41C is turned 'ON', the first program the user interacts with is:

   a. the editor
   b. the run-time interpreter
   c. the operating system
   d. c or b (if flag 11 set)
   e. none of the above

3. HP-41C assembly language is:

   a. organized in 10-bit words
   b. usually programmed into ROM's
   c. unintelligible to the average user
   d. not supported by Hewlett-Packard
   e. all of the above

4. A bare HP-41CV without peripherals has:

   a. 12K of ROM's built in
   b. 319 registers of ROM available
   c. 319 registers of User RAM available (exc. status)
   d. a and b
   e. a and c

5. A byte contains the bit pattern: 0100 0101
   It is:

   a. 45 coded in BCD
   b. the ASCII letter "E"
   c. 69 coded in BCD
   d. a and b
   e. b and c

SECTION I - QUIZ (continued)

6. The register data in Figure 1.8a is the result of which of the following keystrokes:

   a. 3.14159265, ENTER↑, 1E4, *
   b. PI, 1E4, +
   c. PI, 4, Y↑X
   d. 31415.926554 E-8, ENTER↑
   e. 1E4, PI, *

7. Which of the following nibble decoding could not have been the result of an ASTO operation?

   a. 01 41 42 43 44 45 46
   b. 1F 42 4E 4E 4E 49 45
   c. 10 00 00 00 00 00 00
   d. 13 00 00 00 20 20 20
   e. 1F E7 00 00 00 41 2D

8. A value displays in the X register as: -3.14
   Which of the following nibble patterns could produce the display?

   a. 93 14 15 92 54 00 00
   b. 10 2D 33 2E 31 34 20
   c. 93 13 50 00 00 00 00
   d. a and c
   e. all of the above

9. A decoding of the X register yields the following nibbles:

   01 23 45 66 89 09 56

   If the mode is ENG 5, the display will show:

   a. 1.23456 -56
   b. 12.3456 -45
   c. 12.3457 -57
   d. 1.23457 -45
   e. 12.3457 -45

SECTION II

CALCULATOR INSTRUCTION SET/ SAVING BYTES

Section II

Calculator Instruction Set/Saving Bytes

The use of memory space for programming is transparent to the casual programmer in most situations, except when a keystroke yields 'PACKING'.......'TRY AGAIN'. The user is then faced with deleting key assignments, alarms, and programs or resizing his memory for fewer data registers.

The serious programmer of the 41C wants more than his work to fit within the confines of memory, but to use only as much memory as is necessary to perform all of the algorithms required. This is more than a matter of pride; using less memory has many advantages:

More space is available for alarms, key assignments, or data registers.

Smaller, cleaner programs tend to run faster.

Fewer card reads or less extended memory is needed to save a program.

Memory space saved can be used for enhanced user prompting or error checking.

When writing programs for ROM installation, more programs can fit into the space allowed.

If most users will be keying the program in manually, a shorter program would be appreciated.

In order to understand byte saving techniques, the manner in which the 41C stores calculator instructions into memory will be discussed. This section will look at byte utilization of comparable instruction combinations and the classic case of subroutine usage versus in-line code. This section is not an attempt to show a lot of 'tricks' to save memory. There are several excellent texts available that list such 'tricks'. Calculator Tips and Routines, by John Dearing, is an especially good one and is the product of many programmers and many years of calculator programming experience. This section is intended to be an introduction to the 41C's instruction set and good programming techniques.

INSTRUCTION SET

Definitions:

HEX TABLE - a matrix of 256 boxes (16 by 16) containing the 41C calculator instruction set. It may also contain the character displayed or printed by the 41C when that particular byte code is used. A hex table is an excellent guide toward understanding the 41C instruction set. Figure 2.1 shows such a table.

PREFIX - the first byte of a two byte instruction. An example is ARCL 02; here the ARCL is the prefix and is represented by the hex code '9B'.

POSTFIX - the second byte of a two byte instruction that acts as a pointer to a flag or register. If the highest order bit of the postfix is '1', the postfix is understood to be an 'IND'irect pointer. For example, the byte sequence '90 10' means RCL 16; but the sequence '90 90' is interpretted as RCL IND 16.

The 41C calculator instruction set can be organized into the following categories according to the number of bytes required for storage:

ONE BYTE INSTRUCTIONS

TWO BYTE INSTRUCTIONS

THREE BYTE INSTRUCTIONS

FOUR TO SIXTEEN BYTE INSTRUCTIONS

Figure 2.1 shows the 41C instruction set arranged in a table to show the hexadecimal equivalent of the base instruction. The base instruction is the first byte of the program line stored in memory. When refering to a multiple byte instruction, this byte code will be referred to as the 'PREFIX'. The second byte of a two byte instruction will be referred to as the 'POSTFIX'. This terminology will become evident as the various instructions are examined.

HP-41C/CV HEX TABLE

M \ L	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	L \ M
0	NULL 00 0	LBL 00 01 1	LBL 01 02 2	LBL 02 03 3	LBL 03 04 4	LBL 04 05 5	LBL 05 06 6	LBL 06 07 7	LBL 07 08 8	LBL 08 09 9	LBL 09 10 10	LBL 10 11 11	LBL 11 12 12	LBL 12 13 13	LBL 13 14 14	LBL 14 15 15	0
1	0 16 16	1 17 17	2 18 18	3 19 19	4 20 20	5 21 21	6 22 22	7 23 23	8 24 24	9 25 25	. 26 26	EEX 27 27	NEG 28 28	GTO T 29 29	XEQ T 30 30	SPARE 31 31	1
2	RCL 00 32 32	RCL 01 33 33	RCL 02 34 34	RCL 03 35 35	RCL 04 36 36	RCL 05 37 37	RCL 06 38 38	RCL 07 39 39	RCL 08 40 40	RCL 09 41 41	RCL 10 42 42	RCL 11 43 43	RCL 12 44 44	RCL 13 45 45	RCL 14 46 46	RCL 15 47 47	2
3	STO 00 48 48	STO 01 49 49	STO 02 50 50	STO 03 51 51	STO 04 52 52	STO 05 53 53	STO 06 54 54	STO 07 55 55	STO 08 56 56	STO 09 57 57	STO 10 58 58	STO 11 59 59	STO 12 60 60	STO 13 61 61	STO 14 62 62	STO 15 63 63	3
4	+ 64 64	− 65 65	* 66 66	/ 67 67	X<Y? 68 68	X>Y? 69 69	X≤Y? 70 70	Σ+ 71 71	Σ- 72 72	HMS+ 73 73	HMS- 74 74	MOD 75 75	% 76 76	%CH 77 77	P→R 78 78	R→P 79 79	4
5	LN 80 80	X↑2 81 81	SQRT 82 82	Y↑X 83 83	CHS 84 84	E↑X 85 85	LOG 86 86	10↑X 87 87	E↑X-1 88 88	SIN 89 89	COS 90 90	TAN 91 91	ASIN 92 92	ACOS 93 93	ATAN 94 94	DEC 95 95	5
6	1 X 96 96	ABS 97 97	FACT 98 98	X≠0? 99 99	X>0? 100 100	LN1+X 101 101	X<0? A 102	X=0? B 103	INT C 104	FRC D 105	D→R E 106	R→D F 107	HMS G 108	HR H 109	RND I 110	OCT J 111	6
7	CLΣ T 112	X<>Y Z 113	PI Y 114	CLST X 115	R↑ L 116	RDN M 117	LASTX N 118	CLX O 119	X=Y? P 120	X≠Y? Q 121	SIGN ⊢ 122	X≤0? a 123	MEAN b 124	SDEV c 125	AVIEW d 126	CLD e 127	7
M \ L	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	L \ M

Fig 2.1a

HP-41C/CV HEX TABLE

M \ L	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	L \ M
8	DEG IND 00 128	RAD IND 01 129	GRAD IND 02 130	ENTER↑ IND 03 131	STOP IND 04 132	RTN IND 05 133	BEEP IND 06 134	CLA IND 07 135	ASHF IND 08 136	PSE IND 09 137	CLRG IND 10 138	AOFF IND 11 139	AON IND 12 140	OFF IND 13 141	PROMPT IND 14 142	ADV IND 15 143	8
9	RCL IND 16 144	STO IND 17 145	ST+ IND 18 146	ST- IND 19 147	ST * IND 20 148	ST IND 21 149	ISG IND 22 150	DSE IND 23 151	VIEW IND 24 152	ΣREG IND 25 153	ASTO IND 26 154	ARCL IND 27 155	FIX IND 28 156	SCI IND 29 157	ENG IND 30 158	TONE IND 31 159	9
A	XROM 0-3 IND 32 160	XROM 4-7 IND 33 161	XROM 8-11 IND 34 162	XROM 12-15 IND 35 163	XROM 16-19 IND 36 164	XROM 20-23 IND 37 165	XROM 24-27 IND 38 166	XROM 28-31 IND 39 167	SF IND 40 168	CF IND 41 169	FS?C IND 42 170	FC?C IND 43 171	FS? IND 44 172	FC? IND 45 173	GTO XEQ IND IND 46 174	SPARE IND 47 175	A
B	SPARE IND 48 176	GTO 00 IND 49 177	GTO 01 IND 50 178	GTO 02 IND 51 179	GTO 03 IND 52 180	GTO 04 IND 53 181	GTO 05 IND 54 182	GTO 06 IND 55 183	GTO 07 IND 56 184	GTO 08 IND 57 185	GTO 09 IND 58 186	GTO 10 IND 59 187	GTO 11 IND 60 188	GTO 12 IND 61 189	GTO 13 IND 62 190	GTO 14 IND 63 191	B
C	GLOBAL IND 64 192	GLOBAL IND 65 193	GLOBAL IND 66 194	GLOBAL IND 67 195	GLOBAL IND 68 196	GLOBAL IND 69 197	GLOBAL IND 70 198	GLOBAL IND 71 199	GLOBAL IND 72 200	GLOBAL IND 73 201	GLOBAL IND 74 202	GLOBAL IND 75 203	GLOBAL IND 76 204	GLOBAL IND 77 205	X<> _ _ IND 78 206	LBL _ _ IND 79 207	C
D	GTO _ _ IND 80 208	GTO _ _ IND 81 209	GTO _ _ IND 82 210	GTO _ _ IND 83 211	GTO _ _ IND 84 212	GTO _ _ IND 85 213	GTO _ _ IND 86 214	GTO _ _ IND 87 215	GTO _ _ IND 88 216	GTO _ _ IND 89 217	GTO _ _ IND 90 218	GTO _ _ IND 91 219	GTO _ _ IND 92 220	GTO _ _ IND 93 221	GTO _ _ IND 94 222	GTO _ _ IND 95 223	D
E	XEQ _ _ IND 96 224	XEQ _ _ IND 97 225	XEQ _ _ IND 98 226	XEQ _ _ IND 99 227	XEQ _ _ IND 100 228	XEQ _ _ IND 101 229	XEQ _ _ IND A 230	XEQ _ _ IND B 231	XEQ _ _ IND C 232	XEQ _ _ IND D 233	XEQ _ _ IND E 234	XEQ _ _ IND F 235	XEQ _ _ IND G 236	XEQ _ _ IND H 237	XEQ _ _ IND I 238	XEQ _ _ IND J 239	E
F	TEXT 0 IND T 240	TEXT 1 IND Z 241	TEXT 2 IND Y 242	TEXT 3 IND X 243	TEXT 4 IND L 244	TEXT 5 IND M 245	TEXT 6 IND N 246	TEXT 7 IND O 247	TEXT 8 IND P 248	TEXT 9 IND Q 249	TEXT 10 IND ⊢ 250	TEXT 11 IND a 251	TEXT 12 IND b 252	TEXT 13 IND c 253	TEXT 14 IND d 254	TEXT 15 IND e 255	F
M \ L	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	L \ M

Fig 2.1b

PRINTED CHARACTERS

c

DISPLAYED vs. PRINTED CHARACTERS

NOTE: Bytes with a MSN greater than 7 will display as a boxed star and may print with unusual effects!

Each box determined by a hex code (combined MSN and LSN) is arranged in the following format as shown in Figure 2.2. The decimal equivalent of the hex code is in the lower left hand corner of each box. The mnemonic displayed when in PRGM mode is shown in the upper left hand corner of the box, or else an explaination of the instruction appears if the display may vary with subsequent bytes.

Figure 2.2 - Hex Table Format

The middle line in the box contains the register assignment (numeric register or status register) if the byte is interpreted as a postfix. The upper half of the hex table is interpreted as direct storage; that is, a postfix of hex 73 means the instruction will use register X as its argument or a hex 3F means that register number 63 will be used (as in STO X, or RCL 63).

The lower half of the hex table, when interpreted as a postfix, implies that the directed register contents will be used as an INDIRECT pointer. Consider the hex postfix F2; this would mean that the addressing would be IND Z (indirectly based on the contents of stack register Z).

Row 6 of the hex table contains register references above R99, which is the last <u>directly</u> addressable register when using conventional calculator instructions. As can be inferred from these references, R111 is really the last directly addressable numbered register. The code sequence '90 6F' displays as RCL J, and when executed recalls the contents of R111, if it exists.

Row 7 shows more status registers than T, Z, Y, X, and L. It shows 11 other registers that can be used as postfixes. These postfixes all work, with quite predictable results and give the programmer much more power over the 41C. None of these postfixes are normally keyable and must be achieved through 'synthetic' means. The function of these registers and the techniques to use them will be covered in Section V of this course. Keep in mind that these are part of the calculator instruction set and all rules that govern byte usage will apply.

ONE BYTE INSTRUCTIONS

The first one byte instruction is perhaps the most overlooked instruction, but the most used. It is the 'NULL', represented by hex '00'. The NULL will never list out or display when in PRGM mode, but it occupies one byte of memory and takes a certain amount of time to execute. Its execution is transparent to the user except when many nulls are executed in a row, then a delay may be noticed. Many NULL's will also introduce a delay in program editing when stepping from one line to another. You may have noticed such a delay after DEL'eting many program lines and not PACK'ing.

The 41C uses the NULL as a space holder internally during program editing. If program lines are being inserted into memory and there are already instructions in memory after the current line, the 41C will move the instructions lower in memory down by multiples of one register to make room for any added instructions. It places a register full of NULL's at the current line and replaces the NULL's with instructions as they are keyed in, until more room is necessary and then the process is repeated. It can be seen that extensive editing of a program can introduce many NULL's into memory. To regain the space taken by the NULL's, you must 'PACK' memory. PACK'ing will remove all <u>unnecessary</u> NULL's from memory.

There <u>is</u> a case where a NULL has a purpose and cannot be PACK'ed away. Consider the following program lines:

Memory Contents	Program Lines
00 11 12 13	01 123
28	02 RCL 08
40	03 +
14 15 16	04 456
43	05 /

In PRGM mode, the commands GTO .002 and DEL 002 are executed to give:

Memory Contents	Program Lines
00 11 12 13	01 123
00	
00	
14 15 16	02 456
43	03 /

A subsequent PACK will yield:

11 12 13	01 123
00	
14 15 16	02 456
43	03 /

The 41C interprets a numeric entry line as any combination of bytes from hex 10 to 1C as numeric data. This type of line is terminated when any other byte not in this range is found. A null is inserted before every numeric entry line in case it is following another in memory, otherwise the two lines would be combined into a single line. The 41C makes this test when PACK'ing or entering numeric data lines, and inserts them when needed.

It is incorrect to think that the NULL is the equivalent of the 'NOP' ("no operation") instruction on other computers. It does occupy space and takes time to execute, but it is not recognized as a 'line' by the run-time interpreter.

The code 1C is the data entry negation. It should not be confused with CHS which is code 54. It does behave as a CHS to the data entry string in that it will negate a mantissa or exponent when entered into the string. Negate is the correct terminology for what it does, because if two 1C's appear in the same portion (mantissa or exponent) of the data string, it will negate the negative to yield a positive number. This case is only seen when using synthetic programming or reading HP-67/97 program cards.

The remainder of row 0 contains the local numeric labels, LBL 00 through LBL 14. Rows 2 and 3 contain the short form STO's and RCL's, from register R00 through R15. Since most programs will have less than 15 labels and use the first 16 registers, HP made these into one byte instructions to save space in most programs.

Rows 4 through 8 all contain one byte functions that are adequately described by HP's documentation, such as HMS+ or BEEP.

There is one remaining one byte instruction. It is the TEXT 0 instruction. It stands alone in memory with a code of F0. It is a synthetic instruction and must be created by synthetic means. When executed, it has no effect on the contents of the ALPHA register. This makes it a contender as a 'NOP' instruction.

There are three hex codes that are not used as instructions on the 41C. These are 1F, AF, and B0. These are referred to as 'SPARE' instructions because no useful purpose to date has been found for them.

TWO BYTE INSTRUCTIONS

Rows 9 and A of the hex table contain most of the two byte instructions. The simplest of these reference a storage register such as: 90 25, which means RCL 37; or VIEW IND X, which is stored as 98 F3. Another simple type contains one byte of data as its postfix. FIX 3 (9C 03), TONE 9 (9F 09), and FC?C 29 (AB 1D) are examples of this type.

Row A contains the catch-all instruction of the 41C; the one that enables it to have an ever increasing instruction set......the XROM. Those users who have used peripheral functions in programs will remember the mysterious XROM instruction that appears when the peripheral is not attached. The 41C encodes the 'XROM' number of the peripheral function into the last 11 bits of this two byte instruction. The 'XROM' number may be decoded by converting the first 5 bits of the 11 and the last 6 bits into their decimal equivalent from binary.

For example, the code sequence 'A7 46'.......

Written in binary:	1010 0/111 01/00 0110		
	XROM	29	06

Yields: XROM 29,06. The printer documentation tells us that this is a BLDSPEC instruction. This applies to all peripheral functions that are stored into the 41C's program memory.

In Row B we find the branching equivalent of the short form STO's and RCL's, the two byte GTO. Notice the values range from GTO 00 to GTO 14 for the codes B1 to BF. If one byte is sufficient to define the label to branch to, what is stored in the second byte? A branch in an interpreted machine is usually a slow function, especially if it has to search far for the desired label. HP knew that a battery operated CMOS device would have a relatively slow processor, and that interpreted instruction codes would take away precious CPU time. HP uses this second byte to contain a 'compiled' branch length. After the first execution of the branch, it will store the direction and distance to the label within the GTO so that subsequent branches will be executed without searching and therefore much faster.

GTO	Label + 1	Direction	# of bytes	# of registers
1011	0011	1	000	1111

Figure 2.4 - Structure of Two Byte GTO

The first bit of the branch byte determines the direction of the branch: a '0' means forward (to a lower absolute address) and a '1' means backward (to a higher address). The remaining 7 bits contain a 3-bit byte count and a 4-bit register count. The maximum distance for a two byte GTO is then: 7 bytes + 15 registers (105 bytes) or 112 bytes. In order for the interpreter to know whether a branch is compiled or not, the zero value is reserved. Compilation occurs during a program running or during SST'ing the GTO in RUN mode. Decompilation occurs after program editing when PRGM mode is exited.

Note: <u>On some older 41C's, decompilation will NOT occur if the calculator is turned OFF while in PRGM mode</u>. This explains the strange unreproducible behavior some programmers have experienced when testing newly edited programs. The following example will test for this behavior:

In PRGM mode key in the following program:

```
01 LBL 01
02 GTO 01
```

1) Exit PRGM mode by pressing the the PRGM key.
2) Press R/S twice.
3) Enter PRGM mode.
4) GTO .001
5) Use the backarrow to delete LBL 01
6) Enter a BEEP instruction
7) Turn the 41C OFF, then ON
8) Press R/S. If you have this BUG you will hear continuous BEEP's, else NONEXISTENT will display.

Notice that the 'goose' in the display remains stationary. It will move only when it executes a LBL instruction.

branch, it will store the direction and distance to the label within the GTO so that subsequent branches will be executed without searching and therefore much faster.

GTO	Label + 1	Direction	# of bytes	# of registers
1011	0011	1	000	1111

Figure 2.4 — Structure of Two Byte GTO

The first bit of the branch byte determines the direction of the branch: a "0" means forward (to a lower absolute address) and a "1" means backward (to a higher address). The remaining 7 bits contain a 3-bit byte count and a 4-bit register count. The maximum distance for a two byte GTO is then: 7 bytes + 15 registers (105 bytes) or 112 bytes. In order for the interpreter to know whether a branch is compiled or not, the zero value is reserved. Compilation occurs during a program running or during SST'ing the GTO in RUN mode. Decompilation occurs after program editing when PRGM mode is exited.

Note: Decompilation will NOT occur if the calculator is turned OFF while in PRGM mode. This explains the strange unreproducible behavior some programmers have experienced when testing newly edited programs. The following example will demonstrate:

In PRGM mode key in the following program:

Processor changed in Aug 1980

```
01 LBL 01
02 GTO 01
```

1) Exit PRGM mode by pressing the the PRGM key.
2) Press R/S twice.
3) Enter PRGM mode.
4) GTO .001
5) Use the backarrow to delete LBL 01
6) Enter a BEEP instruction
7) Turn the 41C OFF, then ON
8) Press R/S. You should hear continuous BEEP's.

Notice that the 'goose' in the display remains stationary. It will move only when it executes a LBL instruction.

9) Press R/S to stop the noise.
10) Enter PRGM mode to examine the program, you will see:

```
01 BEEP
02 GTO 01
```

11) Press the PRGM key to exit the mode.
12) Press R/S. You may hear a BEEP, but the program will halt and show 'NONEXISTENT'.

Since there is a possibility that a program may contain packable nulls when it is run, a PACK'ing and/or relocation of the program could make compiled branches erroneous. To save those precious compilations and reduce PACK'ing time, the editor will not decompile a 'PACKED' file during a PACK. The file's END contains bits that tell whether the file is packed or not.

At the end of Row C are two more two byte instructions X<>__ (CE) and LBL __ (CF). The 'exchange x' instruction is used like the two byte RCL's and STO's. The LBL here is the two byte local label: LBL 00 through LBL 99 or LBL A through LBL e (Note that the two byte LBL's 00 through 14 must be synthetically created). It can have any byte as its postfix and should not be confused with the global LBL's.

At location AE, there is a multi-purpose instruction, it can be either the GTO IND __ or the XEQ IND __ instruction, depending upon the highest order bit in its postfix. This is the only exception to the INDirect postfix rule. If this bit is a 0, it will be interpreted as a GTO IND, else it is an XEQ IND. Notice that an indirect branch cannot be compiled.

The last type of two byte instruction is the one character text string of the form, F1 xx, where 'xx' is any byte value.

THREE BYTE INSTRUCTIONS

Rows D and E contain the three byte GTO's and XEQ's, respectively. These are long form instructions in that there are more bits to store a compiled branch. Notice that these are <u>local</u> instructions. The information is encoded similar to the two byte GTO:

GTO or XEQ	# of bytes	# of registers	direction	label
1110	111	0 0000 0011	0	111 1111

Figure 2.5 - Structure of Three Byte GTO or XEQ

The above example shows an 'XEQ e' compiled for a forward branch of: 7 bytes + 3 registers or 28 bytes. With 9 bits for the number of registers, branches are limited to 512 registers....more than the limit of program memory!

The END (file end) or .END. (permanent end) instructions are three bytes long. They begin with a byte in the range of C0 to CD. These prefixes are also used for global labels, which are four or more bytes long. Because END's and global labels share the same prefix, they will be discussed together. When a 'CAT 1' is executed, the 41C displays all of the global labels, END's, and the permanent .END. from the top of memory down. To avoid scanning all of memory for these catalog entries, the 41C stores the distance from an END or label to the nearest END or label above it, beginning with the permanent .END.. The 41C maintains a special pointer to know where the permanent .END. is at all times, so traversing the 'global label chain' is a fairly easy matter. The information for this 'chaining' is encoded in the second, third and fourth nibbles of the instruction in the same manner as the three byte GTO's. The third byte of the instruction is where an END is distinguished from a global label. If the first nibble of the third byte is an 'F', the instruction is a global label with the second nibble in this byte being the number of characters in the label <u>plus one</u>. Thus a third byte of 'F8' would denote a global label of length 7. The reason for this offset of one is to tell the number of bytes following the third byte. The fourth byte is always used to record the keycode when the global label is assigned to a key. When no assignment is made, a value of '00' is stored.

When the third byte is not of the type 'Fn', the instruction is interpreted as an END. The first nibble of that byte tells the type of END. A '0' denotes a normal file END, and a '2' denotes the permanent .END.. The second nibble in this byte contains information that tells whether the file has been packed, a '9' means packed and a 'D' means unpacked. This nybble is set up when the file is editted.

FOUR TO SIXTEEN BYTE INSTRUCTIONS

The global labels discussed above are examples of instructions that can take four or more bytes. From the instruction coding one would expect to find global labels with text lengths from 0 to 14. These extremes can exist only with synthetic programming, because the editor will only allow one to seven bytes for text. Even then, the editor will make local alpha labels out of those that have single characters from 'A' to 'J' and 'a' to 'e'.

The global GTO's and XEQ's are found at '1D' and '1E' in the hex table. The second byte of these instructions is a text byte of the form 'Fn', where 'n' is the length of the text string following. The instruction XEQ "NPR" would be coded:

1E F3 4E 50 52

and would occupy five bytes. Like the global labels, there can be synthetic GTO's and XEQ's with lengths outside the range allowed by the editor. Note that there could also be a global GTO "a" or XEQ "J". This would allow indirect branching based on a one character text string, which is not possible with a two byte, local alpha label. With synthetic techniques, these instructions become a reality.

Text entry program lines are of the form: Fn b1 .... bn, where 'n' is the number of characters (bytes) that follow the prefix. This value can range from 0, for a null text string, to 15 characters in length. These characters can be any byte value from '00' to 'FF'. The editor will only allow the normal keyboard characters to be entered, but again synthetic programming will permit the introduction of any byte into a text line. This is especially useful for mixing all upper and lower case letters for printer output, or adding special display characters like the stick man or ampersand.

The last type of multiple byte instruction is the numeric entry line. Normally the editor will limit the length of numeric entry lines to 16 bytes: a signed mantissa of ten digits and a decimal point, and a signed exponent of two digits; however the 41C will accept numeric entry lines of any length. Abnormally long entry lines can be made synthetically or may be found upon reading or merging an HP-67/97 program card (when a separator null is lost). They have little useful value for data entry because after ten digits of the mantissa are loaded, all other mantissa digits are ignored. Multiple signs, negate previous signs and after an exponent is encountered, only two digits are accepted, with its sign toggled as well.

COUNTING BYTES

In order to minimize the amount of memory a program takes, a programmer must be able to calculate how many bytes are consumed by various combinations of instructions. The following pages contain examples of programs lines and the number of bytes contained in each. They are categorized by instruction type for comparison of different variations. Some general rules for byte counting are:

1. Numeric entry lines contain as many bytes as there are digits, decimal points, signs, and E's. Do not forget an unpackable null if the line is preceded by another numeric entry line.

2. Text lines contain 'n'+ 1 bytes where 'n' is the number of characters in the line, including the append symbol.

3. Global labels contain 'n'+ 4 bytes, where 'n' is number of characters in the label. These lines are distinguished from other labels by the text 't' between the LBL and the text string.

4. Any IND'irect instruction is two bytes long.

5. END's occupy three bytes. The permanent .END. uses three bytes, but a CAT 1 printer or video interface listing will include the bytes used to make up a full register.

6. All 'functions' that require no argument (postfix) are one byte long.

7. Look for short form STO's, RCL's, and LBL's which use only one byte. These are on rows 0, 2, and 3 of the hex table.

8. All peripheral functions and ROM calls use only two bytes.

9. Local (numeric and alpha) GTO's and XEQ's take up three bytes, except for short form GTO's from GTO 00 to GTO 14, which use only two bytes.

10. Any instruction not included above, that requires a register reference, a flag reference, or a single digit of data generally takes up two bytes.

