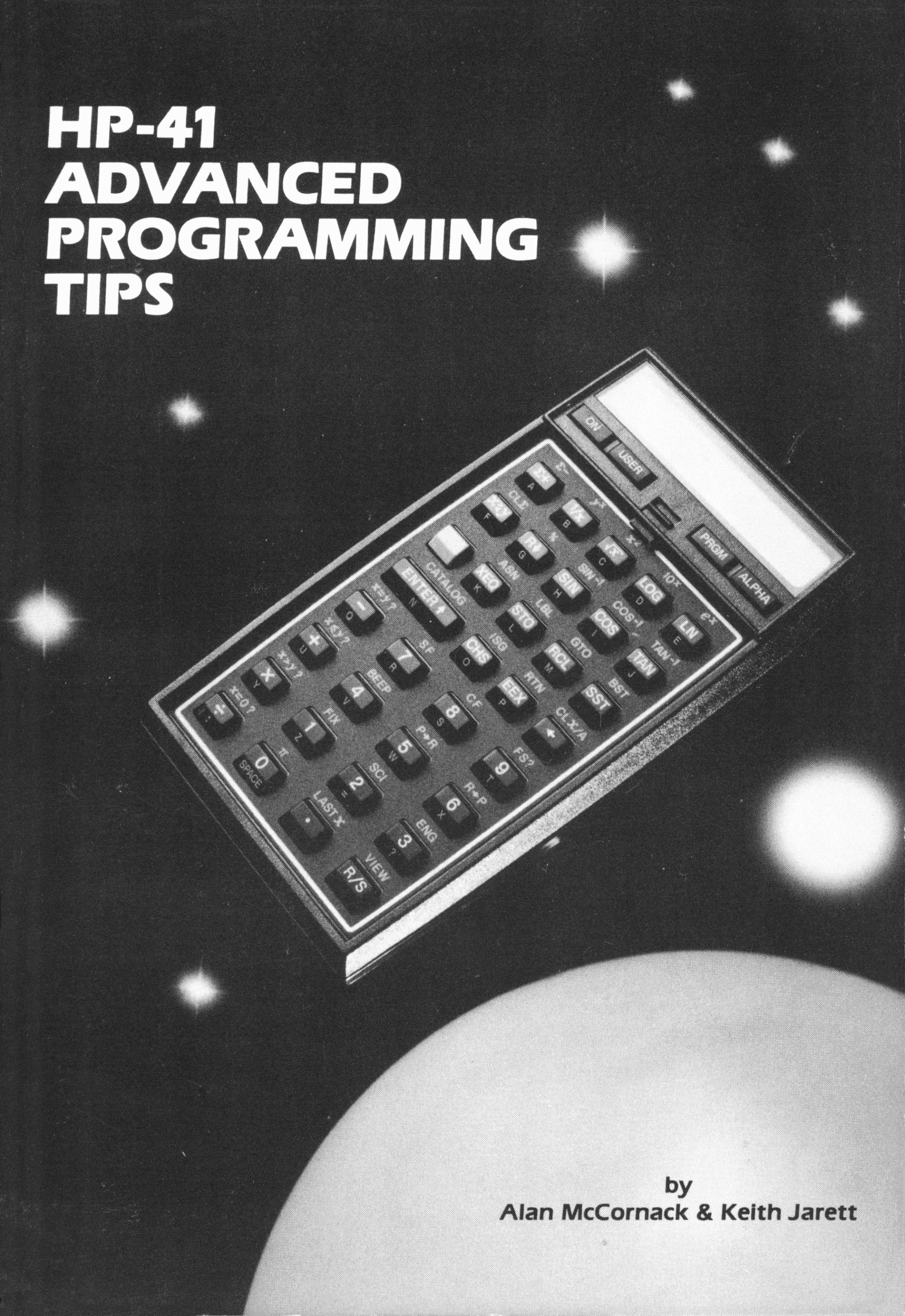
HP-41 ADVANCED PROGRAMMING TIPS



by

Alan McCornack & Keith Jarett

HP-41
ADVANCED
PROGRAMMING
TIPS

by Alan McCornack
and Keith Jarett

published by:
SYNTHETIX
P.O. Box 1080
Berkeley, CA 94701-1080 USA

ISBN 0-9612174-6-4
Library of Congress 85-61882

The material in this book is supplied without representation or warranty of any kind. Neither the publisher nor the author shall have any liability, consequential or otherwise, arising from the use of any material or suggestions in this book.

TABLE OF CONTENTS

PAGE

CHAPTER 1 - INTRODUCTION

CHAPTER 2 - STRUCTURED PROGRAMMING

CHAPTER 4 - ADVANCED SYNTHETIC PROGRAMMING

CHAPTER 5 - ZENROM UTILITIES AND M-CODE

CHAPTER 6 - MORE UTILITY PROGRAMS

CHAPTER 1 - INTRODUCTION

1A. UTILITY MODULES

Many of the sections in this book make reference to the ZENROM, the CCD Module, or the PPC ROM. These devices are modules that have been fabricated by Hewlett-Packard to the specifications of a third party. They are available from EduCALC (See Appendix B).

The ZENROM is a 4K byte module written in machine code (M-code), the natural language of the processor within the HP-41. Like the Extended Functions, ten of the ZENROM functions can be used within programs. There are two "operating modes" in the ZENROM: RAMED and MCED. RAMED allows you to edit bytes within the user memory (data, programs, buffers, Extended Memory, etc.) of the HP-41. With it, you can review or replace bytes, which are shown in hexadecimal (base 16). You can also insert bytes within program memory with RAMED.

MCED (Machine Code EDitor) is "A full machine language programming and editing environment including facilities for disassembly of M-Code routines and creation of new routines using the M-Code hex-loader (when used with 'Quasi-ROM' (Q-ROM) in a machine language storage device)." Even if you don't have a Q-ROM device like the MLDL (See Appendix B), MCED will allow you to view or print the contents of any plug-in ROM module.

The nicest feature of the ZENROM for synthetic programmers is the ability to key in instructions like RCL M directly. Another helpful modification that extends the capabilities of the HP-41 operating system is called "SYNTEXT entry". This allows entry of any character by pressing SHIFT, ALPHA followed by two hexadecimal digits. Also for entering special Alpha characters (both in and out of PRGM mode), the

ZENROM adds two USER mode keyboards.

The CCD Module was developed by the Computer Club of Germany. It is an 8K byte M-Code module with many powerful utilities. Operating system enhancements include directly keyed synthetic instructions, just like the ZENROM, plus the ability to directly execute or assign any two-byte function by specifying its decimal byte codes. Of course enhanced Alpha modes allow easy entry of synthetic alpha characters, and a lower-case mode that allows you to enter lower-case characters without pressing the shift key for each one.

The CCD Module provides an enhanced CATalog function, similar to the one on the HP-41CX. Catalog 2 allows you to press XEQ while an entry is displayed to enter the function in a program (PRGM mode) or execute it (run mode). Catalogs 8 through F allow you to start Catalog 2 at a particular port. The number 8 though F refers to the starting ROM address (see page 22 of this book).

Matrix functions allow you to create and operate on matrices either in main memory or extended memory. The matrix functions of the CCD Module are also available in HP's 12K byte Advantage Module. Logical functions provide some of the capabilities of the HP-16C calculator, including rotation, Boolean operations, and viewing the hexadecimal equivalent of X. Each bit of a number can be individually tested, set, or cleared, making it easy to use a number as a set of flags.

Other utilities help you to efficiently prompt for various types of keyboard input in your programs, automatically rejecting incorrect entries. Other added functions involve printer and alpha register output and input. The FIX/ENG display mode (see pages 139-143 of this book) can be set directly. Data registers or extended memory files can be sorted. I/O Buffers and Catalog 2 function Key Assignments can be saved in and retrieved from Extended Memory, and Key Assignments can be merged.

The ZENROM and CCD Module both have utilities which support advanced synthetic programming. These are the familiar register conversion to and from hexadecimal (coding and decoding), and register storage and non-normalized recalling. The CCD Module has additional byte-oriented storage and recall functions, functions to locate programs in main memory, and functions to manipulate program pointers.

Having either a ZENROM or a CCD Module will allow you to experiment with the ideas in this book more fully, especially where synthetic programming is involved. Extended Functions (built into the HP-41CX) are also highly recommended to the serious HP-41 programmer.

The PPC ROM is an 8K byte ROM module written in user code (the programming language described in the Owner's Manual, augmented by synthetic programming). The PPC ROM contains many powerful routines, half of which make use of synthetic programming. The 122 routines are divided into fifteen categories: Alpha Register, Block Operations, Curtain, Display, Key Assignments, Load Bytes, Mathematics, Matrix, Memory, Miscellaneous, Non-normalized Numbers, Peripherals, Program Pointer, Return Stack, and Sorts. Some of the functions within the PPC ROM have been superseded by faster Extended Functions, ZENROM, or CCD Module functions. The PPC ROM is several years old, but it remains a milestone both in programming and documentation. Study of the documentation should be a must for synthetic programmers wishing to learn more.

1B. GENERAL TERMS AND CONCEPTS

In order to understand the concepts used in this book, you should know a few basic facts about the HP-41 and how it works. The material presented in this section is intended to summarize the necessary terms and concepts. References to other chapters and books are included for further reading on these subjects.

If you are already familiar with hexadecimal (base 16) notation, the byte table, the HP-41's internal status registers, and key assignment registers, all of which come under the general heading of **Synthetic Programming**, you may want to skim this section quickly and move on to Chapter 2. There is enough information overlap that you can read the various sections in this book in whatever order you like. But you should only skip this section if you have a good working knowledge of synthetic programming on the HP-41. If this material is new to you, it would be a good idea to read it more than once.

Key terms

To get the most out of this book, you need to be familiar with certain key terms and abbreviations. Many of these terms relate to the memory of the HP-41 and to the number systems used to represent the information it contains.

The **byte** is the basic unit of memory on the HP-41. All data, whether numbers, characters or program lines, can be expressed as a byte or a series of bytes. The internal organization of the HP-41's user memory is based on registers, each of which contains seven bytes. In addition, bytes can be subdivided and represented in several ways, as we shall see later in this section.

Number Bases

We are all familiar with decimal, or base ten. Single digits range from 0 to 9, and 10 means ten. This familiar number system has little relationship to the internal workings of a computer. Other number systems are better suited than decimal to express number values in a computer.

Binary, or base two, uses only the two digits 1 and 0 to represent numbers. Usually binary digits (**bits**) are grouped in sets of four. As you shall see, this grouping makes the conversion between binary and hex simple. Each bit represents digital logic high (1) or low (0) voltage state. Like base ten, the rightmost digit is the "ones place". The next digit to the left is base two to the first power, or $2^1 = 2$. Two to the second power, or 4, is next, followed by two to the third power, or 8. As an example, the decimal number eleven is written as 1011 in binary. If you align each digit with the value it represents, you'll see the correspondence more clearly:

```
8 4 2 1
1 0 1 1
```

To convert this number to decimal, add the decimal values for each position multiplied by the binary digit in that position. In this case, 1x8 + 0x4 + 1x2 + 1x1 = 11. Thus we just add the decimal values for each positon whose bit is one.

The HP-41 uses 8-bit bytes in user memory. The maximum value of a byte is 1111 1111 base two, or 255 decimal. The leftmost bit represents 128, the next 64, then 32, 16, followed by 8, 4, 2 and 1. The sum is 255.

Even though the machine works in binary, this notation is too cumbersome for general use. **Hexadecimal**, also called **hex** or base sixteen, is a convenient compromise between the decimal system we all know, and binary, which computers use internally. Each group of four binary digits can be represented by a single hex digit, which is also known as a **nybble**. Thus, any 8-bit byte can be conveniently expressed as two 4-bit nybbles.

Each hex digit symbolizes a decimal value from 0 to 15. Digits above 9 use the letters of the alphabet A through F. The binary equivalents of the 16 different hexadecimal digits are shown below:

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

These equivalences are also shown at the bottom of the byte table (discussed later in this chapter).

Hexadecimal is used throughout this book to express addresses in memory, contents of registers, and values of bytes in memory. The word hex will generally accompany any value that is given in hexadecimal. The nybbles will generally be separated into groups of two or three for easier reading. When you see a number which does not have any digits from A to F, whose digits are not grouped, and which is not referred to as a hex number, you may assume that it is a decimal number.

Hex is a natural choice for ZENROM owners since special Alpha characters can be keyed in using hex values, and the CODE, DECODE, MCED and RAMED functions use hex. Decimal equivalents are provided in almost all cases. This will be helpful if you are using XTOA, ATOX, or some of the PPC ROM functions like LB and MK to construct synthetic instructions. However the hex values more clearly indicate the underlying binary structure. This structure is important in many of the examples, so you should be sure to look at the hex values even if you are not using them.

Register addresses are three hex digits long. To convert these three nybbles to decimal requires a little math. If you aren't familiar with these conversions, you should read the following example carefully. You may want to check your results by using the PPC ROM routines "TB" and "BD" (the PPC ROM manual has some helpful material in the writeups for these programs). Or you could use an HP-16C, switching between HEX and DEC modes with the number you wish to convert in the display. The HP-16C manual can also help you understand number bases and conversions.

To convert address 1AB from hex to decimal, you first need to figure out the decimal quantity each digit represents. The 'B' is in the ones place (16 to the 0 power). 'A' represents the number of sixteens (16 to the first power). Finally we have '1' in the two-hundred-fixty-sixes place (16 to the second power). Now multiply each nybble by the appropriate value and sum them:

B =	11 * 1 =	11
A =	10 * 16 =	160
1 =	1 * 256 =	256
		427 decimal

Try several examples if you find this difficult to understand. You will need a working knowledge of base conversions to get the most out of some sections of this book.

The Byte Table

The Byte Table (see Figure 1.1 on page 8) collects a variety of useful reference information on byte structure for advanced HP-41 users. A durable plastic version of the byte table is available (see Appendix B). This plastic card, which fits inside the HP-41's carrying case, is called the QRC (Quick Reference Card for Synthetic Programming). In addition to the QRC, serious synthetic programmers should purchase Jeremy Smith's "SYNTHETIC Quick Reference Guide". It has a wealth of additional information on the internal workings of the HP-41, most of which you will understand after reading this book.

HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING © 1982, SYNTHETIX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	CAT	@c (GTO..)	DEL	COPY	CLP	R/S	SIZE	BST	SST	ON	PACK	←(PRGM)	USR/P/A	2 __	SHIFT	ASN	
0	NULL 00 ‾ 0 ◆	LBL 00 01 x̄ 1 ×	LBL 01 02 ▓ 2 x̄	LBL 02 03 ▓ 3 ←	LBL 03 04 x̄ 4 α	LBL 04 05 x̄ 5 β	LBL 05 06 Γ 6 Γ	LBL 06 07 ▓ 7 ↓	LBL 07 08 ▓ 8 Δ	LBL 08 09 ▓ 9 σ	LBL 09 10 ▓ 10 ◆	LBL 10 11 ▓ 11 λ	LBL 11 12 μ 12 μ	LBL 12 13 ∡ 13 ∡	LBL 13 14 ▓ 14 τ	LBL 14 15 ▓ 15 Φ	0
1	0 · 16 ▓ 16 Θ	1 17 ▓ 17 Ω	2 18 ▓ 18 δ	3 19 ▓ 19 Å	4 20 ▓ 20 å	5 21 ▓ 21 Ä	6 22 ▓ 22 ä	7 23 ▓ 23 Ö	8 24 ▓ 24 ö	9 25 ▓ 25 Ü	. 26 ▓ 26 ü	EEX 27 ▓ 27 Æ	NEG 28 ▓ 28 œ	GTO ᵀ 29 ≠ 29 ≠	XEQ ᵀ 30 ▓ 30 £	W ᵀ 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 0 48 0	STO 01 49 1 49 1	STO 02 50 2 50 2	STO 03 51 3 51 3	STO 04 52 4 52 4	STO 05 53 5 53 5	STO 06 54 6 54 6	STO 07 55 7 55 7	STO 08 56 8 56 8	STO 09 57 9 57 9	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 A 65 A	✱ 66 B 66 B	/ 67 C 67 C	X<Y? 68 D 68 D	X>Y? 69 E 69 E	X≤Y? 70 F 70 F	Σ+ 71 G 71 G	Σ− 72 H 72 H	HMS+ 73 I 73 I	HMS− 74 J 74 J	MOD 75 K 75 K	% 76 L 76 L	%CH 77 M 77 M	P→R 78 N 78 N	R→P 79 O 79 O	4
5	LN 80 P 80 P	X↑2 81 Q 81 Q	SQRT 82 R 82 R	Y↑X 83 S 83 S	CHS 84 T 84 T	E↑X 85 U 85 U	LOG 86 V 86 V	10↑X 87 W 87 W	E↑X-1 88 X 88 X	SIN 89 Y 89 Y	COS 90 Z 90 Z	TAN 91 [91 [	ASIN 92 \ 92 \	ACOS 93] 93]	ATAN 94 ↑ 94 ↑	→DEC 95 _ 95 _	5
6	1/X 96 ᵀ 96 ᵀ	ABS 97 a 97 a	FACT 98 b 98 b	X≠0? 99 c 99 c	X>0? 100 d 100 d	LN1+X 101 e 101 e	X<0? A ▓ 102 f	X=0? B ▓ 103 g	INT C ▓ 104 h	FRC D ▓ 105 i	D→R E ▓ 106 j	R→D F ▓ 107 k	→HMS G ▓ 108 l	→HR H ▓ 109 m	RND I ▓ 110 n	→OCT J ▓ 111 o	6
7	CLΣ T ▓ 112 p	X<>Y Z ▓ 113 q	PI Y ▓ 114 r	CLST X ▓ 115 s	R↑ L ▓ 116 t	RDN M [▓ 117 u	LASTX N \ ▓ 118 v	CLX O] ▓ 119 w	X=Y? P ↑ ▓ 120 x	X≠Y? Q _ ▓ 121 y	SIGN ⊢ ᵀ ▓ 122 z	X≤0? a ▓ 123 π	MEAN b ▓ 124 |	SDEV c ▓ 125 →	AVIEW d Σ 126 Σ	CLD e ⊦ 127 ⊢	7
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	
	00 01 02 03	04 05 06 07	08 09 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	32 33 34 35	36 37 38 39	40 41 42 43	44 45 46 47	48 49 50 51	52 53 54 55	← bit numbers in a 7-byte register		

Figure 1.1a The HP-41 Byte Table, Rows 0 to 7

HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING © 1982, SYNTHETIX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8	DEG IND 00 128 ◆	RAD IND 01 129 ×	GRAD IND 02 130 x̄	ENTER↑ IND 03 131 ←	STOP IND 04 132 α	RTN IND 05 133 β	BEEP IND 06 134 Γ	CLA IND 07 135 ↓	ASHF IND 08 136 Δ	PSE IND 09 137 σ	CLRG IND 10 138 ◆	AOFF IND 11 139 λ	AON IND 12 140 μ	OFF IND 13 141 ∡	PROMPT IND 14 142 τ	ADV IND 15 143 Φ	8
9	RCL IND 16 144 Θ	STO IND 17 145 Ω	ST+ IND 18 146 δ	ST− IND 19 147 Å	ST ✱ IND 20 148 å	ST/ IND 21 149 Ä	ISG IND 22 150 ä	DSE IND 23 151 Ö	VIEW IND 24 152 ö	Σ REG IND 25 153 Ü	ASTO IND 26 154 ü	ARCL IND 27 155 Æ	FIX IND 28 156 æ	SCI IND 29 157 ≠	ENG IND 30 158 £	TONE IND 31 159 ▒	9
A	XR 0-3 IND 32 160	XR 4-7 IND 33 161 !	XR8-11 IND 34 162 "	X12-15 IND 35 163 #	X16-19 IND 36 164 $	X20-23 IND 37 165 %	X24-27 IND 38 166 &	X28-31 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 0	GTO 00 IND 49 177 1	GTO 01 IND 50 178 2	GTO 02 IND 51 179 3	GTO 03 IND 52 180 4	GTO 04 IND 53 181 5	GTO 05 IND 54 182 6	GTO 06 IND 55 183 7	GTO 07 IND 56 184 8	GTO 08 IND 57 185 9	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 A	GLOBAL IND 66 194 B	GLOBAL IND 67 195 C	GLOBAL IND 68 196 D	GLOBAL IND 69 197 E	GLOBAL IND 70 198 F	GLOBAL IND 71 199 G	GLOBAL IND 72 200 H	GLOBAL IND 73 201 I	GLOBAL IND 74 202 J	GLOBAL IND 75 203 K	GLOBAL IND 76 204 L	GLOBAL IND 77 205 M	X<>-- IND 78 206 N	LBL -- IND 79 207 O	C
D	GTO -- IND 80 208 P	GTO -- IND 81 209 Q	GTO -- IND 82 210 R	GTO -- IND 83 211 S	GTO -- IND 84 212 T	GTO -- IND 85 213 U	GTO -- IND 86 214 V	GTO -- IND 87 215 W	GTO -- IND 88 216 X	GTO -- IND 89 217 Y	GTO -- IND 90 218 Z	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 a	XEQ -- IND 98 226 b	XEQ -- IND 99 227 c	XEQ -- IND100 228 d	XEQ -- IND101 229 e	XEQ -- IND102 230 f	XEQ -- IND103 231 g	XEQ -- IND104 232 h	XEQ -- IND105 233 i	XEQ -- IND106 234 j	XEQ -- IND107 235 k	XEQ -- IND108 236 l	XEQ -- IND109 237 m	XEQ -- IND110 238 n	XEQ -- IND111 239 o	E
F	TEXT 0 IND T 240 p	TEXT 1 IND Z 241 q	TEXT 2 IND Y 242 r	TEXT 3 IND X 243 s	TEXT 4 IND L 244 t	TEXT 5 INDM[245 u	TEXT 6 IND N \ 246 v	TEXT 7 INDO] 247 w	TEXT 8 IND P↑ 248 x	TEXT 9 INDQ_ 249 y	TEXT10 IND⊢ᵀ 250 z	TEXT11 IND a 251 π	TEXT12 IND b 252 |	TEXT13 IND c 253 →	TEXT14 IND d 254 Σ	TEXT15 IND e 255 ⊢	F
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

Figure 1.1b The HP-41 Byte Table, Rows 8 to F

Each of the 256 squares in the byte table represents one of the 256 possible byte values. The QRC is printed on both sides, each with eight rows and sixteen columns of squares. Each square contains four to five pieces of data about that particular byte value. The color-shaded areas designate prefixes for multi-byte instructions, examples of which are given on the back of the card. The QRC also has a list of flag functions on the front.

The byte table and QRC allow conversion between hex, decimal, binary, user code instruction prefixes, instruction suffixes, display characters, and printed characters. The hex value of a byte is in the row/column format. For instance, to find the byte corresponding to hexadecimal 49, look at row 4, column 9. At the bottom of this box is the decimal equivalent 73. Thus 73 equals hex 49. This can be checked as follows: 73 = (4*16) + 9.

The top of each square gives the identity of the byte when it is treated as an instruction. If the byte can be used as the first byte of a multi-byte function, this prefix is listed as it would be shown in PRGM mode (FIX, GTO, etc.) or described briefly (GLOBAL, TEXT 7, etc.). By the time you finish this book, you will be fully informed on these latter, less recognizable prefixes.

The front side of the QRC is mostly white, with only one group of three squares shaded in blue (indicating text-related prefixes). The middle row of each square on the front side shows two facts about each byte. On the left is the way the byte would be shown as the second byte (called the suffix or postfix) of a two-byte instruction. To the right is an actual reproduction of the byte as it appears in the Alpha register as a character. Because all of the characters in the second half of the byte table and QRC are displayed as a starburst or "boxed star" (all 14 display segments lit), this is not printed on the reverse. See decimal byte 16, the first byte in row 1, if you aren't sure what a starburst looks like. The postfixes listed on the back are IND 00 through IND e, which shows you that the postfix byte values from rows 8 through F are used for indirect addressing.

The bottom left of each box has the decimal equivalent listed, with byte values 0 to 127 on the front, and 128 to 255 on the back of the QRC.

The bottom right of each box shows how the byte is printed as a character by the HP-82143 printer (the one that plugs directly into the HP-41). The HP-IL printer treats a few of these characters as control characters, but most of the characters are printed the same as on the HP-82143. Characters on the reverse of the QRC disappear from text lines in program listings, and characters that have a shaded background cause additional strange printer behavior when they are listed as part of a text line in a program.

Other terms

Backarrow refers to the key with an arrow on its face, pointing to the left. It is located on the rightmost key in the third row of the keyboard. The backarrow key serves several functions including correcting errors while entering numbers and characters or while filling in prompts. In PRGM mode it is used to delete lines.

A NOP (No OPeration) function mainly serves as a place holder. NOPs are often used after ISG or DSE instructions when you want the same instructions to be executed regardless of whether a skip occurs. Many functions can be used as NOPs, including unused labels (LBL 01, etc.), STO X, or CLD. Each of these has restrictions or disadvantages. The best choice is the synthetic instruction TEXT 0 (F0, decimal byte 240). ZENROM owners can execute "NOP" to insert a Text 0 byte in PRGM mode. CCD Module owners can press XEQ ENTER 240 240 to get two NOPs. The only effect of this NOP instruction is clearing the stack lift disable. (The ALPHA register is not disturbed.)

NOMAS is an acronym meaning NOt MAnufacturer Supported. Synthetic functions were not intentionally designed into the instruction set of the HP-41. Some have quirky, unpredictable or undesirable effects. For these and other reasons, Hewlett-Packard has decided not to fully support and guarantee these functions. Therefore, if you have problems with the material in this book, please do not contact HP. They have kindly provided an unusual amount of inside information to the synthetic programming community with the agreement that they have no obligation to provide the same level of support that they give to (nonsynthetic) functions discussed in their manuals.

Abbreviations

Because of the fact that many references are made throughout this book to other books, three abbreviations are used to keep from repeating the same information many times over. See Appendix B for sources of these books and other materials.

SPME -- "HP-41 Synthetic Programming Made Easy" by Keith Jarett
XFME -- "HP-41 Extended Functions Made Easy" by Keith Jarett
EYHP -- "Extend Your HP-41" by W.A.C. Mier-Jedrzejowicz
SQRG -- "HP-41 SYNTHETIC Quick Reference Guide" by Jeremy Smith

Memory Structure

Within the HP-41, there are two types of memory: RAM and ROM. RAM (Random Access Memory) is also known as user memory because it can be altered by the user. Refer to Figure 1.2, RAM Memory Structure on the next page (or the similar figure on page 4 of SQRG), which shows the various areas which make up user memory. ROM is Read Only Memory, the kind of memory which is contained in application modules. This memory cannot be altered.