11. Do not count packable nulls. These will disappear after PACK'ing and have no bearing on routine comparisions.

SUBROUTINE USAGE

As on every computer that has a subroutine stack, the question always arises as whether to use a subroutine call or repeated in-line code. Most programmers use intuition to make this decision, and they are right in most cases. The obvious cases are of long instruction combinations with many repetitions. To determine in the not-so-obvious cases, an equation can be derived for the computation of actual bytes saved. Figure 2.3 shows graphically the byte savings or wasting from use or misuse of subroutines. For a 5 byte sequence, 4 repetitions are necessary before a subroutine will save any bytes; but for sequences above 8 bytes, there will always be saveable bytes. It is important to note that the equation is derived for short form numeric labels, the least comsumptive of labels. Additional bytes will have to be considered for longer labels. The derivation of such an equation will be encountered as a problem in the section test.

Other considerations must be made when introducing subroutines. If there are six RTN's pending on the subroutine stack, another XEQ will push the sixth one out of the stack and it will be lost. Subroutines tend to be slower than equivalent in-line coding. Will the execution time difference be acceptable for the application?

The following pages contain examples of subroutine usage versus in-line coding. Some of the examples are obvious, others are not. Most cases that are passed over for consideration by most people require modification to the procedure used in order to realize any byte savings. If there are a number of similiar, but not identical, lines repeated, think about rewriting them to use a subroutine. The byte savings are harder to calculate for these cases, but they are often worth the effort.

Equation for Saving bytes

$(N-1)(K-3)-5$ = bytes Saved

# of Rep. # of Bytes.

2byte label = 3
3byte label = 4

GENERAL RULES FOR SAVING BYTES

1. Use short form instructions whereever possible. This can lead to heavy competition for storage registers R00-R15. Consider that an additional storage register will consume 7 bytes. If short form labels are used, remember the maximum branch length that can be compiled. If the branch is too long for the instruction type, it will never be compiled and will execute much slower, especially in large programs.

2. Use text lines and global labels sparingly. Be conservative in the number of characters in your labels and prompts, but don't sacrifice usability for memory unless it is absolutely necessary.

3. Make use of subroutines when they will save bytes. If you don't have equations handy to make a comparision, count them by hand. In this manner you will develop an intuition as when to use or not use them.

4. Examine numeric entry lines for space wasting ordering. If an exponent is used, a decimal point may not be necessary. Avoid trailing zeroes where possible.

5. Look for poor use of the stack; there may be too many STO's and RCL's in the middle of your calculations. Try to recognize applications for register arithmetic (ST+, ST*, etc.).

6. Look for special instruction combinations and save them for future use; such as: 1 % (2 bytes) instead of 100 / (4 bytes). <u>Calculator Tips and Routines</u>, by John Dearing is an excellent source of such tips.

7. Try to program in a 'top-down' or 'structured' manner. Many times there are as many branches in a program as there are calculations; a poorly placed program section can waste bytes. Flow-charts are an excellent tool to study program flow.

SECTION II - QUIZ

1. A peripheral function uses:

    a. 1 byte
    b. 2 bytes
    c. 3 bytes
    d. 4 or more bytes
    e. varies

2. A two byte GTO can branch a maximum of:

    a. 105 bytes
    b. 15 registers
    c. 112 bytes
    d. b and c
    e. a and b

3. Compilation occurs for what combinations of the following instructions:

    I. 2 byte GTO's
    II. 3 bytes GTO's
    III. XROM's
    IV. alpha XEQ's
    V. indirect GTO's

    a. I, II, III
    b. I, II, IV
    c. all except V
    d. I and II
    e. all of the above

4. The global label LBL"ABC" occupies:

    a. 5 bytes
    b. 7 bytes
    c. 4 bytes
    d. 8 bytes
    e. none of the above

5. The following numeric entry line -1.2345678 E-4 contains:

    a. 13 bytes
    b. 12 bytes
    c. 11 bytes
    d. 14 bytes
    e. none of the above

SECTION II - QUIZ (continued)

6. Which of the following instructions is not 2 bytes long?

   I. GTO IND 00
   II. PRA
   III. TONE 9
   IV. LBL 14
   V. LBL"A"

   a. I only
   b. I and II
   c. I, II and IV
   d. IV and V
   e. none of the above

7. The byte sequence: AB 85 is which of the following HP-41C calculator instructions?

   a. FS?C IND 05
   b. FC?C 85
   c. FC?C IND 05
   d. FS?C IND 85
   e. none of the above

8. The byte sequence F2 7F 20 is:

   a. a peripheral function
   b. an alpha entry line
   c. a global label
   d. an unclassified instruction
   e. none of the above

9. Which instruction is not from Row 4 of the hex table?

   a. ABS
   b. *
   c. HMS+
   d. Σ
   e. MOD

10. The longest HP-41C instruction type (16 bytes) is:

   a. global label
   b. numeric entry line
   c. alpha entry line
   d. a and b
   e. b and c

SECTION II - QUIZ (continued)

6. Which of the following instructions is not 2 bytes long?

   I. GTO IND 00
   II. PRA
   III. TONE 9
   IV. LBL 14
   V. LBL"A"

   a. I only
   b. I and II
   c. I, II and IV
   d. IV and V
   e. none of the above

7. The byte sequence: AB 85 is which of the following HP-41C calculator instructions?

   a. FS?C IND 05
   b. FC?C 85
   c. FC?C IND 05
   d. FS?C IND 85
   e. none of the above

8. The byte sequence F2 7F 20 is:

   a. a peripheral function
   b. an alpha entry line
   c. a global label
   d. an unclassified instruction
   e. none of the above

9. Which instruction is not from Row 4 of the hex table?

   a. ABX
   b. *
   c. HMS+
   d. %
   e. MOD

10. The longest normal 41C instruction type (16 bytes) is:

   a. global label
   b. numeric entry line
   c. alpha entry line
   d. a and b
   e. b and c

SECTION III

INSTRUCTION TIMING / FASTER PROGRAMMING

SECTION III

Instruction Timing/Faster Programming

In the last section, we learned how to compare different instruction sequences in order to use the least amount of memory; but there are further considerations to make when writing a program. Consider the simple problem of multiplying a number by two. Here are three possible ways to do it, all of which use exactly two bytes:

2	ENTER↑	ST+ X
*	+	

Most people would have programmed it as shown on the far left, using a multiply. This is a convenient way of doing it from the keyboard and comes to mind first. The second way may be used by some programmers who reason that an addition should be much faster than a multiply. It is just as easy to use from the keyboard. The third method takes four keystrokes to implement (if the ST+ function is not assigned) and is shunned by most people, except when doing register arithmetic. Which of the above should be the choice of the advanced programmer? Examine the amount of time required to execute each combination:

61.4 mS	38.6 mS	35.5 mS

Most people would have selected the combination that takes almost twice as long to execute. The intuitive programmer would have been better off, but would not have achieved the fastest technique. It is evident that there is more to better programming than using the least amount of bytes.

This section is devoted to instruction timing of the basic HP-41C instruction set and how to calculate relative execution times of different instruction combinations. The instruction timings given here were derived by the author on his system (SN# 1952A manufactured December 1979) and will differ from 41C to 41C.

Time Cat 3 and provide this value.

SECTION III

Instruction Timing/Faster Programming

In the last section, we learned how to compare different instruction sequences in order to use the least amount of memory; but there are further considerations to make when writing a program. Consider the simple problem of multiplying a number by two. Here are three possible ways to do it, all of which use exactly two bytes:

2	ENTER↑	ST+ X
*	+	

Most people would have programmed it as shown on the far left, using a multiply. This is a convenient way of doing it from the keyboard and comes to mind first. The second way may be used by some programmers who reason that an addition should be much faster than a multiply. It is just as easy to use from the keyboard. The third method takes eight keystrokes to implement (if the ST+ function is not assigned) and is shunned by most people, except when doing register arithmetic. Which of the above should be the choice of the advanced programmer? Examine the amount of time required to execute each combination:

61.4 mS	38.6 mS	35.5 mS

Most people would have selected the combination that takes almost twice as long to execute. The intuitive programmer would have been better off, but would not have achieved the fastest technique. It is evident that there is more to better programming than using the least amount of bytes.

This section is devoted to instruction timing of the basic HP-41C instruction set and how to calculate relative execution times of different instruction combinations. The instruction timings given here were derived by the author on his system (SN# 1952A manufactured December 1979) and will differ from 41C to 41C.

There are four factors, other than programming, that will affect execution time:

Battery strength

Ambient temperature

41C ROM revision

Peripherals attached

Like most CMOS devices, the 41C will run slightly faster with a higher voltage. This is not to suggest that one should put a higher voltage on their calculator; CMOS circuits are delicate and are easily damaged. It does suggest that a fresh set of batteries will make a difference when comparing execution speed.

The 41C uses an LC oscillator circuit to generate its CPU clock, and like most clock circuits, temperature will make a difference in speed. Remember that there are temperature constraints set by the manufacturer if you experiment with various temperatures.

The 41C internal ROM's contain the programming that actually performs the operations that you program into it. HP has been revising its ROM programming since the 41C was first introduced and these changes can be expected to make some 41C's run faster or slower on some operations than others.

When the 82143A printer is attached to the 41C, a marked reduction in speed is noticed. Whenever flag 55 is set, the 41C will check printer status to determine whether it needs to send it information; such as during TRACE mode when everything must be sent to the printer. There are also occasions in the other two printer modes, MANUAL and NORMAL, that information must be sent. The information transfer, of course takes time, but even the checking itself takes time. The HP-IL peripherals are another example of this speed reduction. The video interface and the 82146A printer tend to slow down the 41C. An extreme case is when power is not applied to one of these loop devices; the time it takes to perform an operation will nearly double.

In view of these factors, the user should try to keep a fresh set of batteries around and keep flag 55 clear. This second goal may be accomplished by not having the printer attached (remember to power all devices down before connecting or disconnecting), or by synthetically clearing flag 55. This flag, if cleared synthetically, must be cleared within a program, because the 41C checks for printer existence during all modes except RUN mode. Any time that your program stops, it will set flag 55 if a printer or HP-IL output device is present.

There are some 41C owners who have replaced their timing capacitor in order to increase execution speed. <u>This is not recommended by Hewlett-Packard.</u> It can increase speed by a factor of two, but there are problems associated with doing this. The current drain on the power supply is higher and there is less time available after the BATT annunciator comes on to replace the batteries. This is especially bad for NiCad users. The operation of peripherals creates another problem. When writing data or a program to magnetic cards using the card reader, the 41C must be operating at its design speed, or the data may be written incorrectly.

Another problem arises with the digital cassette. When accessing a file that crosses a track boundary, a rewind must be made to access the second part of the file. To determine if the drive is functioning correctly, the HP-IL uses a counter to check the amount of time required for the positioning. With a faster clock rate, the counter expires sooner, giving an error.

To add to these problems, some 41C's are more prone to 'crashes' at higher speeds than others. A 'crash' is when the 41C seems to get lost; the keyboard will hang up and even the ON button will not respond. Battery removal for a short period of time will recover from most crashes, but memory contents may have been altered or lost. At any case, this course is intended to teach 'software' techniques and solutions. Hardware 'tinkering' is not for everyone and should only be undertaken by a skilled technician using utmost caution.

In view of these factors, the user should try to keep a fresh set of batteries around and keep flag 55 clear. This second goal may be accomplished by not having the printer attached (remember to power all devices down before connecting or disconnecting), or by synthetically clearing flag 55. This flag, if cleared synthetically, must be cleared within a program, because the 41C checks for printer existence during all modes except RUN mode. Any time that your program stops, it will set flag 55 if a printer or HP-IL output device is present.

There are some 41C owners who have replaced their timing capacitor in order to increase execution speed. This is not recommended by Hewlett-Packard. It can increase speed by a factor of two, but there are problems associated with doing this. The current drain on the power supply is higher and there is less time available after the BATT annunciator comes on to replace the batteries. This is especially bad for NiCad users. The operation of peripherals creates another problem. When writing data or a program to magnetic cards using the card reader, the 41C must be operating at its design speed, or the data mey be written incorrectly.

Another problem arises with the digital cassette. When accessing a file that crosses a track boundary, a rewind must be made to access the second part of the file. To determine if the drive is functioning correctly, the HP-IL uses a counter to check the amount of time required for the positioning. With a faster clock rate, the counter expires sooner, giving an error.

To add to these problems, some 41C's are more prone to 'crashes' at higher speeds than others. A 'crash' is when the 41C seems to get lost; the keyboard will hang up and even the ON button will not respond. Battery removal for a short period of time will recover from most crashes, but memory contents may have been altered or lost. At any case, this course is intended to teach 'software' techniques and solutions. Hardware 'tinkering' is not for everyone and should only be undertaken by a skilled technician using utmost caution.

DERIVATION OF INSTRUCTION TIMING

A relative timing table for the 41C had been derived before by Ernie Gibbs in 1981 and appeared in the February 1981 issue of PPC Technical Notes (V1 N6 p3), published by the Melbourne Chapter of the PPC Club.

The times presented here have been derived by the author using synthetic techniques and the 82182A Time Module. The basic technique was to set up a program in memory that contained: some lines to set up and record an initial stopwatch (SW) time, 140 bytes of memory aligned on 20 register boundaries for storage of instructions, and some lines to recall the stopwatch time after the instructions execution.

The 140 bytes were omitted the first time through to determine the overhead time for the storage and recall of the stopwatch time. The 140 bytes were then entered and repeatedly filled with different instructions. The elapsed time less the overhead time, divided by the number of instructions executed yielded the single instruction execution time. All calculations were performed by the 41C and the process was automated using the PPC ROM. With the aid of the PPC ROM programs, arbitrary byte sequences were stored into the 20 consecutive registers and the base instruction times were determined.

Many of the functions required special data in the X register in order to function correctly. Whereever possible, a range of data was examined to determine timing variance for different parameters. Some instructions, such as the GTO's, were stored in compiled form with single line forward branches. Other instructions required much patience, such as 140 AVIEW's with 0 to 24 characters in the alpha register.

These instruction times are relative, and are provided for program speed comparisions. There will be differences depending upon the installation and situation in which they are encountered. If there are any major differences, please inform the author or the PPC of these differences for further investigation. Most of the information presented in this course was derived by independent users by experimentation. In order for the body of information on the 41C to continue to grow, everyone should carefully record their observations for other users to examine and use.

The system configuration used was a 41C, Quad Memory Module, PPC ROM, Time Module, and an Extended Functions Module all running on alkaline cells.

HP-41C Instruction Timing

All times are in milliseconds (mS).

Numeric Data and Manipulations

Numeric entry lines:

base time	29.5
# of numeric digits	31.9
exponential 'E'	25.4
decimal point	19.9
each sign character	36.9

For example: -1.234 E-9 would be

29.5 + 5*(31.9) + 25.4 + 19.9 + 2*(36.9) = 308.1 mS

Note: unusual lines that contain more than 10 mantissa digits, one decimal point, one exponent, or two signs take much longer than expected.

Stack Manipulations:

CLST	10.7	RDN	17.4
CLX	10.1	R↑	12.4
ENTER↑	12.0	X<>Y	10.6
LASTX	13.3		

Register Manipulations:

	Numbered Register	Status Register
STO	22.5 (21.4)	17.0
RCL	26.0 (24.8)	21.9
ST+	42.7	35.5
ST-	44.0	38.9
ST*	48.6	43.3
ST/	49.4	44.2
X<>	25.5	19.9

Note: For STO and RCL, the values in parentheses are for short form instructions using R00-R15.

Indirect References:

IND add 15.3 mS

HP-41C Instruction Timing (continued)

Flag Operations:

AOFF	18.8	ENG n	16.7
AON	18.8	FIX n	16.7
		SCI n	16.6
DEG	20.3		
GRAD	21.1	CF nn	32.5
RAD	20.4	SF nn	28.4

	True	False
FC? nn	24.0	38.0
FS? nn	23.4	37.3
FC?C nn	35.0	49.7
FS?C nn	38.2	45.1

X Conditionals:

	True	False
X=Y?	10.6	21.4
X>Y?	24.4	38.0
X<Y?	27.8	35.4
X<=Y?	23.8	35.9
X≠Y?	10.6	21.2
X=0?	12.5	23.4
X>0?	12.6	24.1
X<0?	13.5	23.6
X<=0?	11.8	23.1
X≠0?	12.5	23.1

Statistical Functions:

ΣREG n	31.0
Σ+	229.2 *
Σ-	235.1 *
CLΣ	24.5
MEAN	139.9 *
SDEV	481.2 *

* - dependent upon contents of X, Y and the summation registers

HP-41C Instruction Timing (continued)

Trigonometric Functions:

	DEG mode	RAD mode	GRAD mode
ACOS	572	471	546
ASIN	546	489	520
ATAN	343	255	317
COS	458	379	459
SIN	567	471	568
TAN	320	242	321
P-R	672	574	672
R-P	215	162	189

Note: all of the trig functions vary based upon the data that they are processing. All of the times above are for an angle of 45 degrees and a radius of 1.

Labels and Branches:

LBL, numeric, 1 byte	10.9
LBL, alpha or numeric 2 byte	13.2
Global label, n chars.	41 + (n * 4)
GTO nn, compiled, 2 bytes	33.4
GTO nn, compiled, 3 bytes	29.5
XEQ nn, compiled	40.8
RTN (or END after XEQ)	25.0
GTO alpha	(varies)
XEQ alpha	(varies)

Note: the alpha GTO and XEQ are dependent upon the position of the instruction and the label and upon the length of the desired label.

Null byte (00): 5.9 mS

HP-41C Instruction Timing (continued)

Looping Conditionals: (evaluated FALSE)

	Numbered Register	Stack Register
DSE	75.7	67.9
ISG	73.9	66.6

Note: these instructions depend upon the values that are incremented or decremented. If the loop increment is 00 (default 1), execution is slightly faster. When evaluated TRUE, execution time depends upon the instruction skipped.

Functions:

+	26.6		HMS	27.7	
-	32.9		HMS+	69.1	
*	37.3		HMS-	70.1	
/	38.1		HR	40.8	
ABS	15.1		INT	22.1	
ADV	9.4		LOG	46-280	*
10↑X	102-229	*	LN	21-252	*
1/X	39.0		LN1+X	193	*
BEEP	1070. (F26 set)		MOD	17.6	
BEEP	15.3 (F26 clear)		OCT	124.7	*
CHS	12.9		%	36.4	
CLRG	11.8 + 2.8*(SIZE)		%CH	61.4	
CLD	21.1		PI	18.1	
DEC	53-94	*	PSE	1379.	*
D-R	82.9		RND	21.8	*
E↑X	77-242	*	SIGN	21.8	
E↑X-1	125	*	X↑2	36.4	
FACT	21 + 4.7*(X)		Y↑X	111-552	*
FRC	20.2				

Note: most functions above vary slightly, but the ones noted with '*'s vary slightly more. If a range is given, the variation is even greater.

The following instructions defy timing measurement:

OFF PROMPT STOP

The TONE n function varies greatly depending upon its postfix. It can range from a few milliseconds to over five seconds. With F26 clear it executes in 16.5 milliseconds.

Looping Conditionals: (evaluated FALSE)

	Numbered Register	Stack Register
DSE	75.7	67.9
ISG	73.9	66.6

Note: these instructions depend upon the values that are incremented or decremented. If the loop increment is 00 (default 1), execution is slightly faster. When evaluated TRUE, execution time depends upon the instruction skipped.

divides by 600

Functions:

+	26.6		HMS	27.7	
-	32.9		HMS+	69.1	
*	37.3		HMS-	70.1	
/	38.1		HR	40.8	
ABS	15.1		INT	22.1	
ADV	9.4		LOG	46-280	*
10↑X	102-229	*	LN	21-252	*
1/X	39.0		LN1+X	193	*
BEEP	1070. (F26 set)		MOD	17.6	
BEEP	15.3 (F26 clear)		OCT	124.7	*
CHS	12.9		%	36.4	
CLRG	11.8 + 2.8*(SIZE)		%CH	61.4	
CLD	21.1		PI	18.1	
DEC	53-94	*	PSE	1379.	*
D-R	82.9		RND	21.8	*
E↑X	77-242	*	SIGN	21.8	
E↑X-1	125	*	X↑2	36.4	
FACT	21 + 4.7*(X)		Y↑X	111-552	*
FRC	20.2				

.01
HMS
60
/

good way to divide by 100

Entering data

-1 → 98.5 ms
1 CHS → 73.9 ms

AVIEW
SF 25?
XEQ T?
LBL 01
GTO 01
whatever text is, it will start scrolling the message

Note: most functions above vary slightly, but the ones noted with '*'s vary slightly more. If a range is given, the variation is even greater.

The following instructions defy timing measurement:

OFF PROMPT STOP

The TONE n function varies greatly depending upon its postfix. It can range from a few milliseconds to over five seconds. With F26 clear it executes in 16.5 milliseconds.

HP-41C Instruction Timing (continued)

Alpha Data and Operations:

```
    Alpha data                        37.4 + 4*n
       here 'n' is the text length, including appends.

    CLA                               9.8

    ARCL nn

     Alpha data in register
       n characters                   37.4 + 4*n   (as above)
       if status register           - 5.9
       if indirect                  +13.3 (ASTO/ARCL only)

     Numeric data in register
       base timing                    53.9
       each numeric digit              9.8
       decimal point                   8.3
       each comma separator            8.2
       exponent E                      1.3
       if rounded                      1.2
       leading sign "-"                5.7
       exponential sign "-"           10.2
       if indirect                    13.3 (ASTO/ARCL only)
       if status register             -5.9

For example:           ARCL 05
                       R05= -1.234567890 E-59
                       FIX 4
                       SF 28, SF 29
```

53.9 + 9.8*(5) + 8.3 + 1.3 + 1.2 + 5.7 + 10.2 = 129.6 mS

ASTO nn (IND add 13.3 mS ASTO/ARCL only)

Alpha length	Numeric Register	Status Register
≤ 6	32.0	26.7
7	31.5	26.2
14 > > 7	41.4-P	36.0-P
14	30.9	25.5
21 > > 14	39.9-P	34.5-P
21	29.2	23.9
> 21	28.2-P	32.9-P

where P = 1.5 * MOD(alpha length, 7)

HP-41C Instruction Timing (continued)

AVIEW

base timing	211.6
each full body char	10.4
each comma	7.8
each period	8.4
each colon	7.6
each character scrolled	582.

For example: alpha contains AB=5.34 E-6, 2:00 AM

The string 'AB=5.34 E-6, 2' will display before scrolling, leaving ':00 AM' (6 scrolled characters).

The calculation:

211.6 + 17*(10.4) + 7.8 + 8.4 + 7.6 + 6*(582) = 3904.2 mS

A full body character is any character that does not use the punctuation dots of the display. If there are two consecutive punctuation characters, one full body character must be added to account for the included space between the two punctuation symbols. The semicolon does not use the punctuation dots of the display.

VIEW nn

For ALPHA DATA: 118 + 7.5 * (# characters)

For numeric data:

SCI n mode: 127.4 + n

ENG n mode: .use SCI mode for total mantissa digits that are displayed (i.e. for 234.23 E03 use SCI 4 (5 digits).)

FIX n mode: 128.2 + 1.3 * (total digits)

add 2.6 for each ","

For overflow, use SCI equivalent

<u>ALL modes</u>:

leading "-"	0.8
exponent "-"	1.3

SPEED CALCULATIONS

The relative speed of a set of calculator instructions may be calculated by looking up the instruction timing for each instruction and adding them together. For example:

```
01 LBL"POLY"     41 + 4*4                  =  57.0
02 STO 01        short form STO               21.4
03 X↑2                                        36.4
04 3.5           29.5 + 2*31.9 + 19.9 = 113.2
05 *                                          37.3
06 .78           29.5 + 2*31.9 + 19.9 = 113.2
07 RCL 01        short form RCL               24.8
08 *                                          37.3
09 .432          29.5 + 3*31.9 + 19.9 = 145.1
10 +                                          26.6
11 END                                        25.0
                                           --------
                                           637.3 mS
```

If a short form RCL is substituted for each numeric entry line in the program, the execution time would be:

these are for the ~~[illegible]~~ rcl of the

637.3 - 113.2 - 113.2 - 145.1 + 3*24.8 = 340.2 mS numbers

This would make the routine almost 47 % faster. The overhead time for storing each of the constants was not considered in this calculation. If the overhead time is considered, a single pass execution would take more time, but if the routine is executed repeatedly, the time savings could be calculated as follows:

Ignoring the time of the calling program...

Old routine, n repetitions: n * 637.3

-New routine, n repetitions: n * 340.2

-Overhead set-up (assuming short form STO's): Initial setup.
113.2 + 113.2 + 145.1 + 3*21.4 = 435.7

The result:

Time saved = n * 297.1 - 435.7

The breakeven point is at:

435.7/297.1 = 1.47 repetitions

It can be seen that for two repetitions of the routine POLY, there will be a savings in execution time. There is a consideration for the number of bytes used by the extra instructions and the storage registers. The choice of saving time or saving memory will depend upon the program environment.

The easiest place to increase execution speed is in numerical calculations. There are usually two or three ways to accomplish the same task computationally. The case of doubling of a number at the beginning of this section is one example. Consider the case of dividing a number by 100:

4bytes	(1)	100 /	29.5 + 3*31.9	= 125.2 38.1 -------- 163.3 mS
4bytes	(2)	1E2 /	29.5 + 2*31.9 + 25.	= 118.7 38.1 -------- 156.8 mS
4bytes	(3)	.01 *	29.5 + 2*31.9 + 19.9	= 113.2 37.3 -------- 150.5 mS
3bytes	(4)	E2 /	29.5 + 31.9 + 25.4	= 86.8 38.1 -------- 124.9 mS
2 bytes	(5)	1 %	29.5 + 31.9	= 61.4 36.4 -------- 97.8 mS
4bytes	Another Case	~~ABS~~ ABS switch Sign Last X %	86.6 ms	

These five cases are all fairly short (2 to 4 bytes) ways to divide a number in X by 100. The most straight-forward way used by most programmers, (1), is also the slowest. The more experienced programmer might have chosen either (2) or (3). The synthetic programmer would have saved one byte and 31.9 mS over (2) by using the truncated exponent shown in (4). The speed demon programmer would have sought yet a better way as shown in (5); it uses the fewest bytes and takes advantage of the '%' instruction which divides the number in X by 100 and then multiplies the result times the number in Y.

THINGS TO CONSIDER

Do not forget that there are other considerations than speed when comparing instruction combinations:

What will be left in the LASTX register after execution? Will that value save time afterward?

How many RPN stack levels will the combination consume? Will additional STO's and RCL's be necessary? Will the stack lift be left enabled or disabled after the calculation?

Will accuracy be affected? Smaller numbers should always be added together before adding to a much larger number.

Is the speed increase really necessary? Short calculations between prompts should be reduced to the fewest bytes because the speed difference is not easily noticed.

How much more memory will be used by the faster combination? Each storage register consumes 7 bytes that could have been used otherwise.

Will streamlining a program detract from the program's usability? Error trapping before a time consuming calculation would prevent the user from wasting time on a bad answer.

Does the useful lifetime of the program justify the time spent in optimizing execution speed? Many programs are used only for several hours; quick and dirty solutions are generally the most efficient in terms of man-time saved.

0/1 00 00 00 00 0/0 00
0/6 90 00 00 00 0/9 94
0/2 20 00 00 00 0/9 84

when you add these numbers together you need to be careful to get the precision.

GENERAL RULES FOR SPEED IMPROVEMENT

1. Use status registers (RPN stack, and M, N, O, and P) for computations whenever possible, they use less time than numbered storage register operations.

2. Limit the use of in-line numeric data entry lines, especially within loops. This class of instruction is particularly time consuming. If they must be used, consider the different ways of writing them to increase speed. If the number is merely a power of ten, use a synthetic exponent, if possible. If the number has an exponent, a decimal point can be omitted if the resulting exponent does not increase by a digit. For example: 12.3 E-6 vs. 123 E-7.

3. Try to use VIEW or AVIEW instead of PSE. Remember that a VIEW'ed value will remain in the display until another value is VIEW'ed or a CLD is executed. In long calculations, the VIEW may be displayed longer than a PSE. A VIEW can also display the contents of any register, whereas the PSE only displays the X or ALPHA registers.

4. Use local branches whereever possible. The use of short form LBL's and GTO's are recommended for byte savings, but remember that a short form GTO that exceeds the compilable distance will not compile, and will always take longer to execute. The three byte GTO/LBL combination executes slightly faster.

5. If global LBL's are necessary, try to order the program so that LBL's most often accessed are lowest in memory. This will shorten the global label search. Keep the number of characters to a minimum; each one reduces speed and takes more memory.

6. Avoid IND'irect branches as much as possible. They must always search for their corresponding LBL and do not compile.

7. Keep text strings short. The amount of time spent scrolling one character is more than that needed to display another 12 character line.

8. PACK your programs before execution. Every null takes 5.9 milliseconds to execute.

9. The local branches in a program can be compiled without running the program by SST'ing in PRGM mode to every GTO nn or XEQ nn, switching to NORMAL mode and pressing SST. This insures that every branch is compiled, without having to run any data through the program.

   To compile without running goto GTO _ _ then in Run mode SST GTO _ _ and it will compile.

PROGRAM EDITING

The 41C chains all of the END's and global LBL's together from the permanent .END. backwards to the first global LBL or END. It has been noted that the global LBL search proceeds from the lowest to the highest in memory and that a lower LBL would result in a faster search time. In PRGM mode, the 41C must compute the current line number from the beginning of the program file. When BST'ing a large program, large delays are noticed. These delays can be shortened by adding a 'dummy' global LBL just above the lines to be editted. The editor uses the next higher global LBL to recompute the line number when BST'ing. The LBL can be easily found with a CAT 1 and deleted after editting.

Nulls are inserted by the editor when instructions are deleted or when the adding of instructions forces those below it to be moved down to make room. If there is a large contiguous block of nulls within a file, there can be a noticeable delay when SST'ing or BST'ing. Periodic packs of a program file can compress these nulls out and lead to faster single stepping.

Another way to speed program entry is to remove any ROM's or peripherals that are not used in the program being keyed in. Whenever an XEQ "xxxxx" is used to invoke an instruction not on the keyboard, the 41C must search the global label chain for a match, then any ROM or peripheral catalogs, and finally the basic 41C instruction set. Most functions that are used in programs are in the basic set.

The last way to speed the entry of program instructions makes use of a special feature of the 41C, key assignments. If an instruction not on the normal keyboard is used repetitively, assign it to a convenient key. Remember that you must be in USER mode to make it work. If you are keying in normal keyboard functions from rows 1 and 2 of the keyboard, it would be better to have the USER mode off to prevent a bothersome search for auto-local key assignments (LBL A through LBL J and LBL a through LBL e). Synthetic programming allows unusual key assignments, such as X<> 00 or LBL A to be made. The techniques used to generate these assignments will be discussed in Section V.

SECTION III - QUIZ

1. Which of the following will <u>not</u> affect execution speed?

   a. battery strength
   b. peripherals
   c. date manufactured
   d. temperature
   e. phase of the moon

2. The fastest flag operation requires:

   a. 23.4 mS
   b. 18.8 mS
   c. 16.6 mS
   d. 16.7 mS
   e. none of the above

3. A conditional loop will execute faster if the conditional evaluates:

   a. true → correct
   b. false
   c. a or b
   d. all of the above
   e. none of the above

4. With flag 26 clear....

   a. BEEP is faster than TONE 0
   b. TONE 0 is faster than BEEP
   c. BEEP is faster than CLD
   d. a and c
   e. b and c

5. In the modified routine "POLY", if long form RCL's are used, how many iterations are required before a speed increase is realized?

   a. exactly 1.5
   b. less than 2
   c. more than 2
   d. never will
   e. none of the above

3 RCL, STO;

RCL Δt 1.2ms * 3 = 3.6
STO Δt 1.1ms * 3 = 3.3
6.9

Δt = diff. between long and short form.

n * 2x93.5 − 434.6 = time saved or net time.

0 = 434.6 = n * 293.5 => n = 1.48 at least 2 repatitions

SECTION III - QUIZ (continued)

6. How many BEEP's could be executed in the time it takes to scroll one alpha character in an AVIEW?

   a. 1
   b. none → if flag 26 set
   c. 38 → if flag 26 clear
   d. a and c
   e. b and c

7. What is the expected execution time of the byte sequence 1B 4C? E %

   a. 91.3 mS → correct 29.5 + 25.4 + 36.4 = 91.3 ms
   b. 61.8 mS
   c. 91.6 mS
   d. 37.7 mS
   e. none of the above

8. The fastest trigonometric mode is:

   a. DEG
   b. RAD
   c. GRAD
   d. all of the above
   e. none of the above

9. An alpha entry line that appends three characters to the ALPHA register takes: + the append

   a. 40.4 mS = 37.4 + 4 * 4 => 53.4
   b. 41.4 mS
   c. 42.4 mS
   d. 43.4 mS
   e. none of the above

10. The fastest instruction is:

   a. CLA – 9.8
   b. ADV → 9.4 ms
   c. CLX – 10.1
   d. the null → not a valide instruction
   e. none of the above

SECTION IV

EXTENDED FUNCTIONS / EXTENDED MEMORY

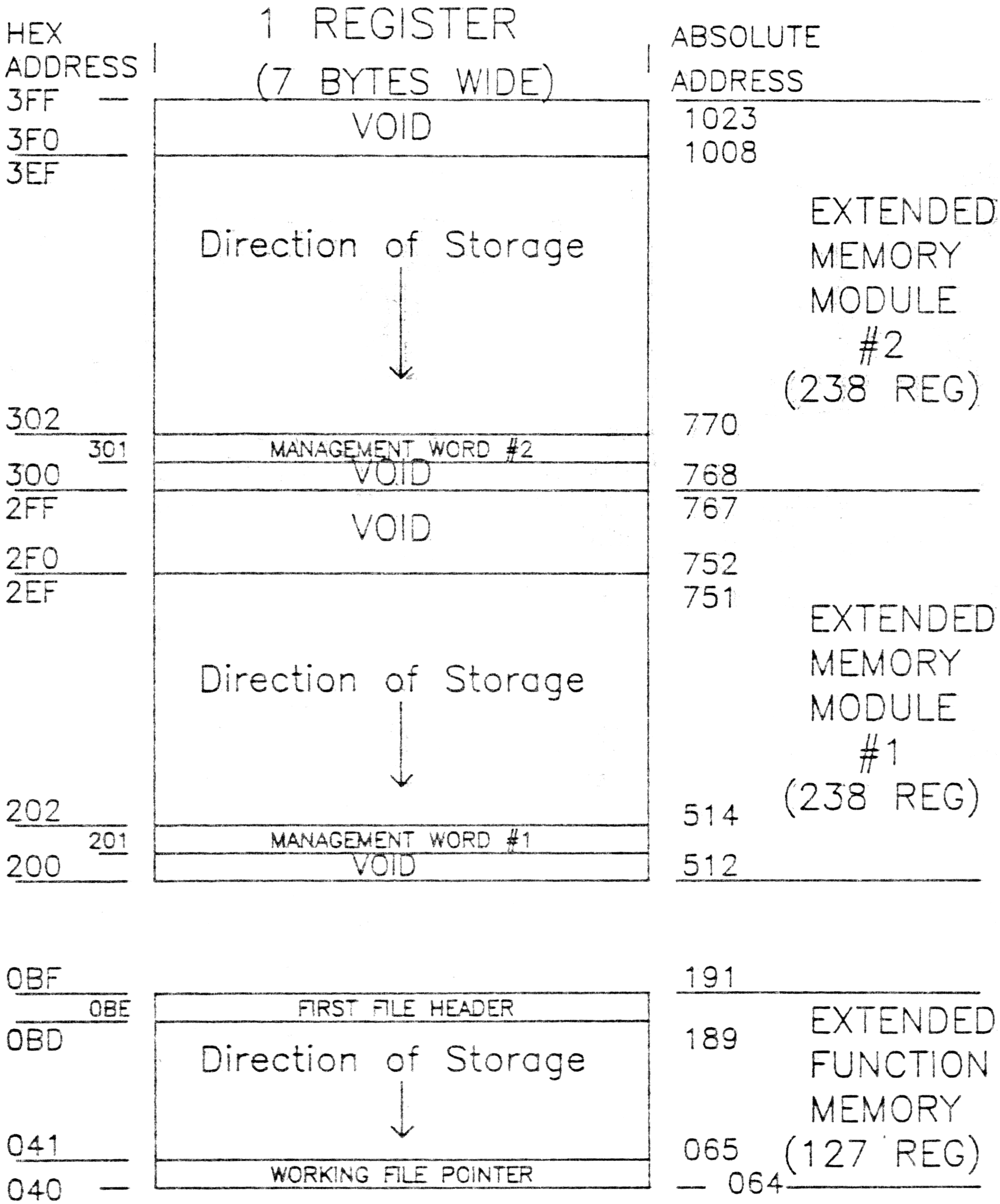


Figure 4.1 — Extended Memory Addressing

SECTION IV

Extended Functions Module/Extended Memory

The Extended Functions Module (XFM) and the Extended Memory Modules (XM's) increase the 41C's RAM capacity by 603 registers. This RAM is not available for key assignments, numbered storage, or program editting. Its purpose is to give the 41C file handling capability. The XFM also brings some excellent data ~~and~~ handling program, ~~handling~~ functions to the 41C's instruction set. These include programmable SIZE, programmable key assignments, block operations, and alpha manipulations. The arrangement of of extended memory is shown in Figure 4.1.

XFM/XM ARRANGEMENT

The 127 registers that reside within the XFM partially fill the void between the User RAM and the status registers. It begins at register address 0BF and continues down through register 040. Location 040 contains the most important pointer for XM, the working file pointer. Its seven bytes contain the current working file number, the top address of the next XM module, if present and in use, and the top address of the current block of memory. If this register is disturbed, a EMDIR (extended memory directory) will display DIR EMPTY. The exact layout is as follows:

Nibble Number

13	12	11	10	9	8	7	6	5	4	3	2	1	0
.	.	I	D	#	.	.	.	N	X	T	C	U	R

this should be drawn out better.

Working File Pointer Format (Ø4Ø)

NXT and CUR, top of next block and top of current block, respectively, are in register hex address form. The file ID# is set when a file is made the working file. When a ASCII or DATA file is invoked by name or by a R/S during an EMDIR, this value is set.

The first file header is stored in locations 0BF and 0BE and file storage progresses downward within the XFM. The storage in all of extended memory begins at the top of the module and progresses downward to the last available register in the module; then storage begins in a higher XM.

The format of XM's 1 and 2 is similar to the XFM memory; just below the last available register for storage is a memory management word. In the XM's the register contains the bottom address of the previous module (BOT), the top address of the next higher module (NXT), and the top address of the current module (CUR).

Nibble Number

13	12	11	10	9	8	7	6	5	4	3	2	1	0
.	.	.	.	.	B	O	T	N	X	T	C	U	R

much better alignment of fields.

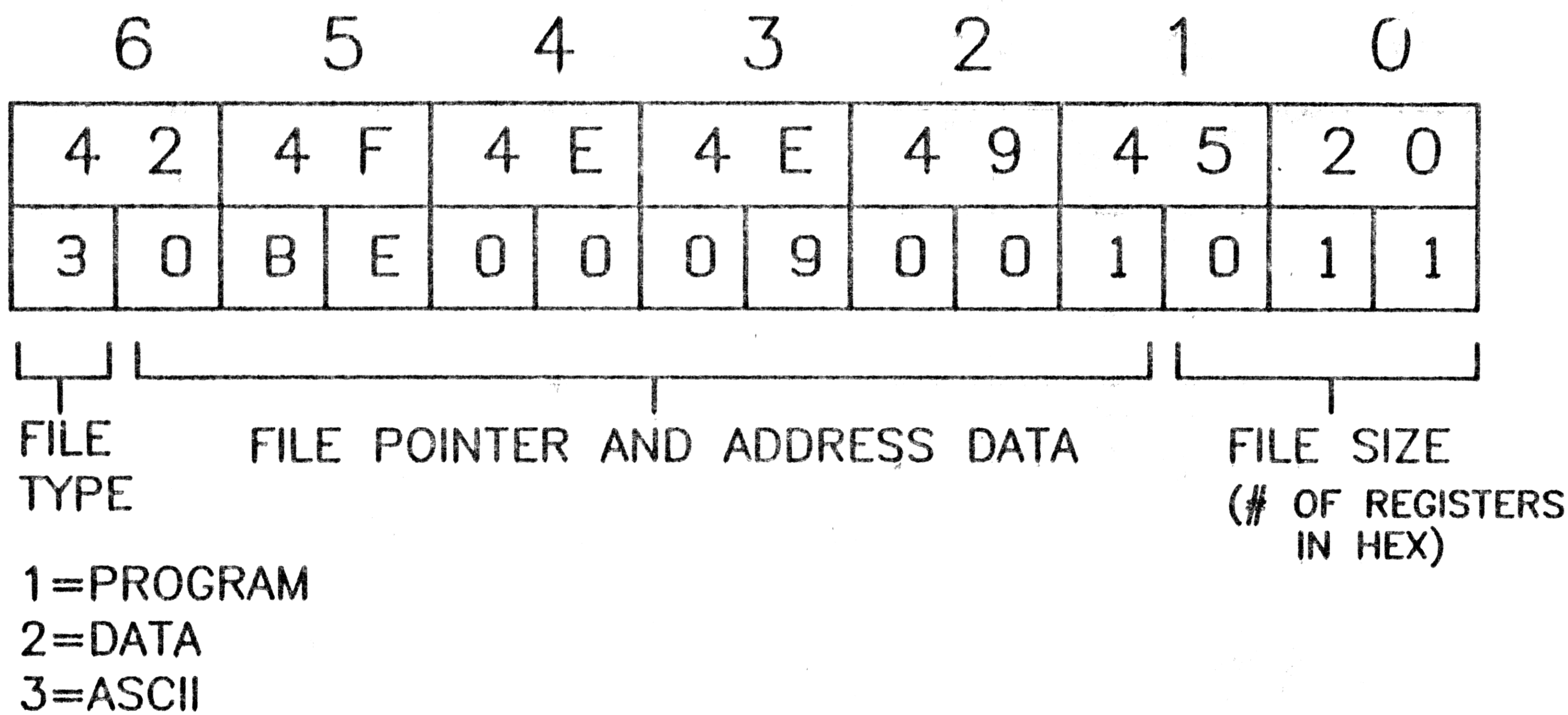
Memory Management Word Format

These registers have little value to the synthetic programmer, because when their contents are altered, subsequent XM accesses will usually correct them. The addresses stored in these registers have the same register hex address format but are offset from the actual address by one or two registers. It has been postulated, by Steve Wright, PPC Calculator Journal V9 N3 pp 22-24, that the purpose of this is to facilitate data movement when files are deleted from XM.

There are several VOID's, or addresses of registers that 'don't exist', within XM. The VOID's at locations 200 and 300 probably exist to prevent a downward read in these modules to continue into the module below. In the case of XM #1, the 'module below' would be the top of User RAM. There are two 16 register VOID's at the top of the two XM's and their use has not yet been determined.

The combination of seven FF bytes is used by the 41C to define a 'partition register'. This partition register appears where a filename would be expected and denotes the end of XM used. It is analogous to an end-of-file marker for all of extended memory. The 41C searches for this partition register whenever a file is deleted. If the last file is deleted, its file header is written over with seven FF's. The old contents of memory are not cleared. If the file to be deleted is not the last file in memory, the 41C searches for this marker and moves everything from the marker to the beginning of the first file after the deleted one forward to fill the gap. If this byte pattern appears within a file as data, a PURFL could result in lost data.

Register #1 – File Name: 7 characters, right justified padded with spaces (20_{16})



Register #2 – File Attributes and Pointers

Figure 4.2 – File Attributes and Pointers

FILE HEADERS

The 41C distinguishes between 3 types of XM files: program files, ASCII files, and DATA files (P, A, and D). The format of a generic file header is shown in Figure 4.2.

Figure 4.2 - File Header Registers

The first register in the two header registers contains the file name: from one to seven characters, left justified and right padded with spaces (hex 20), by the 41C, to make seven bytes. The characters that make up the name can be any byte value from 00 to FF.

WARNING: if you use any characters in a filename that cannot be absolutely determined from the display, you will have problems when you want to delete the file. In order to PURFL (purge a file from XM), you must specify the exact filename. A filename cannot be recalled from XM by normal means.

The second register of the file header contains the file type (P, D, or A), the file pointer and address data, and the file extent. The word 'extent' is used here because the file 'SIZE' shown in an EMDIR is really the number of registers in the extent. The actual file size is the extent plus two registers for the file header. The extent is the number of registers available for file storage under that filename expressed in hex notation. The file type, in the first nibble of the second header register, will have the value of 1 for a Program file, 2 for a Data file, and 3 for an ASCII file. This nibble can be made to take on other values, and the EMDIR will show a '@xxx' for file type and extent; but the file will not respond to any of the XFM file manipulation instructions.

The values in nibbles 12 through 3 of the second file header register differ from one file type to another and will discussed with each type of file.

The EMDIR instruction behaves like the global label CATALOG in User RAM, except that the chain of filenames is NOT kwards. The 41C examines location 0BF for a valid FILENAME, displays it with the file type and extent from the SECOND register, calculates the address of the next possible FILE der from the current extent, and repeats the process UNTIL A partition register is found. This same search TECHNIQUE is used for all file access instructions. When the EMDIR is complete, it leaves the number of XM registers AVAILABLE in the X register.

PR FILES

A program type XM file is just that...the image of a program in User RAM moved to XM. Every feature of the original file is there: compiled branches, the program END, global key assignments, and even nulls that were not packed. In order to prevent a corrupted XM program file from being downloaded into User RAM, a three byte checksum is stored into nibbles 5, 4, and 3 of the second header register. This checksum is computed from the bytes within the program. NIBBLES 12 through 6 of the second header register are not USED any purpose with a program file.

Nibble Number													
13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	.	.	.	.	.	.	.	CHECKSM			EXTENT		

↳ (MOD 256)

Second Header Register Format for Progam Files

The program file extent is the number of registers REQUIRED to hold all of the bytes of the program. It is CALCULATED as the number of bytes in the program divided by SEVEN AND rounded up to the nearest whole number.

Since the END is stored with a program file, the INFORMATION stored to indicate whether a program is COMPILED, or packed is also included with the copy. PRIVATE files may also be stored in XM.

The XFM instructions valid on a program file are:

GETP GETSUB PURFL RCLPT RCLPTA SAVEP

It has been discovered that within certain constraints, a program file may be executed within extended memory. Synthetic techniques for branching outside of User RAM are necessary in order to uses this technique. Some of the problems involved are:

Global label references can only be made to labels within User RAM. Any global labels in the stored program are not part of the global label chain, and cannot be called. XEQ's to global labels in User RAM will not have the correct return address stored on the subroutine return stack.

Numeric branches should be compiled, in order to work. If they are not compiled, a RUN of the program may compile them and make the checksum invalid. The program file cannot then be recalled back to User RAM. If the program file crosses over any XM module boundaries, the compiled branch lengths will not work correctly.

Program files in extended memory can be editted if the program pointer is placed there correctly (by synthetic means), but changes to program content will invalidate the program file's checksum.

In short, it is impractical to use extended memory for running or editting programs. The techniques necessary to do this can be informative as to how program execution and addressing occurs, but are beyond the scope of this course.

DATA FILES

The Data file consists of consecutive records that are treated like numbered storage registers. Each record will hold as much information as a single register, for each one is a register.

Records may be accessed from one at a time sequentially or randomly, to all of them at once, User RAM permitting. When records are read or written to a data file, a pointer is maintained and incremented after every read or write. This pointer is maintained in nibbles 5, 4, and 3 of the second header register as a three digit hex address of the current record or end-of-file. The address of the first record (record # 0) is kept in nibbles 12, 11, and 10 of this header register.

```
                    Nibble Number
 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 ------------------------------------------

  2  FilePtr  .  .  .  .  Reg Ptr   Extent
                          Address of
                          next pointer
```

Second Header Register Format for Data Files

The extent is the hex number of records available for storage in the file. Since the record numbers are based from 0, the highest record number is the extent less 1. This is like the correspondence of highest numbered storage register and the actual SIZE of User RAM. Nibbles 9 through 6 of the second header register are not used.

The XFM instructions valid on a data file are:

CLFL CRFLD FLSIZE GATR GETRX GETX PURFL RCLPT

RCLPTA SAVER SAVERX SAVEX SEEKPT SEEKPTA

ASCII FILES

An ASCII file allows the storage of variable length records, from 1 to 254 bytes in length. Each byte stored can take the range from 00 to FF hex, but again seven consecutive FF bytes should be avoided to prevent false partition registers. The ASCII file manipulations allow the file pointer to be set at any byte position within any record in the file.

The file pointer and register pointers are maintained in the same position within the second header register as with data files, but another pointer is necessary for the alphabetic operations...a character pointer. Information is retrieved from an ASCII file and placed within the ALPHA register. Since a maximum of 254 bytes may be contained in a single record, a character pointer must be maintained to coordination reads and writes to the file. This character pointer is encoded in hex and stored in the 7th and 6th nibbles of the second header register.

Nibble Number

13	12	11	10	9	8	7	6	5	4	3	2	1	0
3	FilePtr			.		. CharPt		RegPtr			Extent		

Second Header Register Format for ASCII Files

The information in an ASCII file is stored in a streaming fashion without regard to register boundaries. The first byte following the second header register is a hex byte count of the number of bytes in the first record. The 41C adds this byte count to the address of that byte to compute the location of the byte count for the next record. The records are numbered starting at record 000 as the data file records are.

For the 41C to access the 31st record in an ASCII file, it must start at the beginning of the file and read each byte count byte and compute the address of the next one thirty times. After the desired record is reached, the character pointer is added to the address to find the desired access point within the file. It is understandable how ASCII file accesses can be time consuming.

The advantages of the ASCII file are that records can be of variable length and may contain any byte combinations. Records may be inserted and deleted at will. More records are easily appended to the end of the file without regard to seeking a file pointer as with data files.

The use of a record byte count requires the user to included them into the size of the file when it is created. The size ~~on~~ extent of an ASCII file is specified in whole registers. The computation of an ASCII file memory requirements are:

records are compacted.

Records + total Characters +1
round up.

$$\text{ROUND UP TO WHOLE REGISTER} \left| \frac{(\#\text{ of bytes of data}) + (\#\text{ of records}) + 1}{7} \right|$$

Sizing Calculation for ASCII Files

Instructions valid on ASCII files:

APPCHR APPREC ARCLREC CLFL CRFLAS DELCHR DELREC

FLSIZE GETAS GETREC INSCHR INSREC POSFL PURFL

RCLPT RCLPTA SAVEAS SEEKPT SEEKPTA

Both DATA files and ASCII files have many instructions for storing and recalling information from the files, but the biggest drawback of their implementation is that a file's size cannot be increased directly. The information must first be transfered to an external storage device or to User RAM, then the file must be deleted and recreated. Or the information may be written to another file of the same type with a larger size. The problem with direct file transfer is that there must be more than twice as much storage left as the original file occupied.

The instructions for manipulation of each of the file types will now be discussed. <u>It should be kept in mind that several instructions are common between two or more file types and that their behavior can be markedly different</u>.

ASCII FILE INSTRUCTIONS

CRFLAS X=# of registers ALPHA=1 to 7 character filename

This instruction creates an ASCII file of the size specified under the name specified and makes it the working file. Possible error messages are:

DATA ERROR	X contains a zero register size
DUP FL	Filename already exists
NAME ERR	ALPHA register is empty
NO ROOM	Not enough space is left in XM

CLFL ALPHA=1 to 7 character filename

This instruction will set the number of records in the file specified to zero and make the named file the working file. Possible error messages are:

FL NOT FOUND	Named file does not exist.
FL TYPE ERR	Named file is a program file
NAME ERR	ALPHA register is empty

PURFL ALPHA=1 to 7 character filename

This instruction will purge the named file from extended memory and move all files after it forward to fill the space left by the file. **WARNING:** After this instruction, there will be no working file selected. Any attempt to reference a file without a current working file will cause the loss of all files. Possible error messages are:

L NOT FOUND	The named file does not exist.
NAME ERR	The ALPHA register is empty.

SEEKPTA X=file pointer ALPHA=1 to 7 character filename

Seek pointer by alpha. The file pointer is of the form rrr.ccc where rrr is the record number and ccc is the character position within the record. Both consider zero to be the first record and first character. Possible error messages are:

DATA ERROR	X is greater than 999
END OF FILE	The desired pointer is beyond the end of the file. The file is selected as the working file, but its pointers are not changed.
END OF REC	The desired pointer is beyond the end of the desired record. The file is made the working file and the pointer is set after the last character in the desired record.
FL TYPE ERR	The filename specifies a program file.

11
12 RCLPT
13 SEEPT
14
15

Xeq [illegible]
Alpha "_____" Alpha
10
XEQ CKFLAS
Explain THE SPEED of EXECUTION
RCLPT => $\emptyset$
XEQ EMDIR
NAME then hold any key down to show the entry
Explain the significance of the display and then
the number left in the display
[illegible]

ASCII FILE INSTRUCTIONS (continued)

SEEKPT X=file pointer rrr.ccc

This instruction is similar to SEEKPTA except that there is no filename specified. The file is assumed to be the working file. The possible error messages are:

DATA ERROR	X is greater than 999
END OF FILE	The desired pointer is beyond the end of the file. The file is selected as the working file, but its pointers are not changed.
END OF REC	The desired pointer is beyond the end of the desired record. The file is made the working file and the pointer is set after the last character in the desired record.
FL TYPE ERR	The filename specifies a program file or there is no working file.

Do Example on other Page

From this point on demonstrate the functions.

RCLPTA ALPHA=<0 to 7 character filename>

This instruction returns the current file pointer of the file specified into the X register. The named file is made the working file. The pointer is in the format: rrr.ccc as defined for SEEKPTA above. The only error message is:

FL NOT FOUND	Named file does not exist.

RCLPT (no parameters)

This instruction functions like RCLPTA except that the working file is assumed. The error message:

FL NOT FOUND	This indicates that there is no working file.

FLSIZE ALPHA=1 to 7 character filename or is empty

This instruction returns the size in registers of the filename specified, or of the working file if the ALPHA register is empty, to the X register. Possible error messages are:

FL NOT FOUND	Named file does not exist or no working file exists.

ASCII FILE INSTRUCTIONS (continued)

APPREC ALPHA=1 to 24 characters of text 1. have then store a record.
2. then do a RCLPT 0. — —

This instruction will append the contents of the ALPHA register to the end of the current working file making it a new record. The file pointer is adjusted to after the last character in this new record. If the ALPHA register is empty, no action takes place. Possible error messages are:

remember the its zero based.
0 → 255
3. CLX
SEEKPT

END OF FL	Attempt to write past end of file.
FL TYPE ERR	Working file is not an ASCII file.

DELREC Can insert and delete these records.

This instruction deletes the record at which the working file pointer is positioned at. All records after the deleted record are moved forward by one record number. The file pointer is set to the beginning of the record that was deleted. Possible error messages are:

END OF FL	Attempt to delete past end of file
FL NOT FOUND	There is no working file.
FL TYPE ERR	Working file is not an ASCII file.