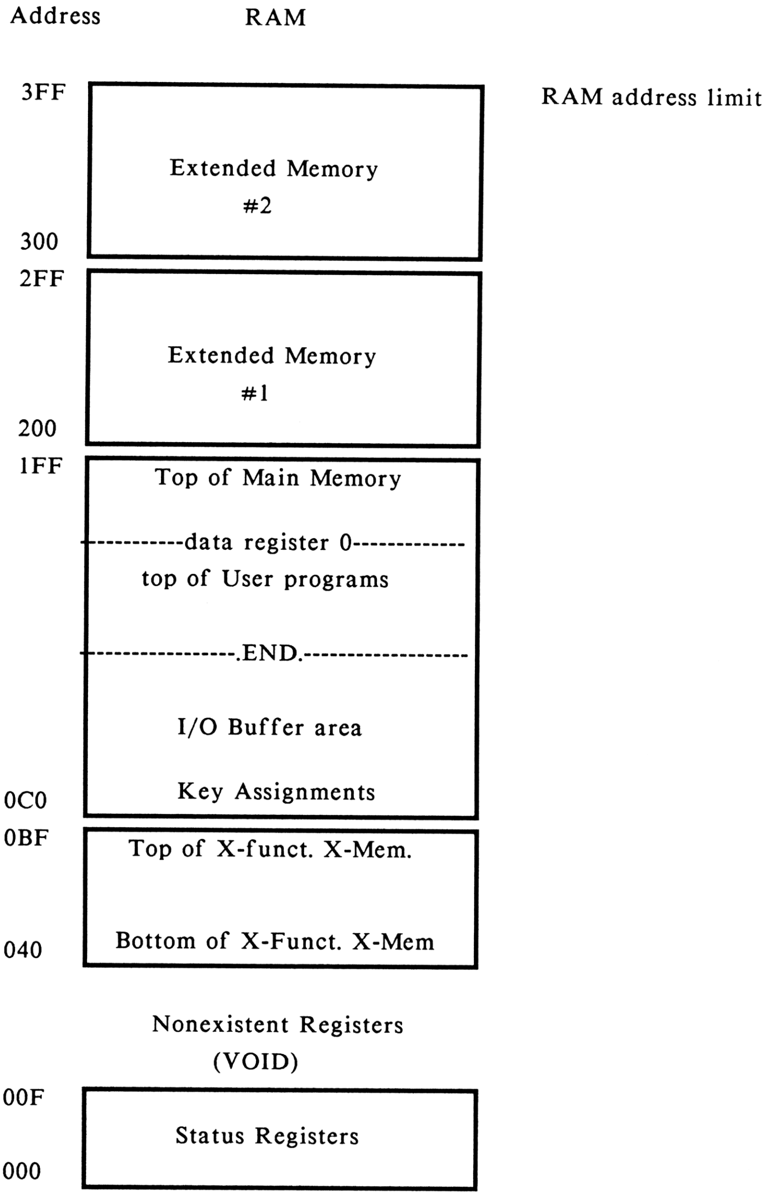


Figure 1.2 RAM Memory Structure

The topmost registers are reserved for data. The number of data registers is set by the SIZE (or PSIZE) function. The SIZE can be set as low as zero, in which case there are no data registers. After MEMORY LOST, the number of data registers is 100 for an HP-41CX, or 273 for an HP-41CV (or HP-41C with Quad memory module). This area contains the numbered data registers, in which data from operations such as STO 00 are held. The address of data register 00 is also known as the 'curtain'.

Data registers up to 99 can be addressed directly by normal functions. Plugging in the ZENROM will extend this capability up to register 111 by making postfixes 100 to 111 keyable (but don't go beyond 111 unless you realize that you are accessing other, and dangerous, areas of memory!). Synthetic instructions with postfixes up to 111 can be created with the Byte Grabber, as will be explained in Section 4B.

The structure of data in a register is important to understand. This subject is covered in SPME at an introductory level, and in EYHP in slightly more detail. In the next few paragraphs we will briefly review this subject.

Within a register, the bytes are numbered 6, 5, 4, 3, 2, 1, and 0, from left to right. Nybbles are numbered 13 to 0, also from left to right. For example, the number -2.349817 E-98 is stored in a register as follows:

Byte	6	5	4	3	2	1	0
Nybble	13 12	11 10	9 8	7 6	5 4	3 2	1 0
Contents	9 2	3 4	9 8	1 7	0 0	0 9	0 2
Meaning	- 2.	3 4	9 8	1 7	0 0	0 E-	9 8

Negative signs are represented by the nybble 9, positive signs by the nybble 0. Nybbles 12 through 3 of a register contains the mantissa in Binary Coded Decimal (BCD) form. The mantissa of a number is simply the 10 numeric digits without sign, exponent, or decimal point. Each nybble (4 bits) represents one of the ten decimal digits that make up the number. These nybbles range in value from 0 to 9 unless the register contains Alpha data or a Non-Normalized Number (NNN). These will be discussed near the end of this section.

The next nybble to the right, nybble 2, is used for the sign of the exponent. Like the sign of the entire number, its value will be 0 for positive exponents and 9 for negative exponents. 9 is used for numbers with absolute values between zero and one. The exponent is represented in a special way, by adding 1000 and then taking only the last three digits. This speeds internal arithmetic. In the above example, 1000 + (-98) = 902.

Alpha data is stored in a register as hex 1x, followed by 6 character bytes. The value of the x nybble is normally 0, but anything will do. If there are fewer than 6 characters in the string, nulls (hex 00) are added after the leading 1x byte to pad the result to a full 7 bytes. For example, the string "ABCD" is stored in a register as hexadecimal 10 00 00 41 42 43 44.

Below the data registers are your user (Catalog 1) programs. The number of registers this area takes up will vary, depending on the total byte count of the programs. Program instructions are stored in sequence working from higher-numbered registers to lower-numbered registers, and from byte 6 to byte 0 (left to right) within a register. The last instruction in program memory is the permanent .END. (refer to Section 4G), which occupies bytes 2, 1, and 0 of its register.

The free registers are below the .END. . These are unused registers, containing only nulls (hexadecimal 00 bytes). The number of free registers can be seen by pressing RTN in run (non-PRGM) mode and pressing PRGM. You will see 00 REG 46 (or something similar), in this case indicating that 46 registers (322 bytes) are unused. One full free register is needed to insert program lines in an existing program unless lines have been deleted at that position in the program. The only other exception is when an addition is being made after the last line of the last program on Catalog 1 (just before the permanent .END.).

Below the free registers are the buffers. The order of the buffers, from hex address 0C0 (decimal 192) upwards is as follows: Key assignments, followed by alarms and other I/O buffers. One exception is that the temporary buffers used by the Solve and Integrate functions of the Advantage Module are placed below the key assignments. Unless you own an HP-41CX, all buffers except those for the key assignments are used by plug-in devices.

F0 (240 decimal) is placed in byte six of each key assignment register. Each key assignment register can hold two assignments. Bytes 2 and 1 (for the first assignment) contain the function code. Two-byte functions (XROMs and synthetic assignments) are stored just as they would be as instructions within a program. One-byte functions are preceded by an 04 filler byte. Byte 0 contains a keycode for the assigned key (see SQRG p. 36, ZENROM p. 44, or EYHP p. 213). An identical scheme is used for the second assignment, with the function code in bytes 5 and 4, and the keycode in byte 3.

The HP-41 always fills the key assignment registers from right to left. When the HP-41 deletes an assignment, it clears only the keycode from the assignment registers. The appropriate bit from the assigned key indicators in status registers ⊢ or e is also cleared (see SQRG pg. 6). The two remaining bytes are not cleared, but they will be overwritten by a new assignment. New assignments are placed in absolute address 0C0 (192 decimal) unless a keycode of zero (indicating an unused half-register or a cleared assignment) is found within the key assignment registers. The old key assignments (and buffers) are moved up one register and 0C0 is used for the new assignment.

ZENROM or CCD Module users may want to alter each F0 in byte six of the key assignment registers to F6, using RAMED or POKEB. This technique prevents the contents of these registers from being altered by packing or from being executed as program steps. Execution of the key assignment registers is possible using STO b, the Byte Grabber, when status register c does not point to the .END. (see Section 4F), or the conditions which may arise while using the methods in Sections 4G and 4I. The HP-41 will not alter this leading byte of the key assignment register unless both assignments are deleted from the register and you PACK, in which case the contents will be discarded.

All of the previous segments of RAM have variable lengths and adresses, except for the upper and lower limits. RAM from 1FF to 0C0 (511 to 192 decimal) contains 320 registers. However, three bytes of this are used for the permanent .END., which is not included in the maximum byte count of 2,237. And because the .END. cannot be removed, one register will always be allocated to program memory, making 319 the maximum SIZE. The remaining RAM addresses contain the Extended Memory of the Extended Functions module (which is internal to the HP-41CX) and the status registers. These addresses are fixed.

Status registers

The status registers occupy the 16 RAM addresses at the bottom of memory from 0 0 0 to 0 0 F (0 to 15 decimal). Their names are T, Z, Y, X, L, M, N, O, P, Q, ⊢ , a, b, c, d and e, respectively. Refer to Figure 1.3 on the next page (also SQRG p. 6, SPME p. 110).

<table>
<tr><th>Nybble</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>e</td><td colspan="9">Shifted Key Assign. Bit Map</td><td colspan="2">PTEMP2</td><td colspan="3">Line #</td></tr>
<tr><td>d</td><td colspan="14">56 User Flags</td></tr>
<tr><td>c</td><td colspan="3">REG start</td><td colspan="2">unused</td><td colspan="3">Cold start</td><td colspan="3">Reg. 0 addr.</td><td colspan="3">.END.</td></tr>
<tr><td>b</td><td colspan="10">Return stack</td><td colspan="4">Prgm pointer</td></tr>
<tr><td>a</td><td colspan="14">Return stack</td></tr>
<tr><td>⊢</td><td colspan="9">Unshifted Key Assign. Bit Map</td><td colspan="5">Scratch</td></tr>
<tr><td>Q</td><td colspan="14">Scratch</td></tr>
<tr><td>P</td><td colspan="8">Scratch</td><td colspan="6">Alpha Characters 22 to 24</td></tr>
<tr><td>O</td><td colspan="14">Alpha Characters 15 to 21</td></tr>
<tr><td>N</td><td colspan="14">Alpha Characters 8 to 14</td></tr>
<tr><td>M</td><td colspan="14">Alpha Characters 1 to 7</td></tr>
<tr><td>L</td><td colspan="14">Last X Register</td></tr>
<tr><td>X</td><td colspan="14">X Register</td></tr>
<tr><td>Y</td><td colspan="14">Y Register</td></tr>
<tr><td>Z</td><td colspan="14">Z Register</td></tr>
<tr><td>T</td><td colspan="14">T Register</td></tr>
</table>

Figure 1.3 The Status Registers

The status registers are also known as the 'system scratch' or simply 'scratch' registers. This is because the HP-41 uses these registers for temporary information storage and various housekeeping functions. They are used to keep track of which keys have been assigned, data/program partitioning, the modes and flags of the 41, pending subroutine returns, your location in program memory, and so on. Both internal functions and plug-in devices use various areas within the status registers for temporary data storage during operations. This section describes the information contained in the status registers. For tips on using some of these registers within your programs, see Section 4A of this book (also SQRG p. 7, SPME p. 111-118).

The five registers from address 000 to 004 make up the stack. You should be familiar with T, Z, Y, X, and L from using the HP-41 as a calculator, though the absolute addresses of these registers may be new to you.

The internal structure of the numbers and Alpha data in a stack register is the same as described above for numbered data registers. The value of the leftmost nybble is 0 for positive numbers, 9 for negative numbers, and 1 for Alpha data. Of course, other values are possible, such as after recalling the contents of status registers other than the stack. These values are known as Non-Normalized Numbers (NNNs). Such numbers can be safely held within the stack and are not subject to alteration.

However, an NNN will be altered by functions that recall it after it is stored in a numbered data register. This alteration process is called normalization, and is part of the test the HP-41 performs to ensure the register actually exists. The normalizing functions are RCL, X<>, VIEW, and ARCL. Because the status registers are always present, this test is skipped when a status register is recalled or viewed. As a consequence, operations involving the status registers are a little faster than those which use numbered data registers (see Appendix A of SPME, or SQRG p. 11). Also see page 35 in the SQRG for more details on normalization.

Status registers M, N, O, and part of P make up the Alpha register. The HP-41 makes them appear as one continuous register, holding up to 24 characters. The P register only contributes bytes 2, 1, and 0 to the Alpha display. The leftmost four bytes are used during number entry or display, and by the CATalog and WSTS functions. The other 21 of the 24 characters are contained within registers O, N, and M.

When characters are keyed into Alpha or recalled using ARCL, they fill the Alpha register from right to left. The first character entered will occupy byte 0 (the rightmost byte) of register M. As more characters are added, existing bytes are pushed to the left. Register M holds the first seven Alpha characters. When more than seven characters are brought into Alpha, they will push the leading characters into register N. This continues into registers O and P in a similar way. When the 24th position (in Alpha register P) is filled, a warning tone sounds (provided that flag 26 is set), indicating that addition of further characters will cause loss of data. However, the entire count of 28 characters in these four 7-byte registers can be safely used if you remain in ALPHA mode (See SPME pg. 112).

Register Q (address 009) is used by the HP-41, the Extended Functions and the Printer for Alpha scratch. This is the reason normal LBL, GTO, and XEQ functions are limited to seven characters. When synthetic GTO and XEQ functions are created with more than 7 characters, the HP-41 will search for a label that matches the last seven characters of the Alpha GTO or XEQ instruction.

Register ⊢ (append), also known as register 'R' to ZENROM users, contains a bit map of the unshifted key assignments. Its hex address is 00A (decimal 10). ZENROM owners should refer to page 44 of their manual, or see SQRG, p. 6 and 36. Each unshifted key corresponds to one of the bits in this register. The HP-41 uses this register like an index in a book; it checks this register whenever an unshifted key is depressed. The HP-41 then searches for an assignment only if the corresponding bit is set, indicating that an assignment is present. All unshifted key assignments can be temporarily suspended by storing zero in this register. See Section 6A (pages 231-235 of this book), pages 119-124 of SPME and pages 193-198 of XFME for suspend/restore key assignment ("SK"/"RK") programs. Also see Section 4A.

The append register also contains an area used as scratch by the HP-41. Nybbles 4, 3 and 2 contain the last function executed, while nybbles 1 and 0 hold the keycode during PASN.

Register a (address 00B, decimal 11) contains part of the subroutine return stack. Since each pending return is represented by two bytes, this seven-byte register contains three and a half return addresses. Each two-byte return can be written as four hexadecimal digits or sixteen binary bits. The RAM return address format in binary is:

0 0 0 0, b b b r, r r r r, r r r r.

This format, minus the leading zeroes, is the same as that used within three-byte GTOs, three-byte XEQs, and global functions. The nine r's above represent the absolute register address. (For the other functions listed above, this would be relative to the instruction itself rather than the bottom of memory.) The digits bbb identify the byte within that register. RAM return addresses point to the last byte of the XEQ or XROM instruction that caused this address to be pushed into the subroutine return stack. Since there are nine bits representing the number of registers, up to hex 1FF = decimal 511 (two to the ninth power minus one) registers can be encoded. Since 1FF is the highest possible register address in main memory, any address or jump distance can be stored using 9 bits.

ROM return addresses have a different represenatation:

p p p p, b b b b, b b b b, b b b b.

ROM return addresses consist of the page number (p p p p) followed by a twelve-bit byte (or word) number within the 4096 (4K) bytes of that "page" of ROM. The page addresses are not the same as the physical port number. See Figure 1.4 for a list of the sixteen page addresses (SPME p. 115, SQRG p. 38).

Most ROMs to date use the Lower 4K of their respective ports. Exceptions are the Auto-Start/Duplication (AUTOST 1A) module, ZENROM, and all 8K ROMs (which use both Upper and Lower 4K). See the material later in this section for more details on ROMs.

Register b contains two and a half pending return addresses plus the program pointer. The hex address of register b is 00C (decimal 12). As you can see in Figure 1.3, the first, second, and half of the third return address are contained within status register b.

Page Address	Device or physical port
0	Internal ROM 0
1	Internal ROM 1
2	Internal ROM 2
3	Internal ROM 3 (HP-41 CX X functions)
4	Diagnostic ROM
5	Time Module (internal for CX, page switched with extra CX Time and X functions)
6	Printer (either 82143 or HP-IL)
7	HP-IL Module Mass Storage
8	Port 1, Lower 4K
9	Port 1, Upper 4K
A	Port 2, Lower 4K
B	Port 2, Upper 4K
C	Port 3, Lower 4K
D	Port 3, Upper 4K
E	Port 4, Lower 4K
F	Port 4, Upper 4K

Figure 1.4 ROM page numbers

Register b also contains the program pointer in the two rightmost bytes. If you are in a ROM program, the program pointer has the same format as for a ROM return address. In a RAM program, the program pointer has a slightly different format than the RAM return addresses format. The bbb field is positioned differently:

0 b b b, 0 0 0 r, r r r r, r r r r.

In ROM, programs are stored from byte 000 of a page up to byte FFF. Therefore when executing a ROM program, the program pointer is incremented at each non-branching instruction. In RAM, this situation is reversed. Programs progress from higher to lower addresses, and the program pointer is normally decremented as instructions are executed.

The program pointer always indicates the last byte of the previous instruction in memory rather than the address of the function currently shown in PRGM mode. This becomes especially apparent when you RCL b in run mode while positioned to the .END.. Several nulls may exist between the last instruction and the .END., despite packing. Register b never points to nulls in PRGM mode; it always points to the last byte of the instruction that precedes the group of nulls.

ZENROM users can find evidence of this behavior by using the RAMED function in PRGM mode. The left-hand "window", showing one of three bytes visible in hexadecimal form, is the byte indicated by register b when RAMED is first executed. RAMED always returns to the line where you started because the contents of register b are maintained.

If you are still doubtful, set the SIZE to 000. Now CAT 1 and R/S immediately. RCL b in run mode. Decode this address using "PD" (PPC ROM), DECODE (ZENROM), DCD (CCD Module) or STO M, ATOX, ATOX. The result will be 3584 (=7*512), 02 00, or 2, 0. All these symbolize an address of byte 0, register 512:

0 0 0 0, 0 0 1 0, 0 0 0 0, 0 0 0 0,

the byte preceding the first instruction in program memory. This is the only case in which a register address of 512 (hex 200) is used.

When an XEQ or XROM (execute ROM program) instruction is encountered, the address of the last byte of the XEQ or XROM instruction is pushed onto the return stack in the appropriate return address format (ROM or RAM). The program pointer is then moved to the program that was specified by the XEQ or XROM instruction.

When executing a RTN or END, the HP-41 halts if the return address is zero, indicating that no return is pending. If there is a nonzero return address, the subroutine return stack is dropped, and the first return address becomes the new program pointer. (In the case of a return to a RAM address, the bbb bits are moved to the left to transform the return address format into normal program pointer format.) This results in the pointer indicating the byte prior to the instruction which is to be executed next.

Status register c, at address 00D (13 decimal), contains vital information relating to memory allocation. This includes the address of the summation registers, register 00, and the permanent .END.. The absolute address of the summation registers indicates the location of the lowest of the six registers in the summation block. If the number of data registers was decreased after the ΣREG function allocated these registers, the first three nybbles (13, 12, and 11) of register c may specify NONEXISTENT registers. Addresses above hex 1FA (with full memory) result in this error message when a function that refers to the summation registers is executed.

Nybbles 5, 4, and 3 of register c contain the address of the first data register. This data/program memory separation, named the "curtain" by its discoveror, William C. Wickes, will be hex 200 when the SIZE is zero with full memory. The curtain can be synthetically set to any value desired. However, if the register immediately <u>below</u> this address does not exist, MEMORY LOST will take place the moment the HP-41 enters standby mode (not running the program).

Nybbles 2, 1, and 0 contain the address of the .END.. Don't mess with this. Altering this address will result in an inability to access Catalog 1. Global branching instructions will be unable to find RAM programs, and packing may cause MEMORY LOST. See Section 4F for a discussion of the ways you can circumvent this. Setting this pointer to a nonexistent address will also cause MEMORY LOST.

The surest way to cause MEMORY LOST is to disturb the "cold start constant" in nybbles 8, 7, and 6. Anytime the HP-41 finds something other than hex 169 here, the result is swift and final. Nybbles 10 and 9 of register c are claimed in some references to be used by the printer for scratch, but in fact they are not disturbed by the HP-41. The CCD module does make use of some of these otherwise unused bits.

An expert synthetic programmer can alter register c in a running program, perform some operations, and restore it to its former value. Until such time as you have the necessary skills, it's best to leave register c alone. Programs which use register c are both powerful and dangerous. Register c should <u>never</u> be used for a scratch pad.

Register d (address 00E) contains the 56 flags of the HP-41 system. Flags 00 to 29 are called user flags, while flags 30 to 55 are termed system flags. The system flags cannot be altered directly by SF and CF instructions. Flag 00 is at the extreme left of register d; flag 55 is at the far right. Each flag occupies one bit of the register. The correspondence between bit positions and flags can be seen at the bottom of the upper half of the QRC. Figure 1.5 briefly describes the use of each flag.

Many of the powerful PPC ROM routines use register d in order to manipulate individual bits. Before the capabilities of the Extended Functions AROT, ATOX, XTOA, and others were available, this was the primary way used to edit NNNs. Flags 00 to 29 can be set, cleared, or tested directly. This allows the first thirty bits to be modified after swapping the bytes to be altered into d.

Flag		Function	See Note No.
00 -	10	General purpose	1
	11	Auto execute	2
	12	Double-wide print	2
	13	Lower case print	2
	14	Overwrite card (CARD RDR)	2
15 -	16	HP-IL printer modes:	2
0	0	MANual	
0	1	NORmal	
1	0	TRACE	
1	1	TRACE with STACK	
	17	Record incomplete (HP-IL)	2
	18	Interrupt enable (IL Development)	2
19 -	20	General purpose	2
	21	Printer enable	4
	22	Numeric entry	2
	23	Alpha entry	2
	24	Range error ignore	2
	25	Error ignore	2
	26	Audio enable	3
	27	USER mode	1
	28	Decimal point/comma	1
	29	Digit grouping	1
	30	CATalog mode	2
	31	DMY / MDY (Time Module)	1
	32	HP-IL manual I/O	1
	33	HP-IL lock	1
	34	HP-IL ADRON / ADROFF	1
	35	Disable autostart (AUTOST ROM)	1

Figure 1.5 HP-41 Flags (register d)

Flag		Function	See Note No.
36 -	39	# of digits displayed (BCD)	1
40 -	41	Display modes:	1
0	0	SCIentific notation	
0	1	ENGineering notation	
1	0	FIX	
1	1	FIX / ENG (see Section 4E)	
42 -	43	Trigonometry modes:	1
0	0	DEGrees	
0	1	RADians	
1	0	GRADians	
1	1	RADians	
	44	Continuous ON	2
	45	System data entry	2
	46	Partial key sequence	2
	47	SHIFT	2
	48	ALPHA mode	2
	49	Low BATtery	5
	50	Message	2
	51	SST	2
	52	PRGM mode	2
	53	I / O request	5
	54	PSE enable	2
	55	Printer existence	5

Note number: 1 Maintained by continuous memory
2 Cleared at turn-on
3 Set at turn-on
4 Clear to disable printer
5 Device-dependent, tested at turn-on

Figure 1.5 (continued) HP-41 Flags (register d)

The PPC ROM routine "IF" and the ZENROM function TOGF allow any individual flag to be inverted (set if clear, clear if set). Extended Functions RCLFLAG and STOFLAG also allow groups of flags or individual flags to be set or cleared, using a control number in X. The Extended Function X<>F permits control of the first eight flags, 00 to 07. A decimal number similar to ATOX/XTOA is used, but the bit correspondence you would normally expect is reversed. Rather than the leftmost bit being the most significant digit (128), the rightmost bit (flag 07) is. This opens up some interesting possibilities for reversing flag patterns. Section 4H has more information on ATOX and XTOA, and the relationship between bits and decimal values.

The e register (address 00F) contains a bit map for the shifted keys which is identical to that used in register ⊢ (R for ZENROM users). The last part of Section 5A shows how to alter an existing key assignment to any one- or two-byte function using RAMED. ZENROM owners who do not have a CCD Module or PPC ROM don't really need to know the hex keycodes or have synthetic key assignment programs to make synthetic assignments. By altering an existing assignment, the appropriate bit is set for you in the bit map. Another way to reconstruct the proper bit maps is to read in any program from mass storage, extended memory, wand, or magnetic card. See XFME pages 193-198 for more details. A detailed knowledge of the internal bit map structure is not necessary to use this method.

The three rightmost nybbles (2, 1, and 0) of register e are used to keep track of the PRGM mode line number. After pressing RTN in run mode or when stopping by encountering an END with no pending returns, the value is set to 000. While a program is running or whenever the line number needs to be updated, the value will be FFF. Since the HP-41 is a 2.2K byte (main memory) machine, the processor knows this number (4095 decimal) cannot be a legitimate line number. Even a full 4K page of ROM cannot contain a program of more than about 4080 lines.

Unlike BASIC programs, line numbers in the HP-41 are not a part of the program. This is fortunate, because it would probably double the byte count of your programs. Line numbers are internally generated solely for user convenience.

Extended Memory

The memory of the Extended Functions Module (XFM) is built into the HP-41CX, and is available in a separate module for the other HP-41 models. Look between hex addresses 0BF and 040 (191 to 64 decimal) in Figure 1.2. Not all of the 128 registers contained within this memory space are available for data, programs or text (ASCII) files. A certain amount of overhead is required to keep things in order, such as maintaining the illusion that this memory is one continuous block that includes any existing Extended Memory modules.

After Extended Memory (XM) is cleared, executing EMDIR shows 124 registers available. The bottom register in the memory of the Extended Functions module (hex address 040) contains a "header" which links XM to any existing Extended Memory modules. Another register is filled with seven FF (decimal 255) bytes to mark off the end of occupied XM. This leaves 126 registers available. However, two is subtracted from this count because any file created requires one register for the file name and another to hold the file type and the number of bytes and number of registers the file takes up. So the count returned by EMDIR is exactly the number of registers available for CRFLD or CRFLAS (CReate FiLe -- Data, or AScii).

However, for program files there is a minor exception which makes this count inaccurate. One byte is added after the END instruction in a program file for a checksum. This byte is the sum of all of the bytes within the program, including the END, modulo 256. Since this extra byte is added, don't be surprised when your 112 byte program uses 17 registers of XM (113/7) instead of 16.

Program files within the Extended Memory of the XFM may be executed synthetically by constructing the proper pointer and storing it in register b with STO or ASTO. However, you need to know what you're doing, or you can make the program file unreadable by GETP or GETSUB. Running the program after it is saved in XM is OK if all of the GTO and XEQ instructions were executed (or SST'ed) before it was saved. This is known as compiling the branching instructions. (See SPME pgs. 59-60). If they are not compiled, running the program in XM will compile them. This will change the byte values of the branching instructions, making the checksum incorrect. Trying to read the program into main memory after that will show CHKSUM ERR, and the program will not be read in.