INSREC ALPHA=1 to 24 characters of text

This instruction inserts the contents of the ALPHA register ahead of the current record pointer as a new record moving subsequent records back to make room for it. Possible error messages are:

END OF FL	Attempt to expand file past end of file.
FL NOT FOUND	There is no working file.
FL TYPE ERR	Working file is not an ASCII file.

"NIC 4"

POSFL → position pointer to first occurance of string
If not found then -1

CLX

ASCII FILE INSTRUCTIONS (continued)

APPCHR ALPHA=1 to 24 characters of text

This instruction will append the contents of the alpha register to the end of the record specified by the file pointer. The file pointer will now be positioned after the end of the current record. An empty ALPHA register has no effect. Possible error messages are:

END OF FL	Attempt to expand or write beyond the end of the file.
FL NOT FOUND	There is no working file.
FL TYPE ERR	Working file is not an ASCII file.
REC TOO LONG	The resulting record would exceed 254 bytes in length.

DELCHR X=# of characters to delete

This instruction will delete X characters starting at the current file pointer position in the working file. The file pointer remains the same. Possible error messages are:

END OF FL	Attempt to delete past end of file
FL NOT FOUND	There is no working file.
FL TYPE ERROR	Working file is not an ASCII file.

INSCHR ALPHA=1 to 24 characters of text

This instruction will insert the contents of the ALPHA register into the working file starting at the current file pointer position. The file pointer will be positioned after the last character added. Possible error messages are:

END OF FL	Attempt to expand file past end of file.
FL NOT FOUND	There is no working file.
FL TYPE ERR	Working file is not an ASCII file.
REC TOO LONG	Attempt was made to expand a record beyond 254 characters.

POSFL ALPHA=1 to 24 characters of text see pg 11

This instruction causes the 41C to scan the working file from the current file pointer position to find a match for the text in the ALPHA register. If a match is found, the file pointer is set to the first character of the matching string and the pointer is returned to the X register. If no match is found, the file pointer remains the same and a value of -1 is returned to the X register. An empty ALPHA register will always return a -1. Possible error messages are:

FL NOT FOUND	There is no working file.
FL TYPE ERR	Working file is not an ASCII file.

GETREC (no parameters)

This instruction will replace the contents of the ALPHA register with 1 to 24 characters from the working file starting at the current file pointer position up to the end of the current record. The file pointer will be set to the next character to be sent. Flag 17 will be set if the end of record was not reached. It will be clear if the end of record was reached. This coordination with flag 17 allows text files to be easily used with the HP-IL Video Interface. If flag 17 is set, the video interface will not place an automatic carriage return and line feed after the OUTA (output alpha register) instruction. Possible error messages are:

END OF FL	Attempt to read past end of file.
FL NOT FOUND	There is no working file.
FL TYPE ERR	Working file is not an ASCII file.

ASCII FILE INSTRUCTIONS (continued)

ARCLREC (no parameters)

This instruction will append characters from the current file pointer position of the working file until either the ALPHA register is full or an end of record has been reached. The file pointer will be left pointing at the next character to be sent. Flag 17 is manipulated as with GETREC; it is set if the end of record was not reached, and cleared otherwise. Possible error messages are:

END OF FL	Attempt to read past end of file.
FL NOT FOUND	There is no working file.
FL TYPE ERR	Working file is not an ASCII file.

SAVEAS ALPHA=ASCII file name <,mass storage filename>

This instruction will transfer an ASCII file to a mass storage device like the Digital Cassette, if one exists. The mass storage filename is optional; if omitted, the name of the ASCII file will be used as the mass storage filename. <u>The name must have been previously-initialized using the CREATE command in the HP-IL module</u>. Possible error messages are:

END OF FILE	The destination file was smaller than the source file. Partial transfer of the contents was made.
FL NOT FOUND	The ASCII filename does not exist.
FL TYPE ERR	The named file is not an ASCII file.
NAME ERR	ALPHA register is empty.
NO DRIVE	The HP-IL is not present or there is no mass storage device on the interface loop.

GETAS ALPHA=mass storage filename <,ASCII file name>

This instruction will retrieve an ASCII file from a mass storage device and place the contents into an ASCII file in extended memory. <u>The ASCII file in XM must have been created previously</u> If the end of either file is reached, the transfer will stop. Possible error messages are:

END OF FL	The end of the XM file was reached before the transfer was complete.
FL NOT FOUND	The named file does not exist in extended memory.
FL TYPE ERR	The named file was not an ASCII file.
NAME ERR	The ALPHA register is empty.
NO DRIVE	There is no HP-IL module or mass storage device on the interface loop.

DATA FILE INSTRUCTIONS

CRFLD X=# of registers ALPHA=1 to 7 character filename

This instruction creates a data file with the specified name and of the specified extent. The created file becomes the working file. Possible error messages are:

DATA ERROR	X-register contains a 0.
DUP FL	A file already exists with the name specified.
NAME ERR	The ALPHA register is empty.
NO ROOM	There is not enough extended memory to create the size of file specified.

CLFL ALPHA=1 to 7 character filename

This instruction will write the value of zero into every record within the data file. Possible error messages are:

FL NOT FOUND	The file specified does not exist.
FL TYPE ERR	The file named is a program file.
NAME ERR	The ALPHA register is empty.

FLSIZE ALPHA=<0 to 7 character filename>

This instruction will return the number of records in the named file or working file to the X register and makes the file referenced the working file. Possible error messages are:

FL NOT FOUND	The named file does not exist or there is no working file.

PURFL ALPHA=1 to 7 character filename

This instruction will purge the named file from extended memory and move all files after it forward to fill the space left by the file. **WARNING:** After this instruction, there will be no working file selected. Any attempt to reference a file without a current working file will cause the loss of all files. Possible error messages are:

FL NOT FOUND	The named file does not exist.
NAME ERR	The ALPHA register is empty.

DATA FILE INSTRUCTIONS (continued)

SEEKPTA X=rrr ALPHA=1 to 7 character filename

This instruction will select the named file as the working file and set the file pointer to the record value specified in the X register. Only the integer portion of the X register is used. Possible error messages are:

DATA ERROR	The number in X is greater than 999.
END OF FL	Attempt to position file pointer beyond the end of file. The file will be selected as the working file, but the file pointer will not be change .
FL NOT FOUND	The named file does not exist.
FL TYPE ERR	The named file is a program file.

SEEKPT X=rrr

This instruction will set the file pointer of the working file to that specified in the X register. Possible error messages are:

DATA ERROR	The number in X is greater than 999.
END OF FL	Attempt to position file pointer beyond the end of file. The file pointer will not be changed.
FL NOT FOUND	There is no working file.

RCLPTA ALPHA=1 to 7 character filename

This instruction will return the file pointer of the named file to the X register and selects it as the working file. The file pointer is of the form rrr as in SEEKPTA and SEEKPT. The only error message:

FL NOT FOUND	The named file does not exist.

DATA FILE INSTRUCTIONS (continued)

RCLPT (no parameters)

This instruction functions the same as RCLPTA except the working file is assumed. The value of the working file pointer is returned to the X register. The only error message:

FL NOT FOUND	There is no working file.

SAVER ALPHA=<0 to 7 character filename>

This instruction will copy all of the current data storage registers to the named data file or to the working file if the ALPHA register is empty. The contents of the first data register R00 will be saved as record 000, R01 will be saved into record 001, etc. The named data file will be made the working file. After the transfer, the file pointer will be pointing to the next available record or at the END OF FL. If there are more data registers than records an END OF FL error will result. This instruction is analogous to the Card Reader WDTA instruction. Possible error messages are:

END OF FL	There were more data registers than records in the file.
FL NOT FOUND	The named file does not exists or there is no working file.
FL TYPE ERR	The named or working file is not a DATA file.

DATA FILE INSTRUCTIONS (continued)

SAVERX X=block control word: bbb.eee

This instruction copies the data registers denoted by the block control word in the X register (bbb=beginning register, eee=end register) into the current working file beginning at the current file pointer. The transfer will not take place if there is not enough room from the file pointer to the end of the file to accomodate the block of data registers. The file pointer will be left pointing to the next available register or to the end of file. This function is analogous to the card reader's WDTAX. The possible error messages are:

END OF FL	Attempt to write past the end of file. The file pointer remains unchanged.
FL TYPE ERR	The working file is not a DATA file.
NONEXISTENT	Attempt to save a register that does not exist in the current SIZE.

SAVEX (no parameters)

This instruction will save the contents of the X register to the current working file at the current file pointer. The file pointer will be advanced to the next available record or to the end of file. Possible error messages are:

END OF FL	Attempt to save a record beyond the number of records available.
FL NOT FOUND	There is no working file.
FL TYPE ERR	The working file is not a DATA file.

CREATE a DATA FILE

21 GETX

22 SAVE X

23

24

DATA FILE INSTRUCTIONS (continued)

GETR   ALPHA=<0 to 7 character filename>

This instruction will copy the contents of the named file or working file, if filename not present in ALPHA, to the data storage registers. The first record (000) in the file is transferred to R00, the second record to R01, and so on until no registers are available in User RAM or an end of file is reached. The file pointer will be left pointing to the next available record or the end of file, depending upon the condition that terminated the transfer, the exhaustion of records or of data registers. The named file will become the working file. This instruction is analogous to reading a data card with the card reader. Possible error messages are:

FL NOT FOUND	The named file does not exist or there is no working file.
FL TYPE ERR	Either the named file or the working file is not a DATA file.

GETRX   X= block control word, bbb.eee

This instruction will copy data from the working file, beginning at the current file pointer, to numbered data registers beginning at register bbb and continuing to re ister eee. The transfer will cease when the registers specified have been filled or the end of file is reached. The file pointer will be positioned to the next record or the end of file. This instruction is the complement of SAVERX or of the card reader's RDTAX. Possible error messages are:

END OF FL	Attempt to read beyond the end of the file.
FL NOT FOUND	There is no working file.
FL TYPE ERROR	The working file is not a DATA file.
NONEXISTENT	At least one register in the range specified does not exist in the current SIZE.

GETX (no parameters)

This instruction will copy the record pointed at by the file pointer of the working file to the X register. The file pointer will be incremented by one. Possible error messages are:

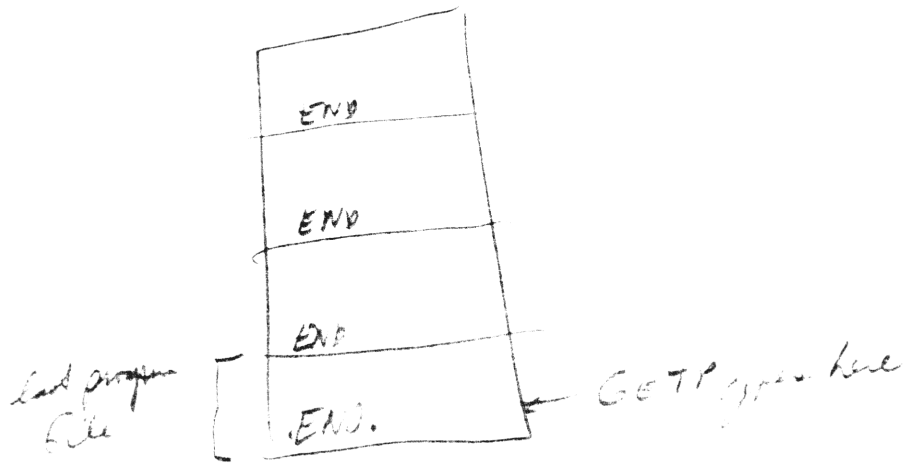
END OF FL	Attempt to read beyond the end of the file.
FL NOT FOUND	There is no working file.
FL TYPE ERR	The working file is not a DATA file.

PROGRAM FILE INSTRUCTIONS

GETP ALPHA=1 to 7 character filename

The named program file is copied from extended memory into program memory. The copy will be placed between the permanent .END. and the last program END, replacing any program in between. If this instruction is executed from a running program, control will be returned to the calling program, unless that program has been replaced by the execution of this instruction, then program execution will resume at the first line of the copied program. If executed from the keyboard, the program pointer is positioned to the first line of the transferred program. If USER mode is on before the transfer, any key assignments recorded with the file will be activated. This instruction is similar to reading a program card with the card reader. Possible error messages are:

CHKSUM ERR	The program file checksum is not correct for the program file. The contents of the file may have been corrupted by: XFM/XM modules removal, static memory loss, or editting or execution with the program pointer in extended memory.
FL NOT FOUND	The named file does not exist.
FL TYPE ERR	The named file is not a program file.
NAME ERR	The ALPHA register is empty.
NO ROOM	There is not enough room in User RAM for the program. Try reSIZEing for fewer data registers, deleting key assignments, or clearing programs. This message is only seen during program execution.
PACKING	The same as 'NO ROOM' except that this message is seen if GETP is executed from the keyboard.



END
END
END
.END.
last program file
GETP goes here

GETSUB ALPHA=1 to 7 character filename

This instruction copies the named file into User RAM after the last program and just before the permanent .END.. Control is returned to the calling program if executed within a program. The program pointer is not set to the fi st line of the transferred program. If USER mode is on, any key assignments recorded with the program will be activated. This instruction is analogous to the card reader's RSUB. Possible error messages are:

CHKSUM ERR	The program file checksum is not correct for the program file. The contents of the file may have been corrupted by: XFM/XM modules removal, static memory loss, or editting or execution with the program pointer in extended memory.
FL NOT FOUND	The named file does not exist.
FL TYPE ERR	The named file is not a program file.
NAME ERR	The ALPHA register is empty.
NO ROOM	There is not enough room in User RAM for the program. Try reSIZEing for fewer data registers, deleting key assignments, or clearing programs. This message is only seen during program execution.
PACKING TRY AGAIN	The same as 'NO ROOM' except that this message is seen if GETP is executed from the keyboard.

PURFL ALPHA=1 to 7 character filename

This instruction will purge the named file from extended memory and move all files after it forward to fill the space left by the file. **WARNING:** After this instruction, there will be no working file selected. Any attempt to reference a file without a current working file will cause the loss of all files. Possible error messages are:

FL NOT FOUND	The named file does not exist.
NAME ERR	The ALPHA register is empty.

RCLPT (no parameters)

This instruction will return the number of bytes in the working file, if the working file is a program file. A program file can be a working file if: an EMDIR is stopped at the file, a RCLPTA or SAVEP with the filename has been executed and no other file has been purged or made the working file since. This instruction can be confusing because it is allowed on all file types. The only error message is:

FL NOT FOUND	There is no working file.

RCLPTA ALPHA=<0 to 7 character filename>

This instruction will return the number of bytes in the named program file. If the ALPHA register is empty, the instruction functions like RCLPT. The only error message is:

FL NOT FOUND	The named file does not exist, or if ALPHA empty, there is no working file.

SAVEP ALPHA= <program name><,filename>

This instruction will copy the named program, or current program if program name is omitted, to a program file specified by the filename. If the filename is omitted, the file will be given the same name as the program. In either case, the ALPHA register must have a name in it. If a program file already exists under that filename, it will be purged from extended memory and recreated. Possible error messages are:

DUP FL	A file of the same name exists in extended memory, but is not a program file. The existent file is made the working file.
NAME ERR	The ALPHA register is empty.
NO ROOM	There is not enough room in extended memory to save the specified program.
ROM	The named program resides in ROM.

EXTENDED MEMORY DIRECTORY

EMDIR (no parameters)

This instruction will display a 'directory' of extended memory and place the number of extended memory registers that are not used into the X-register. If no files exist, the message "DIR EMPTY' will be displayed.

Each directory entry is shown as a single line, containing the filename, a filetype letter (Program, Data, or ASCII), and the extent of the file. A sample directory is shown below:

```
NPR      P138
XXXXXXX  D010
L        A001
MPG      P098
CAR1     D020
CAR2     D020
CAR1X    D020
```

Each entry is 'paused' in the display like a CAT'alog. If any key other than R/S or ON is pressed during the directory, the display will 'hold' at that entry for review for as long as the key is held down. On releasing the key, the directory will continue.

If the ON key was the key pressed, the directory will 'hold' until the key is released, then the 41C will turn off. If the R/S key is pressed, the directory will hold until the key is released, control is then returned to the keyboard. The file that was displayed when either of these two key was pressed is made the working file. After the directory is stopped, the X register will contain the number of XM registers that are unused. If the directory is not stopped, the working file remains the same as before the EMDIR.

The EMDIR instruction is programmable, but control does not return to the calling program if the directory is stopped. The stoppage of the directory is a good way for the user to select a file without having to name the file, but how to guide the user through to restart the program? The following program lines show a technique for using this file feature:

```
"R/S AT FILE"
AON
TONE 7
PSE
"PRESS R/S"
EMDIR
CLD
AOFF
```

Convient Way to Select Working file.

The first message "R/S AT FILE" is displayed for a short time with an attention getting TONE 7. The ALPHA register is then loaded with the message "PRESS R/S". Since ALPHA mode was selected, if the EMDIR is stopped with a R/S, the display will show "PRESS R/S". The user then presses R/S to continue. If the EMDIR was not stopped, the last file displayed would be 'frozen' in the display until another view type instruction or CLD is executed, hence the CLD. The AOFF, of course, is to exit ALPHA mode. Remember...if the EMDIR is stopped, the file will become the working file. If it continues without stoppage, the previous selected working file is still active. If the previous operation was a PURFL, any subsequent file operation will cause all files to be lost.

EXTENDED ALPHA REGISTER INSTRUCTIONS

The XFM has several instructions that enhance the ALPHA capabilities of the 41C. These instructions are similar to string handling functions in the BASIC language that is used on other computers. The correspondence of these functions are as follows:

XFM ALPHA FUNCTION	APPROXIMATE BASIC EQUIVALENT
ATOX	ASC(LEFT$("string",1))
XTOA	"string"+CHR$(X) OR "string"+X$
ALENG	LEN("string")
ANUM	VAL("string")
POSA	INSTR(1,"string",CHR$(X)) OR INSTR(1,"string",X$)
AROT	A$=LEFT$("string",X) N=LEN("string")-X B$=RIGHT$("string",N) B$=B$+A$

These instructions are quite useful when working with ASCII files, ALPHA input data, peripheral codes, and special displays. A description of each of the functions follows:

XFM ALPHA INSTRUCTIONS

ALENG ALPHA=0 to 24 character string

This instruction will return a decimal value in the ran e of 0 to 24 to the X register. This represents the number of characters present in the ALPHA register. The character count starts from the first non-null character in the alpha register, and counts all characters thereafter, including nulls. There are no error messages for this instruction.

XFM ALPHA INSTRUCTIONS (continued)

ANUM ALPHA=0 to 24 character text string.

This is a very powerful instruction that is easily misused. It will scan the ALPHA register for any consecutive set of characters that might be interpretted as ASCII encoded numbers and returns a normalized value to the X register that represents the decoding of that number. In addition, if a valid string is found, flag 22, the data entry flag will be set. This instruction will not clear flag 22. The decoding is controlled by the status of flags 28 and 29.

If flag 28 is set, a "." will be interpretted as the decimal point. If flag 28 is clear, a "," will be interpretted as the decimal point. Flag 29 controls whether or not comma digit group delimiters, if flag 28 set (or periods, if flag 28 is clear) will be accepted as numeric data entry.

Also do S--5 "ANUM" you get +5 not -5

For example, ALPHA="S55,362.8/12"

If F28=set and F29=set, ANUM returns: 55,362.8

If F28=set and F29=clear, ANUM returns: 55

If F28=clear and F29=set, ANUM returns: 55,3628

If F28=clear and F29=clear, ANUM returns: 55,362

depending on what is ~~flag~~ set depends on what Number you get,

The processing commences from the first data entry character (0 1 2 3 4 5 6 7 8 9 E + - , .) based upon the statuses of flags 28 and 29 and continues until a non-data entry character is encountered. The functioning is like that of a numeric entry line. If there are multiple plus's or minus's, their net effect is considered. The 'E' behaves differently, in that a numeric digit must proceed the 'E' Any embedded spaces will stop numeric conversion. At least one numeric digit must be found before any processing will occur. If there is no convertible string in the ALPHA register, the X register is left unchanged as well as flag 22. There is no error message for this instruction.

the , and . are not necessarily used to set where the proper break down is.

AROT ALPHA=0 to 24 character string , X=$\pm$ 0 to 255

This instruction will rotate the contents of the ALPHA register by the number of characters and direction specified by the X register. A positive number in X, rotates to the left, and a negative number rotates to the right. This enables the permutation of the ALPHA register in a keychain type manner. That is, each character can be thought of as a key on a circular keychain; every time a rotation is performed, some keys are moved about the keychain to the other side. Their relative order is maintained.

If there are null bytes (00) within the ALPHA register, they will be lost if a rotation stops or any manipulation results in those nulls becoming the leftmost characters. The ALPHA register uses the null byte to fill the space ahead of any characters entered into the register. The first non-null character from the left is the start of all data. Any nulls following that character are included in the workable contents of the register. The only error message is:

DATA ERROR	The absolute value of X exceeds 255.

ATOX ALPHA=0 to 24 characters

This instruction will return the decimal equivalent of the byte code for the first character in the ALPHA register. The first byte of the ALPHA register will be removed, making the length of the string one less (see comments on nulls in the explaination of the AROT instruction). If the ALPHA register is empty, a value of 0 will be returned. There are no error essages for this instruction.

XFM ALPHA INSTRUCTIONS (continued)

POSA X= $\pm$ 0 to 255 or ALPHA data

if you note follow w/ AROT this character will now be at the front of Alpha

This instruction will return to the X register the first position of the absolute value of the byte code, or ALPHA string, within the ALPHA register. The first character position is 0, like the ASCII file character pointer. If the string or byte is not found, a -1 is returned to the X register. The only error message is:

used to [illegible] a file deli[illegible]

DATA ERROR	The absolute value of X exceeds 255.

XTOA X=decimal value (0-255) or ALPHA data

This instruction will encode the decimal value in X into a hexadecimal byte and append this byte to the contents of the ALPHA register. If the X register contains ALPHA data, the string will be appended to the contents of the ALPHA register. The only error message is:

DATA ERROR	The absolute value of X exceeds 255.

These functions can lend themselves to quite advanced techniques. Consider the following program lines:

```
ANUM
STO IND 00
LBL 01
ATOX
47
X≠Y?
GTO 01
```

This set of instructions will decode a number from the contents of the ALPHA register and then take bytes from the front of the ALPHA register until a delimiting "/" is found. This technique enables the use of delimited numbers in ASCII files. A DATA file uses 7 bytes to save a single number, but if the number is less than 7 bytes when ASCII encoded, an ASCII file record may save room.

Either of the instructions, AROT or XTOA may be used in conjuction with flag 25, the error ignore flag, to check a number in the range of -255 to +255. These limits are quite common in computer applications.

FLAG REGISTER INSTRUCTIONS

The XFM contains three instructions that enable extended control of flags F00 through F43. These functions are explained below:

RCLFLAG (no parameters)

This instruction will return an ALPHA data type value to the X register containing the statuses of flags F00 through F43. The resulting register is encoded as follows:

Nibble Number													
13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	F	F	.	.	.	.	.	.	.	.	.	.	.

The nibbles 10 through 0 contain the binary status of flags F00 through F43, respectively in 4 bit groups. The two F values stored in nibbles 12 and 11 distinguish this data type from normal ALPHA data in an apparent attempt to prevent indiscriminate storing of values into the flags. Bug 7, the ASTO bug, allows a method of creating this data type in the alpha register. Recall that with 7 characters in the alpha register, the second nibble of the seventh character will be stored with the ALPHA data value of an ASTO operation. If BUG 7, the ASTO bug, is present in your 41C a STO/RCLFLAG data type may be created by entering 6 characters into the ALPHA register with the first nibble of the first character being an 'F'. This may be easily done with the XTOA instruction. A byte is then appended to the end of these 6 characters with any one of several instructions: an append text line, XTOA, or ARCL. This byte will be found in the 16th column of the hex table. A good append text line would be one containing a '?'. An ASTO X is then executed to yield the RCLFLAG data type in register X. In this manner a RCLFLAG value may be ARCL'ed for storage into an ASCII file and reconstructed for a later STOFLAG operation; but only if BUG 7 is present.

FLAG REGISTER INSTRUCTIONS

The XFM contains three instructions that enable extended control of flags F00 through F43. These functions are explained below:

RCLFLAG (no parameters)

This instruction will return an ALPHA data type value to the X register containing the statuses of flags F00 through F43. The resulting register is encoded as follows:

Nibble Number													
13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	F	F	.	.	.	.	.	.	.	.	.	.	.

The nibbles 10 through 0 contain the binary status of flags F00 through F43, respectively in 4 bit groups. The two F values stored in nibbles 12 and 11 distinguish this data type from normal ALPHA data in an apparent attempt to prevent indiscriminate storing of values into the flags. Bug 7, the ASTO bug, allows a method of creating this data type in the alpha register. Recall that with 7 characters in the alpha register, the second nibble of the seventh character will be stored with the ALPHA data value of an ASTO operation. In order to create a RCLFLAG data type, 6 characters must be entered into the ALPHA register with the first nibble of the first character being an ′F′. This may be easily done with the XTOA instruction. A byte is then appended to the end of these 6 characters with any one of several instructions: an append text line, XTOA, or ARCL. This byte will be found in the 16th column of the hex table. A good append text line would be one containing a ′?′. An ASTO X is then executed to yield the RCLFLAG data type in register X. In this manner a RCLFLAG value may be ARCL′ed for storage into an ASCII file and reconstructed for a later STOFLAG operation.

FLAG REGISTER INSTRUCTIONS (continued)

STOFLAG X=RCLFLAG data type or control word, bb.ee
Y=RCLFLAG data type if X contains control word

This instruction will cause the restoration of the statuses of flags F00 through F43 if X contains the RCLFLAG data. If X contains a control word, bb.ee, where bb and ee can be from 00 to 43, the Y register must contain the RCLFLAG data. Only the flag statuses defined by the control word are restored. For example a control word of 36.43 would only restore the trigonometric mode and display fix flags. The other flags F00 through F35 would be unaffected.

If more than one group of consecutive flags need restoration, multiple execution of this instruction with different control words will accomplish that. Possible error messages are:

DATA ERROR	The value in either X (or Y if a control word is used) does not conform to the RCLFLAG data format.
NONEXISTENT	The control word specified in X references flags out of the range of 00 to 43.

X<>F X=± 0 to 255

This instruction will exchange the contents of the X register and the statuses of flags F00 through F07, interpreting each flag to represent its binary power of 2. For example, flag seven would be 2 to the 7th power or 128, flag six would be 2 to the 6th power or 64, etc. This instruction allows for binary encoding or decoding of data without mathematical operations. This instruction should not be confused with the X<> __ instruction within the 41C's basic instruction set. The only error message is:

DATA ERROR	The absolute value of X exceeds 255.

It might at first be thought that this function would lend itself to setting bit 7 of a single byte for reverse video on the Video Interface; however, a timed execution reveals that the instruction sequence: X<>F, SF 07, X<>F is in fact slower than 128, +.

BLOCK REGISTER INSTRUCTIONS

The XFM contains two functions that enable blockwise manipulation of contiguous blocks of numbered storage registers in User RAM. They are:

REGMOVE X=control word, sss.dddnnn

This instruction will copy a block of registers of length 'nnn', starting at register number 'sss' to a block of registers starting at register number 'ddd'. If there is overlapping between the blocks, the transfer is ordered such that no register contents are lost. If the 'nnn' of the control word is zero, one register will be copied. The only error message is:

NONEXISTENT	The control word specifies data registers that are not included in the current SIZE.

REGSWAP X=control word, sss.dddnnn

This instruction will swap the contents of two blocks of data registers specified by the control word as defined for the REGMOVE instruction. Again, if the blocks overlap, the transfer is ordered such that no registers are lost. If nnn is zero, one register will be exchanged. The only error message is:

NONEXISTENT	The control word specifies data registers that are not included in the current SIZE.

These two instructions allow manipulation of mass amounts of data. Possible applications are: rapid sorting of data registers, array manipulation and sharing of registers R00 through R15 by multiple programs to take advantage of the speed and byte saving of short form STO's and RCL's.

Do not confuse the 'nnn' block length with the increment or decrement used with the ISG and DSE instructions.

GETKEY (no parameters)

This instruction stands out among the other XFM instructions. It will cause the 41C to wait approximately 10 seconds for a key to be pressed. The keycode of the pressed key is returned to the X register. If no key was pressed before the instruction 'timed out', the value 0 is returned. Note that the keycode of any key, including R/S, OFF, USER, PRGM, ALPHA, and SHIFT will be returned.

The keycode is computed as:

row * 10 + column

where the row containing the OFF key is row 0 and the row containing the R/S is row 8. Rows 0 and 4 contain only four columns, all of the others contain five, the leftmost key being column 1. Note that the keycode returned will never take a negative value as shown when using the ASN function, since the SHIFT key would return a value of 31. This holds true even if the SHIFT flag is set before executing GETKEY.

This instruction lends itself to specialized data entry. Only a single keystroke is allowed, and the value returned has a limited range. It could be used for example for: setting up programmable key assignments, games input such as pinball paddles, and single keystroke recall of arrayed data.

The first test that most programmers put this instruction to is similar to the following program:

```
01 LBL 01
02 GETKEY
03 VIEW X
04 GTO 01
```

They would begin punching keys and examining the keycodes displayed for any of the keys on the 41C. They would soon find that a simple R/S will not stop their program, but return an 84. An OFF keypress would return an 01. How do you stop the program? The 41C recognizes the R/S key as a program STOP when any other instruction but GETKEY is executed. If two R/S's are pressed in rapid succession, the program should stop at lines 01, 03, or 04. The fear introduced by this first confusing encounter causes most programmers to shy away from this instruction, but there is no reason for caution. This instruction should be used as any other instruction would to make the best use of data entry technique.

PROGRAMMABLE KEY ASSIGNMENTS

CLKEYS (no parameters)

This instruction will clear all key assignments from the 41C, including all global label assignments. There is no error message for this instruction.

PASN ALPHA=0 to 7 character name X=signed keycode

This instruction allows the selective assignment or clearing of USER mode key assignments. Upon execution, the instruction or global label specified in the ALPHA register is assigned to the keycode in X. If the value in X is negative, the assignment will be to a shifted key; if positive, to an unshifted key. If the ALPHA register is empty, the assignment at the specified keycode will be deleted. Only primary 41C instructions, peripheral instructions, or global labels may be assigned to a key. This instruction will not allow assignments to be made to the SHIFT key or any key on row 0 of the keyboard. Possible error messages are:

KEYCODE ERR	Attempt to assign to a prohibited or non-existent keycode.
PACKING TRY AGAIN	There is not enough memory to implement the key assignment.

This instruction is quite powerful, but difficult to wield in programs, since a function name can be up to 7 characters in length and a keycode takes 2 or 3 bytes for a numeric entry line. A possible solution is a generalized key assignment routine that stores the key assignments in extended memory for subsequent recall and implementation. This would enable easy key assignment setups as required for special menus and Application Pacs.

PROGRAMMABLE SIZE CONTROL

The XFM has finally given the progammer what he or she has been craving for since the 41C was first introduced...a programmable SIZE.

PSIZE X=size to be sized, 0 to 999

This instruction will resize the numbered data storage registers as specified by the number of registers in X. Remember that the highest numbered register is one less than the SIZE. This instruction can also be executed from the keyboard with the same effect. Possible error messages are:

DATA ERROR	The value in X is greater than 999. This may not be a portent of future expansion to more than 319 registers. It may have been an expediency since the file pointers are limited to the same value.
NO ROOM	There is not enough room for the SIZE specified (during program execution only).
PACKING TRY AGAIN	There is not enough room for the SIZE specified (during keyboard execution only).

SIZE? (no parameters)

This instruction returns the current SIZE to the X register as a positive integer. There is no error message for this instruction.

These two instructions can be used together to set a minimum size within which a program will execute:

```
SIZE?
7
X>Y?
PSIZE
```

Very this size according to what you want.

This sequence will resize User RAM for 7 data registers only if the current size is less than 7.

The most obvious use of SIZE? is to determine how much storage space is available for a program to use.

PROGRAMMABLE CLEAR PROGRAMS

PCLPS   ALPHA=0 to 7 character program name

This instruction will clear the named program, or the current program if the ALPHA register is empty, and all programs afterward down to the permanent .END. from memory. If the currently executing program clears itself, execution terminates as would be expected. Possible error messages are:

NAME ERR	The named file does not exist in User RAM.
ROM	The named file is in ROM memory or the program pointer is positioned in ROM and ALPHA is clear.

This instruction, combined with the GETP and GETSUB of the Extended Functions Module, the RSUB and MRG of the Card Reader, and the READA, READP and READSUB of the HP-IL enable the use of 'dynamic programming'. It is now possible to have programs coordinate the execution and storage of other programs...a feature only found before on larger computers. A program can be 'transient', only residing in User RAM when it is needed; allowing more space for other programs and data.

Does not pack or Clear end.

SECTION IV - QUIZ

1. With only an XFM, a single file that uses all of the extended memory would have an extent of:

   a. 127 registers
   b. 125 registers
   c. 124 registers
   d. < 124 registers
   e. none of the above

2. If a file header is decoded to read:

   2 . . . . 0 0 2 0 1 0

   one may deduce...

   a. a data file
   b. length 10
   c. an ASCII file
   d. length 16
   e. a and d

3. To select a working file with EMDIR, it is necessary to press which of the following keys?

   a. R/S
   b. ON
   c. SST
   d. a and b
   e. a and c

4. After a PURFL operation, a RCLPT instruction will give:

   a. the value of the current working file pointer
   b. the number of bytes if the working file is a program file
   c. the value 0.00000
   d. an error
   e. none of the above

5. The highest record number in a DATA file of extent 16 is:

   a. 16
   b. 15
   c. 999
   d. none of the above

SECTION IV - QUIZ

1. With an XFM, a single file that uses all of the ext memory would have an extent of:

   a. registers
   b. registers
   c. registers
   d. registes
   e. of the above

   -2 Header
   -1 working file pointer

2. If header is decoded to read:

   2 . . . . 0 0 2 0 1 0

   ← data   pointer   extent 16   Hex

   one educe...

   a. a file
   b. n 10
   c. CII file
   d. h 16
   e. d

3. To a working file with EMDIR, it is necessary to which of the following keys?

   a.
   b.
   c.
   d. b
   e. c

4. Aft URFL operation, a RCLPT instruction will giv

   a. value of the current working file pointer
   b. umber of bytes if the working file is a am file
   c. alue 0.00000
   d. ror
   e. of the above go off and come back.

5. The st record number in a DATA file of extent 16 is:

   a.
   b.
   c.
   d. of the above

6. If the program pointer is between the last END and the permanent .END., a GETP instruction will:

a. copy the named file after the .END.
b. copy the named file after the program pointer
c. copy the named file after the last END
d. overwrite any file between the END and .END.
e. c and d

7. If the X register contains ~~0.9999~~ 0.099099 and a REGMOVE instruction is executed, providing there is sufficient numbered storage registers, where will the contents of register 88 be moved to?

a. R188
b. R187
c. R99
d. R89
e. none of the above

8. If the X register contains the following decoded value

SCI DEG

1F F0 00 00 00 80

\# of Digits.

1	0	0	0	0	0	0	0
36	37	38	39	40	41	42	43

40	41	
0	0	SCI
0	1	Eng
1	0	FIX
1	1	FIX/ENG

what is the display fix set by a subsequent STOFLAG?

a. SCI 8, DEG
b. FIX 8, DEG
c. ENG 8, RAD
d. SCI 2, GRAD
e. none of the above

1. FIX 0-9
2. RCLFLAG
3. ENG 0-9
4. 41.41
5. STOFLAG

9. After a DELCHR instruction in an ASCII file, the file pointer is positioned to:

a. the end of file
b. the start of the record number deleted
c. the end of the previous record
d. depends on the contents of x
e. stays the same

10. After a GETREC instruction on an ASCII file, if flag 17 is set, it indicates:

    a. the end of file has been reached
    b. the alpha register is full
    c. the end of record has not been reached
    d. b and c
    e. none of the above

PROGRAM LISTING

☐ 67 ☐ 97 ☒ 41C

STEP/LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
01	♦LBL "MPG"		Miles Per Gallon
02	SIZE?		SIZE greater than 7
03	7		required
04	X>Y?		
05	PSIZE		
06	RCLFLAG		save flag statuses
07	STO 00		
08	CF 21		set printing off
09	FIX 0		set display modes
10	SF 28		and ANUM decoding
11	CF 29		
12	♦LBL 15		prompt for car #
13	"CAR NO."		
14	XEQ 90		
15	"CAR"		build filename
16	48		
17	+		
18	XTOA		
19	ASTO 01		
20	CF 06		save name
21	SF 25		F06=extension file
22	CLX		zero first odometer
23	STO 06		
24	SEEKPTA		select file as
25	FS? 25		working file
26	GTO 16		
27	CF 25		if nonexistent,
28	"NEW FILE"		display "NEWFILE"
29	AVIEW		message and create
30	TONE 8		a new file
31	CLA		
32	ARCL 01		
33	20		
34	CRFLAS		
35	GTO 18		go directly to
36	♦LBL 16		DATA ENTRY
37	"⊢X"		try to open the
38	SEEKPTA		extension file
39	FC? 25		if there, use it
40	SF 06		else the first
41	SF 25		file is used.
42	♦LBL 17		read to end of file
43	CLA		initial CLA
44	GETREC		needed for empty
45	FS? 25		file.
46	GTO 17		
47	ATOX		loop until found
48	ANUM		..read last odometer
49	STO 06		save for MPG calc.
50	♦LBL 18		
51	"ENTER DATA"		ask if data entry
			desired.
52	XEQ 80		default=YES
53	FC?C 05		F05 set=yes
54	GTO 30		
55	♦LBL 19		enter Oil/Gas Co.
56	"GAS CO.?"		letter
57	AON		
58	TONE 5		
59	PROMPT		
60	AOFF		
61	ATOX		get leading letter
62	STO 02		
63	"ODOMETER"		prompt for odometer
			reading
64	XEQ 90		
65	STO 03		
66	"$/GAL"		prompt for price
67	XEQ 90		per gallon for
68	STO 05		gasoline
69	"PRICE$"		prompt for total
70	XEQ 90		purchase cost
71	STO 04		
72	RCL 03		calculate MPG from
73	RCL 06		cost data and
74	X=0?		miles traveled
75	GTO 20		if last odometer
76	-		=0 then no calc.
77	RCL 04		is displayed
78	RCL 05		
79	/		
80	/		
81	FIX 0		
82	"MPG="		display tank MPG
83	ARCL X		
84	AVIEW		
85	TONE 9		
86	♦LBL 20		
87	RCL 03		shift current
88	STO 06		odometer into
89	SF 25		last odometer
90	XEQ 70		try to save data

PROGRAM LISTING

☐ 67 ☐ 97 ☒ 41C

STEP/ LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
91	FC?C 25		if error try to open extension
92	XEQ 72		
93	"MORE"		ask if more data
94	XEQ 80		
95	FS?C 05		if so, loop back
96	GTO 19		
97	♦LBL 30		printout section
98	FC? 55		check for printer
99	GTO 98		if none, skip
100	"PRINTOUT"		prompt for print-out desired
101	XEQ 80		
102	FC?C 05		
103	GTO 98		
104	CF 06		if so, do it
105	CF 12		set printer modes
106	CF 13		
107	SF 21		
108	ADV		build heading
109	"MILEAGE FOR "		
110	ARCL 01		
111	PRA		
112	CLX		
113	STO 06		
114	ADV		
115	"C ODOM. VOLUME "		
116	"⊢PRICE"		
117	PRA		
118	"O MILES GALS "		
119	"⊢ $ MPG"		
120	PRA		
121	"- ------------ "		
122	"⊢--------"		
123	PRA		
124	CLA		
125	ARCL 01		
126	♦LBL 40		open base file for reading
127	CLX		
128	SEEKPTA		
129	♦LBL 41		
130	SF 25		read a record for printing
131	GETREC		
132	FC?C 25		if error, look for extension
133	GTO 96		
134	DELREC		after print, delete
135	ATOX		decode for print formatting
136	STO 02		
137	ANUM		
138	STO 03		
139	XEQ 73		
140	ANUM		
141	STO 04		
142	XEQ 73		
143	ANUM		
144	STO 05		
145	CLA		build print image
146	RCL 02		
147	XTOA		gas co.
148	"⊢ "		
149	FIX 0		
150	ARCL 03		odometer
151	"⊢ "		
152	FIX 3		
153	RCL 04		
154	RCL 05		
155	/		
156	10		align decimal pt.
157	X>Y?		
158	"⊢ "		
159	ARCL Y		volume
160	"⊢ "		
161	FIX 2		
162	RCL 04		
163	X<Y?		align decimal pt.
164	"⊢ "		
165	ARCL X		price (total)
166	"⊢ "		
167	FIX 0		
168	RDN		
169	RCL 06		
170	X≠0?		blank out first mpg calculation
171	GTO 42		
172	"⊢ -"		
173	GTO 43		
174	♦LBL 42		calculate mpg
175	RCL 03		
176	-		
177	CHS		
178	RCL T		
179	/		

Note: Refer to "HP-41C OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for specific information on keystrokes. The Function index is found at the very back of the Handbook. Refer to Appendix E in 67 or 97 "OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for exact keystrokes.

PROGRAM LISTING

☐ 67 ☐ 97 ☒ 41C

STEP/LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
180	ARCL X		mpg
181	♦LBL 43		
182	PRA		print the record
183	RCL 03		save current odom-
184	STO 06		eter as the last
185	GTO 41		loop back
186	♦LBL 70		routine to build
187	CLA		file record
188	RCL 02		$#####/##.##/#.###
189	XTOA		where $=char
190	FIX 0		#=number
191	ARCL 03		.=decimal pt
192	"⊢/"		/=delimiter
193	FIX 2		
194	ARCL 04		
195	"⊢/"		
196	FIX 3		
197	ARCL 05		
198	APPREC		append to working
199	RTN		file
200	♦LBL 71		routine to open
201	FC? 06		and or create
202	GTO 72		extension file
203	"PRINT/R		msg for both files
	EDO"		full.
204	AON		use printout to
205	STOP		regain room
206	♦LBL 72		build extension
207	CLA		file name
208	ARCL 01		
209	"⊢X"		
210	20		
211	CRFLAS		create it
212	SF 06		
213	GTO 70		if ok, retry save
214	♦LBL 73		routine to delete
215	47		all characters in
216	♦LBL 74		ALPHA until the
217	ATOX		delimiter is
218	X<>Y		reached
219	X≠Y?		
220	GTO 74		
221	RTN		
222	♦LBL 80		Yes/No prompting
223	CF 05		routine
224	AON		Default=Yes
225	"⊢?"		
226	CF 23		
227	TONE 7		prompt for Y/N
228	PROMPT		
229	AOFF		
230	89		
231	FS?C 23		
232	ATOX		
233	89		
234	X=Y?		
235	SF 05		
236	RTN		
237	♦LBL 90		routine to prompt
238	"⊢=?"		for numeric
239	♦LBL 91		data
240	CF 22		it will loop
241	TONE 5		until a value is
242	PROMPT		keyed in.
243	FC?C 22		
244	GTO 91		
245	RTN		
246	♦LBL 96		completion test
247	CLA		
248	ARCL 01		
249	FC? 06		
250	PURFL		if working file
251	"⊢X"		is base, purge
252	FS? 06		if working file is
253	GTO 97		extension, skip
254	SF 06		else open the
255	SF 25		extension if it
256	FLSIZE		exists, else
257	FS? 25		done
258	GTO 40		
259	SF 25		
260	♦LBL 97		purge extension
261	PURFL		file when done
262	CF 06		
263	ADV		
264	ADV		paper out
265	CF 21		reset print mode
266	♦LBL 98		
267	"ANOTHER		ask if another
	CAR"		car is desired
268	XEQ 80		
269	FS?C 05		
270	GTO 15		
271	♦LBL 99		exit routine
272	RCL 00		restore flags
273	STOFLAG		clear X,ALPHA,
274	CLX		and display
275	CLD		
276	END		

PROGRAM LISTING

☐ 67 ☐ 97

STEP/ LINE	KEY ENTRY (CODE only)	COMMENTS
	01◆LBL STS	Status program
	02 "LOAD OR NEW?"	prompt for load or new status
	03 AON	
	04 PROMPT	
	05 ATOX	
	06 "NEW" 76	default is LOAD
	07 "NEW"	begin build of program name to call
L	08 X=Y? load	
HSTS	09 "L" either	
	10 "STS" NSTS	
	11 GETP	get the program
	12 ASTO X	transfer control to the program
GTO IND X	13 GTO IND X	Indirect access of a ALPHA LABEL
	14 END	

STEP/ LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
	LBL STS		
	LOAD OR NEW?		

This program is 45 bytes long and would require an XM file of 7 registers.

No flags are used and no registers are used.

Note: Refer to "HP-41C OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for specific information on keystrokes. The Function Index is found at the very back of the handbook. Refer to Appendix E in 67 or 97 OWNER'S AND PROGRAMMING GUIDE for exact keystrokes.

PROGRAM LISTING

☐ 67 ☐ 97 ☒ 41C

STEP/LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
01	◆LBL "NSTS"		New Status file
02	RCLFLAG		generating
03	STO 05		program
04	CF 21		Save current flag
05	SF 28		statuses and
06	CF 29		set those required
07	◆LBL 00		by the program
08	"FILENAME?"		Prompt for the filename to
09	AON		store under
10	STOP		
11	ASTO 00		save filename
12	SF 25		
13	CLX		set the file flags
14	X<>F		
15	SF 07		
16	CLX		try to select the
17	SEEKPTA		file to determine
18	FC?C 25		existence
19	GTO 01		if so, ask whether
20	"OK TO PURGE?"		to purge it
21	XEQ 11		Y/N continue
22	X≠Y?		default=NO
23	GTO 00		
24	CLA		else, purge it
25	ARCL 00		
26	PURFL		
27	◆LBL 01		
28	"NO. OF KEYS?"		number of key assignments to
29	AOFF		make. this is
30	PROMPT		needed to size
31	STO 04		the file
32	9		
33	*		
34	19		
35	+		
36	7		
37	/		
38	CLA		
39	ARCL 00		
40	CRFLAS		create the file
41	"SIZE"		Ask whether to
42	XEQ 10		SIZE..default=NO
43	X≠Y?		
44	GTO 02		

STEP/LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
45	SF 06		
46	"MIN OR ABS?"		ask whether absolute size or minimum
47	PROMPT		size required
48	ATOX		default=minimum
49	65		
50	X=Y?		
51	SF 05		
52	"SIZE=?"		Ask for the desired
53	AOFF		size
54	PROMPT		
55	STO 01		
56	◆LBL 02		
57	"FLAGS"		Ask whether a flag
58	XEQ 10		status is desired
59	X≠Y?		default=no
60	GTO 03		
61	SF 04		save current status
62	RCLFLAG		
63	STO 02		
64	"SET THEN R/S"		Prompt for flags to be set as desired
65	PROMPT		
66	RCLFLAG		
67	X<> 02		save these and get
68	STOFLAG		the previous
69	CF 22		Prompt for a flag
70	"MASK=?"		mask, default=
71	AOFF		no mask
72	PROMPT		
73	FS?C 22		
74	SF 03		
75	STO 03		
76	◆LBL 03		
77	"CLKEYS 1ST?"		Ask whether to clear all key assignments
78	XEQ 11		before making the
79	X=Y?		current ones.
80	SF 02		default=no
81	CLA		
82	FIX 0		
83	X<>F		get file flag byte
84	XTOA		
85	X<>F		restore it for use
86	FS? 06		here
87	ARCL 01		build the file data
88	APPREC		and save it
89	CLA		

Note: Refer to "HP-41C OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for specific information on keystrokes. The Function index is found at the very back of the Handbook. Refer to Appendix E in 67 or 97 "OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for exact keystrokes.

LBL NSTS

PROGRAM LISTING

☐ 67 ☐ 97

	STEP/LINE	KEY CODE (67/97 only)	COMMENTS
ARCL	90	02	
FS?	91	4	
APPREC	92	C	
CLA	93		
FIX 2	94		
ARCL 03	95	03	
FS? 03	96	3	
APPREC	97	C	
1 E3	98		generate a loop
ST/ 04	99	4	control number
SIGN	100		for key entry
ST+ 04	101	4	
FIX 0	102		
LBL 04	103	4	
"PRESS KEY"	104	S K	prompt for desired
ARCL 04	EY		key assignment
AVIEW	105	04	
LBL 05	106		
31	107	5	
GETKEY	108		
X<0?	109	Y	
GTO 05	110		
X≠Y?	111	5	
GTO 07	112		
LBL 06	113	7	
31	114	6	account for shifted
GETKEY	115		keys and shifted
X<0?	116	Y	shifts
GTO 06	117		
X≠Y?	118	6	
GTO 05	119		
CHS	120	5	
LBL 07	121		shifted
55	122	7	normalize to 1 to
+	123		169 for use as a
STO 01	124		leading byte
"NAME"	125	1	
ARCL 04	126	"	Prompt for function
"⊢=?"	127	04	or program name
AON	128		for that key
AVIEW	129		
CLA	130		Allows you to also
STOP	131		clear keys.
RCL 01	132		
XTOA	133	1	build key record
SIGN	134		and save it
CFS	135		
	136		

STEP/LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
137	AROT		
138	APPREC		
139	ISG 04		loop until done
140	GTO 04		
141	RCL 05		
142	STOFLAG		restore flags to entry states
143	CLST		
144	CLA		clean up by clearing
145	AOFF		program from
146	PCLPS		user memory
147◆	LBL 10		General YES/NO
148	"⊢ Y/N?"		prompting routine
149◆	LBL 11		second entry point
150	AON		
151	PROMPT		
152	ATOX		
153	89		lets calling sequence handle the logic
154	.END.		

Note: no normal end since this is a transient program.

Uses flags: F00-F07, F21, F22, F25, F28, F29, sets FIXes and uses AON/AOFF.

Uses registers R00-R05

354 bytes in RAM

51 registers in extended memory

Note: Refer to "HP-41C HANDBOOK AND PROGRAMMING GUIDE" for specific information on keystrokes. The Function Index is found at the very back of the Handbook. Refer to Appendix E in 67 or 97 "OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for exact keystrokes.

PROGRAM LISTING

☐ 67 ☐ 97 ☒ 41C

STEP/LINE	KEY ENTRY	KEY CODE (67/97 only)	COMMENTS
01	◆LBL "LSTS"		"Load Status"
02	"FILENAME?"		prompt for filename
03	AON		
04	STOP		
05	AOFF		
06	CLX		
07	SEEKPTA		select file as
08	GETREC		working file.
09	ATOX		get file flags.
10	X<>F		
11	FC? 06		size?
12	GTO 00		
13	SIZE?		find size
14	ANUM		get saved size
15	FC? 05		if ... absolute
16	X>Y?		or too small
17	PSIZE		then resize
18	◆LBL 00		
19	FC? 04		skip flag status
20	GTO 00		
21	GETREC		
22	FS? 03		
23	GETREC		and mask for now
24	◆LBL 00		
25	FS? 02		clear keys first?
26	CLKEYS		
27	◆LBL 01		loop until F25
28	SF 25		cleared
29	GETREC		get key assignmnt
30	FC?C 25		
31	GTO 02		F25 exit
32	ATOX		get key code
33	85		normalize to
34	-		-84 to +84
35	PASN		assign it
36	GTO 01		
37	◆LBL 02		take care of
38	FC? 04		flags if
39	GTO 04		desired
40	1		position to flag
41	SEEKPT		record
42	GETREC		use BUG 7 tech.
43	"├?"		to generate
44	ASTO X		STOFLAG data
45	FC? 03		
46	GTO 03		
47	GETREC		get mask
48	SF 28		set flags for
49	CF 29		proper ANUM
50	ANUM		
51	◆LBL 03		
52	STOFLAG		set flag status
53	◆LBL 04		
54	CLA		clear LSTS from
55	PCLPS		User RAM
56	.END.		

(Since this program is retrieved from extended memory under program control, there will never be a regular END, just the permanent .END.

USAGE:

Uses no data registers.

Flag statuses are either set by the program or left at:

F00-07=file flags
F25-clear
F28-set
F29-clear

Program execution is directed by another program, control will be returned to the keyboard upon completion.

XFM, SIZE=017
114 bytes

Note: Refer to "HP-41C OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for specific information on keystrokes. The Function index is found at the very back of the Handbook. Refer to Appendix E in 67 or 97 "OWNER'S HANDBOOK AND PROGRAMMING GUIDE" for exact keystrokes.

SECTION V

SYNTHETIC PROGRAMMING

Σ Reg 00

CLΣ

1

STO 00

1E67

STO 02

1E-97

STO 05

SF 25

SDEV or MEAN

SECTION V

Synthetic Programming

The power of the HP-41C/CV system is truly overwhelming. Up to this point only the standard HP documented techniques; and some common sense have been used to increase the capabilities of the 41. But with all of this there is still a more powerful set of functions available to the user; these are the synthetic functions.

Synthetic functions cannot be keyed in by normal techniques and thus are called synthetic because their code is 'synthesized' in the calculator's memory. The beginnings of synthetic programming (SP) on the 41 can be traced back to the first machines sold. Their microcode contained some anomalies that allowed the user to STO/RCL IND to very large register numbers (704-999). Another anomaly was the ability to SF/CF IND for all of the 41's 56 flags. These two anomalies became known as "BUGS" 2 and 3[1]. There are, or were, several other bugs in the box; a list of these is shown in Figure 5.1. For now, we will only deal with Bugs 2, 3, and 9.

Note: There are over 100 HP documented BUGS. This list is only those that are more interesting, useful, or damaging.

Make a test for the Bugs.

Bug No.	Discription	Referance
1	Σ+ and Σ- don't save X in LASTX	V6 N5 P28
2	STO/RCL IND 704 through 999	"
3	SF/CF IND all 56 flags	"
4	Small angle SIN error	V6 N5 P30
5	Incomplete CLP of large programs	"
6	Digit termination 41 translation of HP-67/97 programs	V6 N8 P23
7	Fragmented seven character alpha	"
8	Non-compile if OFF in PRGM mode	"
9	Too small or too large line numbers	V7 N9 P25
10	Statistics/Error ignore	

Figure 5.1 - Table of Documented Bugs

Bug two is a very useful critter. It makes the calculator appear to have a full 999 registers available to store data in. If you have a 41C with a Quad RAM, or a 41CV, what it is really doing is wrapping around in memory. With the XFM and XM modules, it is actually accessing those registers. This will be discussed again in the section on addressing. The advantage of BUG 2 was that it gave access

1. As numbered and classified by PAC.

Addressing Memory:

In section one we talked about the structure of the 41's memory. Let us look again at the 41's memory structure and how the 41 keeps track of instructions ~~shown~~ in memory. Remember the 41C has ~~available~~ both RAM and ROM memory ~~available~~

These two area's ~~are~~ of memory are accessed by two completely separate I/O lines. This fact is one of the reasons why port extenders work and are advantageous. This allows both a RAM module and a ROM module to be addressed to the same port on the calculator. For example both the Math Module and the Quad Memory can be in the same port at the same time.