Depending on the number of Extended Memory modules (0, 1, or 2), XM may contain a total of 128, 367, or 606 total registers. Due to the fact that the bottom register of each XM device is used to link it to the next, and one register containing FF bytes is needed, this translates into 126, 364, or 602 registers available. When the directory is empty, the count returned by EMDIR will always be two less than these numbers. For further information on XM, see Keith Jarett's "HP-41 Extended Functions Made Easy."

ROM

ROM is an abbreviation for Read Only Memory. All HP-41s contain internal ROMs which have instructions that tell the processor what to do. These instructions are not the same as those used by RAM in PRGM mode in three fundamental ways.

First, ROM uses ten-bit bytes which are also called words. Since they are ten bits long, they cannot be expressed by just two hex nybbles. Serious programmers can study the documentation HP made while developing the internal (and other HP) ROMs. This material, called the "source code", or VASM, is NOt MAnufacturer Supported (NOMAS). The individual instructions are in octal (base 8) notation in the early VASM listings, but they are in hex if you get a more recent version. The VASM listings are available from PPC (see Appendix B).

Second, these instructions are written in machine-code (M-Code). This applies to the internal ROMs, Extended Functions, Time, HP-IL, and others. Not all ROMs are in M-Code; many contain the user code that you are familiar with. As an example, all of the Wand functions are in M-Code except "WNDTST". For a good introduction to M-Code, see Ken Emery's book "HP-41 M-Code for Beginners".

The third major difference between ROM and RAM is the direction of program execution. Whereas RAM decrements the address (with execution moving from higher to lower addresses), ROM increments the address as the program runs. This lower-to-higher scheme has little effect on you unless you start programming in M-Code, but you should be aware of it. Compiled GTO's and XEQ's in ROM are also different since there are no register boundaries. The jump distance is recorded simply as a number of bytes (up to ±127).

The HP-41 has four ports for external ROMs and devices which contain ROM. These ROMs can have a mixture of M-Code programs, data, and user code programs. There are 31 possible XROM numbers to identify the ROM: 01 through 31. No XROM may have ID number 00 (which identifies an empty port), and two ROMs may not be plugged in simultaneously if they have the same ID number. See SQRG p. 15-19 and 22-30. Each 4K block of ROM must have its own ID number, and it may contain up to 64 functions.

Whenever you key in an Alpha XEQ instruction, the HP-41 quickly checks Catalog 2 (and then Catalog 3). If a matching label is found in CAT 2, the instruction placed in memory is a two-byte XROM function. Removing the peripheral (ROM) from its port will cause this line to be shown as XROM xx,yy , where xx is the ROM ID number and yy is the function number within the ROM. Many synthetic two-byte key assignments will preview as an XROM until the key is released. See SQRG p. 14, ZENROM manual p. 31, and SPME p. 58-59 and 81-83.

CHAPTER TWO - STRUCTURED PROGRAMMING

2A. OUTLINING THE PROBLEM

The most crucial step in writing a program is to define what the program is supposed to do, and the method it will use to perform the function. Often there are several ways of doing the same job. The approach you take will usually have the most noticeable effect on the length and speed of the program. "Byte shaving" techniques are not nearly as effective as good algorithm design.

As a first step, make a list of the inputs your program will need. If there are only two or three numeric inputs, you may want to assume they are in the stack when program execution begins. For user convenience, or if there are many inputs, you may want to use Alpha prompts. Then decide what kinds of outputs the program will give. A few special programs won't have any direct outputs at all, but this isn't often the case. List the outputs, where they go, and the format they need to be in.

Next, determine what modules and peripherals the program needs. If a printer will be used, you need to keep this in mind as you write the program. Your program documentation should mention all of the devices that the program requires, or has optional provisions for, above and beyond a "bare" HP-41C. (These days it's probably OK to assume a CV or quad memory if you need to.)

If the program is divided up into short sections or if it isn't very long, you should try to preserve all or part of the stack if that's practical. This is especially valuable if the program may be used by another program as a subroutine. It's strictly a matter of your judgement as to whether this is important. It generally takes a few extra bytes to save part of the stack. Nevertheless, it's considered good programming practice to do so.

List the different pieces of information used by the program, what they are, and exactly where they belong. This includes the use of data registers, status registers (see Section 3A), Extended Memory files, etc. The more complicated the program, the easier it is to lose track of data. This won't be necessary for a simple program which only uses the stack. Try to avoid using any data registers if the program will be executed as a subroutine. If data registers are used, make a note of the minimum SIZE and what each register is used for (in case you want to change the program later). Some of these details might seem trivial now, but you probably won't remember them in six months, so <u>write it down</u>.

Before you begin to write a complex program, you need to write down the steps needed to accomplish the work. An example of this would be to write down the process you would go through to do the task by hand. It's helpful to break the job down into small, manageable chunks. It makes the job of translating the process from words into program steps a lot easier. If there is a formula that applies to your problem, write it down as one of the steps. You may end up breaking it down into smaller sections and rearranging parts of it when you translate it into program steps. This will depend on the formula and how complex it is.

With all of the preliminary work done, you should have a clear idea of what you need to do. In addition, you may want to use subroutines or a technique known as modular programming. See the Sections 2B and 2C, which follow this section.

Since you already have the program broken down into a number of small tasks, begin by looking at the first step. You may need to add a section prior to this first part to set up the stack or data registers, to test for the existence of the highest numbered data register needed, or to trap other errors. Once the program performs the needed initialization, add the program steps needed to carry out the first task. You'll want to single-step through the program at this point to make sure everything works as planned. Then continue to add to the program one section at a time.

If you have trouble translating one of the tasks you wrote down into program steps, there are several techniques you can try. First, try breaking it down into smaller steps. You may also find your work can be simplified by rearranging the order of the steps or combining two of them. If you're still stuck, try to think of completely different ways of doing this same job. At some point, you may have to revise the constraints you put on the program in order to get the job done.

As the program increases in length, it's a good idea to add a STOP instruction to the end of the program before adding new lines. This lets you run the program through the section that already works. From there, you can single-step the new section to check its performance. Remove the STOP function after the new section works properly. Repeat this technique for each of the tasks you listed.

After you complete the program, you should add a table showing how the stack is used. Once again, this is especially valuable if the program might be used by another program as a subroutine. Be sure to make a note of any status registers that are altered, including Alpha. You can look at the programs in the PPC ROM User's Manual or the forms used by the Hewlett-Packard Users' Library for good ideas on areas to cover in your program documentation.

2B. SUBROUTINES USING XEQ

The three types of XEQ instructions are known as local, indirect (IND) and Alpha. Nonsynthetic Alpha XEQ functions range from three to nine bytes in length. They are composed of an ordinary text line preceded by a byte with decimal value 30 (1E in hex). Because of this simple structure, the jump distance from the XEQ instruction to the label cannot be compiled within the bytes of the XEQ function. This makes Alpha XEQs slower than local XEQs, which do compile this information. When a local XEQ function is encountered in a program for the first time, the direction and distance to the matching label are recorded within the XEQ. Twelve bits are used to record the the relative jump distance, and one bit indicates the direction. See SQRG page 39 and EYHP page 66.

The Alpha XEQ will vary in execution speed according to the position of the label searched for. Alpha labels are a part of the Catalog 1 chain of global labels and ENDs. The search for the correct label begins at the bottom of Catalog 1 and proceeds upward. If the label being executed is near the bottom of Catalog 1, the search won't take long. But when the label is near the top of the catalog, it can take anywhere from one-fourth of a second to well over a second to find the matching label. For this reason, you should use a local label if the label being executed is within the same program. You can accomplish this in an existing program either by replacing the Alpha LBL and XEQ with local functions or by adding a local label next to the existing Alpha label and using a local XEQ. The advantage of this is that the local XEQs within the program will be speedy, and access by other programs is not lost in the process. Refer to Section 3E for more information on labels.

An XEQ IND or GTO IND instruction also cannot be compiled. All indirect XEQ and GTO functions are two bytes long. The first byte has a decimal value of 174, or AE in hexadecimal. This first byte is the same for indirect GTOs and XEQs. The difference between the two types of instructions lies in the fact that indirect XEQs have postfix values from the second half of the byte table. Because of the many similarities between these two functions, XEQ IND will not be covered in detail here. Refer to Section 2D for information and examples of indirect branching using GETKEY followed by GTO IND.

Local XEQ instructions are always three bytes long. The first byte of a local XEQ function has a decimal value from 224 to 239 (E0 to EF), depending on the value of the compiled jump distance. The second byte is only used for this compiled information. The third byte contains two pieces of data. The leftmost bit indicates whether the direction of the compiled jump is up (0) or down (1) from the location of the XEQ instruction. The remaining seven bits represent the number of the local label that the HP-41 will search for. This number can range from 0 to 127, with 102 to 111 showing as A to J and 112 to 127 displaying as T to e. As a consequence of this postfix structure, which is identical to that of a three-byte GTO, a program cannot branch to synthetic indirect labels such as LBL IND e (207, 255). Any two-byte label whose postfix is less than 128 or any one-byte label can be used with XEQ.

The advantage of this structure for three-byte XEQ (and GTO) instructions lies in the fact that twelve bits are used to represent the magnitude of the compiled jump. This allows a compiled jump of up to 511 registers and 6 bytes. Since this number exceeds the maximum allowable main memory size (320 registers), you never need to worry about the branching XEQ being too far from its matching label. The jump distance will be compiled within the instruction when it is first executed, and this makes subsequent executions very quick. The equivalent of GTO and XEQ instructions in ROM use a sixteen-bit field to store the jump address. This allows branching anywhere within the HP-41's 64K ROM address space.

If compiled jump information is not present (for example, the first time the GTO or XEQ instruction is encountered), the HP-41 must search for the corresponding label. The search for a matching local label begins at the branching function (GTO or XEQ), and continues downward through the program until the label is found, or an .END. or END is encountered. If this happens, the search continues from the first line of the program until the matching label is found, or the original address of the branching instruction is reached. If no matching label is found, the NONEXISTENT error message is displayed and program execution halts. Otherwise, the distance is compiled (stored within the GTO or XEQ instruction itself) and program execution continues from the label.

Any time a change is made to a program, the HP-41 resets all of the jump distance bits to zero within branching instructions. This process is known as decompiling. This is discussed in detail in Section 4G, where techniques are presented to avoid losing the compiled information contained within a program read in from a mass storage device. Avoiding decompiling will save the time needed to recompute the jump distances as well as the effort it takes to run or SST each local branching instruction.

XEQ instructions which branch to local labels compile jump distance information, while other types of XEQs do not. For this reason, they are always faster than indirect or global XEQs. Not only are they faster, but they can be up to four bytes shorter than a global XEQ instruction. Local XEQ functions are generally better than these other types when branching within the confines of a single program.

There is a synthetic key assignment which is very useful for local XEQ functions. Its "MK" inputs are 0, 229 (hex 00, E5 for ZENROM users). With this key assignment, you can create XEQ instructions with byte values 229, 0, nnn, with nnn being equal to the decimal number you key in. The number keyed in can range from 000 to 255. Numbers in excess of this amount are interpreted by the HP-41 modulo 256. That is, only the remainder after division by 256 is used. In PRGM mode, the function created is utilized normally. However, in run mode, postfixes from the second half of the byte table correspond to the synthetic two-byte indirect labels mentioned before. As an example, filling in the prompt with 255 will branch to LBL IND e, which can't be reached by any instruction executing in PRGM mode.

When a key assigned with 0, 229 is pressed, the distinctive prompt "$T+N IA _ _" appears. Fill in the prompt with the decimal value of the desired postfix. Using values 100 and 101 will give instructions that display as XEQ 00 and XEQ 01, respectively. Don't let their appearance deceive you. These functions will execute synthetic labels 100 and 101 (which also display only the last two digits!). The decimal values 102 to 127 create XEQ instructions for labels A through e. See the left side of the middle row in each appropriate box of the byte table for the suffix that will be shown for these values.

The XEQ functions are very useful because they allow you to make use of a subroutine without disturbing the flow of your program. Program execution continues (after the subroutine has finished) at the line following the XEQ instruction. Any series of repeated program steps can be made into a subroutine.

Using XEQ in this way may not always be desirable. If speed is of the utmost importance, you may want to leave repeating sequences of instructions as they are. If you're interested in saving bytes, the subroutine should contain nine or more bytes if it is to be executed twice, six or more bytes when executed three times, and at least five bytes when executed four to six times. This byte count does not include the LBL or RTN instructions which are the first and last lines of the subroutine. See page 31 of Jeremy Smith's Synthetic Quick Reference Guide for a complete table of byte savings by creating a subroutine.

To modify an existing program that contains a series of instructions that are repeated, begin by writing down the instructions. Choose a numeric label that is not used by the program. If you can, use one of the one-byte labels, LBL 00 to 14. Replace each occurence of the repeating program lines with a local XEQ instruction which matches the label you selected. Now decide where you want to put the subroutine. The usual place is at the end of the program. Be sure to add a RTN function between the last line of the program and the subroutine. The

RTN prevents program execution from dropping into the subroutine. Then key in the subroutine's numeric label followed by the same series of program steps that you wrote down earlier. The END or .END. will function as a RTN for the subroutine.

Another place a subroutine can be added is after an unconditional GTO. This is a GTO which is not preceded by a conditional function which may cause the GTO to be skipped. In this case a RTN is not needed ahead of the subroutine's starting label.

The only advantage in placing the subroutine as near to the XEQ function as possible is that that time for the initial search is reduced if the subroutine is below the XEQ instruction and nearby. Subsequent executions do not need searches because the distance information is compiled.

2C. DEVELOPING THE MODULES

Modular programming is a helpful approach for you to use whenever you are constructing a large program. The job that the program does first needs to be divided into more than one section. Using this approach goes beyond just saving bytes by replacing program lines with an XEQ instruction and a subroutine. The idea behind modular programing is to develop subroutines which can be used as building blocks for other programs. This approach also speeds debugging; it is simpler to debug two 100-line programs than one 200-line program.

The utility programs contained within the PPC ROM are fine examples of modular programming. The programs are structured so that they can be used as subroutines. In fact, many of these programs are used extensively as subroutines by other programs in the PPC ROM. If you have a copy of the "PPC ROM USER'S MANUAL", you can find ample evidence of this by looking on page 6 under the "CALLS" heading or in the individual routines' documentation in the "TECHNICAL DETAILS" section.

For example, the "PD" routine (page 358 of the PPC ROM Manual) calls both "2D" and "QR". The "QR" routine in particular is used by many of the programs in the PPC ROM, as you can see by checking page 6 of the manual. As a more subtle example, consider line 58 of "PD", which is LBL 14. This is an alternate entry point for "PD", which allows the routine above it, "RT", to use the last 8 lines of "PD".

Though your subroutines need not be written up as thoroughly as the routines in the PPC ROM, you should write down the inputs required, the task performed, the outputs provided and any and all status registers affected. If your routine uses just the stack registers, you can show this easily with a small chart listing the contents of T, Z, Y, X, and L before and after execution.

As an example, let's say you are writing a program within which, at two different places, the rightmost character in Alpha needs to be converted to a decimal number. You have the Extended Functions module or an HP-41CX, which has Extended Functions built in. In the first version of the program, you used the sequence E, CHS, AROT, RDN, ATOX in the two places where this operation was needed. But during testing, you found out that nulls were not being handled correctly. If the rightmost character was a null, the result of this sequence of instructions was not zero as it should be. Instead, the null disappeared when rotated to the left end of Alpha, and the leftmost character was converted to decimal.

It is clear that making this function into a subroutine will save bytes, because the necessary changes to the sequence (which is 7 bytes already) is going to push the byte count over the nine byte minimum. As you may recall from the previous section, a subroutine executed twice should contain nine or more bytes. So you replace E, CHS, AROT, RDN, ATOX with XEQ 10 in both places, go to the last line of the program and key in RTN, LBL 10.

The revised subroutine needs to meet several requirements. The contents of ALPHA must remain unchanged, except for the decoded character, which is removed. The original contents of X and Y (when the subroutine begins) must be preserved, and end up in Y and Z. The decimal character value needs to be in X. Here is one solution:

```
ALENG     Check the length before rotation
DSE X     Decrease X by one and skip rotation if X was 1
AROT      Rotate the righmost character to the left end
ALENG     Get the new length of Alpha
-         and subtract for comparison.
X=0?      If X is zero, a null was dropped during the
          rotation and zero is the result.
RTN       Return with former X and Y in Y and Z.
RDN       Otherwise get rid of the negative one,
ATOX      and convert the character to decimal
RTN       The result is now in X; former X, Y in Y, Z
```

There are always several ways to accomplish the same result in a short routine like this. Often there are two different approaches that achieve the minimum byte count. The important thing is to check that your routine does in fact perform correctly, even if the inputs are unusual. In the case above, the original routine failed to correctly translate a null character. One of the hardest parts of developing a program is to make sure that you have considered all these special cases. This is the source of the worst kind of "bug"; one that lurks in the background, ready to surprise you long after you think the program is working.

Here are a few special cases that your programs should be able to handle, unless you choose to exclude some of them by explicitly stating the restrictions in the program's documentation:

Insufficient SIZE

Use a RCL nn instruction at the start of the sequence, or something fancier, like XROM "VS" from the PPC ROM.

Inputs outside allowed range or of the wrong type

Generating the appropriate error message is normally not a problem unless your program sets flag 25 (for example, to check the SIZE) before it checks for input errors.

Nonexistent files

It is best to check for these before you get too far into the program.

Flag 25 set

Should not disrupt the program. This is perhaps the toughest case to handle. You can take the easy way out by putting a CF 25 instruction near the top of the routine. In the routine above, however, the status of flag 25 is preserved because none of the instructions can clear flag 25 when performed in the given sequence.

When you are writing a particular module that will only be used as part of a larger program, it is fair to make assumptions about the SIZE, input characteristics, and flag settings, if you make sure that these assumptions will be checked earlier in the larger program. Naturally, these assumptions should be documented if you hope to avoid lurking bugs the next time you use the module for another purpose.

2D. PUTTING THE MODULES TOGETHER

After you have a set of program modules, you can put them together in a few different ways. The most obvious way is to string the modules together end to end. This works fine as long as the flow of your program is linear, or "straight through". It also allows one or more modules to be repeated several times. Just enclose them in a loop, with a label at the top and a test instruction and GTO at the bottom.

However, this method needs to be modified if a module is used in more than one place in the program. One way you can handle this situation is to put the module at the end of the program, and put an XEQ instruction at each place in the program where you need to use the module.

But what if you don't know ahead of time which modules will be needed, or in what order they will be used? Then you need to use another trick. First, give each module a different numeric local label. At each step when a new module is needed, you need to specify the label number to identify which module is to be used next.

There are two ways to get the label number of the module to be used. If the module selection depends on the output of the previous part of the program, your program can compute the label number of the needed module, or it can look up the number from a stored table by using a RCL IND instruction. If the module selection depends on the user of the program, he can be prompted to press a key (more about this later) which will make the selection.

Once you have the label number that identifies the module to be executed, a simple XEQ IND instruction will execute the module that you want. Assuming you have the label number in the X register, you could use an XEQ IND X instruction to call the module. But your module may expect one of its inputs in the X register. A better solution is the sequence RDN, XEQ IND T. Another approach is to store the label number in a data register (say register 00), then load the required data in the stack, then XEQ IND 00. The XEQ IND instruction is one of the most powerful programming tools that the HP-41 has to offer. All advanced HP-41 programmers should become fully familiar with the capabilities of this function.

Sometimes you can use GTO IND in place of XEQ IND. For example, if all the modules conclude with a GTO 00 instruction (including all possible input conditions!), then all modules will end up at LBL 00. If, in addition to this, there is only one point from which the modules are called, then this point can be put after the LBL 00 instruction. In this case the GTO IND instruction is preferred. It conserves the subroutine return stack. The GTO 00 instructions at the end of each module take the place of the more typical RTN instructions.

Now back to a point raised on the last page. That is, how can you conveniently get the label number for a module by prompting the user of the program? The user certainly is not going to remember which module has which label! But the user of the program can identify a particular key with a particular module. To bridge the gap from key to label number we use the GETKEY function (provided in the Extended Functions module and in the HP-41CX). When you put a message in the display and execute GETKEY, the calculator waits for the user to press a key, then it puts the row/column keycode in the X register. (The row/column keycode is two digits. The tens digit is the row number, with row 0 being the ON/USER/PRGM/ALPHA switches, and row 8 being the bottom row. The ones digit is the column number, column 1 being the leftmost.)

The numeric labels for the modules can be selected to match the row/column keycodes obtained by the GETKEY instruction. This method is used in some of the programs in "HP-41 Extended Functions Made Easy".

Incidentally, the GETKEY function has a cousin, GETKEYX, which is found only on the HP-41CX. If you have an HP-41CX, you should check your manual for a description of GETKEYX. The additional features provided are a variable wait (compared to the fixed 10-second wait provided by GETKEY), XTOA-compatible character codes for keys pressed in ALPHA mode, and an option to process the key on the downstroke rather than on the upstroke. Of these, the character codes are the most useful. They can, however be calulated from the normal GETKEY keycodes.

2E. A MODULAR PROGRAMMING EXAMPLE

In this section we will develop a program to automatically solve triangles. You will be able to input any known parts of a triangle, and the program will complete the triangle if possible, finding the values of all the missing parts.

Creating this program is clearly a tall order. But let's do it step-by-step:

1) decide how you want the program to appear to the user,
2) outline the logic flow that will be required,
3) identify the modules of the program,
4) program the modules,
5) assemble, test, and debug the program.

Program appearance

First, let's consider the data management aspects of the program, and establish preliminary register usage and input/output characteristics. This application is probably best suited to use of the top-row keys both to enter data and to recall the calculated values. A reasonable choice for top-row key functions is:

Angle A_1	Angle A_2	Angle A_3		Initialize
Side S_1	Side S_2	Side S_3	Area	Solve

Angle 1 is opposite Side 1, Angle 2 is opposite Side 2, and Angle 3 is opposite Side 3. The sides of the triangle are stored in data registers 01, 02, and 03, and the angles are stored in registers 04, 05, and 06. This allows the sides to be recalled using the natural keystroke sequences RCL A (which gives RCL 01), RCL B (02), and RCL C (03). To start the program, press SHIFT e. This clears all angle and side information. Then key in a triangle element and press the corresponding key. Continue until all known elements are entered. Then press the [E] key to solve for the remaining elements. The program will fill in the missing values or indicate that the data supplied is not sufficient for a solution.

Program Logic Flow

Now let's look at the logic flow of the solution portion of the program. This portion, which is the heart of the program, starts by finding out how many sides have been supplied. (A zero value indicates to the program that a result needs to be calculated. A nonzero value indicates that an input has been supplied.)

While checking the data elements, the program needs to rotate the triangle. This could be done by interchanging data register contents, but then we would have to keep track of the rotations so that the original configuration could be restored.

A cleaner approach is to use the indirect addressing capability of the HP-41. Register 00 holds a pointer to select one of the three sides or angles for recall or storage. To rotate the triangle, we need only increment the pointer in the sequence 1, 2, 3, 1, 2, 3, etc.

Because of the program's complexity, a special notation will be used. For each of the three sides, an S indicates that a nonzero value is present, a 0 indicates that a zero value is present, and x indicates the value has not been checked. For example, the notation S0x means that one nonzero side is followed by a zero side, with the third side untested as yet. The pointer is set to the third side. A similar notation is used for the three angles of the triangle.

The cases that the program can solve are:

SSSxxx	(three sides and any, all, or no angles)
SS0xxA	(two sides and the included angle)
SS0AA0	(compute the included angle and solve as SS0AAA)
SS0A00	(two sides and a non-included angle; may have two solutions)

SS00A0   (same as above)

S00 with at least two angles

(compute the third angle; use law of sines to solve)

The above cases include any triangle that can be rotated to match these conditions.

Insufficient data cases are:

000xxx   (need at least one side)

SS0000

S00 with less than two nonzero angles

Setting up a decision strategy is the most important part of writing this triangle solution program. Here is the program's decision tree:

```
Find a nonzero side
    All 3 sides zero... give the message "MORE DATA" and halt.
    Nonzero side found... continue with S̲xx case.
Rotate and check the next side
    If zero... continue the S0̲x case.  Check the next side.
        If nonzero... continue the S0S̲ case.
            Rotate the triangle into the SS0̲ position.
            Continue as the SS0̲ case (below).
        If zero... begin the S00̲ case.  Find two angles
            and calculate the third, using:
                $COS\ A_3 = -\ COS\ (A_1 + A_2)$.
            If two angles are not found, halt with
            the message "MORE DATA". Otherwise continue with
            the S00̲ case after all angles are present.
            Use the law of sines:
                $S_j = SIN\ A_j * S_i/SIN\ A_i$
            to find the second and third sides.

    If nonzero... continue the SS̲x case.  Check the next side.
        If zero.. continue the SS0̲ case.  Start checking angles.
```

If the third angle is nonzero... SS0 xxA case.

Use the law of cosines:

$$S_3 = \text{SQRT}(S_1{}^2 + S_2{}^2 - 2S_1S_2 \text{ COS } A_3)$$
$$= \text{SQRT}((S_1 - S_2 \text{ COS } A_3)^2 + (S_2 \text{ SIN } A_3)^2)$$

to find the third side. Finish the solution by using the side-side-side (SSS) subroutine to fill in any missing angles.

Otherwise if the first angle is nonzero... Ax0 case.

If the second angle is nonzero, use A1 and A2 to compute A3. Then use the law of cosines to find S3 as above.

If the second angle is zero, use the side-side-angle (SSA) subroutine to compute the third angle. Two solutions are possible. (The two solutions may be very close to each other in the case of a near-right triangle. Exact equality of the two solutions is virtually impossible, given the 10-digit accuracy of the HP-41.) The second solution, called Solution 1 in this program, is a smaller triangle that fits within the Solution 0 triangle. The Solution 1 triangle is always more obtuse than the Solution 0 triangle, although the Solution 0 triangle may also be obtuse.

In this case, the program must ask the user to select a solution. The prompt OBTUSE? 1/0 asks whether Solution 1 or 0 is desired. Then use the law of sines to find the third side. Finish the solution by using the SSS subroutine to fill in the missing angles.

Otherwise if the second angle is nonzero... 0A0 case.

Rearrange inputs to the SSA subroutine, and use it to compute the third angle. Then use the law of sines to find the third side. Finish the solution by using the SSS subroutine to fill in the missing angles.

The SSS subroutine computes each angle in turn, using the law of cosines formula:

$$\begin{aligned} \text{COS } A_3 &= ((S_1{}^2 + S_2{}^2) - S_3{}^2)/(2\ S_1\ S_2) \\ &= [1 - (S_3{}^2/(S_1{}^2 + S_2{}^2))]/[2S_1S_2/(S_1{}^2+S_2{}^2)] \\ &= [1\text{-}(S_3/\text{SQRT}(S_1{}^2+S_2{}^2))^2]/[\text{SIN 2ATAN } S_1/S_2] \end{aligned}$$

The triangle is rotated after each computation, so that the same formula can be used for the next computation. After three rotations, an ISG counter halts the loop.