¶ Another feature that ~~bts~~ is taken advantage of by the port extenders is that some ROM's are permanently addressed to the lower 8 pages (0-7) of ROM space. ~~and other~~ Other ROMs (most of the application ROMs) are port addressed. Part of their address is determined by what port it is plugged into. For example The Petroleum Fluids module and Timer Module could be in the same port at the same time. This is because the Petroleum Fluids pak is port addressed and the Time Module is hard addressed to page 5 in ROM memory.

~~Similar~~ Similarly It is possible to combine RAM modules by being sure they all have their proper address wired in. It is obvious that addressing in the 41 is an important ~~deacture~~ feature to understand. A lot of things not normally possible on the 41 become not only possible but ~~understandable~~ desirable when the addressing of memory on the 41 is understood. Let us ~~exaime~~ examine the structure of the 41's address pointer.

The same address pointer is used when the 41 is addressing either RAM or ROM. The structure of the pointer is slightly different for each case. Lets look at the ROM version first. The 41 can address a total of 64K Words of ROM memory. The form of the address pointer used to address rom is the familiar form used by most computers to address memory. It uses the full two bytes of the pointer. Each word is directly addressed from ØØØØ to FFFF.

The RAM pointer is not so straight forward as the ROM pointer. The ~~RA~~ RAM in the 41 is arranged in blocks of seven bytes called registers, This is for convienence in storing and manipulating numbers. But when running a program it is necessary to address the individual bytes within the registers. Because of this the two byte address pointer is broken down into two different feilds. Figure 5.2 shows this break down. The first three nibbles ~~0,1,2~~ 2-Ø are used as the register pointer. The last nibble (3) is used as a byte pointer. Lets examine the register pointer first. It is three nibbles or 12 bits in length, but uses only the lower 10 bits. This makes possible the addressing ~~the~~ of 1024 total registers (2^{10}; ØØØ to 3FF). It should be noticed that when the CPU is accessing main memory the 10^{th} bit is ~~never~~ allways Ø (main memory extends from ØØØ to 1FF), and only when XMemory is accessed is the 10^{th} bit set. When we examine the subroutine return stack we will see why this is important. The rest of the registers from 2ØØ to 3FF are present only when 1 or 2 extended memory modules are present. The register pointer is used to tell the CPU what register in RAM ~~the~~ it is presently accessing.

The byte pointer is the 3^{rd} nibble of the address pointer. A register contains seven bytes each. If the CPU is to know which byte is the next executable program instruction it is necessary to know where it is within a register. Program instructions don't usually take up a full register so one or more instruction are ~~is~~ stored with in any given register. The byte pointer is always set to the ~~last~~ byte before the next executable instruction in a program. The value of the byte pointer can range from $\emptyset$ to 6 which uses ~~s~~ three of the four bits available to it.

To review then the address pointer has two forms. The first form is used when accessing ROM and is the ~~typte~~ typical pointer from $\emptyset\emptyset\emptyset\emptyset$ to FFFF. The second form is used when accessing RAM. This form splits the Pointer into a Register and byte pointer. ~~S~~ This is demonstrated in figure 5.3. The address 40C2 is pointing to register $\emptyset$C2 and byte 4 within that register.

Need to add a section for definitions.

to all of the 41's registers. This includes the Status, Key assignment, program and data registers at the same time. This allowed us to store things directly into memory where we wanted it, to create synthetic codes.

The other useful bug was BUG 3. This bug allowed us to set or clear any of the 56 flags as we choose. This gave us a quick way of decodeing arbitrary bytes stored into registers. It also gave us access to the system flags which allowed us to do things like clear flag 55 when the printer was attached but not needed, or set the CAT flag.

It wasn't long before HP corrected most of these "BUGS" and they were no more. Those who still had the bugs continued to explore with them and make discoveriers. They began to automate their exploration and write programs that would simulate the actions of some of the bugs like 2 and 3 which had desirable effects. Their efforts gave all 41s the capabilities to run synthetics. This made the 41 more and more like a true miniture computer.

Our goal in this portion of the course is to lead you through the forest and let you look at the different speices of trees living there. This will let you create and use synthetics on your machine and save you time trying to learn all of the material from several different sources.

ADDRESSING MEMORY

In section I, we examined the organization of the 41's memory. Lets take another look at this organization and see how its addressed by the machine. There are two ways to address somthing, so lets define a couple of terms before we get started.

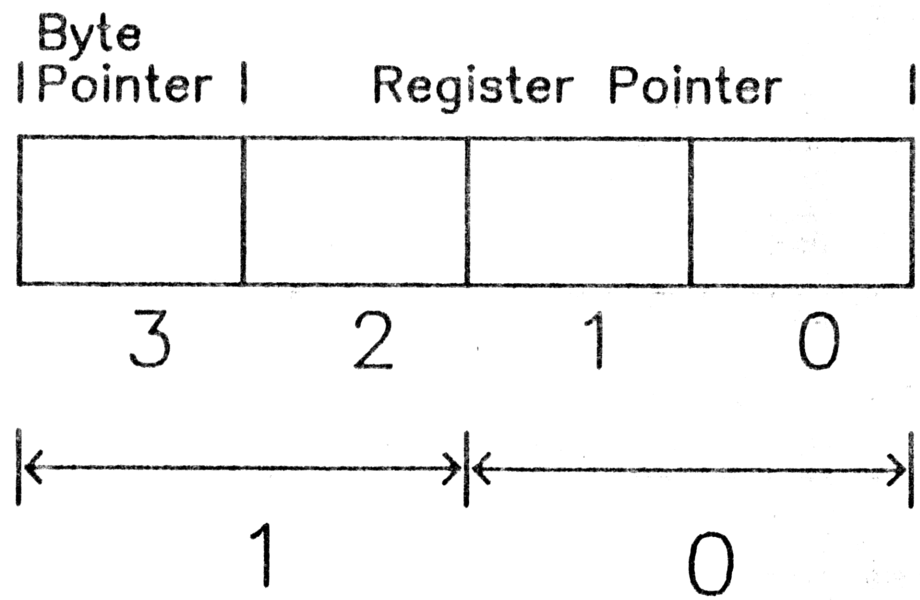
Definitions:

Absolute Address - the address of the register regardless of what is stored in or associated with the register. For RAM ØØØ-3FF, for ROM ØØØØ-FFFF.

Relative Address - the distance between two addresses or the address associtated with certia registers at certian times.

The addressing scheme for RAM is really rather simple. Each address consists of two bytes. These two bytes are broken into two different fields, the byte pointer and the

Structure of a two byte address pointer



This is the form of the address pointer in the 41C for RAM memory.
1. You see variations of it in all of the GTO's that get compiled
2. END statements
3. Subroutine Stacks.

Figure 5.2

register pointer. This is shown in figure 5.2. The register pointer is contained in the first three nibbles, and byte pointer is contained in the last nibble.

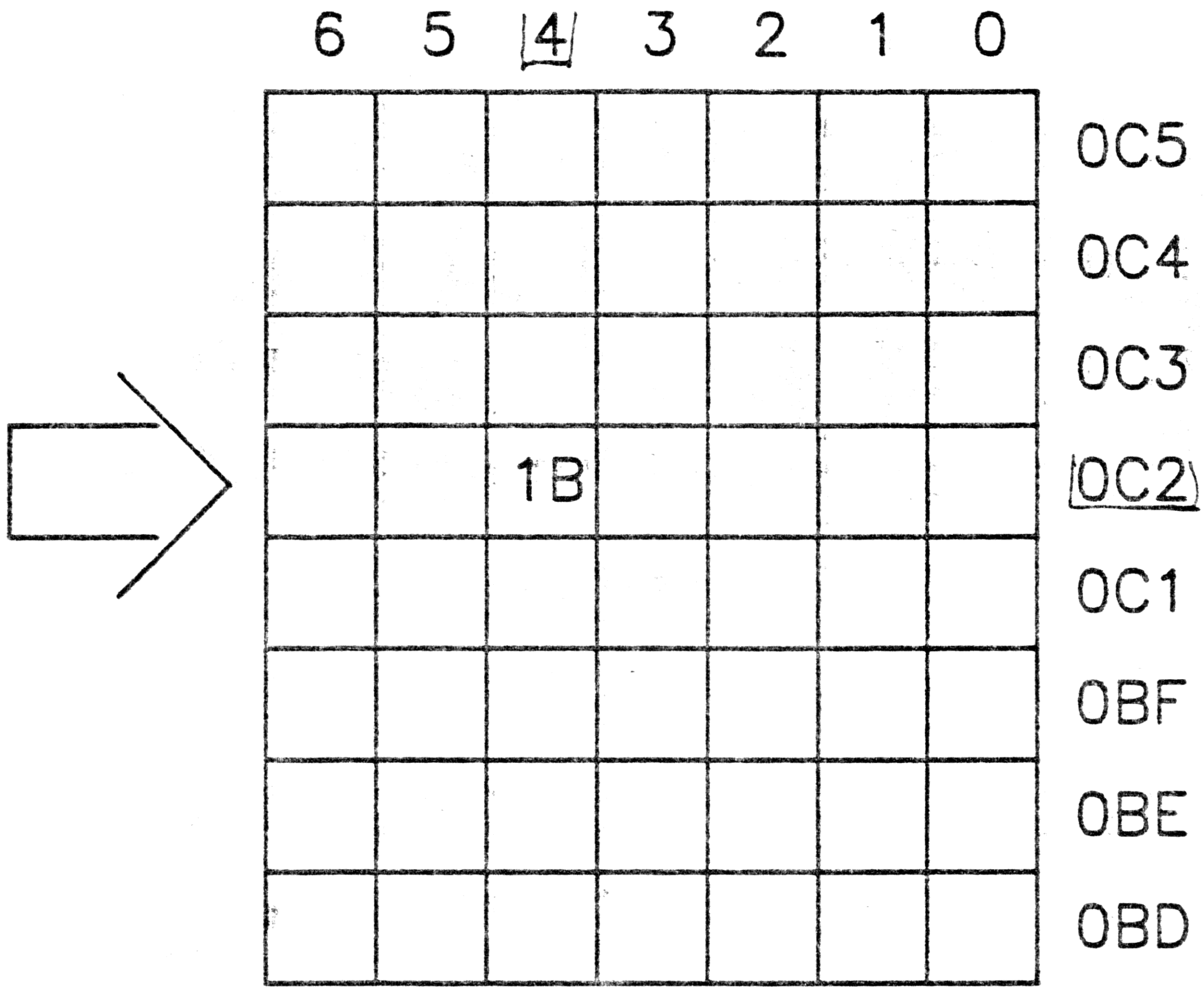
Figure 5.2 - Structure of a Address Pointer

The 41 has register address form HEX ØØØ to 1FF for the User RAM and 200 to 3FF for the extended memory. It is here that the odd behavior of Bug 2 becomes apparent. Notice that 3FF (HEX) is 1024 (DEC). This means the 41 should have a full 1024 registers from ØØØ to 1023. Where are they? Figure 1.3 shows the addresses of the segments of RAM memory, with XM from 040 to 0BF and 200 to 3FF with a void from 010 to 03F. When bug 2 was used to access these large registers it was trying to locate these XM registers but instead wrapped around in memory to STO/RCL into the User RAM and Status registers. With extended memory these registers became legitament and so indirect access was gained to the XM. The register pointer in RAM then can have values from 000 to 3FF in it.

A distinction must be made here between the addressing of RAM and ROM. There are a total of 1024 RAM registers available if one counts XFM/XM modules also. In ROM there are 64K of 10 bit words available to the CPU. The important differance between RAM and ROM is that the 41 uses two different I/O lines to address them. So their Address Pointers are a little different. The ROM Address Pointer doesn't have a byte pointer. It is a 2 byte address from ØØØØ to FFFF. But it is still kept in the same location and an internal flag 10 (CPU) is used to distingish which is being addressed.

The byte pointer for RAM is the 4th nibble of the Address Pointer. It points to which byte is the next to be processed by the CPU. Since there are seven bytes to a register it can have values of 0 to 6. So a given RAM address that had the following values in it

Example of the address pointer



Address Pointer = 40C2

Figure 5.3

'40C2'

Would point into memory as shown in figure 5.3.

Figure 5.3 - Example of RAM Address Pointer

By storing the proper code into the address pointer we can move freely around in both RAM and ROM memory. This is useful for Exploring the system and seeing how things are done.

Insert New material on NNN
Into this area

STATUS REGISTERS

material on Status registers is okay.

One of the most powerful results of SP is the ability to access directly all of the 16 status registers that the 41 uses. These registers are located at the bottom of memory, from address 000 to 00F. Figure 5.4 shows a block diagram of these registers. We will look in more detail at

Figure 5.4 - Status Registers

NNN's

Since we are going to be dealing with data stored in registers, we should know how the 41 determines what is held within its registers. There are several different kinds of information the 41 deals with.

Numeric Data
Alpha Data
Program Instructions
Key Assignments
~~Time Module~~ Alarms Buffers
I/O Buffers.

} Buffer Registers

Review figure 13 at this point to show sections of memory

Lets see how the 41 determines what is stored with in its registers. In general Numeric and/or Alpa data is expected to be found only in the storage registers (RØØ - Rnn)., ~~Program~~ Likewise ~~instructions to~~ program instructions are only expected to be found from just below RØØ to and including the .END. . Key assignments ~~are~~ start at location ØCØ and continue upward toward the .END. . Alarm Buffers ~~are~~ start imediantly after the key assignments. And I/O buffers are immediatly after that. Because the 41 has a general idea of what to expect to find with in certian ~~cetian~~ areas of its memory it uses the 13th nibble or the 13th and 14th nibbles (6th byte) to tell what type of information is stored within the register. Figure 5.9 shows the header byte (6th) for the different types of data stored in memory

Fig. 5.9.

Unlike information ~~data~~ stored in the data registers, information stored in program registers is not always rigidly formatted into a given register. We often come across seven bytes of data in a register that would not correspond to alpha or numeric data. This register is said to be non-normalized.

We will concentrate here on numeric, alpha, and program data. Key assignments will be discussed latter. Data stored in the registers designated for programs is not formatted with a header nibble or byte. Instructions begin at the global label and follow sequentially filling consecative regsisters from left to right. This is unlike information that is stored as numeric or alpha data in the Data ~~Storage~~ registers.

Numeric data splits the 14 nibble register into 4 feilds. The 13th nibble is used as a sign digit for the mantissa. It has the value Ø for a positive number and 9 for a negative number. The next 10 ~~digits~~ nibbles (12 - 3) contain the 10 digit mantissa. each nibble contains one digit of the mantissa, most significant in ~~the~~ nibble 10, least significant in 3. These 10 nibbles can contain values between Ø and 9. The 2 nibble is the exponent sign digit. Like the mantissa sign it can have a value of Ø or 9. Nibbles Ø and 1 contain the exponent, It can have values from Ø to 9.

Alpha data when stored in a register is also formatted The ~~first~~ 6th byte is a 1Ø when alpha is stored. The other 6 ~~characters~~ bytes contain the 6 characters stored in the register, as their ASCII equivalents. If less then six characters are in the register they are left justified and padded with nulls.

An exception to the above information is the status registers. These 16 registers contain a lot of information that is formatted in various ways (see Status Registers). If data was recalled from one of these registers and Normalized by some means the information it contained would be lost. The data stored in most registers other than the data registers is ~~7~~ seven arbitrary bytes. The normalization of ~~an~~ ~~seven arbitrary~~ arbitrary seven bytes of a register into Numeric or alpha data could destroy the information and in some cases this could cause ~~loss~~ ~~lock-up the~~ of the keyboard to lock-up or "memory lost."

A non normalized number is any number that contains a value other then Ø, 1, 9 in the 13th nibble, if the register is numeric data a value A-F in the mantissa or exponent, ~~But a~~ or a value other then Ø or 9 in the exponent sign. (Program instructions, and the information in the status registers are NNN if they are contained or placed into data registers.) ~~The~~ The normalazation of a NNN at an inoppertune time can cause a lot of trouble when one is ~~doing~~ doing synthetic programing, exspecally in the status registers. Because of this we need to be aware of what functions can normalize a NNN.

Normalization of NNN

We need to be aware that normalization can occur and what functions will cause it to occur. The most common ones are those that operate on data stored in the data registers; STO, RCL X<>, VEIW, AVIEW, ASTO, ARCL. We'll look at these one at a time.

STO

Has no effect on the data, it exactly copies the information in the X-register into the specified register. No normalization occurs.

RCL,

RCL will normalize information stored in a register to either numeric or alpha data. How it is normalized depends on the value in the sign digit(nibble 13). If the sign digit contains a Ø or 9 the data is normalized to numeric data. If in this case other nibbles had values greater than 9 (A-F) these revert to their decimal equivalent and carrier a one in to the next digit. This is demonstrated in the following two

ø5 $\boxed{6A}$ 96 43 21 $\boxed{6C}$ øø → Hex Coded NNN
0.5 7ø 96 43 22 " → x-reg b4 ~~STO~~ STO øø
+ 5.7ø 96 32 - øø → x-reg after RCL øø
ø 5 $\boxed{7ø}$ 96 43 21 $\boxed{79}$ øø → Normalized Hex code

examples.

#1:

ø5 6A 96 43 21 6C øø	~~Hex~~ NNN code	
0.5 7ø 96 43 22 "	x-reg before STO øø ~~coded as NNN~~	
5.7ø 96 32 - øø	after ~~STO~~ /RCL øø	
05 $\underline{7ø}$ 96 43 21 $\underline{79}$ øø	~~Decode NNN~~ normalized code	

The A is 1ø decimal and so gives a ø with a carry to the 6 to give 7ø. The C in the exponent sign is treated as a negative exponent and changed to a 9 with a carry. Hence the -øø exponent.

#2

ø5 $\boxed{6A}$ 96 4C ~~0~~ 69 $\boxed{øC}$ ~~Hex~~ NNN code
5.7096522 -ø4 → x reg before STO øø
5.7096522 -88 → x reg after RCL øø
ø5 $\boxed{7ø}$ 96 52 21 69 $\boxed{12}$ → ~~Hex after decode~~. normalized code

the 4C byte is changed to 52. C = 12 dec. gives 4+1 2 ⇒ 52. The C in the exponent along with the negative sign give a -88 exponent.

The effect of normalization by recall can be quite disastourous to NNN if they are changed to ~~numeric~~ numeric data.

If the sign digit contained something other than a 9 or ø, the normallization is to alpha data.

For example: $\boxed{25}$ 64 28 $\boxed{øC}$ 29 ø1 $\boxed{3F}$ NNN ~~Hex~~ code
-5.6428122-71 x-reg before STO øø
ċ (μ) Ā ? x-reg after RCL øø
$\boxed{15}$ 64 28 $\boxed{øC}$ 29 ø1 $\boxed{3F}$ ~~Hex code~~ normalized code

As can be seen only the ~~$~~ 13 nibble was changed but the displaying of the NNN in the x-register was greatly altered. (see displaying NNN).

The effect RCL then is first determined by the sign digit (nibble 13). If Ø or 9 the data is normalize'd to a number and all digits containing values other than Ø-9 are normalized. If the sign digit has any other value only the sign digit is change (made a 1) and the data treated as alpha. The only exception is if the sign digit is an A. The results is a normalized positive number.

VIEW

View behaves the same as RCL. So care should also be taken with it.

X <> __

The ~~value~~ data in the x registers is copied into the designated register without normalization. The ~~number~~ data in the register is treated as if a RCL was performed on it.

ASTO

only six of seven bytes can be saved.

A 1Ø is the 6th byte the others (5-Ø) are copies of the first six bytes of the NNN.

		28	ØC	29	Ø1	3F		NNN
25	64	~~2A~~	~~ØØ~~	~~A5~~	~~Ø1~~	~~78~~		~~hex~~ code
7o	d	(	μ	)	夭	?	■	Alpha before ASTO ØØ
1Ø	25	64	28	ØC	29	Ø1		~~Hex code after ASTO ØØ~~ ~~Alpha after ASTO ØØ~~ Normilized Code

The last byte of information is lost but the other 6 bytes are unchanged. No data is changed only one byte is lost!

ARCL

ARCL behaves simiarly to RCL except in Alpha mode. The 13th nibble is changed to 1 and all other nibbles left unchanged. For example:

25 64 28 ØC 29 Ø1 3F	Hex code
-5.64 28 122 -77	x-reg STO ØØ
d (μ) 夭 ? ●	Alpha display after ARCL ØØ
15 64 28 ØC 29 Ø1 3F	Normilized code

Only the 13th nibble is changed and made a 1. AVIEW behaves the same way.

NNN in The Display

NNN don't always display as expected. If you were following the examples above, you might have noticed that the way some of the NNN displayed was ~~a~~ rather unquie. The displaying of NNN in both the ~~X reg~~ stack registers and Alpha display will be discussed next.

Stack register display.

Again it is the sign digit that partly effects the way an NNN displays. If sign digit is a 1 then the NNN will display as Alpha data. If the sign digit is Ø then a positive number results. Another value will result in the displaying of a negative number.

Another problem is the display mode. Displaying the same NNN in FIX 9, SCI, ENG, 9 ~~to~~ will cause different displays.

Ø5 6A 9 6 4321 6C ØØ	
0.5709643 22 "	FIX 9
5.7096432 -ØØ	SCI 9
57096432. -Ø3	ENG 9

Notice The effect of placincing an abnormal value in the exponent sign has on the way the NNN is displayed and the characters used to display it.

Hex Code	Exponent Sign	Exponent digit
A	blank	▦
B	!	$>
C	"	∠
D	#	=
E	$	≥
F	%	7

The above values in the exponent sign will display as the character shown in the second column for any fix mode. IF the A~~E~~-F appear ~~as the~~ in either nibble of the exponent they display as shown in the third column. If there happens to be a A-F in both the exponent sign and 1st nibble they will combine to give the next higher character ~~the~~ and the remainder of the decimal value in the 1st nibble. It is not easy to decode these displays and their seeming unconcern for the normal numbers.

If the sign digit happens to be a 1 and the NNN displays as alpha data the effect isn't so bad.

Only the last six bytes will display as ~~their~~ ASCII equivalents. If the ~~...~~ display can't display the character it appears as a box star in the display.

1Ø 28 29 28 29 28 29		NNN code
( ) ( ) ( )		~~display in X reg~~. Stack reg
☒ ( ) ( ) ( )		alpha display

The only major problem is the displaying of Null byte's (ØØ). These are suppressed from the display.

1Ø 28 ØØ ØØ 29 28 29		NNN code
( ) ( )		~~X reg~~ Stack reg
(_ _) ( )		alpha display.

This can lead to the misinterpretation of NNN code.

ALPHA REGISTER DISPLAY

When a NNN is displayed in ALPHA mode it is displayed as a seven character string. If an approriate display character can not be displayed the box star is shown. ~~The~~ The main problem here is with leading null bytes they aren't displayed, even though other null bytes in the string are displayed as the over bar. If the NNN can be placed into the Alpha display the xFunction ATOX can be used to get the decimal equivalents of its characters.

Note: The time differences in execution time in the recall of status VS Numbered registers is primarily due to the normalizing of the data when recalled from numbered registers. The status registers are considered "pure and correct" and are not normalized and execute in less time

NON – NORMALIZED NUMBERS

	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Number	SIGN 0/9	MANTISSA 0–9										SIGN 0/9	EXPONENT 0/9	
Alpha	1	0					ASCII							

Explain the

One of the greatest advances in computer Sci was the realization that both data and programs could be stored in the same memory.

KeyAssign	F	∅					
Program	Fill from left to right with instructions			⟶			
Time	A	A	# of registers in Alarm buffer	∅ ⟶			
I/O	B	B	# of registers in I/O Buffer	Pointer	∅ ⟶	∅ or 1	

Figure 5.9

```
                05 6A 96 43 21 6C 00    HEX
                ~~5.7096432-00~~         X-register   0.5709643ZZ "
will become:    ~~Fix 0.57~~ "                         5.709632-00

                05 70 96 43 21 79 00    After Storage and recall
```

if the sign digit contains any other value it is set to 1 and the number is treated as alpha data. For example:

```
                25 64 CA 00 A5 01 78    Hex
                -5.6530010 -22          X-register
                   d ▩  ▩ 大 ▩          alpha
                15 64 CA 00A5 01 78     After normalization by ~~ASTO~~ STO/RCL
```

The affect of ASTO and ARCL is to add a 10 byte to the front. This marks the data as alpha data from then on. It however minimizes the changes made to the NNN. Leading nulls are not displayed and are dropped.

```
                25 64 CA 00 A5 01 78    HEX
                 %  d ▩  ‾ ▩ 大         ALPHA Display
                ~~00 25 64 CA A5 01 78~~
                ~~10 64 CA~~
                10 2564 CA ØØ A5 Ø1     After ASTO/ARCL
```

NNN's IN THE DISPLAY

The displaying of an NNN is rather difficult to explain. They don't always display the same. Lets look first at an NNN in the X register and then an NNN in the Alpha display.

X REGISTER

If the sign digit contains a 1 then the system trys to display the NNN as alpha data in the X register. If the sign digit is 0 then the NNN is displayed as a positive number. If the sign digit is anything else then it is displayed as a negative number. For example the NNN

10 00 01 00 01 01 00

will display as an Alpha string of three little men.

大大大

Notice that all of the nulls were suppress in the display.

The number of digits displayed and the display mode will also affect the way the NNN displays. If the NNN has enough digits in the display the digits greater then 9 will carry a one on to the next digit. If there aren't enough digits to do this then they will display as follows:

A	:
B	;
C	<
D	=
E	>
F	?

The best display mode to use when dealing with NNN's is SCI 9.

ALPHA DISPLAY

When an NNN is in the Alpha display the biggest problem is the displaying of null bytes. Any leading nulls are not displayed. Any nulls in the middle of the NNN or at the end will be displayed as an overline. If a character can't be displayed then it is displayed as the starburst pattern. The Extended Functions module's ATOX function can be used to decode these.

HP-41C/CV Status Registers

6 5 4 3 2 1 0

Register			Address	
Shifted Keys	e		00F	same color as ⊢
Flags	d		00E	All one color.
Memory	c		00D	All one
Subroutine	b		00C	color
Subroutine	a		00B	
Unshift Keys	⊢	⊤ [1]	00A	same color as e
Scratch Q		_	009	
Alpha P		↑	008	make
Alpha O		]	007	another
Alpha N		\	006	color
Alpha M		[	005	
Stack L			004	
Stack X			003	make one
Stack Y			002	color
Stack Z			001	
Stack T			000	

As displayed / As printed

1. These symbols are typed as an upper case "H" and "T" respectively and then removing the right half and bottom using "liquid paper" or equivalent.

Figure 5.4

Alpha Registers

28	27 SCRATCH	26	25	24	23	22	P (008)
21	20	19	18	17	16	15	O (007)
14	13	12	11	10	9	8	N (006)
7	6	5	4	3	2	1	M (005)
6	5	4	3	2	1	0	

BYTE NUMBER

Red #'s denote the numbers corresponding to page 6 top.

Figure 5.5

each associated section of these registers. The first group we encounter is the Stack Registers.

STACK REGISTERS

We won't discuss the stack in detail here because it is covered in great detail in the owners manual and should already be fairly famaliar to you. It has the addresses 000 to 004, corresponding to the L,X,Y,Z,T respectively.

ALPHA REGISTERS

The next group of registers are where the ALPHA display is contained. They are the M, N, O, P registers with the address from 005 to 008 respectively. You may notice that four registers have 28 bytes in them, which is 4 more then the 24 bytes needed for holding the 24 character ALPHA display. Why are only 24 of the 28 available bytes used for the display? Most of the functions that operate on the display work with gorups of 6 characters at a time. ASTO/ARCL handle only six characters because the 7th byte is used to code the data as alpha. In this way it makes sense then that the display be an integral number of six in length. But do we actually have 28 bytes availble for the display? Figure 5.5 shows the display registers in greater detail. Lets look and see how characters are entered into the display and what happens to the characters that are already in the display when a new character is entered.

Figure 5.5 Alpha Display

1. The new characters enter the display in the Ø byte of the M register.

2. The other characters are shifted one byte to the left to make room for the new character.

3. If a character is shifted out of byte 6 of a register it is shifted back into byte Ø of the register above it.

4. If the display is full when a character is entered a tone will sound to remind you that you will lose the first character entered and then the 1st character entered is shifted out of the display, and "lost", as the new character is entere.

One important thing to note here is that the display registers are treated as one long 28 byte register with the 6th byte of P the most significant and the Øth byte of M the least significant. In step four the shifting of the 1st character out of the display and into position 25 causes it to be lost as far as the system is concerned. But it is still there and will not be lost until it is shifted out of the P register. So we really do have 28 characters that can be accessed synthetically. There is however one catch. The system use the last four bytes of P as system scratch. This is most evident during data entry, data display, or status card writing. So anything stored in P synthetically can should be considered volatile.

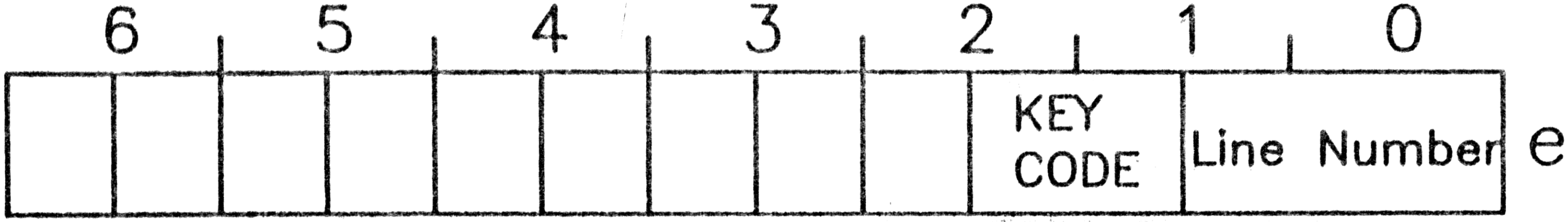
This stipulation however doesn't apply to the other three ALPHA registers M, N, O. These may be used synthetically for many things.

Q REGISTER

The next register is the Q register at address 009. This register is used so frequently by the CPU that it is not very useful as another data register like the ALPHA registers. Most of the time Q contains the characters of the last function executed by name in reverse order. We will see a use for this when we talk about the Q Loader. Another useful feature of the Q register is that when an XROM function is assigned to a key the Q register contains the XROM number in its 2nd and 3rd bytes and the microcode entry point address in its Øth and 1st bytes. We can make some use of the Q register synthetically, although not to the extent we will with the ALPHA registers.

Key Assignment Registers

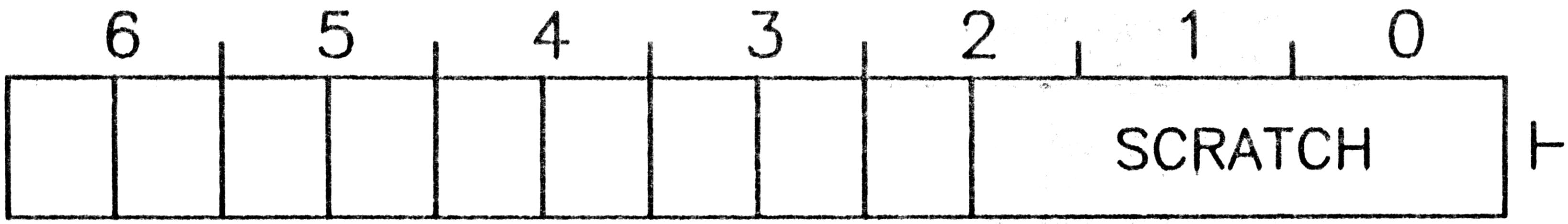
SHIFTED



15	54	14	53	13	52	12	51	11	
25	64	24	63	23	62	22	61	21	
35	74	34	73	33	72	32	71	$\boxed{31}$	shift key shifted.
45	84	44	83	43	82		81	41	

other half of enter key

UNSHIFTED



15	54	14	53	13	52	12	51	11	
25	64	24	63	23	62	22	61	21	
35	74	34	73	33	72	32	71	$\boxed{31}$	shift key
45	84	44	83	43	82		81	41	

other half of enter key.

Figure 5.6

Each row should be color coded to help display it.

KEY ASSIGNMENT FLAG REGISTERS

The next register up is the ⊢ register, 00A. This register and the e register go together to make the Key Assignment Flag registers. They both map their first 36 bits to the keys on the front of the calculator. When a key is pressed in USER mode the system first checks the corresponding bit in these registers (⊢ for unshifted and e for shifted), if the bit is set then it looks for the function assigned to that key, if the bit is clear the the face function is executed. The key assignment flag registers are shown in figure 5.6.

Figure 5.6 - Key Assignment Flag Registers

With only 36 bits used out of each register that leaves 5 nibbles in each to account for. In the ⊢ register they are used as system scratch. In the e register the first three nibbles of e are the line number in the current program. We will see an interesting application of this later on. The next two nibbles (3 and 4) are the key code of the last key pressed.

Of general interest is that in the key assignment flags there are bits corresponding to the SHIFT key and the hidden key under the ENTER↑ button. This means that synthetically we may make key assignments to these keys, although the shift key would then be of little use in USER mode and there is no contact for the hidden key, unless we put one in.

Configuration of Memory

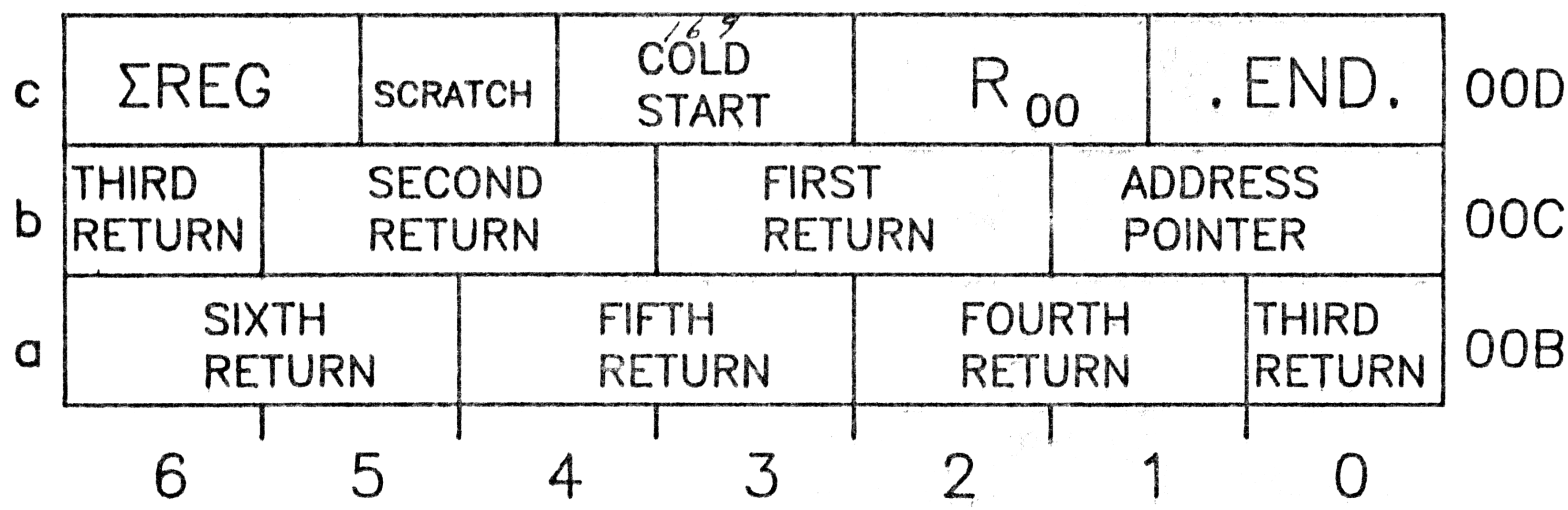


Figure 5.7

MEMORY MAP REGISTERS

The next three registers contain a lot of very interesting and useful information. They are also very dangerous to use. The wrong code stored in here could do anything from locking up your keyboard to "MEMORY LOST". The a and b registers contain the address pointer and the six level subroutine return stack. The c register contains four pointers to different parts of RAM. Figure 5.7 shows how the registers are divided.

Figure 5.7 - Configuration of Memory

Lets look first at a and b registers. The first two bytes (0,1) of b contain the Address Pointer described in the last section. By placing the proper code into this pointer we may wonder through all of RAM and ROM. The rest of b and all of a contains the subroutine return stack. The return addresses are not always copies of the Address Pointer at the time the subroutine was called. If the return is to ROM then the return address is a copy of the Address Pointer. If the return is to RAM then the return address is a coded version of the Address Pointer. Since the byte pointer in RAM never is greater then 6 it uses only three bits of the last nibble of the Address Pointer. Likewise the register pointer never is greater the 3 in the last nibble and so doesn't use the last three bits of the third nibble. So for RAM returns the Address Pointer is compressed by placing the three bits used by the byte pointer into the last three bits of the register pointer. This is illistrated by the following. Suppose a subroutine is called when the Address Pointer is at 3160. The Address Pointer is compressed to 0C60 and added to the return stack.

The c register (system register) contains a lot of useful but dangerous information. A wrong number in this register can cause a lot of fustration. The first three

Flag Register

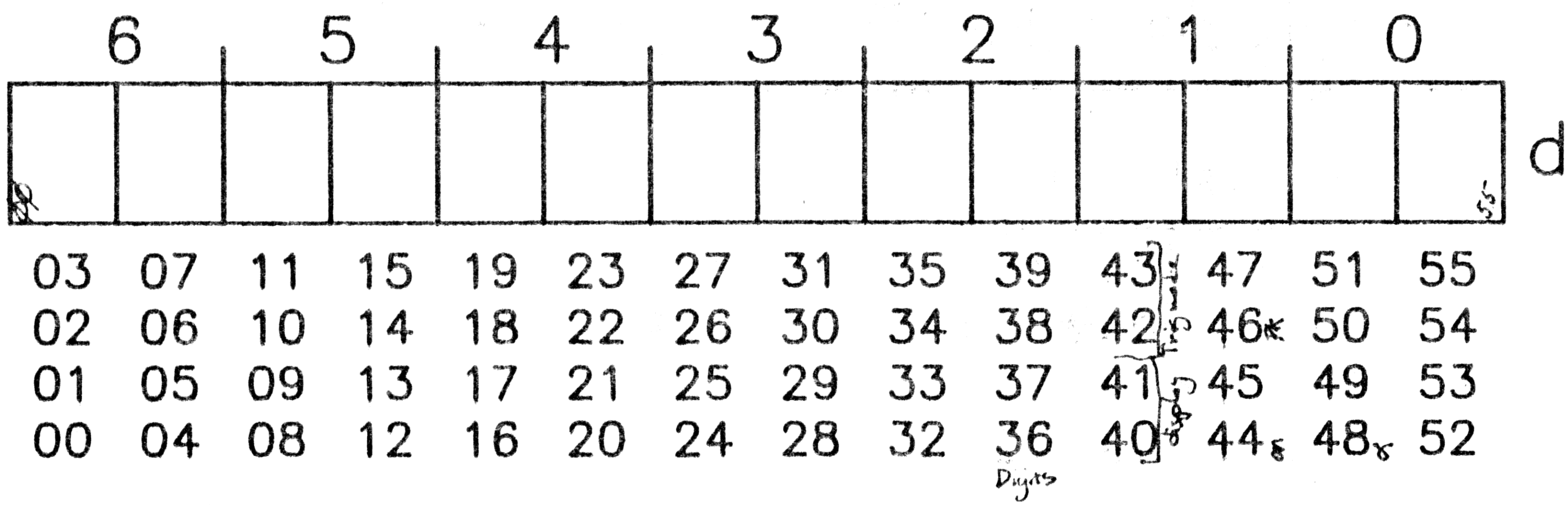


Figure 5.8

nibbles are a register pointer that contains the present location of the permenent .END.. This number is refered to e erytime a global label search or a CAT 1 is executed. The next three nibbles contain another register pointer that contains the present location of the curtain between the data registers and the program registers. Manipulation of this number s ntheticlly can be of use when we want to hide data registers. The next three nibbles contain a "COLD START" constant (169). If that number ever changes the unrelenting "MEMORY LOST" occurs. Nibbles 10 and 11 are used by the printer as scratch. The last three nibbles contain a pointer to the first of the summation registers. We can sometimes use this information to move the curtain up or down by this amount as a fast data hiding technique. As can be seen alot can be done with these three registers, but great care should be used when working with them syntheticly.

FLAG REGISTER

This is the d register (00E). Since a register contains 56 bits it shouldn't be to surprising to find that all 56 flags are contained in one of the status registers. Figure 5.8 shows the correspondance between the register and the flags.

Figure 5.8 - Flag Register

By storing the proper code into this register we can set or clear blocks or single flags all at the same time. We can use STO d to set up our display, angular mode and number of digits for a program all at the same time. There are also some interesting things you can do such as setting flag 30 the CAT flag and list the entire 4096 entries. Also Bug 10 can be used to increment the flags as a loop counter.

SYNTHETIC FUNCTIONS

Part of the definition of a synthetic function was that it was not keyable by normal means. If this is true, then how do we get to them? The origins of synthetics came from the bugs in the microcode that allowed users access to registers and functions that they were not supposed to be able to get at. But as stated before, these bugs were exterminated by HP in due course. Fortunately for us, the first people to use synthetics developed programs and other means to generate synthetics on non-bugged machines. Let us first make a general list of the most common synthetic functions and then a list of how to get at them.

The number of synthetic functions is almost infinite, but the most commonly used ones are shown in the table below:

Function	Description
STO	Store data to any of the status registers. Does not cause normalization.
RCL	Recall data from the status registers. Does not normalize data.
X<>	X exchange with any of the status registers. Does not normalize data.
DSE	Decrement and skip if equal in any of the status registers (usually M, N, and O). Does not normalize data.
ISG	Increment and skip if greater, in any of the status registers (usually M, N, and O). Does not normalize data.
Text	Lines of text that contain characters not available on the keyboard. Save bytes in creating special characters.
Keys	Synthetic key assignments of all functions both synthetic and normal.
GTO, XEQ	Precompiled GTO's and XEQ's that are created synthetically.
Short form exponential	Short form of the EEX does not have the 1 in front of it (E5 instead of 1 E5).

The application of these functions makes for a very wide variety of synthetic programs. Now let us explore how to get these functions into our programs.

There are several ways to load synthetics into the 41C: the card read, wand, Bug 2, PPC ROM, Bug 9, pulling modules, and a few others. We will talk about the card reader, wand, PPC ROM and Bug 9. Let us start with Bug 9 and see how it works and then synthesize our first synthetic function.

BUG 9

When register e was discussed, we saw that the first three nibbles contained the current line number in the PRGM. When a CAT 1 is executed this number is set to 000. Suppose we do a CAT 1 in PRGM mode and stop immediately. The line number is set to 000 because the system does not know what line to expect. Now suppose we delete one line with DEL 001. If you do, you should see a very large line number around 4093. But we did not have any programs with that many line numbers in the calculator! What has happened is that we have deleted and jumped into the top of the key assignments or status registers if no keys are assigned. Well, of what use is this to us? Let us do the following:

these are displayed on the video interface as a ~ because that is the normal ASCII character for that code

Instruction	Display
1. ASN OFF Σ+	ASN OFF 11
2. ASN OFF Σ-	ASN OFF -11
3. GTO ..	PACKING
4. PRGM	00 REG ___
5. CAT 1/STOP	LBL ___ or .END. REG ___
6. DEL 001	DEL 001
7. SST (2)	409 ~~4093~~ ᵀ
	02 ~~4044~~ LBL 03
	03 ~~4045~~ OFF
8. Backarrow (2)	01 ᵀ
9. ALPHA A ALPHA	02 ᵀA
10. PRGM	run mode
11. GTO ..	PACKING 20 reg AS file

Example #1

What have we just accomplished? We have just created in the key assigment registers a synthetic function known as a "Byte Jumper". This function in turn can now be used to create other synthetic functions. We will discuss this function more in the next section. This technique of assigning any function to a key and then using Bug 9 to go in and "edit" the key assignment is one way to generate synthetic key assignments. The most important thing is that all HP-41C/CV's have this Bug. You do not need any peripherals to do it. But it is limited in what it can do.

CARD READER

We have seen that Bug 9 can be used to create synthetic key assigments, but how do we get synthetic code into our programs? One way to do it is to use the card reader. When the calculator writes a program to cards it copies the program bytes onto the card sequentially and so if a multibyte instruction just happens to be at the break between tracks of a card then it will be split between the tracks. We can take advantage of this as follows. Write our program so that the line where our synthetic code will be is at the junction between two tracks. If necessary, filler bytes (ENTER↑ works well) can be temporarily inserted to put the instruction on a junction. First enter the prefix byte (STO __) then record the cards. The STO byte and the byte designating the __ will be split between the tracks. Now go back to the program and replace the STO with ENTER↑, and the __ with the proper postfix bytes and record and second set of cards. Now to get the synthetic code, read in the first track from the first recording and the second track from the second recording. You should now see the synthetic STO function as a program line. Clean up the temporary filler bytes and you have your program. This technique is fairly versatile and can create most of the synthetic program codes by using the appropriate prefix and postfix bytes. It is fairly long and the card reader will drain your batteries, but it can be done when all else fails.

WAND

Another fast way to enter synthetic code into programs is with the wand. Bar code can be made for most of the synthetic functions and characters, then by simply scanning the bar code if is entered into the program. The only really big hitch is that different wand revisions will not read the synthetics the same. In some, the null bytes cannot be read without locking up the keyboard or losing memory. This is most unfortunate. But it will get most of the synthetics into the calculator.

PPC ROM

By far the most versatile and easiest to use are the routines in the PPC ROM. These routines were writen by the people who started synthetics on the 41 and were tested thoroughly before being accepted for the ROM as a program. The ones that are most used are the Load Bytes and Make Key assignment programs. These two will allow you to create any arbitrary set of bytes in program or assign any function to a key (even multibyte functions). Associated programs create the NNN's that can be used to set flags, create special characters or whatever.

The "LB" routine allows a user to place synthetic code into his or her program automatically. The user places a LBL "++" followed by at least 7 bytes of +'s and an XROM "LB". Then with the XROM "LB" as the next executable step, presses R/S. The program will prompt you with "xx of nn?" in the display. You then have only to enter your desired bytes in the correct order and you will have created your synthetic code. This is by far the easiest and most versatile way for creating program code. You can enter any combination of bytes at any location in your program. There are also supporting routines that help create NNN's for the purpose of storing them in the status registers while a program is running, or to create special characters for the printer without using BLDSPEC.

Lets create a short program to swap the M, N, and O registers around in the Alpha display. We will use "LB" to create the synthetic functions required.

28 register ASFL

1.	PRGM	01 LBL "SWAP"	
		02 LBL "++"	
		03-17 +	
		18 XROM "LB"	
2.	PRGM		
3.	R/S		
4.	DEC/HEX INPUT 56		
	#1 OF 14?	ALPHA 90 R/S 51	
	#2 OF 14?	75 R/S 60	RCL M
	#3 OF 14?	CE R/S 69	x<> O
	#4 OF 14?	[illegible] R/S	
	#5 OF 14?	91 R/S	STO M
	#6 OF 14?	75 R/S	
	#7 OF 14?	R/S 102	
5.	SST		
6.	PRGM 5	DEL 05	
7.	~~Backarrow all + and LBL "++", and XROM "LB"~~		
8.	GTO .. 134	Packing	

9. ALPHA ABC - VWX ALPHA
10. XEQ "SWAP"

EXAMPLE #2

The other routine, "MK", is very useful for creating key assignments of the sythetic functions. These key assignments can also be of ROM functions or multibyte functions. When this routine is executed, the program tells you how many registers there are free for assignments, it then prompts you for the prefix, postfix and key code. When these are entered, the appropriate bit in register e or ⊢ registers are set and the function is assigned to the key. This is done without disturbing previous key assigments, and if a key is taken the program prompts for a new key. There are also routines to pack the key assignment registers to recover any partially full assignment registers for system use.

1. XEQ "MK"
2. PRE/POST/KEY 159 ENTER↑
124 ENTER↑
23 R/S
3. PRE/POST/KEY R/S
4. SIN

13 ASFL

Towe b

USER Insert Here

EXAMPLE #3

BYTE JUMPING/GRABBING

One of the most useful functions sythetics has given us is the byte jumper. This function is the result of a synthetic key assignment of any byte from row F with any other byte from the hex table. The function has some interesting and useful properties that we can use to our advantage.

The 41, when presented with a code, must try to execute that code as best it can. The byte jumper has the code "F_ _". The bytes from the F row of the hex table are the TEXT line designators. The F tells the system that text is following and the next nibble is how many characters (0-9 and A-F). The second byte of the instruction (postfix) is not important at this time. When this function is executed in the run mode, it sees the F meaning text and then looks at the last nibble of the byte preceding the address pointer to determine how many characters follow. Figures 5.10a and 5.10b show how a byte jumper works. Let us try a short example to see what happens when we execute the byte jumper.

ENTER

1. Type in program	01 LBL "BJ"
	02 STO 02
	03 "ABC"
	04 ENTER↑
	05 ENTER↑
	06 END
2. GTO .003	ABC
3. PACK	PACKING
4. PRGM	0.00
5. Byte jumper	XROM 05,01
6. PRGM	03 *
7. SST	04 /
8. SST	05 ENTER↑

Example #4   11 regs ASFL

What happened to the text line that we were displaying when the byte jumper was executed? Let us look at the code for the * and / functions. We see that the code for the * is 41 hex and for the / is 42 hex. The byte jumper used the last nibble of the STO 02 function to jump two bytes into the middle of the text ABC and then displayed the B and C as their stand alone functions * and /, respectively. The position of the address pointer can be followed in Figures 5.10a (before jump) and 5.10b (after jump). Okay, now let us make use of the byte jumper to create an actual function.

Suppose that you wanted to do X<>M as line 3 in your program. Let us follow through the next example and see how to generate it.

1. GTO .003	ABC	22 reg ASFL
2. PACK	PACKING	
3. PRGM	0.00	
4. Byte Jump	XROM 05,01	
5. PRGM	03 *	
6. XEQ X<> 22	04 X<> 22	
7. GTO .003	03 AB■	
8. RDN	04 RDN	
9. GTO .003	03 AB■	
10. PACK	PACKING	
11. PRGM	0.00	
12. Byte Jump	XROM 05,01	
13. PRGM	03 *	
14. STO 00	04 STO 00	
15. GTO .003	03 AB0	
16. SST	04 X<> M	

Example #5

Now let us see what we have done. The first 5 lines simply repeat what we did earlier. They place the program pointer into the middle of the text line at the B. The 6th line places a X<> 22 function into the program between what was the B and the C of the text line. The next step (7) reestablishes the text line, but the C is replaced with the code for X<> __. The 22 and the C are left as their stand alone functions of 6 and /. Step (8) places a RDN instruction immediately after the text line. The steps (9) through (13) repeat the byte jumping procedure. Step (14) places a dummy byte between the B and the X<> and when the GTO .003 is done, the dummy byte is incorporated into the text line and the X<> is forced out. When the X<> is forced out it is a prefix byte and needs a postfix byte to make it complete and so it picks up the RDN byte and we get the resulting X<> M. This is shown in Figure 5.10a-e.

This technique can be used to create almost any synthetic code desired in a program. The two program lines STO 02 and the text line are called the control and generator lines, respectively. With the proper combination of these two lines you can create most of the functions. The byte jumper can jump at most 15 characters. This is because the nibble controling the jump distance can have values from 0 to 15 (F). Let us now focus our attention on another aspect of the byte jumper that at first is not obvious.

The byte jumper is a text instruction and when it is executed it trys to do just that...put a text string into the alpha display. When the byte jumper jumps the two bytes into the middle of the text line it copies what is jumped into the display. This can be very useful. We can now use the byte jumper to decode program lines that contain more than one byte. For instance:

Example of Byte Jumper

Our six line program in memory

start here

Figure 5.10a

F3	00	B 42	J 4A	STO Ø2 32	Text+3 F3	A 41
B 42	C 43	ENTER 83	ENTER 83	C0	END 00	90

After GTO 003 our address pointer is pointing to this position and is displaying "ABC"

After BJ

Figure 5.10b

F3	00	42	4A	32	F3	41
* 42	/ 43	ENTER 83	ENTER 83	C0	END 00	90

Now after Byte jumping the address pointer is in this position and displaying 03 * 42

Example of Byte Jumper

Figure 5.10c

F3	00	42	4A	32	F3	41
* 42 B	X<> CE	Ø Ø 00 whatever	/ 43 C	Enter 83	Enter 83	END C0
00	90					

Now GTO . 003 and the text line shows "AB$\boxed{*}$." Now add the line 04 RDN pack and GTO 003

Figure 5.10d

F3	00	42	4A	32	F3	41
42	X<> CE	RDN 75	43	83	83	C0
00	90					

F3	00	42	4A	32	Text 3 F3	A 41
B 42	STO ØØ 30	X<>M CE	75	/ 43	ENTER↑ 83	ENTER↑ 83
END C0	00	90				

LBL "BJ"

STO 02

ᵀAB0

X<>M   Synthetic Code

/

ENTER ↑

ENTER ↑

END

Figure 5.10E

1.	GTO .003	ABO
2.	PACK	PACKING
3.	Byte Jump B5,23	XROM 05,01
4.	ALPHA ALPHA	A
5.	ATOX	243
6.	ATOX	65
7.	Byte Jump	XROM 05,01
8.	ALPHA ALPHA	B
9.	ATOX	66
10.	Byte Jump	XROM 05,01
11.	ATOX	48
12.	ATOX	206
13.	Byte Jump	XROM 05,01
14.	ATOX	117
15.	ATOX	67
16.	ATOX	192
17.	ATOX	2
18.	ATOX	9

Example #6

What we have just done is to decode the entire program from line 03 until the END. The 192/2/9 is the END statement for our program. This technique can be used to decode GTO's and XEQ's after they have been compiled. It can also be used to a limited extent to explore the ROM's. The byte jumper is a very useful tool and practice is suggested so that you can use it.

BYTE GRABBER

It is not necessary that the byte jumper be executed in RUN mode only. When the byte jumper is executed in PRGM mode, it called the "Byte Grabber". This is because of the way it behaves. Suppose we make the following assignment: 247,65 to a key. This is a text seven and the - postfix. What would happen if this was pressed in PRGM mode? Let us follow through the next example and see.

Suppose we have the following program:

```
01 LBL "AA"
02 ENTER
03 STO 22
04 ENTER
05 END
```

Now let us do the following and see what we get for our effort.

1. GTO .00h
2. Byte Grab
3. RDN
4. BST (2)
5. Byte Grab
6. SST and see STO M

EXAMPLE #7

We now have instead of a STO 22, a STO M instruction. Where did it come from. Lets see what we did a step at a time. The first step of the example positioned the address pointer just before the STO 22. We then pressed the key with the byte grabber assigned to it. We saw a text line with a null, a character, 4 nulls, and a box star. The code for this is in HEX is:

F700E10000000092

The F7 byte is the text seven indicator followed by the text string. The last byte of text string (92) is the STO prefix that was byte GRABBED when the byte grabber was executed. This is called prefix masking. The next step was to insert the RDN function after the text line. This RDN will become the new postfix for the STO prefix. We then went back to just before the text line and pressed the byte grabber again. This had the effect of masking the first F7 byte that was executed and thus released that text string to its stand alone functions. Note that most of these were nulls and therefore don't appear in the program as lines. But most importantly the last byte (the STO prefix) is released and looks for a postfix to go with it. What it finds is the RDN byte waiting for it and the result is the STO M function.