The SSA subroutine starts with S_1, S_2, and A_1.
The first task in this subroutine is to use A_2 to find A_3.
The angle A_2 can have two possible values, each of which is a solution to:

$$\text{SIN } A_2 = A_2 * \text{SIN } A_1 \ / \ A_1.$$

If we let

$$\hat{A}_2 = \text{ASIN } (\ A_2 * \text{SIN } A_1 \ / \ A_1\ ),$$

then A_2 can either be $\hat{A}_2$ or $180 - \hat{A}_2$.
Since $A_3 = 180 - A_1 - A_2$, this yields two possible values for A3:

$$A_3 = 180 - \hat{A}_2 - A_1 \text{ or } \hat{A}_2 - A_1$$

The second value is often negative, and thus not valid. However, if it is positive, it represents an obtuse angle solution. In that case, the first value represents an acute angle solution. If $\hat{A}_2 - A_1$ is positive, the user must be prompted to determine whether an obtuse or acute solution is desired. Once A_3 is found, S_3 is computed, using the law of sines formula:

$$S_3 = \text{SIN } A_3 * S_1/ \text{ SIN } A_1.$$

As you can see from the last few pages of discussion, the first step in modular programming is the most important and often the most time-consuming. Proper attention to structuring the solution to the problem will save much time in programming.

Program Modules

The first module needed in this program is the data entry module. Here is a listing of that module, with comments:

```
01 LBL "TRI∡"
02 LBL A               ; Store or view Side 1
03 1                   ; Data is held in register 01
04 GTO 00              ; LBL 00 is used for forward branching
05 LBL B               ; Store or view Side 2
06 2                   ; Data is held in register 02
07 GTO 00              ; LBL 00 is used for forward branching
08 LBL C               ; Store or view Side 3
09 3                   ; Data is held in register 03
10 GTO 00              ; LBL 00 is used for forward branching
11 LBL a               ; Store or view Angle 1
12 4                   ; Data is held in register 04
13 GTO 00              ; LBL 00 is used for forward branching
14 LBL b               ; Store or view Angle 2
15 5                   ; Data is held in register 05
16 GTO 00              ; LBL 00 is used for forward branching
17 LBL c               ; Store or view Angle 3
18 6                   ; Data is held in register 06
19 LBL 00              ; LBL 00 is used for forward branching
20 RDN                 ; Put the data register number in T
21 FS?C 22             ; If a number was entered,
22 STO IND T           ; Put that number in the proper register
23 RCL IND T           ; Recall the value in any case
24 GTO 99 End          ; moves to end of program, for faster
                       ; response to the next top-row key
```

The next module computes the area after the triangle has been solved:

```
25 LBL D               ; Compute Area of the solved triangle
26 RCL 01              ; S1
```

```
27 RCL 02           ; S2
28 *                ; S1*S2
29 RCL 06           ; A3
30 SIN              ; SIN(A3)
31 *                ;
32 2                ;
33 /                ; AREA = S1*S2*SIN(A3)/2
34 GTO 99 End       ; moves to end of program, for faster
                    ; response to the next top-row key
```

The last of the "easy" modules clears the data registers to initialize the calculator in preparation for a new triangle:

```
35 LBL e            ; Initialize for a new triangle
36 CF 22            ; Clear number entry flag, so that
                    ; LBLs A-C and a-c will work properly
37 CLX              ;
38 STO 01           ; Clear registers 01 to 06
39 STO 02           ;
40 STO 03           ;
41 STO 04           ;
42 STO 05           ;
43 STO 06           ;
44 GTO 99 End       ; moves to end of program, for faster
                    ; response to the next top-row key
```

In developing the major portion of the program, the decision tree that determines which solution to apply, mnemonic words will be used in place of the labels. Numeric values will be attached in the listings, but these values, and the line numbers, are normally only filled in after you have completed developing the program on paper.

```
45 LBL E            ; Begin solving the triangle!
46 1.003            ; Set up an ISG pointer for the count
47 STO 00           ; 1,2,3, 1,2,3 in register 00
```

```
                               ; Note: in retrospect it seems that a
                               ; DSE pointer for a count of 3,2,1 3,2,1
                               ; would save bytes, at line 46 and in
                               ; the incrementing subroutine,
                               ; at some expense in clarity of the
                               ; program logic.  This change, if
                               ; desired, is left as an exercise.
48 LBL 01  Findside            ; This loop finds the first nonzero side
                               ; of the triangle
49 RCL IND 00                  ; Recall the current side
50 X≠0?                        ; If it is not zero,
51 GTO 00 Skip                 ; Skip forward
52 ISG 00                      ; Increment the counter to the next side
53 GTO 01 Findside             ; Try the next side

54 LBL 02  Insufficient        ; Termination point if not enough data
55 "MORE DATA"                 ;
56 AVIEW                       ; Display "MORE DATA" message
57 GTO 99 End                  ; Terminate, ready for more data entry

58 LBL 00  Skip                ; First nonzero side found
                               ; Without loss of generality, this will
                               ; be assumed to be Side 1 from here on.
59 XEQ 12 +Si                  ; The +Si subroutine increments the side
                               ; counter i in register 00, then recalls
                               ; Side i (now the current side).
                               ; Here we are checking the next side,
                               ; to see if it is nonzero.
60 X≠0?                        ; If the second side is nonzero,
61 GTO 04 SS̲x                  ; go to entry point that solves the
                               ; SS̲x case, in which sides 1 and 2
                               ; are nonzero, with side 2 being the
                               ; current side.
62 RDN                         ; Remove the zero value for the second
                               ; side.  Put the first side value in X.
```

```
63 XEQ 12 +Si           ; Recall the next side.
64 X≠0?                 ; Otherwise, if third side is nonzero,
65 GTO 05 S0S           ; go to entry point that solves the
                        ; S0S case, in which sides 1 and 3
                        ; are nonzero, with side 2 being zero,
                        ; and side 3 the current side.

                        ; Otherwise, we have the S00 case,
                        ; in which at least two nonzero angles
                        ; are needed.
66 XEQ 15 +Ai           ; The +Ai subroutine increments the
                        ; counter i in register 00, then recalls
                        ; Angle i from register i+3.
                        ; Here we are recalling angle 1,
                        ; to see if it is nonzero.
67 X≠0?                 ; If the first angle is nonzero,
68 GTO 00 Skip          ; skip to the entry point for the
                        ; Axx case, in which angle 1 is
                        ; nonzero, and is the current angle.

                        ; Otherwise, we have 0xx case,
                        ; in which the next two angles
                        ; must be nonzero.
69 XEQ 15 +Ai           ; Recall angle 2
70 X=0?                 ; If angle 2 is zero,
71 GTO 02 Insufficient  ; give the message MORE DATA
72 XEQ 15 +Ai           ; Recall angle 3
73 X=0?                 ; If angle 3 is zero,
74 GTO 02 Insufficient  ; give the message MORE DATA

                        ; Otherwise, we have the 0AA case.
                        ; At this point, angle 3 is in X,
                        ; and angle 2 is in Y.
75 XEQ 13 LastAngle     ; Use the LastAngle subroutine to find
                        ; the missing angle.  The LastAngle
                        ; routine will conclude with the pointer
                        ; set to the formerly missing angle:
                        ; AAA.
                        ; Now, angle 1 is in X, angle 3 in Y.

                        ; These next three lines allow a later
                        ; part of the program to finish this
                        ; case.  Don't bother trying to
                        ; understand them at first reading.
                        ; They are typical of the parts of
                        ; the program that you must fill in
                        ; after the initial draft.
76 XEQ 10 -i            ; Decrement the side/angle counter.
                        ; The case is now S00, with all
                        ; angle values filled in.
77 X<>Y                 ; Put angle 3 in X, angle 1 in Y.
78 GTO 03 S00           ; Solve the S00 case, with
                        ; angle 3 in X, and angle 1 in Y.
79 LBL 00  Skip         ; Resume the Axx case, in which
                        ; angle 1 is nonzero, and is the
                        ; current angle.  (Sides are S00.)
80 XEQ 15 +Ai           ; Recall the second angle.  After
                        ; this step, angle 2 is in X, and
                        ; angle 1 is in Y.
81 X=0?                 ; If the second angle is zero,
82 GTO 00 Skip          ; skip to the entry point for the
                        ; A0x case, in which angle 1 is
                        ; nonzero, angle 2 is zero, and is
                        ; the current angle.

                        ; Otherwise, we have AAx case,
                        ; in which the next angle can be
                        ; computed from the first two.
83 X<>Y                 ; Put angle 1 in X.
```

```
84 XEQ 13 LastAngle      ; Use the LastAngle subroutine to find
                         ; the missing angle.  The LastAngle
                         ; routine will conclude with the pointer
                         ; register set to the formerly missing
                         ; angle: AAA.
                         ; Now, angle 3 is in X, angle 1 in Y,
                         ; the same conditions we set up at
                         ; line 76 above.  As there, we continue
85 GTO 03 S00            ; by branching to the S00 case.

86 LBL 00  Skip          ; Continue the A0x case.
87 RDN                   ; Put angle 1 back in X for later use.
88 XEQ 15 +Ai            ; Recall angle 3 to X.
89 X=0?                  ; If angle 3 is zero,
90 GTO 02 Insufficient   ; the data is not sufficient.
                         ; Otherwise, the case is A0A.
91 XEQ 11 +i             ; Increment i to get A0A.
92 XEQ 13 LastAngle      ; Use the LastAngle subroutine to find
                         ; the missing angle.  The LastAngle
                         ; routine will conclude with the pointer
                         ; register set to the formerly missing
                         ; angle: AAA.
                         ; Now, angle 2 is in X, 3 in Y, 1 in Z.

                         ; To transform these conditions to
                         ; the same conditions we set up at
                         ; line 76 above, we need to set the
                         ; pointer to side/angle 3, and put
                         ; angle 3 in X, angle 1 in Y.
93 RDN                   ; Angle 3 now in X, angle 1 in Y.
94 XEQ 11 +i             ; Increment the counter.  S00 case.
95 LBL 03  S00           ; Entry point for the S00 case.
                         ; Angle 3 is in X, angle 1 in Y.
                         ; Use the law of sines:
                         ; S_j = SIN A_j * S_i/SIN A_i
```

```
96 SIN              ; SIN A3
97 XEQ 12 +Si       ; Recall Side 1.
                    ; At this point the stack (XYZT) is:
                    ;  S1               sinA3     A1
98 RCL Z            ;  A1               S1        sinA3     A1
99 SIN              ; sinA1             S1        sinA3     A1
100 /               ; S1/sinA1          sinA3     A1        A1
101 *               ; sinA3*(S1/sinA1)  A1        A1        A1
                    ;    = A3           A1        A1        A1
102 LASTX           ; S1/sinA1          A3        A1        A1
103 XEQ 15 +Ai      ;  A2               S1/sinA1  A3        xx
104 SIN             ;  sinA2            S1/sinA1  A3        A1
105 *               ; sinA2*(S1/sinA1)  A3        A1        A1
                    ;    = A2           A3        A1        A1
106 STO IND 00      ; Store as Angle 2
107 RDN             ;    A3             A1        A1        A2
108 XEQ 11 +i       ; Increment angle counter, from 2 to 3
                    ;    A3             A1        A1        xx
109 STO IND 00      ; Store as Angle 3
110 RTN
```

The stack management instructions (X<>Y, RDN, and R↑) within the above decision portion of the program would normally be inserted at the last stage of program development, after all the modules are in place and their input/output characteristics are fully known. Stack management needs to be carefully documented along with the program steps, and even then it normally needs some fixing when you start testing the program.

The missing modules are now:

```
LBL 04  SSx         ; Solves the SSx case.
LBL 05  SOS         ; Solves the SOS case.
LBL 10  -i          ; Decrement the side/angle counter.
                    ; Preserve the values in X, Y, and Z.
```

```
LBL 11  +i              ; Increment the side/angle counter.
LBL 12  +Si             ; Increment counter; recall current side.
LBL 13  LastAngle       ; Compute third angle from the other two.
LBL 15  +Ai             ; Increment counter; recall current angle.
LBL 99  End             ; Terminate; prepare for new top-row
                        ; key input.
```

Let's work on the simpler subroutines now, labels 10 through 15. We'll throw in two more that will be needed later:

```
LBL 14  StoreAi         ; Store current Angle without
                        ; incrementing the counter.
LBL 16  Ai              ; Recall current Angle without
                        ; incrementing the counter first.
```

Again, the line numbers would not normally be part of your program development, which starts with pencil and paper.

Insert LBL 10-99 commented listings here

```
222 LBL 10  -i          ; Decrement the side/angle counter.
                        ; Preserve the values in X, Y, and Z.
                        ; In the 1, 2, 3 sequence, decrementing
                        ; can be done by incrementing twice.
223 XEQ 11 +i           ; Increment the side/angle counter.
224 LBL 11  +i          ; Increment the side/angle counter.
                        ; Preserve the values in X, Y, and Z.
225 X<> 00              ; Stack is:      i.003      y     z     t
226 3
227 MOD                 ;        i MOD 3 +.003      y     z     z
228 LASTX
229 SIGN
230 +                   ;       i MOD 3 +1.003      y     z     z
                        ; MOD 3 leaves i=1 and 2 unchanged,
                        ; but reduces 3 to 0.
231 X<> 00              ;                   x       y     z     z
232 RTN
```

```
233 LBL 12  +Si          ; Increment counter; recall current side.
234 XEQ 11 +i            ; Increment counter register.
235 RCL IND 00           ; Recall current side.
236 RTN

237 LBL 13  LastAngle    ; Compute third angle from the other two.
                         ; Stack starts as       A1     A2      z       t
                         ; Pointer starts at S2.
238 XEQ 11 +i            ; Increment counter to S3.
239 RCL Y                ;  A2         A1         A2      z
240 RCL Y                ;  A1         A2         A1      A2
241 +                    ;  A1+A2      A1         A2      A2
242 COS
243 CHS
244 ACOS                 ; ACOS(-COS(A1+A2))
                         ;  A3         A1         z       z
245 LBL 14
246 XEQ 00 Skip          ; Put counter+3 in T for use as an
                         ; indirect address to recall the
                         ; current angle.
                         ;  A3         A1         z       i+3
247 STO IND T            ; Store A3.
248 RTN

249 LBL 15  +Ai          ; Increment counter; recall current angle.
250 XEQ 11 +i            ; Increment counter
251 LBL 16  Ai           ; Recall the current Angle without
                         ; incrementing the counter.
252 XEQ 00 Skip          ; Put counter+3 in T for use as an
                         ; indirect address to recall the
                         ; current angle.
253 RCL IND T            ; Recall the current angle.
254 RTN
255 LBL 00  Skip         ; Put counter+3 in T.
                         ; Stack begins:  x       y       z       t
```

```
256 3                    ; 3      x      y      z
257 X<> 00               ; i      x      y      z
258 ST+ 00
259 X<> 00               ; i+3    x      y      z
260 RDN                  ; Restore stack:
                         ; x      y      z      i+3
261 LBL 99  End          ; Reset to line 00; prepare for new
262 END                  ; top-row key input.
```

Now, let's fill in the modules that handle cases for which more than one nonzero side is present. We must start with LBL 04, the SSx case, and LBL 05, the S0S case, although more modules will be needed. For example, LBL 07 will handle the SSS case.

```
111 LBL 04  SSx          ; Solves the SSx case.
                         ; Side 2 is in X, Side 1 in Y.
112 XEQ 12 +Si           ; Recall side 3.
113 X≠0?                 ; If side 3 is nonzero,
114 GTO 07 SSS           ; Solve SSS case.
                         ; Otherwise, solve the SS0 case.
115 RDN                  ; Put Side 2 in X, Side 1 in Y.
116 GTO 00 Skip          ; Continue below.
117 LBL 05  S0S          ; Continue the S0S case.
                         ; Side 3 in X, Side 1 in Y.
                         ; Next we transform this into an SS0
                         ;  case, so we can solve this and line
                         ; 115 using the same module.
                         ; This transformation is simple.  We
                         ; just make an S0S case, which is
                         ; indistinguishable from the SS0 case.
                         ; The old side 3 becomes side 1.  The
                         ; old side 1 becomes side 2.
118 XEQ 10 -i            ; Decrement the side counter.
119 X<>Y                 ; Put the old side 1 (now side 2) in X,
                        ; the old side 3 (now side 1) in Y.
120 LBL 00  Skip        ; Continue the SS0 case, with
                        ; Side 2 in X, Side 1 in Y.
121 XEQ 16 Ai           ; Recall angle without incrementing
                        ; first.  Recalls Angle 3.
122 X≠0?                ; If Angle 3 (the included angle
                        ; between sides 1 and 2) is not zero,
123 GTO 06 SS0xxA       ; Solve side-angle-side case
                        ; Stack contains  A3, S2, S1.
124 RDN                 ; Otherwise, restore Side 2 to X,
                        ; Side 1 to Y.
125 XEQ 15 +Ai          ; Recall Angle 1.  Stack has A1, S2, S1.
126 X=0?                ; If Angle 1 is zero,
127 GTO 00 Skip         ; Case is SS00x0.
128 XEQ 15 +Ai          ; Otherwise recall Angle 2.
                        ; The stack is then: A2, A1, S2, S1.
129 X=0?                ; If Angle 2 is zero,
130 GTO 03 Skip2        ; case is SS0A00.
                        ; Otherwise, case is SS0AA0.
                        ; Compute Angle 3.  Stack starts as
                        ; A2, A1, S2, S1
131 +
132 COS
133 CHS
134 ACOS
135 XEQ 11
136 XEQ 14 StoreAi      ; Stores Angle 3.  Stack is A3, S2, S1.
                        ; Continue to solve, using the
137 GTO 06 SS0xxA       ; side-angle-side case (SS0xxA).
138 LBL 03  Skip2       ; Status is SS0A00.
139 RDN                 ; Stack is now: A1, S2, S1, 0
140 XEQ 09 FindSide     ; The FindSide subroutine uses the data
                        ; in the stack:    A1   S2   S1
                        ; to compute S3.
```

```
141 XEQ 11 +i             ; Increment the side/angle counter.
                          ; Status is now SS0.
142 STO IND 00            ; Store Side 3.
143 GTO 07 SSS            ; Use the side-side-side solution to finish.
144 LBL 00  Skip          ; SS00x0 case.
145 RDN                   ; Put Side 2 in X, Side 1 in Y.
146 X<>Y                  ; Put Side 1 in X, Side 2 in Y.
147 XEQ 15 +Ai            ; Recall Angle 2.
148 X=0?                  ; If Angle 2 is zero,
149 GTO 02 Insufficient   ; Data is not sufficient.
                          ; Otherwise, the case is SS00A0
150 XEQ 09 FindSide       ; Use data in the stack:   A2   S1   S2
                          ; to compute S3.
151 XEQ 11 +i             ; Increment the side/angle counter.
                          ; Status is now SS0.
152 STO IND 00            ; Store Side 3.
153 GTO 07 SSS            ; Use the side-side-side solution to finish.
```

Now we need several more modules:

```
LBL 06  SS0xxA            ; Solve the side-angle-side case
LBL 07  SSS               ; Solve the side-side-side case.
LBL 09  FindSide          ; FindSide subroutine uses the Law of
                          ; Cosines to compute S3.
LBL 14  StoreAi           ; Store current Angle without
                          ; incrementing the counter.
LBL 16  Ai                ; Recall current Angle without
                          ; incrementing the counter first.

154 LBL 06  SS0xxA        ; Solve the side-angle-side case
                          ; Stack contains:   A3   S2   S1
                          ; or the mirror-image equivalent
155 X<>Y                  ;  S2           A3           S1
156 P-R                   ;  S2cosA3      S2sinA3      S1
157 RCL Z                 ;  S1           S2cosA3      S2sinA3
```

```
158 -                   ; S2cosA3-S1    S2sinA3
159 R-P                 ; SQRT((S1-S2cosA3)^2+(S2sinA3)^2)
                        ; = SQRT(S1^2+S2^2-2S1S2cosA3)
                        ; = S3
160 STO IND 00          ; Store S3.  Continue as SSS.

161 LBL 07  SSS         ; Solve the side-side-side case.
                        ; Initial stack contents arbitrary.
162 RCL 00              ; x.003
163 FRC                 ; 0.003
164 STO 00              ; Reset side counter
165 ISG 00              ; to side 1.
166 LBL 08              ; Top of loop that computes the angle
                        ; opposite each side.
167 XEQ 16 Ai           ; Check the current angle, which we
                        ; shall refer to as A1
168 X≠0?                ; If the angle is already known (nonzero),
169 GTO 00 Skip         ; then skip the computation.
170 XEQ 12 +Si          ; Recall S2
171 XEQ 12 +Si          ; and S3.
172 R-P                 ; Stack is  SQRT(S2^2+S3^2), ATAN(S2/S3)
173 XEQ 12 +Si          ; Recall S1.
174 X<>Y                ; Stack is:  SQRT(S2^2+S3^2), S1, ATAN(S2/S3)
175 /                   ; S1/SQRT(S2^2+S3^2), ATAN(S2/S3)
176 X↑2                 ; S1^2/(S2^2+S3^2), ATAN(S2/S3)
177 1
178 X<>Y
179 -                   ; 1 - S1^2/(S2^2+S3^2), ATAN(S2/S3)
180 X<>Y
181 ENTER↑
182 +                   ; 2*ATAN(S2/S3), 1 - S1^2/(S2^2+S3^2)
183 SIN                 ; SIN(2*ATAN(S2/S3)), 1 - S1^2/(S2^2+S3^2)
184 /                   ; COS A1
185 ACOS                ; A1 (result)
186 XEQ 14 StoreAi      ; Store the current Angle without
```

```
                         ; incrementing the counter.
187 LBL 00  Skip         ; Entry point if calculation is skipped.
188 ISG 00               ; Increment the counter, to compute
189 GTO 08               ; next angle.  Skip when all 3 are done.
190 RTN                  ; Finished!

191 LBL 09  FindSide     ; FindSide subroutine.  Use the Law of
                         ; Cosines to compute S3.
                         ; Stack starts as:  A1, S2, S1
192 ENTER↑
193 SIN                  ;  sinA1        A1          S2          S1
194 R↑                   ;  S1           sinA1       A1          S2
195 /                    ;  sinA1/S1     A1          S2          S2
196 ENTER↑               ;  sinA1/S1     sinA1/S1    A1          S2
197 R↑                   ;  S2           sinA1/S1    sinA1/S1    A1
198 *                    ;  S2*sinA1/S1  sinA1/S1    A1          A1
199 ASIN                 ;  Â 2          sinA1/S1    A1          A1
200 ST+ T                ;  Â 2          sinA1/S1    A1          Â 2+A1
201 R↑                   ;  Â 2+A1       Â 2         sinA1/S1    A1
202 COS
203 CHS
204 ACOS                 ;  180-Â 2-A1   Â 2         sinA1/S1    A1
205 X<>Y                 ;  Â 2          180-Â 2-A1  sinA1/S1    A1
206 R↑                   ;  A1           Â 2         180-Â 2-A1  sinA1/S1
207 -                    ;  Â 2-A1       180-Â 2-A1  sinA1/S1
208 X<=0?                ; If Â 2-A1 is not positive,
209 GTO 00  Skip         ; then there is only one solution.
                         ; Data in Y and Z is  180-Â 2-A1   sinA1/S1
                         ; Otherwise, ask user to select a solution.
210 "OBTUSE? 1/0"        ; The message asking the user to enter
                         ; a 1 if the triangle is to be more
                         ; obtuse, or a 0 if the larger, more acute
                         ; triangle is desired.  Note that the 1
                         ; is closer to the shape of the smaller,
                         ; more obtuse triangle, while the 0 is
```

```
                          ; closer to the shape of the large, more
                          ; acute triangle.
211 PROMPT                ; Display the message and halt.
212 CF 22                 ; Avoids confusing the LBL A-C and a-c
                          ; routines.
213 *                     ;  0 or Â2-A1   180-Â2-A1   sinA1/S1
214 X≠0?                  ; If a 1 was entered (more obtuse),
215 X<>Y                  ;  180-Â2-A1         Â2-A1   sinA1/S1
216 LBL 00  Skip          ; Data for the solution is in Y and Z
217 RDN                   ;  Â2-A1          sinA1/S1    or
                          ;  180-A2-A1       sinA1/S1
218 SIN                   ;  sinA3           sinA1/S1
219 X<>Y                  ;  sinA1/S1        sinA3
220 /                     ;  S1*sinA3/sinA1 = S3
221 RTN                   ; Return the value of the missing side.
```

This completes the program. It is not, in fact, the shortest possible triangle solution program, but most shorter programs require the user to enter all the data each time the program is run. With this program, you can adjust the value of a side or an angle, clear the values from other sides and angles, and try another solution.

Now it's time to test and debug the program. As I mentioned earlier, this is the time when you would normally find any errors in your stack management instructions. I hope you don't find any in this case, because I didn't put in any intentional errors! There are, however at least two improvements possible: If you first add a LBL 03 after line 149, you can replace lines 140-143 by GTO 03. Also, lines 202-204 can be deleted, since sin(A2+A1)=sin(180-A2-A1) at line 218.

To test the program scientifically, you need to identify a complete set of test cases. This is much easier said than done. In fact, an earlier version of this program that I developed in 1979 and which was published in the April 1980 PPC Journal (pages 13 and 14) managed to retain some hard-to find errors.

The test cases should include at least one of each type that the program can solve:

1) SSSxxx (three sides and any, all, or no angles)
2) SS0xxA (two sides and the included angle)
3) SS0AA0 (compute the included angle and solve as SS0AAA)
4) SS0A00 (two sides and a non-included angle; may have two solutions)
5) SS00A0 (same)
6) S00 with at least two angles (compute the third angle, then use the law of sines to solve)

Once you have tested the program's capability to rotate a nonzero side to the S1 position, it should be sufficient to test only cases in the above standard forms. For the side-side-angle cases (numbers 4 and 5), you will need to test obtuse, acute, and right triangle solutions.

You should also test the insufficient data cases:

7) 000xxx (need at least one side)
8) SS0000
9) S00 with less than two angles

Let's test the insufficient data cases first.

Press [e] to clear the triangle data registers.

Case 7: Put in three angles, such as

30 [a], 60 [b], 90 [c]

and press [E] to solve. The result should be

MORE DATA

Case 8: Press

[e], 3 [A], 4 [B], [E]

Again the result should be a display of MORE DATA.

Case 9: Press

[e], 3 [A], 30 [a], [E]

Again the result is MORE DATA.

Now for the sufficient data cases. Set the display mode to FIX 2. For all but cases 4 and 5, a single triangle ought to be sufficient. We will use the 3-4-5 right triangle: S1=3, S2=4, S3=5, A1=36.87, A2=53.13, A3=90.00.

Case 1: Press

[e], 3 [A], 4 [B], 5 [C], [E]

When the program halts, check the results.

[a] gives 36.87, [b] gives 53.13, and [c] gives 90.00

There is one feature that of this program's SSS solution that you should note. If you have supplied angle values in addition to the three sides, the program will assume that these angles are correct and will not recalculate them. If you want to be sure that the angle results are correct and consistent with the sides, you can modify the program to recalculate all three angles when doing the SSS solution. Just delete lines 167-169 and 187, which test for a nonzero angle value and skip the calculation if possible. Naturally this change will increase the excution time. So make your own decision according to your preference.

Case 2: Press

[e], 3 [A], 4 [B], 90 [c], [E]

When the program halts, check the results.

[a] gives 36.87, [b] gives 53.13, and [C] gives 5.00

We can use Case 2 to check the program's rotation capability. Let's give it the same problem, rotated by 1 side:

[e], 3 [B], 4 [C], 90 [a], [E]

The results are:

[b] gives 36.87, [c] gives 53.13, and [A] gives 5.00

Now let's rotate it by another side:

[e], 3 [C], 4 [A], 90 [b], [E]

The results are:

[a] gives 53.13, [c] gives 36.87, and [B] gives 5.00

Case 3: Press

[e], 3 [A], 4 [B], 36.87 [a], 53.13 [b], [E]

When the program halts, check the results.

[c] gives 90.00, and [C] gives 5.00

Case 6: Press

[e], 3 [A], 36.87 [a], 53.13 [b], [E]

The results are:

[c] gives 90.00, [B] gives 4.00, and [C] gives 5.00

[e], 3 [A], 53.13 [b], 90 [c], [E]

The results are:

[a] gives 36.87, [B] gives 4.00, and [C] gives 5.00

[e], 3 [A], 36.87 [a], 90 [c], [E]

The results are:

[b] gives 53.13, [B] gives 4.00, and [C] gives 5.00

Cases 4 and 5: First a near-right triangle,

[e], 3 [A], 5 [B], 36.87 [a], [E]

This one is a little tricky. You will get

DATA ERROR

because the triangle cannot be closed. The problem is that Angle 1 is just a little too large. Adjust it like this:

36.86 [a], [E]

The results for Solution 0 are:

[b] gives 88.77, [c] gives 54.37, and [C] 4.06

If you selected Solution 1:

[b] gives 91.23, [c] gives 51.91, and [C] 3.94

As you can see, the results are sometimes quite sensitive to the inputs. A variation of 0.01 degree in one angle resulted in a change of over 1 degree in the two angle outputs.

Now, an acute triangle case

    [e], 3 [A], 4 [B], 53.13 [b], [E]

The results are:

    [a] gives 36.87, [c] gives 90.00, and [C] 5.00

Finally, a true dual-solution case.

    [e], 4 [A], 3 [B], 36.87 [b], [E]

The results for Solution 0 are:

  [a] gives 53.13, [c] gives 90.00, and [C] 5.00

If you selected Solution 1:

    [a] gives 126.87, [c] gives 16.26, and [C] 1.40

You can see from this example how comprehensive the testing must be for a complex program. Without this testing, the user of the program may encounter errors. Worse, the user may not even notice the errors, and rely on the incorrect results. This is especially embarassing when the user and the programmer are the same person! So be sure to fully test your programs. It is time well spent.

CHAPTER THREE - OPTIMIZING PROGRAMS

The aim of this chapter is to offer some guidelines and ideas on ways to improve your programs. The kinds of improvements that will be discussed fall under two basic categories: those that reduce a program's byte count, and those that reduce its execution time. Often you have to choose whether speed or byte count is more important. Happily, many of the improvements in the material that follows will make your programs both shorter <u>and</u> faster.

Optimizing a program involves a number of tradeoffs in addition to considering the byte count versus speed. Therefore, there aren't any rules written in stone. Since the object here is to improve your programs and programming skills, you'll get the most benefit if you take the time to try techniques that are new to you.

The best way to try out new techniques is to use a program you wrote yourself. This way you will be familiar with the program's use of the stack and Alpha registers. If you haven't used the program for a while, it would be a good idea to "walk through" (SST) the entire program for a simple example to check stack usage before making any changes. It's also good to restrict your changes to a small section of the program at first. After you have tested the effect of these changes and determined that the program still works correctly, you can modify another section.

It's also a good idea to have two copies of your program in main memory when you start. The first version is left unaltered, in case you need to refer to it to figure out some forgotten detail of program operation. The second copy of the program is the one to be modified. Before you make any other changes to the second program, make a small change in the spelling of its main Global label (usually line 01).

This identifies the two different versions and eliminates having to use CAT 1 to move between the two. Be aware, though, that any Alpha GTO or Alpha XEQ instructions in these programs will branch to the matching Alpha label lowest in memory (the one closest to the .END.). So if the program has more than a single Global label, the version you are testing should be closer to the .END. (farther down Catalog 1).

There are several easy ways to alter the main Global label of program you are modifying. One way is to change a letter from capital to lower case, if one of the letters from A to E. Another good idea is to change the last letter to a number, or add a "2" after the last letter. This is used to keep track of the number of times the program is revised. Each time a major change is made, increase the number by one. Be sure to write down all of the changes you make, and don't be in a hurry to throw out the old documentation. It can be useful when it turns out that a case you didn't test on the new version fails to work.

An easy way to test the changes you make in your program is to insert a STOP (press R/S) instruction a line or two before the new program lines. Then run the program and SST through the lines that were changed, R/S and run to completion. If the STOP instruction is added within a loop, you may have to press R/S more than once to finish the program. Press and hold R/S to check the line number before releasing it.

After you're satisfied that the changes you made are a suitable substitute for the original, remove the STOP function you added, make a note of all the changes you made, and move on to the next change, if any. Be sure to PACK and run the program to compile the GTOs and XEQs before you save it on cards, cassette, or in Extended Memory.

Of course, all of this isn't absolutely necessary for simple changes. However, it's good to establish habits that are both systematic and convenient. The time you spend working on a program is practically wasted if you don't document how it works.

The different program optimization categories in this chapter are arranged in an order that is designed to give you good results in reducing byte count and improving speed. The beginning sections yield gains fairly quickly, while those later in the chapter take a bit more work. Whether you just need to trim off a few bytes to make your progam fit neatly onto one card or you want to completely optimize a program for speed, you'll make best use of your time by proceeding in this order.

3A. ALPHA TEXT LINES

A quick way to trim a few bytes from a program is to shorten Alpha prompts, messages or warnings. The easiest approach is to use abbreviations, contractions, or the initials of some of the words in your Alpha text lines. You can often omit the period after an abbreviation. (The period does use a full byte of program memory even though it does not usually occupy a separate character position in the display.) Replace words in the text lines of your programs as long as there will be no loss of meaning.

Sometimes the normally keyable characters just aren't enough. Rows zero and two of the byte table contain many useful synthetic characters. Messages containing the apostrophe ("DON'T R/S"), the ampersand ("YOU & I") and # ("INPUT #1?") are just a few examples. Although these characters are not normally keyable, they can be used in alpha messages. Nonkeyable characters can be appended to the Alpha register using the Extended Function XTOA, the Printer instruction ACX, or the PPC ROM's "DC" (Decimal to Character) program. Many bytes and considerable execution time can be saved over using XTOA, "DC", or ACX by instead constructing synthetic text instructions with the special characters "built in". The byte-building functions XTOA, "DC", and ACX are better suited to cases in which the character to be appended may depend on the outcome of an earlier computation or user selection.

You can construct synthetic text lines using a byte loading program (such as the PPC ROM's "LB"), the CCD Module, the ZENROM, or a synthetic key assignment. More information on this subject appears in Section 4B. Both hexadecimal and decimal inputs are listed in the examples that follow, for use with either the ZENROM or a byte loading program.

Of course, these two techniques aren't the only ways to create text lines containing non-keyable characters. You can use the Byte Jumper (described in Section 4I) to get inside an existing text line. You could also use a Text 0 prefix assignment to release one of the Text bytes from row F of the byte table, which would absorb instructions to make a text line. However, the Byte Grabber and the byte loader are more straightforward to use. They also illustrate the underlying principles more clearly.

CCD Module or ZENROM users can create synthetic text lines as easily as normal text lines, because the USER key acts as a second shift key in ALPHA mode. (Note that these two modules do operate differently from each other in ALPHA mode, however.)

ZENROM users can create synthetic text lines in two additional ways: by inserting bytes with RAMED's "I" mode, or by editing existing text lines with RAMED. Pages 56 through 61 of the ZENROM manual explain this in detail.

If you're intending to use a byte loading program, make sure you have a copy of "LB", "LBX", or a similar program in RAM or ROM. If you're using the byte grabber (a synthetic key assignment with prefix 247 = hex F7), be sure that the postfix value is from 15 to 28, 32 to 191, or 206 to 244. For our purposes here, it would be best if you assign the commonly used byte grabber 247, 63 to a convenient key using either the CCD Module's enhanced ASN function, "MK", "MKX", or another synthetic key assignment program. The postfix value of 63 appears as a question mark in the distinctive text line that is created by the byte grabber.

If you are completely unfamiliar with the byte grabber, you should work through the examples in Section 4B before coming back to finish this section. Even better, you could read Chapters 1 and 2 of Keith Jarett's "HP-41 Synthetic Programming Made Easy", which will show you how to use the byte grabber to create many synthetic instructions.

Using the byte grabber to create synthetic text lines

The approach to putting a nonkeyable character into a text line by using the byte grabber can be summarized as follows. First you key in a normal text line of the length that you want the final result to have. Then you use the byte grabber to absorb the prefix byte of the text line. This is the byte (hexadecimal Fn; see row F of the byte table) which specifies that the next n bytes are to form a TEXT instruction of n characters, where n is up to 15. The characters from the text string are then exposed and appear as instructions on their own. Next you replace some or all of the characters with the instructions corresponding to the non-keyable characters that you want to put in the text line. Finally, you use the byte grabber again to release the previously grabbed prefix byte and re-establish the modified text line. An example will illustrate this procedure:

Start by using GTO .. to get to the bottom of Catalog 1 and pack memory. Enter PRGM mode. Make sure at least five registers are free (00 REG 05 or more). Decrease the SIZE if necessary to achieve this.

Key in LBL "AT", ALPHA A B C D E F G ALPHA, followed by seven + instructions.

SST twice to return to line 01 or press GTO .001.

Press the byte grabber. You should have the following:

```
01 LBL "AT"            (Alpha Text)
02 "‾?‾‾‾‾‾▩"
03 -
04 *                      "LB" inputs:
05 /
06 X<Y?                   247, 65, 66, 35,
07 X>Y?                   68, 69, 70, 0
08 X<=Y?
09Σ+
10 +                       Hex inputs:
11 +                      F7, 41, 46, 23
12 +                      44, 45, 46, 00
13 +
14 +
15 +
16 +
```

Now position yourself to line 05 (the divide instruction that corresponds to the character "C") and backarrow. Key in RCL 03. Go to line 09 and backarrow. Press GTO .001. Byte grab again, SST, and then backarrow twice. XEQ "PACK" (do not GTO . .) .

Looking at line 02 should show "AB#DEF‾". We replaced character "C" in the original text line with "#" (RCL 03, decimal value 35 = hex 23 in the byte table). "G" was deleted and not replaced at all. That position now shows the "overbar" character, representing a null. Had PACKing taken place before using the byte grabber the second time, this null would have been filled in by the first of the following + instructions. Within the text line, the + instruction (decimal value 64) would appear as "@".

The purpose of the extra + instructions is to prevent any possibility of part of the .END. being absorbed into the text line. This would cause you to lose access to Catalog 1, and that would probably lead to MEMORY LOST. You should <u>NEVER</u> byte grab within 5 bytes in front of an Alpha label or the .END. . To avoid the chance of this happening, always add a buffer of instructions between where you are working and any nearby global instruction at a lower address in memory. You might not think you need a protective buffer, but accidents will happen . . .

We can extend the idea of replacing characters to include replacing the entire contents of the text line. This is especially attractive when most of the characters are either lowercase or special characters for use with a printer. Text lines in which most of the characters are nonkeyable are easiest to construct with "LB". And when any of the decimal byte values are from 192 to 205 or from 208 to 239, using "LB" (or the ZENROM) is the only reasonable way to proceed.

Let's use the "AT" program to illustrate replacing the entire set of characters in a text line. You'll need at least one free register in addition to having "AT" in memory. Begin by back-arrowing the text line 02 and keying in the original "ABCDEFG". BST to LBL "AT" and press the byte grabber. Now key in LBL 00, RCL 06, LBL 11, RCL 08, CLST (press XEQ ALPHA C L S T ALPHA), STO 06, and RCL 09. Each of these instructions will become a character in the new text line. Check the byte table for the character correspondence. GTO .001 and byte grab again. SST, then backarrow twice. SST to take a look at the line you created. You should see 02 "Ѫ&μ(⊠6)".

To create this line using "LB", use inputs of 247, 1, 38, 12, 40, 115, 54, and 41. The "full man", ampersand, mu (or "micro"), left parenthesis, starburst, and right parenthesis are all nonkeyable. The two parentheses are especially useful.

If you used the byte grabber, SST'ing the next seven lines will show functions which correspond to characters A through G. The buffer of seven + instructions is just beyond them.

More byte savings

A completely different approach to saving bytes in constructing program messages is to use the system error messages in place of your own warnings. In most cases where this method can be used, you will only need two or three bytes to generate the system error message. The longest needs seven bytes. Of course, flag 25 must be clear in order for these errors to cause the listed error message to be displayed. In addition, flag 24 must be clear for OUT OF RANGE. See the table below for ten different ways to generate various error messages:

	Instruction Sequence	Error Message
With any HP-41		
	CLX, LOG	DATA ERROR
	ASTO X, OCT	ALPHA DATA
	E2, FACT	OUT OF RANGE
	SF 99	NONEXISTENT
With Extended Functions or 41CX		
	CLA, CLFL	NAME ERR
	CLA, SEEKPTA	FL TYPE ERR
	CLX, PASN	KEYCODE ERR
With Time or 41CX		
	CLX, DDAYS	DATA ERROR X
	., DATE, DDAYS	DATA ERROR Y
	E4, ., TIME, XYZALM	DATA ERROR Z

The instruction sequences listed are suggestions only. You may want to make changes in them to suit your particular needs. As an example, you might want to change CLX to the lone decimal point (.) so the contents of X aren't lost. This method is used in the last two sequences above to bring zero into the stack without clearing X. You might want to use a different function than the one listed in the first three examples if that function appears somewhere in the program. This will avoid confusion.

If you need to test particular conditions in your program, you can change these sequences further. For example, if a program is intended to accept non-negative inputs only you can use X<0? LOG at the top of the program. The LOG instruction will give a DATA ERROR right away if a negative input is supplied. Otherwise it is skipped.

3B. NUMERIC ENTRY

The suggestions in this section mainly improve the execution time of your programs. In some cases they will save a byte or two as well.

Instead of using 0 for zero, you should consider some alternatives. A lone decimal point (.) is a little faster, though a little harder to read in a program listing. **CLX** or **ENTER** followed by **CLX** is faster still, as long as leaving the stack lift disabled won't cause problems with subsequent instructions. You can also use **RCL a** (recalling status register **a**) for a zero as long as the subroutine stack has less than three pending returns. You can even use **RCL a** for zero when three returns are pending if the "zero" will be stored in a numbered data register and recalled (hence normalized) and set to a true zero before it is tested. Otherwise a test **X=0?** will give the result NO.

A slightly faster alternative to using 1 for the number one is a solitary **E** (Enter EXponent), decimal value 27 (hex 1B) from the byte table. This technique has become fairly common, though there are other nonsynthetic options available. As long as X contains ordinary non-negative numeric data, you can substitute the function **SIGN**. This is nearly four times faster than the digit entry 1. If the contents of X are not known, you can use the sequence **CLX, SIGN**. In either of these two cases, if you need to preserve X, add an **ENTER** instruction beforehand to push the value of X into register Y.

When your program makes use of a numeric entry line with just a "1" before the exponent, for instance **1 E3**, you can save one byte and reduce execution time by removing the **1**. As an example, **E2** is about one-third faster than **1 E2**, although both perform the same function. These are known as short form exponents. An example using the byte grabber to remove the leading **1** from a similar power of ten is given in the first part of Section 4B. The "LB" inputs for **E2** are 27, 18. Removal of a leading "1" before an exponent instruction is automatic if you have a ZENROM.

The instruction listed as **NEG** (negative), decimal value 28 in the byte table, is relatively slow. It takes about the same time as entering a numeric digit. In most cases, you can avoid numeric entry lines containing this byte value quite easily. If the **NEG** byte is used to make a number negative, use **CHS** instead. This only requires a small change in the way you key in the number. For example, to key in negative eight, instead of pressing **8**, **CHS**, resulting in a line showing **-8**, key in 8, ALPHA, ALPHA, CHS. This produces two program lines, one showing **8** and the other **CHS**. Both versions use two bytes, but the second is 65% faster! Note that L (LASTX) is unaffected by either **NEG** or **CHS**.

Another way to avoid the **NEG** function is to divide instead of multiplying, or vice versa, so that a positive exponent is used. Use **E3**, / in the place of **E-3**, *. This is over 40% faster, and is one byte shorter. This technique can also be used on some numbers which are not exact powers of ten. As an example, **5 E3**, / substitutes for **2 E-4**, *.

In a similar fashion, you can avoid using **NEG** in negative exponents with a reciprocal (**1/X**) instruction. This technique can be used for powers of ten, as well as other numbers. **E6**, **1/X** is faster than **E-6**. However, it does change the L (LASTX) register.

A slightly less obvious example is to use **8 E8, 1/X** in place of **1.25 E9**. There is a shortcut to help you determine whether a particular number can be entered this way. Key in the number or single-step the program line containing the number, then press **1/X**. If the resulting number has the same number of significant digits, this technique will only save a little time. But if the result has fewer significant digits, you'll conserve both bytes and execution time using this method.

In the same manner, the sequence **3, 1/X** is <u>much</u> faster than using .333. It also uses half as many bytes and equals one-third exactly. This can be applied to other numbers as well. Again, taking the reciprocal of the desired result will show if this will be helpful with a particular number. Many fractional numbers are best expressed as a division of two numbers, expecially if accuracy is needed. A series of instructions such as **2, ENTER, 3, /** instead of **.6666**, or **5, ENTER, 6, /** in place of **.8333** is an example of superior programming practice. The **ENTER** instructions aren't actually needed after the numbers are keyed in, and they can and should be removed. This will save a few milliseconds and avoid the problem that the stack lift is re-enabled if you stop the program after the **ENTER**, then restart it. However, even without the **ENTER**, the HP-41 will leave an invisible null between the two numbers as a separator, so the byte count is the same whether you remove the **ENTER** instruction or not.

Using **1/X** or division isn't just for fractions whose digits repeat. The last example can be changed to **5 E4**, (null), **6, /** when the number desired is **8,333.33**. This holds a slight speed and byte count advantage over a numeric entry line of **8333.33**. Use the function execution times from Appendix A of SPME and your best judgement to decide which one is better to use in your program. A simple example of this would be using **2, 1/X** in place of **.5** for one-half. The 30% speed advantage of **2, 1/X** has to be weighed against the fact that **.5** is easier to understand and does not change the LASTX register.

In the case above where the quantity 2 was needed, yet another substitution could have been made. The sequence **SIGN, ST+ X** is 31% faster than using 2. However, this time savings at the cost of two more bytes strictly limits this to applications in which time is the most important factor. Be forewarned that needing to add **CLX** because X may contain alpha data or NNNs, or having to use **LASTX** to recover the old value of X will almost completely cancel out the time savings.

When using either very large or very small numbers, you can speed up operation slightly by eliminating the decimal point. As an example, instead of using **1.23 E12**, substitute **123 E10**. This saves one byte and a little time. For numbers with positive exponents, this will sometimes result in the saving of another byte and a little more time when this change results in the exponent being reduced from two digits to just one. An example using a negative exponent would be to change **7.36 E-21** to **736 E-23**, eliminating the decimal point.

Quite often when program execution is bogged down by numeric entry lines, it's because the program requires a large number of constants to be stored in the data registers during program initialization. This is especially true when the numbers have many digits, an exponent, the number and/or exponent is negative, and so on. This kind of numeric entry is very time consuming because the program bytes have to be translated into an actual number which has very different byte values than the representation in program memory.

Synthetic programming offers a solution to this problem that is practical if the numbers average more than nine bytes apiece, or if saving execution time is more important than saving bytes. The number is brought into Alpha in the form of a text line, and recalled into the stack using RCL M. This avoids the time-consuming formatting that is normally required. Former Alpha contents are lost, but this will usually not matter during initialization.

In order to figure out the correct byte values for the text line that will generate the desired number, there are several approaches you can take. If you have a ZENROM, simply XEQ "DECODE" with the number in X. Write down the fourteen hexadecimal digits of the result. Enter PRGM mode where you want the instruction inserted. Then XEQ "RAMED" and press I for the insert mode. Key in F 7 followed by the fourteen hex digits you wrote down earlier. Press ON to exit the RAM-Editor. Add a RCL M insıruction after the text line and you have everything you need.

With a CCD Module, put the number in X and XEQ "DCD". Write down the fourteen hexadecimal digits of the result. Enter PRGM mode where you want the instruction inserted. Then use the alpha hex entry feature (ALPHA SHIFT ENTER H digitl digit2) to enter each of the seven synthetic characters by their two-digit hexadecimal equivalents. Again, add a RCL M instruction.

If you have a PPC ROM, you can use a slightly longer procedure. Clear Alpha, put the number in X, and STO M using a key assignment or by single-stepping a STO M instruction. Then XEQ "CD" seven times to give the decimal values needed in reverse order (right-to-left). Or you can use the Extended Function ATOX as described below. Add decimal byte 247 as the first of eight bytes that make up the proper text line. When you have the values for all of the numbers you need, use "LB" to create the text lines and follow each text line with RCL M (144,117).

Another approach is the PPC ROM's "NH" routine. XEQ "NH" with the number in X. The result is a list of fourteen hexadecimal digits. Divide them up in seven pairs of numbers to use with "LB". When using this byte loading program, turn ALPHA mode on and use F7 as the first of eight pairs of hexadecimal digits. Repeat the same procedure for each number, adding hex 90 and 75 after each complete number (text line) for the required RCL M instruction.

If you don't have the ZENROM, CCD Module, or the PPC ROM, you can figure out the proper byte values with the Extended Function ATOX if you have STO M or X<> M assigned to a key or contained within a program (such as "LB"). Clear Alpha and then transfer the bytes of your number from X to M. Either press a key assigned with STO M or single-step it as an instruction (Section 4B shows how to use the byte grabber to create this particular function). Enter ALPHA mode and make a note of the position of any nulls ("overbar" characters), because ATOX will not decode these. Exit ALPHA mode and use ATOX repeatedly to get the remaining non-zero byte values, from left to right. You can make the needed text line using a byte loading program or the byte grabber, as described in Section 4B.

Putting the needed byte values together manually isn't that difficult, because the numeric representation is Binary Coded Decimal (BCD). The details of the structure were given in Section 1B. The first of the 14 hex digits is the sign nybble. Use 0 for positive numbers and 9 for negative numbers. The next ten nybbles are the mantissa digits (the number without any decimal point or exponent). The three nybbles that remain represent the exponent and its sign (again, 0 for a positive sign, 9 for a negative sign). A negative exponent -mn is stored as 1000-mn, which equals 9xy, where xy = 100-mn. For example, for E3 use hex 003 for the sign and exponent nybbles. For E-6, 1000-6 = 994, so use hex 994 hex for these three nybbles. The number totals 14 hexadecimal digits (nybbles) which can be used with a byte loading program in pairs. Add a Text 7 byte (F7 hex) before the number, and hex 90 and 75 bytes afterward (to make RCL M).

Let's work through a complete example. Suppose the number you want is the speed of light in miles per hour (186,282 * 60 * 60 = 6.70615200 $E^{08)}$. This is a positive number, so the first nybble is 0. The next ten nybbles are 6 7 0 6 1 5 2 0 0 0. The exponent sign is also positive, so the next nybble is 0. Simply keying in the number and pressing SCI 1 or counting the number of digits between the decimal point and the first digit gives a result for the exponent of 0 8. So the bytes needed (in hex) for 670,615,200.0 are as follows:

```
        06 70 61 52 00 00 08
```

You can use these values with a byte loading program or convert them to instructions and use the byte grabber in Section 4B to replace existing characters in a text line. Note that the easiest way to incorporate nulls in a text line using the procedure in 4B involves using a one-byte function such as ENTER as a place holder. After the other bytes are in place, the one-byte function is deleted, leaving a null. Then the text prefix can be released to re-absorb the edited character bytes.

Using a text line followed by <u>RCL</u> <u>M</u> is three times faster than the numeric entry line 670615200. Slightly more speed can be gained in this example by using the non-normalized numeric representation

```
        00 00 67 06 15 20 11
```

which is 0.006706152 E11. This needs only a 5-character text line rather a 7-character one. The two leading nulls can be omitted, and a Text 5 (hex F5) byte takes the place of Text 7.

If you are using several numbers, you may want to pair them up and create two at a time, using a 14-character text line. This saves half a byte per number. Both numbers are loaded into Alpha at the same time, then the two numbers are brought into the stack with RCL N and RCL M instructions. This is nearly five times faster than normal entry of a negative ten digit number with a decimal point and negative two-digit exponent.

3C. CLEARING REGISTERS

The normal function **CLST** clears stack registers X, Y, Z, and T. If you also want L (LASTX) cleared, use **CLST, +**.

As long as X contains a legitimate number, you can use **STO nn**, **ST- nn** to clear a single data register. The advantage of this technique is that none of the stack registers are disturbed. If the contents of register T are unimportant, or if X contains alpha data or Non-Normalized Numbers, there is a faster alternative which is one byte longer. The sequence **ENTER, CLX, STO nn, RDN** stores zero in register nn and restores the stack (except T, which is cleared). Note that both of these methods can be applied to register 00 through 99 using normally keyable instructions. Synthetic functions can extend direct addressing capacity up to register 111. **STO 111**, which displays as **STO J** has byte values 145, 111. Refer to the byte table and Section 4B for more information.

A block of six registers from nn to nn+5 can be cleared very quickly using **ΣREG nn, CLΣ** . If maintaining the original location of the summation registers is important, the sequence can be slightly modified. **RCL c, ΣREG nn, CLΣ , STO c, RDN** will preserve the summation register location. This second series of instructions can be stopped and restarted with no ill effects. However, if the register c contents are removed from X between **RCL c** and **STO c**, MEMORY LOST will occur when the program stops.

The contents of Extended Memory can be cleared in several ways. If you have an HP-41C or CV (without an internally installed Extended Functions/Memory module), you can turn the calculator off and remove the Extended Functions module for about a minute. When you plug it back into a port and then turn the HP-41 on, all of Extended Memory will be clear. This includes additional Extended Memory modules, as far as the HP-41 is concerned.