Key Assignment Register

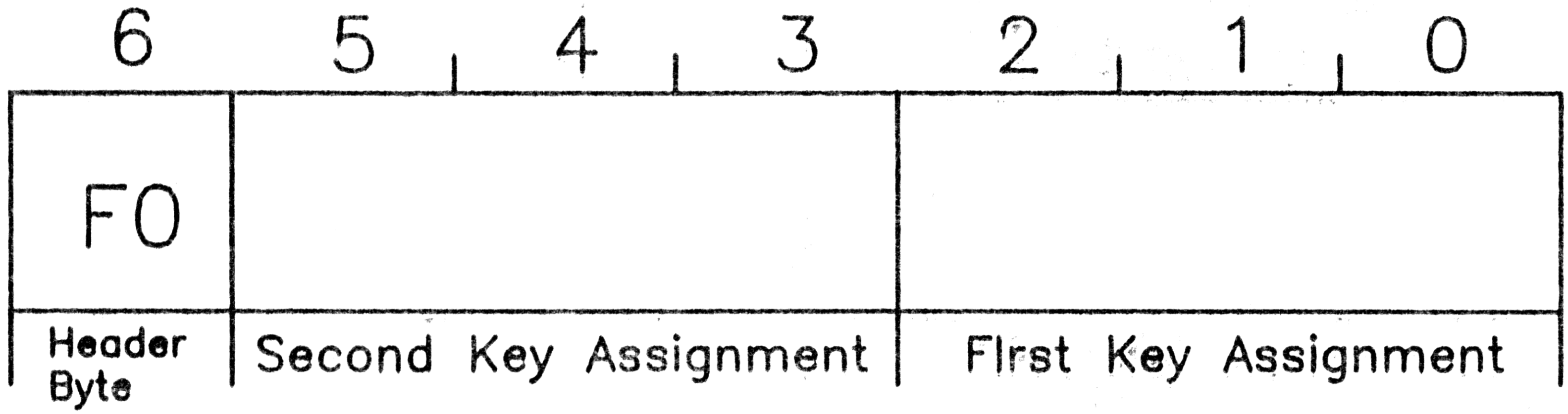


Figure 5–11

Lets try the following example and see what the result is. It will prove very interesting. Follow very closly.

1. GTO .005
2. Byte Grab
3. !!!!!!!!!!!!!!!!!!!!

EXAMPLE #8

SYNTHETIC KEY ASSIGNMENTS

The ability to assign synthetic functions to keys is a real time saver and convienance. To understand how the key assignments work lets take a closer look at what the 41 does when a key assignment is made. Then lets see how we can modify the key assignments and make use of them syntheticly.

The 41 does two things when a function is assign to a key. First it sets a flag in either the ⊢ register or e register depending on whether the key is unshifted or shifted. It then stores a three byte code into the next availble postion in the Key Assignment registers. Lets look closer at these key assignment registers and what their structure is. Figure 5.11 shows what the fields are in a

Figure 5.11 — Key Assignment Register

key assignment register. The 6th byte is the header byte and is always F0. Each key assignment register has this code as its most significant byte. Bytes 0-2 correspond to the first key assignment placed in that register. The bytes 3-5 correspond to the second key assignment made in that register.

The individual key assignments are a three byte code that define the function and the key it is assigned to. Figure 5.12 shows the structure of a three byte key assignment code.

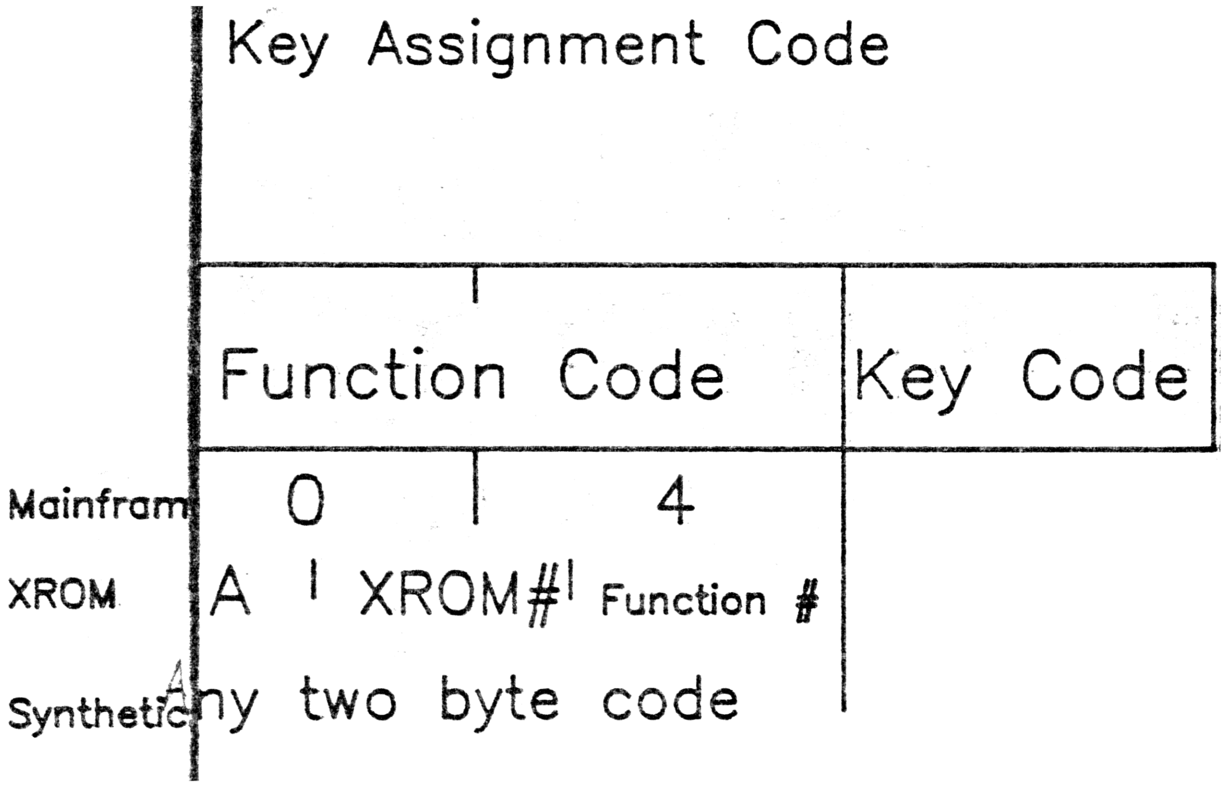

Key Assignment Code
Function Code
Key Code
Mainfram
0
4
XROM
A
XROM#
Function #
Synthetic
Any two byte code

Figure 5.12

Figure 5.12 - Three Byte Key Assignment

Lets examine the key code first. The key code is a coded version of what you get by taking the row and column that the key is in. Suppose that we want to ASN LOG to the LOG key. The LOG key is in the first row and forth column (writen 14). This is writen in general as row A column B (AB unshifted and -AB shifted). The code that is stored in the key assignment is this row, column code changed as follows:

Unshifted = (B-1)A hex
Shifted = (B-1)(A+8) hex

This gives us a two digit hexidecimal number that is stored in the least significant byte of the key assignment.

Now lets look at the coding of the function bytes. There are two bytes availible to store the functions hex code in. Most main frame functions use only the first of these. The other (more significant byte is filled with a 04 byte (any byte from row zero will work). So the assignment of the LOG function to the LOG key would appear as:

04 56 31

The 04 byte corresponds to the filler byte used, the 56 is the Hex code for the LOG function and the 31 is the key code for the LOG key. If only one of the two bytes are used for main frame function, then why are there two bytes availble for the function code. One of the main advantages of the 41 is its abilty to address XROM's, and these XROM functions can all be assigned to keys. These functions do take two bytes when assigned to a key. These two bytes are its XROM number as discribed in section II. Lets take the time to work an example and see if there are any questions about key assignments to this point.

Lets suppose we want to make the following two assignments in the order shown.

Function:	Key:
ISG	Shifted CHS
DSE	CHS

Write out the 7 byte HEX code for these two key assignments appearing in the same register.

Now to further test our skill lets decode an assignment register and see what the assignment register contains.

F7 04 1D 11 04 7B 42

Write the assignments in order that they were made below.

Now that we are fluent in key assignments lets go on and see what they can do for us synthetically. What happens if we synthetically place a code other then 04 or row A in the most significant byte of the assignment. In general we get what are called psuedo XROM's. The normal XROM numbers are 0 to 32, but PXROM's can have numbers from 0 to 64. This is how we get key assignments for multibyte instructions like STO M. You have already seen one example of this in the byte jumper that we created with bug 9. It is the assignment of a F1 and a 41 bytes to a single key. The synthetic function STO M is assigned as 91 75. When pressed in run mode you see a breif display of a PXROM number. Now lets try writing a key assignment for the following two assignments.

ISG M to ISG key (-42)
DSE M to CHS key (42)

Write the assignment for the two functions as the HEX code in the next line.

With the use of NNN's we can store the approiate code in to the block of key assignment registers starting at 0C0. This will give any key assignment we wish to make. The programs like "MK" do just that for us.

Q LOADER

Let us revisit the Q register and see if it can't be of more use to us. This register contains the label of the last executed (XEQ) function or global label spelled in reverse. It was found that when a byte from row one of the table was assigned to a key synthetically, the results were unexpected. When executed in run mode the proper digit was placed in the X register. But in PRGM mode the results were even more interesting. The digit was entered but a text line was also inserted. It was soon found that what ever was in the Q register at the time was loaded into the text line but in correct order. We can make use of this to create synthetic text lines of special characters of up to seven bytes in length. Lets create the following text line in a program. The characters are "( )". As you can see these are normal characters. What we do is to create a NNN that is the code for the text line in reverse order. We then go to the place in our program where we wish to insert the line and go to RUN mode. We store the NNN in Q (STO Q). Then back into PRGM and press the assigned digit key (Q Loader). We see the number as the first line and the text string as the second line.

This is a very useful way to get synthetic text strings into our program. Unfortunately they are of limited length.

GENERAL TECHNIQUES

In this section we will review some general uses of synthetics that we feel are important and regularly used. The techniques discussed will be alpha manipulation, curtain raising, flag control, synthetic tones, byte saving, branching, speed up, and the use of NNN's in synthetics. Let us start our discussion with the use of NNN's.

When we wish to modify the status registers or place a certain code in memory some place, we need to use NNN's to do it. This means that we must be able to easily code and decode NNN's. There are several programs or versions of the same program available to do this. The PPC ROM routine "HN" (Hex to NNN) takes a seven byte NNN in the alpha display (entered as normal 0-9 and A-F) and codes it into a NNN. "NH" (NNN to Hex) does just the opposite. The programs "CODE" and "DECODE" also do this. The use of the programs will only be demonstrated here. More thorough explainations of these routines is in Synthetic Programming Made Easy. Let us see how the two routines, "HN" and "NH" work and then move on to use NNN's in synthetics. Suppose we want to take the following hex code and turn it into a NNN.

C0 FF 00 00 FE A9 FE

We would do the following:

1. ALPHA C0FF0000FFA9FF ALPHA
2. XEQ "HN"
3. ~~-1~~. 6500016-11

To decode the NNN, place it in the X register and do the following:

1. NNN in X register
2. XEQ "NH"
3. ALPHA C0FF0000FFA9FF

These NNN's can now be used for what ever we want them for. As mentioned earlier register operations among the status registers don't normalize the NNN. We can therefore use the NNN to change the status of the machine by direct storage of it into the proper status register. If we want to have more then one NNN around we can save them in the Alpha display until they are needed and then recall them from there. As we proceed through the rest of this section we will see several uses of NNN's in the status registers.

ALPHA REGISTERS AND SYNTHETICS:

The alpha registers serve many purposes in synthetics. They're used as an extension to the stack, to display special characters, temporary storage of NNN's. Lets look first at using them as data registers.

All of the register operations can be used in conjuntion with the alpha registers. This allows us to use them for storage of data, for loop control words. Lets write a short program to go from 0 to 9 and sound the tone of the loop control word each time. Lets use the M register as our loop counter. Write your version in the space below.

We can also use the Alpha registers to enter NNN's into the system. It is possible to store an NNN as a text string in your program, and when it is needed to do a RCL (alpha). This enters the NNN into the X register without normalizing it. The following lines of programing could be

```
+
+
+
20:36   04/27
01♦LBL "PRDEMO"
02 CLA
03 "BE$K"+"
04 RCL [
05 ACSPEC
06 PRBUF
07 END
```

27

```
4:11AM 01/02
01♦LBL "FLAG"
02 1
03♦LBL 01
04 ST+ X
05 VIEW X
06 GTO 01
07 END
```

used to create the special character ± . If we used the normal method of building the character the programing becomes prohibativly byte consuming.

```
CLA
~~"O" "G$±"~~  ~~(F7 1 00 1 22 4 7 1 2 00)~~ F7/10/02/24/4B/F1/22/00
RCL M 90/75
ACSPEG
```

We can also manipulate the alpha string that is in the display. The need to use synthetics for this has been almost eliminated by the advent of the XFM module. The PPC ROM has some utilities programs for doing this though.

SYNTHETIC TONES

As you saw earlier in section II the tone function has the HEX code 9F __. Where the second byte is a postfix from 00 to 09. With synthetics we can now use and byte we wish as the postfix for the 9F. We find by experimentation that using any byte from the lower half of the byte table results in 128 different tones. Using any byte from the top half of the byte table results in TONE IND __.

With further work we find that the tone's frequancy is dependant on the 0th nibble of the post fix, and the tone's duration is dependant on the 1st nibble of the postfix. This means that there are 16 tones with varing durations. However they are different and have some uses.

The very short tones are good for use with an alpha prompt for data input from the keyboard. The prompt is slow enough and a tone can slow it further to the point where you may misenter a number. The shorter tones help here.

Another use is to give the keys a tonale feedback. Each key can be assigned a tone that sounds before the function on it executes.

For hardware bufs the tones can be used to turn devices on and off, different durations setting different conditons.

CURTAIN MOVING

Lets go back and look again at register c. We found that it contained four three nibble pointers. The first always gave the present register of the permantent .END., the second pointed to the absolute address of the R00 data register (curtain between program registers and data registers), the third was a cold start constant, and the fourth was the absolute address of the first REG. At times it would be benifical to hide a group of data registers when more then one program or the same progam has to access the same block of data registers and would destroy the data stored there. Be placing the correct NNN into the c register it is possible to move the curtain up or down and thus cover or uncover a group of registers. Lets see how this works.

1. RCL c
2. XEQ "NH" Have them do a size 10 w/No Programs.
3. 1D90016 91CB1CB
4. Add ten (A) to the 1CB
5. 1D9001691D91CB
6. 01 STO 00 11 STO 11
7. Place HEX code in ALPHA
8. XEQ "HN"
9. STO c
10. RCL 00???????

EXAMPLE #~~8~~9

What happened to the 01 that we stored into the R00? We raised the curtain 10 registers and what was R11 is now R00. Now lets put the curtain back and see if the 01 we stored in R00 is still there. Type in the HEX code for the orginal curtain and XEQ "HN", then STO c. Recall 00 and theres the 01. Now clear X, and STO c. This is the result of a mistake or of late night editing.

FLAG CONTROL

The flag register (d) can also be used for a lot of things. Lets do a couple of examples. Lets load the following program from tape and run it then see what is happening.

1. ALPHA FLAG ALPHA
2. READP
3. RUN
4. RUN

What are all those strange displays that are appearing? We used a special FLX e to increment the flags until flag 30 the CAT flag was set, we then STOP and when we pressed RUN the next time the CAT started up because flag 30 was set.

Now lets use an NNN to set the Printer flag 55. Lets see how this will effect our run time on a program first.

1. ALPHA FL55 ALPHA
2. READP
3. GTO ..
4. XEQ "FL55"
5. 00 00 00 3C 02 80 01
6. XEQ "HN"
7. STO d
8. XEQ "FL55"

[illegible]

0000063C02800
0000043843000

EXAMPLE ~~#X~~ 10

With the flag 55 set the execution is somewhat slower. This is most significant when a long program is running and output is sent to the printer only at the end of the program. We can also use NNN's to setup the flags as we want for a given program by letting the first few lines be a NNN with a STO d following them.

SECTION V - QUIZ

1. If you wanted to assign the synthetic function X<> d to the shifted STO key, what would the hex code be for the assignment?

CE ~~E7~~ 2B
7E

if row 8 then carry one to next significant byte.

   a. EC E7 B2
   b. 71 7E 33
   c. CE 7E 2B
   d. 04 CE 01
   e. none of the above

55 includes

2. If flags 1, 7, 12, 27, and 40 are to be set with a STO d instruction. What is the hex code that should be stored into the d register?

   a. 41 08 00 10 00 80 01
   b. 10 08 00 01 00 80 14
   c. A0 00 00 0C 01 00 01
   d. 01 80 10 00 00 08 10
   e. none of the above

3. The c register may be used as a synthetic register

   a. only in program mode
   b. only when there are no subroutines pending
   c. only if the 169 constant is saved
   d. never
   e. only on Sundays

4. The following address pointer 51CE is pointing at the next executable instruction in your program. What is the register and what byte is that instruction in?

   a. byte E register 51C
   b. byte 5 register EC1
   c. byte 6 register 51C
   d. byte 5 register 1CE
   e. none of the above

5. A NNN is defined as:

   a. a number that has not been operated on by the system
   b. a number that has anomolous digits in its sign digit or other fields
   c. a number that has been changed by the RCL instruction

SECTION V - QUIZ (continued)

6. The byte grabber should not be used in PRGM mode if you are:

   a. close to the .END.
   b. in the middle of a program
   c. in the middle of a ROM
   d. none of the above

7. The Q loader can be used to create a synthetic text line of up to ______ characters?

   a. 2
   b. 9
   c. 7
   d. 1
   e. none of the above

8. The byte jumper uses the Øth nibble of the byte preceeding the address pointer to determine how many bytes it should jump. What is the maximum number it can jump?

   a. 16
   b. 8
   c. 7
   d. 15
   e. none of the above

9. Which of the status registers are used by the system for scratch?

   a. P, M, N, O
   b. Q, a, c, d
   c. P, Q, ⊢, c
   d. a, b, c, ⊢
   e. none of the above

P ⇒ nibbles 13-6
Q ⇒ all of it
⊢ ⇒ nibbles 4-Ø
c ⇒ nibbles 10,9 printer

10. The address pointer may point to any place in RAM or ROM. If it is pointing to ROM it does not have a

   a. byte pointer
   b. register pointer
   c. not part part of the status registers
   d. can't be done

FINAL EXAMINATION

1. The following codes describe what functions...

```
91 75
CF 77
92 75
```

a. RCL M STO 0 X<> M
b. RCL M X<> 0 STO M
c. STO 0 X<> M RCL 0
d. STM LBL O ST+M
e. STO O LBL O ST+O

2. The key assignment registers contain ____ assignments in each?

a. 3
b. 2
c. 7
d. 1
e. none of the above

3. Write out the key assignments for the following functions:

1. byte jumper

2. q-loader

4. The following nibble pattern was obtained by what type of operation?

1 F F 0 0 0 0 0 0 0 0 0 0 8 0

a. a numeric entry line
b. a RCLFLAG operation
c. an ASTO operation on a BUG 7 machine
d. b and c
e. none of the above

5. Write the program lines necessary to create an ASCII file named "QWERTY" with an extent of 16, enter two records of your choice and position the file pointer to the second character of the second record.

6. If the ALPHA register contains "ADVANCED COURSE" and the X register contains 67, what would be the resulting value in the X register from a POSA instruction?

   a. -1
   b. 3
   c. 4
   d. 5
   e. 6

7. An ENG IND 99 instruction would take how long to execute?

   a. 16.7 mS
   b. 16.6 mS
   c. 32.0 mS
   d. 30.0 mS
   e. none of the above

8. The fastest executing numeric entry line to generate the number -1.2345 E-5 would be:

   a. -1.2345 E-5
   b. 1.2345 E-5, CHS
   c. -12345 E-9
   d. 12345 E-9, CHS
   e. none of the above

9. The local alpha label: LBL A occupies:

   a. 2 bytes
   b. 1 byte
   c. 3 bytes
   d. 5 bytes
   e. none of the above

10. The HP-41C is which of the following:

   a. a personal computer
   b. a tool
   c. a toy
   d. none of the above
   e. a, b, and c

SECTION VI

APPENDICES

References and Recommended Reading

Owner's Handbook and Programming Guide, HP-41C/CV, Hewlett-Packard Company.

82104A Card Reader Owner's Handbook, Hewlett-Packard Company.

82143A Printer Owner's Handbook, Hewlett-Packard Company.

Extended Functions/Memory Module Owner's Manual, Hewlett-Packard Company.

Jarett, Keith, HP-41 Synthetic Programming Made Easy, Synthetix, Manhattan Beach, CA, 1982.

Wickes, W. C., Synthetic Programming on the HP-41C, Larken Publications, College Park, MD, 1980.

Dearing, John, Calculator Tips & Routines, Especially for the HP-41C/41CV, Corvallis Software, Inc., Corvallis, OR, 1981.

PPC ROM USER'S MANUAL, PPC, Santa Ana, CA, 1981.

Hewlett-Packard Journal, March 1980, Hewlett-Packard Co.
- Wong and Conklin, "Powerful Personal Calculator System Sets New Standards".
- Steiger, "Packaging the HP-41C".
- Johnson and Marathe, "Bulk CMOS Technology for the HP-41C".

PPC Calculator Journal, PPC, Santa Ana, CA.

This publication is recommended for programmers and users of all levels of expertise. It contains programs and data spanning a wide variety of applications. It is also a forum for HHC programmers to share their discoveries, tips, and opinions. The following articles are some of the more important articles that deal with the topics of this course. The references will be of the form Volume, Number, and Page.

For a price schedule of reprints, send a SASE (9" x 12") to:

PPC, Reprints
2545 West Camden Place
Santa Ana, CA 92704

PPC Calculator Journal

Kennedy, "HP-41C Combined Hex Table"	V6 N5 P22
Nelson, "Bugs in the Box?"	" " P27
McClellan, "Inside the HP-41C - Photos, Schematics"	V6 N6 P 4
Hooper, "Editing Speed"	" " P30
"Bugs Four & Five"	" " P30
Wickes, "HP-41C Status Register Access"	V6 N7 P31
Nelson, "Bug Update - Bugs 6, 7 & 8"	V6 N8 P23
Wickes, "Through the HP-41C with Gun and Camera"	" " P27
Nelson, "HP-41C Tones"	V7 N1 P17
Wickes, "Freedom From Bugs"	V7 N2 P30
"Synthetic Key Assignments"	" " P30
Istok, "Pseudo XROM's on the HP-41C"	" " P32
Kolb, "Two Byte Assignments"	" " P36
Cullings, "Shift Key Reassignments"	" " P38
Close, "Bug 2 - A Practical Application	V7 N3 P 8
Edelen, "eGOBEEP"	" " P15
Cadwallader, "Flag 30 Catalogs"	V7 N4 P25
Wickes, "Byte Jumping"	" " P26
Wickes, "Direct Addressing of ROM Routines"	V7 N5 P55
Albillo, "Universal Byte Jumper"	V7 N6 P40
Jarett, "A Byte-Jumper Cookbook"	" " P43
Story, "Caveats for the Novice Byte-Jumper"	" " P46
McGechie, "Key-Assigned Numbers & Q Register Uses"	V7 N7 P21
Close, "Using Register P"	V7 N8 P27
McGechie, "Register Q Uses, Alpha GTO"	" " P27
Dearing, "41C M N & O Register Operations"	" " P29
Ruble, "HP-41C Saving Bytes"	V7 N9 P 9
MacLean, "Catalog Bus (PPC BUG 9)	" " P25
Stern, "Synthetic Programming Tips"	V7 N10 P20

PPC Calculator Journal

Close, "Romping Thru ROM"	V8	N1	P14
Massman, "Installing Modules Inside the HP-41C"	"	"	P20
Stern, "Program Mode 'Byte Jumpers'"	"	"	P31
Close, "Erasable ROM for the HP-41C"	V8	N2	P45
Cadwallader, "Merging HP-41C Key Assignments"	"	"	P46
Cadwallader, "HP-41C/V Reassigned Keys"	"	"	P47
Nelson, "41 Assembly Language Programming"	V8	N3	P16
Gibbs, "41 Function Timing"	V8	N4	P 5
McGechie, "Numerical Entry Times"	"	"	P 6
Collett, "Everything About Key Assignments"	"	"	P16
Baldrige, "Some Useful Byte-Jumpers"	"	"	P26
Close, "FIX 9 - A DECODE Alternative"	"	"	P28
Close, "BUG 9 Comments"	"	"	P31
Altman, "The HP-41 at 2X Speed"	V8	N5	P 3
Wickes, "Synthetic Programming - A Perspective"	"	"	P 4
Klous, "BUG 8 Synthetics"	"	"	P 7
Close, "More BUG 9"	"	"	P 7
MacLean, "SST Bugs"	"	"	P 8
"Error Ignoring Errors"	"	"	P 9
Brockman, "An Enhanced 'Byte-Jumper'"	"	"	P10
Baldrige, "F3 Byte Jumper"	"	"	P10
Cheeseman, "Long (Synthetic) Labels"	"	"	P10
Gosteli, "Generalized Fn Byte Maskers"	"	"	P11
Stern, "The P Register"	"	"	P13
Jensen, "Decoding Register P"	"	"	P13
Cadwallader, "The 41's 4096 CAT 3 Functions"	"	"	P21
Kendall, "Bare Bones Prefix Masker"	V8	N6	P40
Hill, "Pseudo Names - Synthetic Assignments"	"	"	P45
McCurdy, "HP-41 Long Form Branching to Short Form Labels"	"	"	P52
Schmill, "-2 -2 = 22?"	"	"	P53
Massman, "More HP-41 Speedup"	"	"	P54
Jensen, "The 41 'SCRATCH' Registers"	"	"	P77
Herzfeld, "More GTO Branching"	V9	N1	P 9
Cooper, "FIX e on the HP-41"	"	"	P10
Close, "HP-41 Crash Recovery Tips"	V9	N2	P21
Horn, "Q-Register-Sharing Functions"	"	"	P22

PPC Calculator Journal

Lipschultz, "Extended Memory Topics"	V9 N3 P21
Bailey, "Exploring Extended Memory"	" " P21
Wright, "EFM/EM Structure"	" " P22
Wagner, "A Program File Can Run?"	" " P24
Atwood, "EFM Observations"	" " P24
Cardinale, "EFM Comments"	" " P25
Karras, "How to Use ASTO b to Jump in Memory"	" " P25
Borrebach, "Lonngg GTO's and XEQ's"	V9 N4 P19
Szablowski, "Structures of Complex HP-41 instructions"	" " P19
Maloga, "Upside-Down RAM"	" " P36
Katz, "All Prefix Zero Key Assignments"	" " P67
White, "Addressing the HP-41C"	V9 N5 P18
McCurdy, "3-Byte GTO KAS and Key to IND Labels"	V9 N7 P 9
Bailey, "Time Module Alarms & I/O Buffer"	" " P11
Smith, "XS XEQ in the Stack"	" " P13
Lambert, "Synthetic Programming Using the 41 Card Reader"	" " P14
Mathewson, "EFM Verify Bug"	" " P19
Cadwallader, "256 Assignments on one HP-41 Key"	" " P20
Erickson, "The Care and Feeding of the HP-41 at 2X"	V9 N8 P19
Bernstein, "EFM - Tutorial for Running Programs Stored In"	V10 N1 P44
Close, "HP-41C/CV Normalized RCL"	" " P64