This technique works by erasing the contents of the register at hex address 040. This register is called the link register (each additional Extended Memory module has a similar link register at its lowest address). It contains the pointers to the working file, the top of the next Extended Memory module (if any), plus the address of the top of this module, which is always 0BF. Any time the HP-41 finds this register at hex 040 clear, it will presume Extended Memory is empty.

If you have the PPC ROM, you can clear this link register very quickly. The sequence 64, XEQ "RX" will normalize hex address 040, effectively clearing Extended Memory for subsequent operations. Recovering from accidentally performing this sequence is possible, though difficult. Recovery is made more complex due to the fact that operations such as EMDIR will result in the top register of Extended Memory (hex address 0BF, decimal 191) being filled with FF bytes. These bytes serve to mark the end of the portion of Xmemory that is in use. For more details on this subject, refer to Chapter 10, Section C of "HP-41 Extended Functions Made Easy" by Keith Jarett.

If you have a CCD Module, you can clear register 040 hex (64 decimal) with the sequence 64, ENTER, CLX, XEQ "POKER".

If you have a ZENROM, you can clear register 040 hex with the sequence ALPHA, CLA (shift backarrow), shift ALPHA 4 0, ALPHA, CLX (backarrow), NSTOM. However, it's much easier to just use the CLXM function to actually overwrite all of X Memory with nulls. CLXM is also programmable, though the "XM LOST" message will be suppressed in a running program. This function is very fast, and there is no way to recover Extended Memory contents afterward.

The Extended Function PCLPS (programmable clear programs) function can be used to clear program registers in main memory in two ways. When executed, PCLPS will clear main memory from the program named in Alpha all the way to the bottom of Catalog 1. In this way, you can selectively clear enough room in main memory to read in a program from mass storage when you run out of free registers. PCLPS is rather quick, and in addition there is no need to PACK afterward. The registers which have been cleared will be added to the free registers available, which are shown as .END. REG nnn as the last entry in Catalog 1.

With Alpha clear, the PCLPS function will clear program registers from the first line of the program you are positioned to, down to the bottom of Catalog 1. This may not be what you had intended, so be sure to check the contents of the Alpha register before you execute PCLPS. Be careful not to use PCLPS while positioned to ROM programs if you have revision 1B Extended Functions; in this case you will get MEMORY LOST (unless the Alpha register happens to be empty). If you have revision 1B Extended Functions, use CAT 1 to ensure you are in RAM before executing PCLPS. The sequence CAT 1, R/S immediately, XEQ "PCLPS" will clear all of program memory. Since this clears only program memory, you may prefer this over the ZENROM's CLMM (Clear Main Memory) function.

All of main memory is cleared by the CLMM function. The only difference between this ZENROM function and executing a Master Clear is that Extended Memory is left undisturbed by CLMM. All key assignments, Time module alarms, I/O buffers, data, stack, and program registers are overwritten with nulls. CLMM resets all status registers and flags to their default state. The number of free program registers will be reset to 46 if you have an HP-41C or CV. If you have a CX, the number of free registers will be 219 (SIZE = 100).

3D. LOOPING

A program loop on the HP-41 usually consists of a label function, some number of instructions to be repeated, and a GTO which returns execution back to the original label. The two sections following this one will discuss labels and GTOs in depth. This section focuses on the fact that the time it takes to complete the loop once has to be multiplied by the number of times the loop is executed in order to figure the total execution time. Therefore, any time saved within the loop is multiplied by the number of times through the loop. Even small reductions will add up to significant time savings. Because of this, it is worthwhile to make an effort to reduce the execution time of a loop, even if it takes a few more bytes to do so.

First, reduce execution time by doing whatever tasks you can outside the loop. Avoid numeric entry lines within the loop. Store such numbers in stack, status (such as Alpha registers M, N and O), or numbered data registers before the loop begins. Notice that stack and status registers have a slight speed advantage over the data registers. Also, data registers 00 to 15 have a slight edge over higher numbered data registers in speed. This is related to the fact that the **STO** and **RCL** functions for registers 00 to 15 are one-byte functions.

Rather than using **PSE** to display a number, make use of **VIEW** in a loop. As long as the loop takes enough time to allow you to look at the number, **VIEW X** is superior to **PSE** because program execution continues. However, an **AVIEW**, **CLD**, or **PROMPT** instruction subsequent to the **VIEW** instruction will naturally change the display. **VIEW** can also be used to show a number or six characters of an Alpha text stored in a register without having to recall the information to Alpha or X. You'll probably want to clear the display with **CLD** following the loop that uses **VIEW** or **AVIEW**.

It is common for a result to be accumulated in a status or data register while the loop is being repeated. After exiting the loop, when you need to recall that value, you can simultaneously recall the value and reset the register to zero using **CLX, X<> nn** or **ENTER, CLX, X<> nn**.

If your program contains a single large loop consisting of a label, some instructions, and a **GTO**, you may want to replace this with something a little faster. The label gets replaced by a **RCL b** instruction. The GTO is replaced by **STO b**. In order for this technique to work, you have to keep the recalled contents of register b in the stack and have them in X when the program later encounters the **STO b**. If you use too many stack manipulation instructions to get the register b contents out of the way and then back into X, your time savings will be negated. Check the function execution times in Appendix A of SPME if you have to add a **RDN**, **R↑**, or other stack manipulation to make the **RCL b**, **STO b** looping technique work.

The purpose of this **RCL b/STO b** method is to squeeze just a little more speed from the HP-41. It doesn't save bytes. Whether this method is economical for your particular program or a waste of time depends on how many instructions you have to add to put the recalled contents of register b in X at the right time.

A far more sophisticated application of a similar idea is used in Clifford Stern's high-speed Morse Code progam "MC". Refer to pages 151 to 158 of Keith Jarett's Synthetic Programming Made Easy. Appendix B of SPME thoroughly explains how and why "MC" is so fast, and may spark some ideas that will apply to your programs.

If it is at all possible, write your programs in such a way as to avoid using functions that are time consuming. Use the function times listed in Appendix A of SPME or on pg. 11 of SQRG to make comparisons. In general, logarithmic, statistical, and trigonometric functions should be avoided within a loop because of their inherent slowness. As an example, Σ+ seems like a nice, one-byte way of accumulating X and Y. If that's all you need (with the summation register location = 11), why not use ST+ 11, RDN, ST+ 13? Although it's four bytes longer, it's 60% faster.

Another example involves the use of $Y\uparrow^{X}$ or $10\uparrow^{X}$ in a base conversion (or similar) program. This approach is attractive because the program is more straightforward and easier to write. But instead of recalling the base, recalling the counter and raising Y to the X power each time through the loop, change the nature of the counter. Instead of counting 0, 1, 2 and so on, and raising the base to the power of the counter, start with a counter value of one. Then at the end of each loop, multiply the counter by the base with ST* nn. The overall effect is the same, but the method and speed of the two approaches are much different.

Examining different alternatives will often result in finding ways to save time. This might include rewriting the equation involved or using a different method or equation. Comparing different ways of achieving the same result will often give you new insights, and is time well spent. You can save time in this way even with trigonometric functions, which are notoriously slow.

For example, when you have to multiply complex numbers within a loop, use the rectangular representation directly instead of using the much slower **R-P** and **P-R** conversions. Specifically:

$$(a+bi)*(c+di) = (ac-bd)+i(bc+ad)$$, and

For division it may be faster to use:

$$\begin{aligned}(a+bi)/(c+di) &= (a+bi)*(c-di)/(c^2+d^2)\\ &= [(ac+bd)+i(bc-ad)]/(c^2+d^2)\end{aligned}$$

In general, it isn't easy to get around using trig functions when they're needed, but sometimes a single **R-P** can replace two trig functions. However, don't be discouraged if you find a case in which execution times cannot be improved much.

3E. LABELS

An Alpha (global) label is needed to make your program show up in Catalog 1. But using more characters than are necessary to make the label unique is a waste of bytes. Two or three characters in the name will usually be sufficient. It's also wasteful to use more than one Alpha label in a program unless it the program contains separate sections or subroutines that will be executed by other programs. Most programs require just one Alpha label, which serves as the starting point.

The two-byte local labels A through J and a through e give easy USER mode access from the keyboard to as many as fifteen different sections of your program. It's a good idea to make use of A through E and then a through e before using F through J, because the second row of keys is often reassigned, and you may not want to disturb the default X<>Y and RDN functions of the F and G keys. Remember that any key assignments have priority over local labels. Press and hold a key to check its function. Using these two-byte local labels instead of Alpha labels will save many bytes.

The one-byte labels (LBL 00 to LBL 14) are found in row 0 of the byte table. They should be used in most of the remaining cases when a label is needed. When the HP-41 searches for numeric labels (because the jump distance is not compiled), it begins at the line after the branching instruction. From there it searches downward (toward higher line numbers) until the label is found or until the END of the program is encountered. If the END is reached before the label is found, the search continues from the first line of the program. Because of the way this search proceeds, it is possible for your programs to use more than one label with the same number. You just have to make sure that the correct label is encountered first. Here is a simplified rule for repeated use of a label: Reuse labels for forward branches only (where the GTO or XEQ is at a lower line number than the matching LBL), and make sure that the branches don't cross each other.

Because labels 00 through 14 can be easily reused for forward branching, your programs should seldom need to make use of the two-byte numeric labels 15 to 99. An exception is the need for special LBLs in conjunction with GTO IND or XEQ IND instructions, as discussed in Section 2D. In general, your program should have one Alpha (global) label, and possibly local labels A to E and a to e marking off major sections that need to be separately accessible. The remainder should be one-byte labels 00 to 14, unless you are using labels 15 to 99 as the object of GTO IND or XEQ IND instructions (as in Section 2E).

The only other time when labels 15 to 99 are desirable to use is when you don't want to use synthetic programming techniques. If this is the case, when the distance from a GTO to its matching label is greater than or equal to 112 bytes, use labels 15 to 99 and a GTO with the same number. Failure to follow this advice will result in a GTO whose jump distance is too large to compile, and as a consequence, execution will be slow. More information on this subject is presented in the next section on GTOs.

Synthetic two-byte labels with any of the 256 possible postfix (second byte) values can be made using the byte grabber or Text 0 prefix assignments. Details of these synthetic instruction-building techniques can be found in Sections 4B and 4C. The "LB" inputs (for use with a byte loading program) for a two-byte LBL are 207, nn. Look at the second row in each square within the byte table to see how each of these postfixes will display with decimal value nn. Labels 100 and 101 are exceptions; they display as LBL 00 and LBL 01 respectively.

Most of these labels can also be created using a synthetic key assignment of 207, nn, with nn corresponding to the label number. However, this seems to be limited to postfixes 15 to 254. This behavior may vary from machine to machine. Postfixes of 128 or greater will create labels which display as LBL IND nn. These labels are normally useless because neither XEQ or GTO can branch to them. As an example, GTO IND 00 does not branch to LBL IND 00. Instead, it uses the contents of register 00 to determine the value of the numeric label to search for.

If you are a ZENROM owner, nonsynthetic labels are entered in the same manner described in the HP-41 Owners' Manual. However, you can also key in numeric and global labels which are usually considered synthetic. This is done directly from the keyboard. The ZENROM allows synthetic suffixes for all instructions. In contrast, the CCD Module only allows synthetic suffixes to be directly keyed for RCL, STO, and X<> prefixes. Other two-byte synthetic functions must be keyed in by decimal equivalent, using XEQ, ENTER, byte1, byte2.

The entry of two-byte numeric labels using the ZENROM can best be summed up with the chart below. The decimal byte values for all of these labels have "LB" inputs 207, nnn. The postfix nnn and the keystrokes needed to create LBL nnn are listed below. These keystrokes also apply when creating GTO and XEQ instructions. Because of this, shift LBL is omitted from the list. Remember you need to press shift LBL, shift GTO, or XEQ to create one of these functions.

POSTFIX	DECIMAL	
DISPLAY	POSTFIX	KEYSTROKES
00	100	EEX, 0, 0
01	101	EEX, 0, 1 or EEX, Σ+
T	112	EEX, 1, 2 or (.) T
e	127	EEX, 2, 7 or (.) e
IND 00	128	EEX, 2, 8
IND 71	199	EEX, 9, 9

You can use the features of the ZENROM to make global labels containing any of the 256 possible byte values without any restrictions. The USER ALPHA mode can be used for both the lowercase letters (all but a - e are shown as ▓) and all of the displayable characters which are not normally available. The USER ALPHA keyboard is shown on one of the two overlays supplied with the ZENROM, and on page 57 of the ZENROM manual.

In addition, the ZENROM has a character entry feature called "SYNTEXT". This allows the entry of any character in a label, global assignment, text, GTO or XEQ function, no matter which Alpha mode you are in. Simply press shift, ALPHA and fill in the two prompts with hexadecimal digits (row, column) from the byte table. The corresponding byte is entered as if it had just been keyed in.

3F. BRANCHING USING GTO

Avoid Alpha GTO

Don't use an Alpha GTO in a program except to branch to a different program in Catalog 1. An Alpha GTO sends the 41 searching for a matching label, beginning at the bottom of Catalog 1 and continuing upwards. The time it takes to find a matching label will be fairly short when the program is at the bottom of Catalog 1, such as when first developing the program. But the search time gets longer as programs are added between the bottom of the catalog and the program searched for. This search time for a matching Alpha label (roughly .016 seconds per Catalog 1 LBL or END) can exceed a full second in some cases. If you need to return to the beginning of your program, rather than using an Alpha GTO, you should add a numeric label immediately following LBL "xx" and use a matching numeric GTO.

There are two reasons why you shouldn't use an Alpha GTO or XEQ to branch to a ROM (plug-in or module) program. The first reason is speed. All of Catalog 1 is searched before the HP-41 checks Catalog 2 for the program. In Catalog 2 there are often several modules that have to be checked before the HP-41 even gets to the module that contains the function you want. This search process is therefore virtually guaranteed to be slow.

Another reason to avoid the Alpha GTO or XEQ for ROM functions is its greater byte count. An XROM "xx" instruction uses just two bytes. Compare this to the four bytes taken by an Alpha GTO or XEQ for a function that has a two character name. Using an XROM instruction will save even more bytes for longer program names.

Synthetic three-byte GTOs

Use the normally keyable two-byte GTO 00 to 14 for branching to one-byte labels 00 to 14 if the jump distance is less than 112 bytes. See the following section, 3G, for help in counting bytes to determine the jump distance. When the jump distance is greater than or equal to 112 bytes, a three-byte GTO is needed to contain the compiled distance. Three-byte GTOs that branch to the one-byte labels 00 to 14 aren't normally keyable. Nevertheless, there are four synthetic methods available to create these GTOs.

The first technique involves using a byte loading program to make the needed instruction. Byte loading programs such as "LB" or "LBX" will create the required function when you use inputs 208, 0, n, where n equals the label number.

The second procedure used to create this synthetic GTO requires the byte grabber key assignment. At the location of the needed GTO instruction, in PRGM mode, key in **RCL IND 88**, **ENTER**, **LBL 00**, BST to the step above the **RCL IND 88** instruction, and byte grab to make a three-byte **GTO 01**. The same procedure will create a three-byte **GTO 00** if you backarrow the **LBL 00** before byte grabbing. For other values, look in row 0 of the byte table for the correct third instruction. Byte grab the RCL byte and backarrow to clean up. If this isn't clear to you, refer to Section 4B for more complete instructions on using the byte grabber.

The third approach to synthesizing a three-byte GTO involves the use of a Text 0 prefix (240, 208) assignment. The process is too lengthy to repeat here. Refer to Section 4C for instructions and a full explanation of Text 0 prefix assignments.

The fourth approach uses a completely different class of key assignments than the last two methods. Credit goes to Gregor McCurdy for publishing an excellent article on these prefix 4 assignments in the Oct./Nov. 1982 (V9N7P9d-10b) PPC Calculator Journal. Assign 4, 213 to a convenient key with a synthetic key assignment program like "MK" or "MKX" before you continue.

Several different GTOs can be created with this assignment, depending on the keys you press to fill in the numeric prompt. Check that the assigned key shows d<__ when pressed. Backarrow to cancel it. Now position yourself to the place where you want to put the synthetic GTO and enter PRGM mode.

As an example, let's create a three-byte GTO 01. Press the assigned key followed by SHIFT 0 1 (or SHIFT Σ+). You should now see GTO 01. You can confirm that this is a three-byte function using the byte counting methods described in the next section, or with the help of more advanced techniques presented in Section 4I.

To synthesize a three-byte GTO T, press the assigned key followed by SHIFT . T. This method is limited to the five postfixes 112 to 116, which correspond to T, Z, Y, X, and L (unless, of course, you have a ZENROM, which permits synthetic suffixes to be entered from the keyboard for all instructions). All matching labels for these GTO instructions with row 7 suffixes are synthetic.

The table below lists all of the various possible inputs, and the GTOs which are produced by this assignment.

PRGM mode input	GTO produced
00 to 14	two-byte GTO 00 to 14
15 to 99	three-byte GTO 15 to 99
IND 00 to IND 99	three-byte GTO 00 to 99
IND (.) T, X, Y, X or L	three-byte GTO T, Z, Y, X or L

Feel free to experiment with this. If you own a ZENROM, you will find that other inputs are possible with this function. Also, there are three other functions, with different prompts, that act the same as the 4, 213 assignment. They also have a prefix of 4, and their postfixes are 209, 219 and 222. 4, 208 is the normal GTO function. Unexpectedly, assignment 4, 203 can be used for GTO .___, GTO .. ,

GTO Alpha (including local labels A to J and a to e), GTO IND 00 to 99, GTO IND T, Z, Y, X, or L. More in accord with its location in the byte table, this assignment can also be used to create a normal END with inputs 00 to 99.

The run mode behavior of the prefix 4 assignments with postfixes 209, 213, 219, and 222 is somewhat unpredictable. Most of the time, they will act like a regular GTO and branch to the label that corresponds to the number you key in. You can GTO labels 00 to 99, synthetic labels IND 00 to IND 99 as well as IND T, Z, Y, X or L. But once in awhile, the assignment will behave like a compiled GTO, and unexpectedly jump some number of bytes in either direction.

Note that the COMPILE function of the ERAMCO MLDL Operating System ROM will automatically change a two-byte GTO into a synthetic three-byte GTO when a two-byte instruction is not sufficient to hold the jump distance to the matching label.

Label-less Programs and Avoiding Decompiling

One of the most difficult tasks to perform is putting together a working version of a program completely without labels, doing the work of computing the jump distances of the branching instructions by hand. This tedious job is made easier by the ZENROM's RAMED function. At least the business of changing a GTO or XEQ function to be a compiled instruction is made easy.

This is one of the more esoteric aspects of synthetic programming. For the few bytes saved, much effort is expended. One wrong move, and all of the pre-compiled jump distances are lost. The first thing you need to learn before you can construct a program without labels is how to avoid decompiling.

The HP-41 clears the jump distance information from branching instructions after they are invalidated by an operation such as using DEL, backarrow, inserting instructions (in PRGM mode), and so on.

This is controlled by nybble 6 of the END (or .END.) instruction immediately below your location in program memory. Whenever you do something to invalidate the jump distances, the HP-41 Operating System (OS) sets nybble 6 of the END or .END. to hex value F (See Table 4.1, page 150 in Section 4G). Later, while leaving PRGM mode, packing, or at turn-on, the OS will zero jump distance information within the domain of any END or .END. functions if the appropriate bit (third) of nybble six is set. This erasure is known as decompiling. Section 4G discusses ways to avoid decompiling. Only the use of RAMED is covered here.

Besides preventing the OS from erasing jumps to nonexistent labels, you can avoid decompiling to preserve the compiled status of a program brought into main memory from mass storage or Extended Memory. It sidesteps slowing the program execution down to recompute the jump distances.

After reading in the program, enter PRGM mode. Be sure NOT to exit PRGM mode until finished, or decompiling will take place. Now press BST twice. You should see the last line of the program in the display. Now press XEQ, ALPHA, E N D, ALPHA. Then execute RAMED, press USER, and change the hex 0F byte to 0 9 or 0 0. Press ON to exit RAMED mode and press PRGM to return to run mode. That's all there is to it.

The "ReNFL" program below is simply a rewritten version of the program appearing in 5B. A total of 5 bytes are saved by the removal of the one-byte labels within the program. You will need to create a packed END as per the above procedure if the program is to be read in from mass storage or Extended Memory. Failure to do this will result in NONEXISTENT until the proper values are stored within the nybbles that hold the jump distances.

```
01 LBL "ReNFL"       20 X=Y?         39 X<> M
02 44                21 GTO 04       40 ALENG
03 POSA              22 RDN          41 XEQ 02
04 X<0?              23 X<>Y         42 X<>Y
05 "⊦, ▩▩▩▩▩▩▩"      24 CLA          43 X<> M
06 X<0?              25 DSE X        44 NSTOM
07 GTO 01            26 XTOA         45 RTN
08 AROT              27 X<>Y         46 7
09 ATOX              28 NRCLM        47 X<>Y
10 RDN               29 STO M        48 -
11 XEQ 02            30 ASHF         49 DSE X
12 X<> N             31 RDN          50 X≠0?
13 STO a             32 X<>Y         51 X<0?
14 X<> M             33 ATOX         52 RTN
15 191               34 -            53 "    "
16 CLA               35 DSE X        54 GTO 05
17 XTOA              36 GTO 03       55 END
18 X<>Y              37 RTN
19 NRCLM             38 RCL a        107 bytes
```

Line 07 = hex B2, C2. Line 11 = hex E4, 08, 02.
Line 21 = hex B5, 23. Line 36 = hex B4, B4.
Line 41 = hex E2, 01, 02. Line 54 = hex B6, B1.
Line 53 appends one space.

As before, line 05 appends a comma and seven FF bytes to Alpha (hex F9, 7F, 2C, FF, FF, FF, FF, FF, FF, FF), and line 53 appends a single space (hex F2, 7F, 20). Programs read from barcode by the wand do not have compiled jump distances, so you will have to do the work of changing the GTO and XEQ instructions using RAMED. If you are keying it in by hand, you might as well use GTO . . to attach the END to the program. The program must be packed (don't use the previous procedure, as the nulls before lines 02 and 46 won't be removed, throwing off the jump distances), unless you use RAMED to enter lines 02 and 46.

Like the program in 5B, "ReNFL" can be used in two ways. You can rename a file whose header register is within the Extended Memory of the Extended Functions module / HP-41 CX. With "ABC,DEF" in Alpha, "ReNFL" will change file "ABC" to "DEF". If Alpha contains just a single filename (no comma), the header register with a matching name will be replaced with seven hex FF bytes. This is interpreted by the OS as the end of XM, effectively removing that file and all those that follow.

There are two basic methods that can be used to compute the values for the jump distances within the branching instructions. The first would be to use the byte counting techniques described in 3G to determine the size of the jump (RCL b method using "CB"). A look at the program listing tells the direction of the jump (compare with the version that has the matching labels in 5B). Then a little conversion from decimal to binary to hex with SQRG page 39 (Function Structure) as a guide should give you the right values to key in with RAMED.

The second, far easier way is to look at a compiled version of the program that has the labels in place, and figure out how the relative jump will be changed by the deletion of the labels. In the case of the jump at line 41, there is no change! The jump is still 2 bytes, direction = 0, 1 register from the first byte of the XEQ instruction to the instruction where execution should resume (7 in this case).

The structure of the GTO at line 07 needs only a small change to make it jump upward one byte less. The original compiled value of B2, D2 needs to have 1 subtracted from the third nybble (D), giving hex B2, C2. The new line 11 jumps two bytes less because of the deletion of labels 03 and 04. Hex E8, 08, 02 becomes E4, 08, 02. The GTO 04 instruction at line 21 does not change, since there are no labels deleted to change the jump distance. However, the GTO 03 instruction at line 36 needs to account for the deleted label because of the direction of the jump. Hex B4, C4 becomes B4, B4. The GTO at line 54 also jumps one byte less, changing from B6, C1 with label 05 in place, to B6, B1 without the label.

There are other interesting possibilities for branching functions with pre-compiled jump distances. Since the HP-41 OS does not check for a matching label when the jump distance (or direction) bits are nonzero, you can go wherever you like. And because the program subroutine return stack stores absolute addresses, you could even have an XEQ instruction jump into the middle of another program and return properly! There is no problem jumping over ENDs with pre-compiled jumps. This behavior is similar to that in 4I. Short form (two-byte) GTO instructions are limited to jumping 112 bytes when pre-compiled. The long-form GTO and XEQ instructions have a 12-bit field for the relative jump distance, making 512 register jumps possible. Direct jumps into Extended Memory are possible, though it requires a fixed absolute address for the program that makes the jump. Because of these severe restrictions, no example will be given here to demonstrate this. We leave it to those interested to explore.

GTO .000

An interesting synthetic assignment can be used to move to line 000 of a program, just like executing RTN in run mode. This saves keystrokes if you are in PRGM mode. The technique was discovered by Roger Hill.

In order to use one of these assignments, you must have a non-prompting, non-programmable function present. The Card Readers' VER function and the MCED function of the ZENROM are good examples. Letting Aa bc represent the hexadecimal bytes of the selected non-programmable, non-prompting function, the synthetic assignment Ca bc will execute a non-programmable RTN.

For example, MCED is XROM 05,06, which is hex A1 46. If you have a ZENROM present and you assign C1 46 to a key, the assignment will preview as MCED. Similarly, VER is XROM 30,05, or hex A7 85. If a

Card Reader is present and you assign C7 85 to a key, the assignment will preview as VER. Both assignments clear all pending returns and move you to line 000 of the current program, just as if you had executed RTN or END in run mode. To get to line 01 in PRGM mode, press the assigned key followed by SST.

3G. BYTE COUNTING TECHNIQUES

There are several different methods which can be used to count program bytes. The most obvious method is to count the bytes by hand, looking up the byte count for each instruction and tallying the total. This is fine for short programs. A copy of Jeremy Smith's HP-41 Synthetic Quick Reference Guide (SQRG page 39) or a copy of Keith Jarett's "HP-41 Synthetic Programming Made Easy" (pages 57 to 63) will help with this. When you need the byte count of long instruction sequences or whole programs, other methods are faster and easier. Also, synthetic GTO instructions can be three bytes rather than the normal count of two, and you cannot tell the difference by just looking at the display or a printed listing.

Byte Counting with the Printer or with the HP-CX

If you have either a printer or an HP-41CX, you can find out very easily how many bytes an entire program uses in main memory. First, be sure to PACK to eliminate any leftover nulls, and set flag 21 to enable the printer. Select TRACE mode on the printer and list CATalog 1. If you have an HP-41CX, just list CATalog 1 (you don't need a printer). All of the labels and ENDs in main memory will be listed, with a byte count accompanying each END. If you want to count the bytes within a program, you can insert END instructions to partition that section of the program off from the rest. In PRGM mode, XEQ "END" where needed, PACK, and CATalog 1 as before. The byte count returned includes the bytes from the line after the previous END in Catalog 1 (if any; otherwise from the first line of program memory) down to and including the END that accompanies the listed number of bytes. When you are done counting bytes, backarrow any extra END lines you inserted and PACK again. This method is a little clumsy, and it decompiles your program, but it gets the job done.

Byte Counting with the CCD Module

The CCD Module's functions PLNG (Program Length) and PPLNG (Programmable PLNG) will give you an immediate byte count of the program area that contains a particular Alpha label that you specify. If you don't specify a global label name, you'll get the length in bytes of the current program.

Byte Counting With Extended Memory Files

If you have Extended Functions or an HP-41CX, you can get the byte count of a program after a copy of it is saved in Extended Memory. If a copy already exists, key the program file name into Alpha and XEQ "RCLPTA". If there isn't a copy in XMemory, key the name into Alpha and execute the sequence SAVEP, RCLPT. A short program which automates this procedure appears in "HP-41 Extended Functions Made Easy" (XFME). Refer to the index on page 264 of XFME under "CBX".

One very useful feature of SAVEP is that you are allowed to save programs in Extended Memory under names other than the ones used within the program. You simply key into Alpha the same letters that appear in a Global label within the program, add a comma, and then key in the name you want it saved under. When you execute SAVEP, only a program with an identical file name will be overwritten. This is useful when you make changes to a program after saving it with SAVEP, and you want to compare byte counts.

Let's use a program named "EM" as an example. Assuming a label "EM" exists, press SHIFT CAT 1 (to make sure you are not in a ROM module; this avoids a potential MEMORY LOST with the buggy Revision 1B Extended Functions), then ALPHA E M ALPHA and XEQ "SAVEP". This saves a copy of the unmodified program in case you don't like the results of your changes. Make the modifications and PACK after testing the program. Now key in "EM,EM1" and XEQ "SAVEP". Do not put a space after the comma! Now retrieve the byte count of "EM1" (the "working file") with RCLPT. Compare this to the count for the original "EM" by using RCLPTA. RCLPTA will ignore the comma and following characters, and use the letters before the comma as the file name.

This method is efficient because you avoid ever needing to go back and undo changes you made. This cuts down the number of times you have to PACK, and saves time. It also helps in case you forgot something along the way or deleted a line too many in the process. It gives you something to fall back on. If you don't like the changes, get rid of the modified version in main memory with CLP or PCLPS and read in the original from X Memory. If you like the changes, you don't need to do anything to the program in main memory.

Whether you like the changes or not, you still have an X Memory file that may not be needed. If you don't want this file, put the name in Alpha and execute PURFL. If a revision 1B Extended Functions module is in your machine, follow this with EMDIR as demonstrated in the "PK" program in Section 5B. Otherwise you may lose your entire Extended Memory directory.

Byte Counting Using RCL b in RAM or ROM

The RCL b function recalls the current program pointer to X. The two rightmost bytes contain the current address within program memory. The other five bytes may contain pending returns if RCL b is used in a running or prematurely halted program, but this does not concern us here. We need to decode the pointer information in the rightmost 2 bytes. Using this capability you can find the distance between two pointers in bytes by subtracting their decimal equivalents.

In order for this technique to work, you need to be able to execute a RCL b key assignment, or you need a ZENROM or CCD module. These last two are custom modules which allow you to execute synthetic functions (in particular, RCL b) from the keyboard in the same way normal instructions are entered. These modules also perform other special functions.

To make the synthetic RCL b key assignment (not necessary if you have a CCD Module or ZENROM), you need a synthetic key assignment program like the PPC ROM's "MK", Tapani Tarvainen's "MKX" from SPME, "ASG" from XFME. Use inputs of 144, 124 and the keycode of the key to which you want RCL b assigned. You will need to execute RCL b from the keyboard to try the examples in this section.

Next you need to know how to convert the recalled program pointer to a decimal value. If you have the PPC ROM, you can use the "PD" (Pointer to Decimal) routine to convert a RAM pointer to a decimal address. (This pointer decoding program does not work properly with ROM addresses because the ROM pointer format is different.)

If you don't have the PPC ROM, you can use the following program instead. You'll need an HP-41CX or the Extended Functions module. If you don't have either of these, you can use the set of programs on page 86 of SPME to decode RAM and ROM pointers and count bytes.

```
01 LBL "CB"
02 X<>Y
03 XEQ 10
04 X<>Y
05 XEQ 10
06 -
07 RTN
08 LBL "PD"
09 LBL 10
10 "*"
11 X<> M
12 STO N
13 RDN
14 ENTER↑
15 CHS
16 ALENG
17  E
18 -
19 X≠0?
20 ATOX
21 ABS
22 ATOX
23 R↑
24 STO M
25 RDN
26 LASTX
27 16
28 ST/ T
29 MOD
30 LASTX
31 X↑2
32 *
33 +
34 7
35 *
36 +
37 INT
38 RCL M
39 X<>Y
40 CLA
41 END
```

Line 02 of "CB" swaps the two pointers to decode the one in Y first. Line 03 executes label 10 which is "PD". This converts the pointer to a decimal number. Line 04 swaps this decimal pointer number with Y, putting the pointer that hasn't been decoded back into X. Line 05 converts it to decimal. Line 06 subtracts the two decimal values, yielding a byte count.

Label "PD" and label 10 are one and the same, but Alpha XEQ instructions use more bytes and are slower than numeric executes. Lines 10 through 12 and 15 have the effect of putting the contents of X in Alpha register M and removing all but the two rightmost bytes. Lines 13 and 14 get rid of the pointer that was in X, replacing it with a copy of Y. Line 15 follows the ENTER↑ instruction to disable the stack lift, otherwise a RCL X function would have to be used here.

Line 16 returns the length of Alpha to X, which will be zero or one after lines 17 and 18 decrease it by one. If there is only one character in Alpha, line 19 will skip the first Alpha TO X instruction. Line 21 functions only to copy X into L. Line 22 decodes the rightmost character in Alpha, leaving Alpha empty and the decimal character code in X.

Lines 23 and 24 push the other pointer into X and save it in Alpha register M, so it isn't lost. Lines 25 and 26 replace it with the decimal character code of the left-had pointer byte. After line 27, this number is also in T. Line 28 divides this by 16, extracting the left nybble. Line 29 separates the right nybble from the decimal value, discarding the rest. Lines 30 and 31 recover the number 16 from L and square it, giving 256. 256 is multiplied by the right-hand nybble of the left byte and added to the right byte value by lines 32 and 33. The result is multiplied by 7 by lines 34 and 35. Line 36 adds this to the value of the leftmost nybble which was isolated by line 28. Line 37 removes any fractional artifact of that division. Lines 38 to 40 recall the other pointer from Alpha register M, place it in Y, and clear Alpha.

The "CB" (count bytes) and "PD" (pointer to decimal) routines listed above total 67 bytes. If you don't need the byte counting part of the program, you can delete lines 01 to 07 and line 09. The byte count for "PD" alone is 51 bytes.

When you use either the PPC ROM version of "PD" or the version above, you may want to press ENTER after you RCL b. Do this when you're decoding one address. Both programs preserve the contents of Y, so after you're done decoding this address, you can use the contents of Y to return to the address you decoded. If you used the PPC ROM version of "PD", press CATalog 1 and stop it anywhere. This returns you to RAM. Then for either version, press X<>Y, STO b. You avoid going all the way through Catalog 1 to find your program, or using an Alpha GTO. This saves time.

Both the PPC ROM's "CB" routine and the "CB" program here execute their respective versions of "PD" twice, and subtract the decoded decimal addresses to give the byte count between these two addresses. For a correct count, position yourself in program memory to the first line you want counted, and RCL b. Then move to the line after the last line you want to be counted and RCL b again. Now XEQ "CB". The result is the number of bytes from the line at address Y up to and including the line before address X.

If you can't position yourself to the line after the last one you want to be counted, manually add the number of bytes in that line to the number returned by "CB" after using the last line's address for the second entry. This will ordinarily be necessary only when the last instruction is the .END. or an END. If the last instruction is an END, just add three bytes to the total. If you use the address of the .END. for the second entry, the byte count for the program will be correct. However, you still need to add three bytes for a correct total, because a three-byte END will have to be added to this program before more programs can be added to main memory.

As an example of using "CB", position yourself in main memory by executing CATalog 1 or an Alpha GTO to a program whose first line is an Alpha label. Press RCL b. Now use CATalog 1, stopping at the Alpha label following the END of the first program. RCL b again. Now XEQ ALPHA C B ALPHA. The byte count will be returned for the first program. This value will be correct, including the END, as long as the second Alpha label was also at line 01. It may be easier for you to start at the first line of the program whose bytes you want to count, press RCL b, BST (to the END), RCL b, XEQ "CB", and then add three to the total returned.

When you count bytes within a program, the easiest method is to RCL b in run mode at the first line you want to count, go to the last line you want to count, SST (watch that this doesn't cause a jump beyond the next line) and RCL b again. Then execute "CB" to count the bytes. Remember that the byte count returned includes all of the bytes starting from <u>and including</u> the first address, extending through the line before the second address.

Both versions of "PD" are made to decode RAM addresses. Trying to use "PD" to decode a ROM address will yield an incorrect number. The reason for this is that RAM and ROM pointers have different formats. In RAM the first of the four nybbles in the program pointer indicates the byte number within the register specified by the remaining three nybbles. However, in return addresses and ROM addresses, this first nybble indicates the port or internal ROM number. For more information on this, refer to pages 22-23 of this book, or page 115 of SPME or page 38 of SQRG. The details of this arrangement aren't important here. What <u>is</u> important is that you know that even though ROM addresses are not correctly decoded by "PD", you can still use the "PD" output to compute the correct result.

Because ROM programs execute opposite the direction of RAM programs, from lower to higher addresses, the sign of the difference in bytes will be incorrect. Also, "PD" interprets the difference in ROM bytes as a difference in RAM registers of 7 bytes each. So when you use RCL b and "CB" to count bytes in ROM, divide the result by -7.

Let's use an example of counting bytes that does not require the PPC ROM, but uses a ROM you probably have available. Turn your HP-41 off and plug the Optical Wand into one of the ports. Press ON and GTO "WNDTST". RCL b, GTO .999 and RCL b again. XEQ "CB". Now press 7, CHS, / to divide by negative seven. The resulting byte count of 55 includes lines 01 to 29 of "WNDTST", and does not include the END. You can confirm this is correct by using the COPY function to make a duplicate of this program, GTO .. , and follow the same procedure as before. This can be applied to any ROM containing user-code programs. Just don't cross over END instructions in the ROM, and the byte counts will be correct.

Counting bytes with RCL b has an advantage over other methods in that any portion of a program can be counted, not just the whole program. And no extra copies of these programs or instruction sequences have to be made. Unless you have a CCD Module with its built-in PLENG and PC>X functions, I'm sure you'll find it worthwhile to use a little RAM space for a copy of "CB" and "PD".

Figuring Tracks and Records

If you have any sort of mass storage device, counting bytes can be very valuable. The number of bytes a program takes up on the storage medium can have several effects. The controlling factor is the way the program bytes are grouped on the recording medium.

If you own the Card Reader, you need to keep in mind that 112 bytes fit onto one track (one side) of a magnetic card. Up to 224 program bytes fit on a single card. If you mainly use the card reader to store your programs, you may want to try to tailor your programs with this fact in mind. As an example, if you have a program whose length is 230 bytes, it's easy to see that trimming off six bytes would enable you to fit the program onto a single card, completely filling both tracks. This eliminates the need for a second card. It also gets rid of a half-used card, which is a nuisance.

Another approach would be to trim off just three bytes from the program. Then when your record the card or read it back in, simply press backarrow when you're prompted for the third track. This leaves off the three-byte END, and has no important effect besides avoiding the need for another card. You should mark the card or cards to remind you of this for when you later read them in, so you don't go looking for the "missing" card. A backarrow symbol next to "#3" can serve as a reminder on the card.

Of course, if you make any changes to the program that add bytes, you will once again need another card. But for programs that don't often change, the time it takes to shorten a program enough to fit on one less card is well spent when the byte count is just over a multiple of 224.

If you save programs on the Digital Cassette Drive, you should be aware that all of the recorded information on the tape is grouped into chunks, called records, of 256 bytes. When you execute NEWM, what is actually happening is the tape is being marked off electronically into these records. The tape is unuseable by the Drive without the record markings.

You should know that one mini data cassette holds 131,072 (128K) bytes. This is composed of 512 records, each containing 256 bytes. The first two records are used for housekeeping, leaving 510 records for the directory and files. One record in the directory will hold up to 8 filenames. For this reason, the number used to fill in the NEWM prompt should be a multiple of eight. Since the microprocessor within the Cassette Drive keeps track of one entire record, access to another file within the same block of eight files will usually be swift after an initial operation (such as DIR or READP), because the directory registers at the beginning of the tape will not have to be read for the starting address of the file.

The last byte of the last record of a program on cassette tape is used for a checksum. Therefore, the longest program that can be stored in a single record is 255 bytes long. The next increments are 511, 767, and 1023 bytes. This fact may seem trivial when you first record the program, but it may make a difference later. The other thing you should know is that that Cassette Drive will not re-use vacated records unless the vacated records were once a part of the original program file or it is the last file on the cassette tape. Each file must begin with a new record.

Let's use an example to illustrate a point. Suppose you have a fairly large program that you use often. You usually read it in, use it for a little while, and clear it from memory. You recorded it at the beginning of a cassette to make access quick. You use it for some time, and it seems to work well. In the meantime, you add more programs and write-all sets after the program on the tape. One day you find an error that went unnoticed. The fix for the problem pushes the byte count up from 750 to 771 bytes, and you record the new version on the same tape. It seems to take a long time for the WRTP to finish. Now every time you need the program it takes a lot longer to read it in.

You can avoid this kind of problem by "padding" a program with a small section at the end of the program that does nothing before you first record the program. A series of nulls will serve this purpose as long as you are careful not to remove them by PACKing. Add enough nulls (or NOPs) between the last line and the END to tip a borderline byte count over the next multiple of 256. You can figure out the bare minimum number of bytes you need to add by taking the present byte count and press 256, MOD, LASTX, X<>Y, -. This way, if you later make some changes to the program which require more bytes, the position of the program on the tape can stay as it is.

An easy way to add nulls to a program is to position yourself to the END while in PRGM mode and key in any function. Backarrow the function and there are seven nulls, assuming the program was initially packed. You can SST to the END again, key in a function, backarrow it, and another seven nulls will be added. Just be careful not to backarrow the END. To find out how many registers you need to open (which is the number of times you need to repeat the above procedure) divide the number of bytes that you computed above by seven, and round up to the next higher number.

If you have the ZENROM, use RAMED to change the third byte of the END to hex 09 (from 0F) while still in PRGM mode. SST to the END (or .END.) of the program, and (still in PRGM mode) XEQ "RAMED". Press PRGM until you see 00 Cx xx in the display, then press PRGM twice more. The third byte of the END (or .END. is now in the center of the display. Replace it by keying in X9 where X is the same nybble that currently appears at the left side of the center portion of the display. This change keeps the HP-41 from erasing the compiled jump distance information as it exits PRGM mode. Otherwise, you will need to recompile all branching functions before you WRTP by running the program or SSTing each local GTO or XEQ instruction. See Section 4G (pages 160 to 161) for more information on avoiding decompiling. Press ON and PRGM to return to run mode.

GTO Instructions

Section 3F mentioned that a two-byte GTO can store a jump distance from -111 to +111 bytes. When a larger jump distance is needed, you should either create a synthetic three-byte GTO 00 to 14, or use a higher numbered numeric label with a non-synthetic three-byte GTO instruction. To count bytes for GTO jumps, RCL b at the GTO first, SST, RCL b at the label, and XEQ "CB". This procedure is convenient and easy for you to execute. For a successful two-byte GTO, the acceptable "CB" count is from -109 to +113 bytes, inclusive.

There are more direct methods to test if the two-byte GTO will work well, or if a three-byte GTO is needed. They don't really involve byte counting at all, but they deserve to be mentioned here. Section 4I covers a synthetic assignment called the Byte Jumper which allows you to copy program bytes into Alpha without altering program memory. After you PACK and SST the GTO to compile the jump distance (if it can be compiled), Byte Jump over the GTO, check the bytes in Alpha. If the second byte of the GTO appears as a null, indicating no stored jump distance information, then the jump distance was not compiled. Note that you should <u>look</u> at Alpha to do this, as the null will disappear when you use ATOX.

If you have the ZENROM, you can directly look at the second byte of the GTO using the RAMED function in PRGM mode. If the two hex digits of the second byte are 00 after packing and trying to compile as described above, the jump is too long to compile.

3H. MISCELLANEOUS TIPS

This section contains some tips for your programs which do not fit neatly into the previous categories of Chapter 3. An example would be to figure out the best way to double X. Problems like this are best approached by listing all of the different ways you can think of doing the job. Tally the byte counts and execution times on paper using Appendix A or SQRG page 11. Then decide which method has the most desirable characteristics for that particular program.

For an example, let's look at a simple problem, doubling the number in X. Some of the possible ways to double X are:

2	63 ms	ENTER↑	11 ms		
*	37 ms	+	30 ms	ST+ X	35 ms
2 bytes,	100 ms	2 bytes,	43 ms	2 bytes,	35 ms

Since they all use two bytes, the byte count won't be a factor in your decision. Execution time (speed) is the other important quantity to weigh. The first, and most obvious way to double X (2, *), is not very fast. The sequence ENTER↑, + is more than twice as swift. However, ST+ X beats them all. At 35 milliseconds (ms), it's nearly three times faster than the first way of doubling X.

Though this is a simple example, there is another characteristic which has been overlooked: stack usage. Each of the three approaches has a different effect on the stack. The first example will leave 2 in LASTX (L). If the use of 2, * allows you to replace a later number entry line containing 2 with a LASTX instruction, you may want to use this. Speed, including any possible beneficial effects "downstream", should be taken into account for each case.

The second sequence, ENTER↑, +, leaves the original value of X in L. If this is of any value, it will clearly outweigh the 8 ms speed advantage of ST+ X.

ST+ X is unique among these three in that it is the only one that has no effect on L. Since it's the fastest, you'll most likely choose this one anyway. But the fact that L is left unchanged may be helpful in your application. Always consider stack usage.

As was pointed out in Section 3B, numeric entry lines are rather slow. Avoiding them will almost always save time. That is the primary reason the first approach of the three in the previous example was so slow compared to the others.

Another example of this is to use ENTER↑, SIGN, % to divide by 100. This is more than two times faster than either 100, * or E2, *. If X may contain negative numbers, change this series of instructions to ENTER↑, CLX, SIGN, %. Otherwise, all results will have a positive sign, as though ABS were performed. The percent function will leave the original value of X in Y, X/100 in X, and unity (1) in L when used this way.

This method of using the % function can be extended to include several other cases. When dividing by a number which is evenly divisible by 100, you may want to replace the sequence n, / with the quantity (n/100), followed by %. As an example, replace 50, / with 2, %. This doesn't apply to very many cases, but it can be another possible way of shortening the number of digits needed in a numeric entry line.

A similar situation arises when you need to raise the number in X to an integer power. The HP-41 has a dedicated function to square X (X↑2), which avoids the need to use 2, Y↑X. This will save bytes as well as time. You can also use this function to raise to the third, fourth, fifth, sixth, eighth, or other integer powers, alone or combined with multiplication. Some examples are listed below, which you will find are all time savers. None save bytes, though.

Power	Instruction sequence	L contents
3	X↑2, LASTX, *	X
3	ENTER↑, X↑2, *	X↑2
4	X↑2, X↑2	X↑2
5	ENTER↑, X↑2, X↑2	X↑2
6	X↑2, ENTER↑, X↑2, *	X↑4
8	X↑2, X↑2, X↑2	X↑4

When writing programs, there is often a need for powers of 2. Remember that you can often save time, if not also bytes, by recovering a number from L. You can also turn one power of 2 into another. The techniques previously listed apply, and, in addition, ST+ X can be used (only for 2) to increase any power of 2 by one power. I recently rewrote a program to make good use of the methods described here. A portion is listed below:

```
Before          After
 8               8
 AROT            AROT
 RDN             ST+ X
 256             X↑2
 MOD             MOD
 16              LASTX
 /               SQRT
 XTOA            /
                 XTOA
```

Notice that 8 becomes 256 by first being doubled (ST+ X) to equal 16. Sixteen is then squared. LASTX recovers 256 after MOD, and SQRT turns it back into 16. In this way, a byte is saved (in spite of the fact that the program is one line longer), and two numeric entry lines are eliminated. This may give you some ideas which apply to your programs.

STO L should rarely be used in your programs. This wastes a byte. To store the contents of X in L, substitute the SIGN function. (This will, of course, alter X.) ABS can also be used if X does not contain NNNs or Alpha data. Otherwise, it may give OUT OF RANGE or ALPHA DATA errors.

The HP-41 inherited many characteristics from previous HP calculators, including "stack lift disable". One of the unfortunate effects of this feature can be that the stack contents are other than what your program expects. This is because the running program has no way to keep track of the stack lift status in a program which is halted by the user (or by alarms) and restarted. As an example, the sequence CLX, RCL 01 in a program should put the contents of register 01 in X without disturbing Y. But if the program is stopped and restarted between the CLX and RCL 01 functions, Y will have zero in it! The stack lift is re-enabled when you use R/S (but not SST). For this reason, it is a good idea to use RDN instead of CLX before a RCL instruction if your intent is not to disturb Y. This HP-41 "bug" and its solution were discovered by Steve Wandzura.

CHAPTER FOUR - ADVANCED SYNTHETIC PROGRAMMING

4A. USING THE STATUS REGISTERS IN A PROGRAM

Many of the status registers can be used within a program in the same manner you would a normal data register. There are a number of reasons to use status registers for temporary storage in a program: to avoid destroying needed data in the storage registers, to preserve stack contents, to be able to use SIZE = 000, for a slight speed advantage, or to avoid the normalization of non-normalized numbers (NNNs).

You need not be a master of synthetic progamming to make good use of the status registers. You just need to be able to create synthetic two-byte instructions, and follow some simple guidelines. The two sections following this one will help you to create the synthetic functions needed to access these internal registers. And this section will acquaint you with using the status registers for scratch (temporary storage) and similar purposes.

The 16 status registers reside at the very bottom of the HP-41 memory, at decimal addresses 0 through 15. The register names are T, Z, Y, X, L, M, N, O, P, Q, ├ , a, b, c, d and e, respectively. The first five (T through L) are the stack registers. They are discussed in your Owner's Manual, and will not be covered here.

Stack register operations are keyable on any HP-41. If you own the CCD Module or ZENROM, functions using the eleven status registers in addition to T, Z, Y, X and L can also be keyed in directly. This "prompt-expansion" feature works on all HP-41s except very early HP-41Cs with serial numbers prior to 2035xxxxxx (unless internal ROM 0 was replaced during servicing with revision G or later). But if you make frequent use of status registers operations, having a ZENROM or CCD Module will speed up keying in a program tremendously. Frequently used key assignments for the Byte Grabber, RCL M, STO b, etc. can be eliminated forever.

With the ZENROM, STO, RCL, X<>, ISG, VIEW and other functions which access any of the sixteen internal status registers can be keyed in the same way you key in STO Z: press STO, . , Z. The one exception, append (⊦), is keyed in as R. You can also use decimal postfixes: press RCL, EEX, 24 (124) for RCL b. Indirect status register operations are keyed in by pressing shift after the function; for VIEW IND M, press shift, VIEW, shift, . , M.

With the CCD Module, only the STO, RCL, and X<> functions allow synthetic suffixes. The append suffix is keyed as . shift XEQ. Decimal input for synthetic suffixes is not supported.

ALPHA REGISTER

The Alpha register is composed of status registers M, N, O, and P. The first three can be used virtually without restriction for scratch purposes. If you are going to be using these registers in a program, it is usually a good idea to use CLA first. This is especially true if your program uses ST+ or ST- to operate on these registers. Register P can be used if your program does <u>not</u> have digit entry lines (including the lone decimal point or E) or cause a number to be displayed (PSE or STOP without Alpha ON, VIEWing numbers, etc.). These conditions, if they occur, will alter the first two bytes of P.

Of course, if you use any of these four registers for holding data, you give up the ability to use the Alpha display for messages, PROMPTs, and similar uses until the Alpha register is no longer needed by the program for scratch.

REGISTER Q

Status register Q is frequently used by the HP-41 system. Both the HP-41 itself and plug-in devices use Q for temporary storage of Alpha arguments. For this reason, Q is of very limited usefulness as a scratch register in a program.

The HP-41 uses Q for Alpha arguments with Alpha labels, Alpha GTO's, Alpha XEQ's, Alpha W's (byte 31), and INDirect GTOs and XEQs when the register addressed contains Alpha data. Q is also used by AVIEW and PROMPT, or just being in ALPHA mode (AON).

A numeric entry line alters Q, as the number is constructed in Q before it is brought into the stack (X). Interestingly, the function PI does not alter Q. Several trigonometric functions make use of Q: ACOS, ASIN, COS, P-R, and SIN. In addition, SDEV and Y↑X also use Q. The TAN and ATAN functions do not use Q.

The Time Module (or HP-41CX) functions ALMCAT, CORRECT, SETDATE, SETIME, T+X, and XYZALM make use of Q. And Extended Functions (or HP-41CX functions) ANUM, CRFLAS, CRFLD, GETAS, GETP, GETSUB, PASN, PCLPS, PSIZE, PURFL, SAVEAS, and SAVEP all wipe out the contents of Q.

The HP-82143 printer also makes frequent use of Q even if no printer functions are used. For this reason, Q shouldn't be used as a scratch pad register in a program if it is likely that this printer will be connected and active.

REGISTERS ⊢ AND e

Registers ⊢ and e contain bit maps of the assignments for unshifted and shifted keys. If these maps are erased, the HP-41 will not know to search for your key assignments, and the normal (non-USER mode) functions will be executed instead. If the bit corresponding to a certain key is set while no real assignment exists, the key will usually preview XROM 04, 02 and function as ABS.

If the data in these registers is accidentally altered, there is an easier way to restore it than reconstruction of the proper byte values by hand. Simply GTO . . and (if necessary) read in a program from any source. Programs recorded in Extended Memory, on magnetic cards, or on a cassette will all restore ⊢ and e to their proper values. This simple technique was discovered by Clifford Stern.

If you use one or both of these registers in your program, you should devise a scheme to restore their original contents within the program. If there were no unshifted (register ⊢ , or no shifted (register e) key assignments, that register could be used by your program as long as you clear it after you're done. Another way to restore ⊢ and e is to use (Extended Functions) GETSUB followed by PCLPS, with a program named in Alpha that specifies a program file in Extended Memory. The only side effects of this are an END added at the bottom of CATalog 1, and the clearing of any assignments made to keys that have Global label assignments in the program. These side effects can both be avoided if you use a special synthetic zero-byte program file invented by Clifford Stern. See pages 194-195 of "HP-41 Extended Functions Made Easy" and the "IN" program on page 198.

REGISTER c

Status register c contains the absolute address of the .END., the location of R_{00} and the summation registers. It also contains the "cold start constant". If this number does not equal 169 hexadecimal when the HP-41 checks it (such as when returning from a running program), MEMORY LOST will occur immediately. If the address of the .END. is altered, you will lose access to CATalog 1, and most likely eventually wind up with MEMORY LOST.

Don't get the idea that you can't alter register c and get away with it. On the contrary, you can even clear this register in a running program as long as you restore it before the program halts. The processor makes certain routine checks, including testing the values in register c, whenever returning from a running program, turning on, and so on. Many synthetic programs alter register c temporarily and later restore it. Most are interruptible while others aren't. The PPC ROM routine "OM" changes the curtain address to hex 010 (16 decimal). This makes it possible for key assignment programs to address the bottom key assignment register (hex 0C0, decimal 192) as data register 176. So altering register c is both useful and dangerous. Never experiment with it haphazardly: know what you're doing.

REGISTER b

It isn't practical to use register b for scratch because the two rightmost bytes contain the program pointer. If you alter them, you'll change your location in program memory. However, in Section 4H, we make use of this fact to execute bytes in the Alpha register as though they were program steps. Register b also contains the first two and a half addresses in the subroutine return stack.

REGISTER a

The **a** register normally contains pending returns 3 through 6, but it can be used for scratch under the right conditions. First, the program using register a must not be called as a subroutine by another program if the subroutine stack exceeds two pending returns. If this were to happen, the HP-41 could return to an address almost anywhere in RAM. The exponent (rightmost byte) in register a would be interpreted as the left half of the third pending return.

Second, the program may branch using GTO, but XEQ or RTN must not be used. Otherwise the contents of register a will be shifted two bytes. Lastly, register a should not be used as a scratch register in a program using the Extended Funtion (or Data Logger) PSIZE. PSIZE will alter the data in register a as though it contained legitimate return addresses in need of updating because of the shift in location of R_{00} (the curtain) and all of CATalog 1.

REGISTER d

Register d contains information about the status of the flags that is best left undisturbed. But while it isn't practical to use register d for scratch, limited information can be easily stored there for later use. An efficient way to influence decisions made later in a program is to set a flag in register d. As an example, suppose you want to terminate a program if a variable exceeds a certain limit, but the variable is not in the stack when you want to test it.

The solution is to test the value earlier in the program and set a flag if the limit is exceeded. Then, at a good place to terminate, test the flag using FS?C and terminate if the flag is set. Flags 0 to 10 are "general purpose" flags, but don't forget flags 18 to 20. Since flags 11-20 are cleared at turn-on, you can sometimes use them without having to use a CF instruction first.

You can store an integer from 0 to 255 in flags 0 to 7 using the Extended Function X<>F. The number in X need not be a positive integer when you use X<>F. However, it should be less than 256. When you use X<>F to recover that number, it will be as though ABS and INT have been performed (with no change to the LastX register).

Armed with this information, you should be able to make good use of the status registers. Registers M, N, and O can be used freely. Registers P and a require a little more care. Registers ⊢ , e and Q have serious difficulties associated with them, and registers d, b, and c should be left until you have considerable experience using the other status registers.

You may want to explore the possibility of saving the contents of the status registers in an Extended Memory data file using SAVEX and later restoring them with GETX. This requires lowering the curtain temporarily to 000. A similar technique can be used on data registers with the functions SAVER, SAVERX, and GETR. None of these instructions normalizes your data.

4B. SYNTHETIC INSTRUCTIONS USING THE BYTE GRABBER

The synthetic key assignment known as the byte grabber has been widely used by synthetic programmers to manipulate bytes in program memory. The most common byte grabber has prefix and postfix values of 247 and 63, respectively. You should be familiar with this assignment before reading either this section or Section 4I. See pages 6 to 9 of SPME or pages 231 to 244 of "Extend Your HP-41" by Mier-Jedrzejowicz if you need to refresh your memory. The conventional practice of using the byte grabber to absorb the prefix from a multi-byte instruction in program memory will not be described in detail here.

You will need to have a byte grabber assigned to a key in order to experiment with the ideas in this section. Making synthetic assignments to a key is an easy process if you have either the CCD Module, a synthetic key assignment program ("MK", "MKX", "ASG", etc.), or the ZENROM. Refer to pages 69 and 70 of SPME for a description of the use of the synthetic assignment program which appears in barcode form on pages 174 and 175. The CCD Module's enhanced ASN function is equivalent to a built-in synthetic key assignment program.

If you use the ZENROM rather than a synthetic key assignment program or the CCD Module, creating the byte grabber is relatively easy. With the ZENROM plugged in, start by assigning the TAN function to the key where you want the byte grabber assigned. Then press ALPHA, SHIFT, ALPHA, C 1, ALPHA and execute RAMED. Press PRGM twice, and if F0,04,5B is displayed, key in the two bytes of the assignment in hex. Otherwise, press PRGM three more times (00,04,5B should be shown) and key in the two hex bytes of the assignment. Use F7, 3F for the common (decimal 247, 63) byte grabber. Now press ON to exit RAMED, and you're done. This technique will work as described above so long as the TAN assignment was not placed in a vacant key assignment register above location 0C0. If this is the case, you can still locate the TAN assignment (hex 04 5B) by pressing USER several times, but it may take a little searching.

The byte grabber is most commonly used to remove a prefix byte from a two-byte function. This frees the second byte to attach itself to the byte or bytes that follow. An example of this is the creation of a synthetic RCL instruction using STO IND 16. When the STO prefix is grabbed, the IND 16 suffix becomes a RCL prefix. If the STO IND 16 was followed by RDN (hex 75, decimal 117), the new RCL instruction will access status register M (decimal postfix 117).

Using the byte grabber this way, you can make instructions which are not normally keyable. Examples include direct operations on the data registers from 100 to 111 and status registers above L. This approach will work well as long as the instructions were just keyed in, or if the program is PACKed before pressing the key assigned with the byte grabber. The byte grabber is pressed while in PRGM (and USER) mode, and while positioned to the line before the byte (or bytes) you wish to grab. The byte or bytes are absorbed into a text line, which is deleted if not needed.

The byte grabber can also be used to release bytes. One way this can be done is to grab the Text prefix byte from a line of text. As an example, the text line "aBK1$C" has decimal byte values 246, 97, 66, 75, 49, 36, 67. After you byte grab the Text 6 (see byte 246 in the table on page 9), the six characters within the line in program memory are free to stand alone. This results in the instruction sequence ABS, *, MOD, STO 01, RCL 04, /, which could save you quite a few keystrokes if you wanted this sequence in your program. Of course, this technique is rather limited because you are confined to the characters that are keyable. But if you have the byte grabber assigned, you may prefer using keystroke sequences like ALPHA, I, ALPHA, BST, byte grab, backarrow, SST in place of the usual XEQ, ALPHA, H M S (shift) +, ALPHA.

There is another way to use the byte-releasing capability of the byte grabber. Using a synthetic key assignment program, the CCD Module, or the ZENROM, assign the 247, 142 (hex F7, 8E) byte grabber to a key. Make sure that the key previews XROM 30, 14 when pressed and held before continuing. (Note: If you have the Card Reader plugged in, this will preview as 7DSP2.)

We are going to use this key assignment to make PROMPT lines. Press GTO . . and enter PRGM mode. Press ENTER↑ twice to serve as a buffer (to protect the .END.), BST, byte grab, BST, byte grab and backarrow. SST and there is our PROMPT instruction. Repeat it again, if you like. Just byte grab, BST, byte grab, and backarrow. So, in addition to grabbing and releasing bytes, we can use the byte grabber to create an instruction corresponding to the value of the suffix (postfix) of the assignment. After you're done experimenting, delete all of the lines in the program.

More than one byte can be absorbed by the byte grabber. In a packed program, the byte grabber normally absorbs only one byte. You can absorb two bytes if you key in any one-byte function before pressing the byte grabber. This technique of inserting bytes before pressing the byte grabber can extend its range to absorb up to 5 bytes.

We can combine the technique of keying in bytes before pressing the byte grabber with byte grabber assignments whose postfixes are from rows 9 through F in the byte table. The resulting combination allows us to synthesize two- or three-byte functions. It is necessary to key in 4 bytes before using the byte grabber to properly align the prefix (supplied by the byte grabber) and the one- or two-byte postfix (which already exists within program memory). This is a very powerful synthetic technique. The fact that you can see the bytes as Alpha characters after you byte grab means you can make a visual check to insure that there are no stray nulls before backstepping and byte grabbing again to release the byte.

Clear the byte grabber key assignment and replace it with the 247, 206 (hex F7, CE) byte grabber. (ZENROM owners can simply modify the postfix byte of the existing assignment.) The key assignment should preview as XROM 31,16 when you hold it down. We will use this assignment to create several synthetic two-byte X<> instructions.

Begin with GTO . . , and enter PRGM mode. Make sure at least 8 registers are free. Then key in ENTER↑, CLX, LASTX, RDN, and a buffer of at least four ENTER↑ or 1/X instructions. Then SST twice or GTO .001.

Press EEX and CHS to put the four bytes hex 00, 11, 1B, 1C. ZENROM owners use EEX, 8, CHS, because the ZENROM removes the hex 11 byte. Keying in these four bytes will cause the byte grabber to absorb the maximum of five bytes. Now byte grab, BST, byte grab, and press backarrow twice. SST to see the new line 02 X<> O. Repeat the process by pressing EEX, CHS, byte grab, BST, byte grab, backarrow twice, and SST. You should see X<> N. Repeat once more, and we're through byte grabbing. EEX, CHS, byte grab, BST, byte grab, backarrow twice, and SST. Line 04 is now X<> M. Get rid of the buffer of instructions protecting the .END. using DELete after you SST again. Then SST twice (to line 01), backarrow the ENTER↑, key in LBL "SHR", add CLX, and PACK. You should have:

```
01 LBL "SHR"
02 CLX
03 X<> O
04 X<> N
05 X<> M
```

This little program performs an Alpha register shift to the right ("SHR") seven characters. The number of characters that you start with in the Alpha register should not exceed 21. The program clears the original value in X, and leaves the former contents of Alpha register M in its place.

To generate the three synthetic lines for "SHR" we keyed in the second bytes (postfixes) of the functions we wanted, and used a byte grabber with an assignment of 247, 206, because 206 was the first byte (prefix) of the desired X<> function. The technique demonstrated here is a variation of a procedure that you will use heavily in the next section, where you will learn about Text 0 prefix assignments. As you will see, the Text 7 prefix (byte grabber) and the Text 0 Prefix assignments are similar in many ways. Each has its strengths, and working in concert they are an extremely powerful combination.

4C. USING TEXT 0 PREFIX ASSIGNMENTS

One, two, three, or multi-byte synthetic instructions can be made quickly and easily using a key assigned with decimal values 240, XXX, where XXX is the decimal value of the first byte (or prefix) of the function you wish to create. You will need to have a CCD Module, a ZENROM, or a synthetic key assignment program such as the PPC ROM's "MK" or Tapani Tarvainen's "MKX" (barcode on pages 174 and 176 of "HP-41 Synthetic Programming Made Easy") to make use of this powerful class of synthetic key assignments.

This key assignment group is most useful when you need to make several synthetic instructions which share the same value for the first byte and have a variety of postfix byte values. Under these conditions, you would either have to make several key assignments or go through a fairly long procedure using the byte grabber. Text 0 prefix assignments will save keystrokes compared to the byte grabber because less setup is required.

Let's say for an example that you need to make a number of synthetic TONEs. Start by assigning 240, 159 to a key (159 is the TONE prefix). Check that the assigned key previews XROM 02,31. Then GTO . . , enter PRGM mode, and key in LBL "LL". We are going to create a series of TONEs with second byte (postfix) values of 19, 4, 69, 118, 69, and 86.

The first thing we have to do is to look up the instructions which correspond to these postfix bytes, and key them in after label "LL". Therefore, key in 3, LBL 03, X>Y?, LASTX, X>Y?, and LOG. You should also add a buffer of several ENTER instructions after LOG, to protect the .END. from possible alteration or loss.

You should PACK at this point because of the invisible null introduced before the "3" instruction. This is only necessary when some of the byte values are from 16 to 27 (decimal). Now SST twice to return back to label "LL". You should still be in program mode.

Now press EEX (use 9, EEX if you have a ZENROM, to offset the ZENROM's removal of the 1 byte in front of the E), then press CHS, the assigned (240, 159) key, backarrow twice, and SST. There is your TONE 19, displayed as TONE 9. The 159 prefix grabbed the 19 suffix to create this TONE. Let's examine this process more closely before creating the other five TONEs. Below is a diagram which shows the bytes in program memory, represented in hexadecimal notation. "4C 4C" below indicates the last two bytes of LBL "LL" (which is in the display when EEX is pressed). The effect of each of the six steps, which are shown on the left, can be clearly seen. The underline indicates the bytes displayed.

	4C 4C 13 04 45 76 45 56 83	
EEX	4C 4C 00 11 1B 00 00 00 00	13 04 45 76 45 56 83
CHS	4C 4C 00 11 1B 1C 00 00 00	13 04 45 76 45 56 83
240, 159	4C 4C 00 11 1B 1C F0 00 9F	13 04 45 76 45 56 83
backarrow	4C 4C 00 11 1B 1C 00 00 9F	13 04 45 76 45 56 83

```
backarrow    4C 4C  00 00 00 00 00 00 9F  13 04 45 76 45 56 83
SST          4C 4C  00 00 00 00 00 00 9F  13 04 45 76 45 56 83
```

Whenever an instruction is inserted in a packed program, the HP-41 quickly shifts all of program memory below the insertion down one register to make room. This will be repeated, if necessary, when more than seven bytes are needed. Any four bytes could be used ahead of the (240, 159) assignment to align the hex 13 (decimal 19) byte as a suffix for the 159 byte. The bytes in the above diagram are shown in groups of seven to help you visualize how this works. To complete the other five TONEs, simply repeat those six steps listed at the left side of the diagram five more times. Don't worry if you lose track, the buffer of ENTER instructions will alert you if you go too far, because you will see TONE IND 03 (the result of the 159 prefix grabbing the ENTER suffix) when you SST. When you're finished, delete all of the ENTER (and TONE IND 03) instructions. Now PACK the program, return to run mode, and run the program. You may recognize this series of TONEs. Charge!

There is a generalized procedure which can be applied to all 256 possible postfix values in a Text 0 prefix key assignment. The only exceptions in this generalized procedure are that the three-byte functions 192 to 205 and 208 to 239 (GLOBAL, GTO, and XEQ) are treated as two-byte functions because they only need to absorb one, rather than two, of the bytes that follow. The second (middle) byte of the resulting three-byte instruction will then be a null.

The general procedure, after making the needed key assignment, is as follows:

1. Position yourself in program memory, in PRGM mode, where the instruction is to be created.
2. Key in the last byte of the function in the form of an instruction.
3. BST.
4. Press EEX, CHS (ZENROM users press 8, EEX, CHS) for any

instruction except three-byte functions. For them, press only EEX. (ZENROM users press 8, EEX).

5. Press the key with the Text 0 Prefix assignment.
6. Backarrow twice.
7. SST and inspect the results. If it is an END, you must PACK or GTO . . immediately.

The two lines deleted by step 6 are Text 0 and 1 E-. There is also an invisible null between the two deleted instructions. The Text 0 instruction is a useful NOP (No OPeration) function created as a consequence of the Text 0 Prefix assignment. This is the same NOP created by the ZENROM's NOP function. EEX, CHS (or 8, EEX, CHS with a ZENROM) is a convenient, two (or three) keystroke way of inserting four bytes. The number of bytes absorbed, if any, depends upon the number of bytes keyed in at step 4 and the number of postfix bytes normally required by the instruction prefix.

Let's go through another example, this time creating a packed, compiled END (192, 0, 9). This is useful in avoiding decompiling, the loss of the compiled jump distances contained in the GTO and XEQ instructions of a program with no END. More details on this will be given in Section 4G.

Assign 240, 192 to a key, and check that the preview is XROM 03, 00. Then go to a convenient place in memory to create a new END, such as the "LL" program. If you wish to test that this procedure preserves compiled information, add a LBL 01 after LBL "LL", and GTO 01 after the series of TONEs. PACK the program, execute it once, and position yourself to the last program line (step 1 of the above procedure).

Since LBL 08 corresponds to the decimal byte value of 9 that we want, key in that instruction (step 2). Then BST (step 3). Press EEX (step 4). Now press the key assigned with 240, 192 (step 5). Backarrow twice to clean up the two excess lines (step 6), SST, and you should see an END. PACK or GTO . . (step 7) and return to the "LL" program.

If you added the LBL 01 and GTO 01, you can confirm that the jump distance information was not lost upon PACKing through the use of another feature of the Text 0 prefix assignments. Position yourself to the GTO 01, which should be line 07. In run mode, press any key assigned with a Text 0 Prefix. The Alpha register should now contain six bytes, the first two of which are the GTO 01 instruction. The next three are the END we created, followed by a null. Visual inspection of Alpha is enough to confirm that the compiled information was not lost, as the second byte would be a null (overbar character), not a starburst, had the jump distance been cleared. If you like, you can decode these bytes using the extended function ATOX or the PPC ROM's "CD" routine. More information on the nature and use of this "byte jumping" feature of all Text prefix key assignments is given in Section 4I.

In the example above, we could have used a two-byte instruction such as ISG 09 or TONE 9 in the place of LBL 08. The only other change necessary would have been to press EEX and CHS at step 4, to cause two bytes rather than one to be grabbed. The fact that the middle byte of the END would not be a null in that case makes no difference. PACKing changes the middle byte to the proper value. More information on this subject is in Section 4G.

If you should forget which Text 0 Prefix assignment was assigned to a particular key, there is an easy solution. Take the first value of the two numbers in the XROM preview, multiply it by 64, and add the second value to that quantity. For example, if the XROM preview shows XROM 02, 24, you would multiply 2 by 64 and add 25. The result of 152 tells you that the assignment is 240, 152. This will work for any Text 0 Prefix assignment.

There are many uses for the Text 0 Prefix assignments. The 240, 245 assignment works much like the byte grabber key assignment (247, xxx). When pressed in program mode, one byte is absorbed by a Text 5 instruction. This assignment inserts a Text 0, a null, and a Text 5 byte when pressed in the program mode. Normally these three bytes are followed by 4 nulls to complete the register that is opened for the inserted instruction. The text line includes these 4 nulls and a single absorbed byte.

Both this byte grabber and the Text 7 prefix byte grabber can be made to absorb more than one byte by inserting instructions prior to pressing them. Up to four bytes of instructions can be keyed in before pressing either byte grabber, resulting in a maximum of five bytes being absorbed. Because of the Text 0 function created by the Text 0 Prefix byte grabber (240, 245), a little more cleanup is necessary than with the usual Text 7 prefix byte grabber.

If you decide to create one-byte instructions using the Text 0 prefix assignments, you may omit step 4 in the general procedure. It really isn't necessary to press EEX and CHS. However, it is possible to combine bytes 16 to 28 in rather unusual ways, with unexpected results. If you want to experiment with adding one of these byte values as a prefix on an existing numeric entry instruction, use EEX, CHS, 7 at step 4. The 7 is necessary if a null exists before the numeric entry line. This will be the case while building an instruction using the Text 0 prefix assignments. However, if the program containing the numeric entry line has been PACKed, and the HP-41 removed the null because it wasn't needed to separate two adjacent numeric entry instructions, then the 7 is unnecessary. If you aren't sure, try it with just EEX, CHS at step 4 in the procedure. If that doesn't work, use EEX, CHS, 7.

One way that the Text 0 Prefix assignments are an improvement over the byte grabber lies in the fact that bytes 228 to 239 can be synthesized. This is impossible with just the byte grabber. The reason for this is that the usual procedure for bytes above 143 is to byte grab the first byte from an instruction such as RCL IND 16. This releases the second byte, which will then link up with whatever follows. The problem is that without a ZENROM, bytes 228 through 239 (and 245 to 255) are not keyable as postfixes, since the HP-41 does not allow the entry of three digit numbers as postfixes (no STO IND 100).

The 240, 228 assignment could be used to create a text line such as 247, 127, 40, 228, 120, 144, 104, 41. However, this is a very complex undertaking (using the byte-loading program "LB" or the ZENROM's RAMED would be far easier). Not only is a byte grabber needed (or the 240, 247 assignment), the task is complicated by the need to PACK at least once. Because of the nature of PACKing, it is difficult to avoid the alteration of the sequence 228, 120, 144 (XEQ 16) after it is synthesized. The best approach is to also create a synthetic END (192, 0, 9) as the last program line. Do this after setting up all of the bytes except 228, and then PACK (only once!) after inserting the 228 byte and doing the cleanup. This tricky procedure is a good test of your ability to use F0 prefix assignments.

You may want to postpone this exercise until the end of this chapter. There is a lot more information to be learned, and you should be proficient before you try. When you can make this text line with only Text 0 prefix assignments 192, 228, and 247, you have truly mastered the use of the Text 0 prefix assignment.

4D. THE Q LOADERS

A Q-loader is one of a class of synthetic key assignments used to create program instructions which contain up to seven Alpha characters. This includes all of the various functions containing Alpha characters, from Alpha text lines and Alpha labels to Alpha GTO, XEQ, and W instructions. (The W instruction uses the synthetic prefix 1F.)

When a Q loader key assignment is pressed, the contents of status register Q are used, in byte-reversed order, to form the Alpha portion of the instruction. Because of the fact that the characters are used in reverse order, trailing nulls are suppressed. (This contrasts with the display of characters in the Alpha register, where leading nulls are suppressed.) This, and the seven character limitation, are the only significant constraints that the Q loaders have.

An important advantage that the Q loaders have over the byte grabber is that there is no difficulty making Alpha functions containing bytes from the last four rows (C to F) of the byte table. Bytes from 192 to 205 and 209 to 255 can be quite difficult to handle using the byte grabber: the Globals, GTOs, and XEQs swallow the two bytes following them, and often change their value upon PACKing. Furthermore, bytes 228 through 239 are unkeyable as postfixes. Bytes above 250 or so are tricky to manipulate too. A Text 15 prefix can swallow your .END. and lead to MEMORY LOST after PACKing, before you realize what's happening. The Q loader (or a byte loading program) is a good way to get around these difficulties.

Generally, a Q loader is used by building the needed bytes in Alpha in reverse order, moving these bytes to status register Q, entering PRGM mode, and pressing the key assigned with the Q loader that corresponds to the type of instruction desired.

The synthetic key assignments RCL M and STO Q are handy to move the accumulated bytes from Alpha register M to register Q. To learn how to use the Q loaders to make the various Alpha instructions, use the CCD Module or a key assignment program ("MK", "MKX", etc.) to make the following key assignments:

INPUTS	PREVIEW	FUNCTION
144, 117, 15	XROM 01, 53	RCL M
145, 121, 14	XROM 05, 57	STO Q
27, 4, -11	XROM 44, 04	E, "Q load"
205, 4, -12	XROM 52, 04	LBL "Q load"

4, 29, -13	@AHHH	GTO "Q load"
4, 30, -14	@NQ	XEQ "Q load"
4, 31, -15	2 _ _	W "Q load"
	(fill in the prompt with any number)	

Now use the normal ASN function to assign the Extended Function (or Data Logger function) XTOA to key 11 (Σ+). If you do not have a Data Logger or Extended Functions module, or an HP-41CX, you can substitute the PPC ROM's "DC" routine, or the program on page 77 of "HP-41 Synthetic Programming Made Easy". Be sure to assign one of these to key 11 (Σ+).

Now GTO . . , enter program mode, and key in LBL "QL". Return to run mode and clear Alpha. Use the assigned key (11) to build the following bytes in the Alpha register:

41	XTOA (Σ+)
70	XTOA (Σ+)
12	XTOA (Σ+)
32	XTOA (Σ+)
49	XTOA (Σ+)
52	XTOA (Σ+)
40	XTOA (Σ+)

Inspection of Alpha should show ")Fμ 14(". If you used "DC", you need to return to the program "QL" by using either GTO or CAT 1. After you are properly positioned, press RCL M (15) and STO Q (14). Then enter program mode and press shift A (-11). The first line created by this key assignment is line 02 E. The lone E, which is a slightly faster way of producing 1 than using the instruction "1", is a result of the prefix of the assignment used (see decimal value 27 in the byte table). Any value in row 1 may be used in place of 27 when making a Q-loader assignment, with a resulting instruction corresponding to that byte.

Now SST to see the second result of this Q loader. Line 03 "(41μ F)" is composed of a Text 7 byte followed by the reverse of the bytes that were stored in register Q. This particular text example was selected to demonstrate some of the non-keyable characters that can be used for text, labels, and the other Alpha instructions.

With line 03 still in the display, press shift A (-11) again. Once more we have a solitary E line. SST to the next line. This is a Text 0 instruction, also known as F0 or decimal byte value 240. It is similar to the <u>N</u>o <u>OP</u>eration instruction available on some machines, in that it does nothing (except re-enable the stack lift if it was disabled). However, such a place-holder function can be useful after an ISG or DSE instruction where the decision capability is not needed.

Whenever a Q loader is used, register Q is cleared. So if you used the wrong Q loader, or you want to create another instruction with the same bytes, you need to load Q again. If the stack hasn't been disturbed, just STO Q. If it has, check Alpha to see that the bytes are still there and use RCL M, STO Q.

There are two other methods for using the Q loaders. They have two advantages in that it is unnecessary to reverse the order of the characters in Alpha, and no synthetic assignments are needed to load Q. The disadvantage is slight; you can only use up to six characters in the first case, and nulls can't be used.

For the first method, begin by assembling the characters you want in Alpha. Use up to 6 characters, in normal (non-reversed) order. In ALPHA mode, ASTO X to transfer these characters to X. As Alpha data, the first of the seven bytes in the register is the Alpha identifier, leaving six bytes for the text. Return to run mode and key in GTO IND X. This should show NONEXISTENT, but it has the vital side effect of putting the label name in Q. If there happens to be a matching label, the GTO IND X will have caused the calculator to branch to the label. In that case, you must return to the place you want to put the instruction. You should only use CAT 1, GTO .nnn (GTO line number), SST or BST to do this. Do not use GTO Alpha, any functions with Alpha inputs, or any other function that might alter Q. Then press one of the Q loaders in program mode.

The second alternative method to use the Q-loader is simply to key in LBL ALPHA, spell out the characters needed, press ALPHA again, and continue as you would with any Q loader. You can key in the LBL in run mode, then switch to PRGM mode to press the Q loader. Because of the requirement that all of the characters are keyable within a normal Alpha label, the usefulness of this technique is somewhat limited.

Following these methods, you should be able to use the Q loaders to produce non-keyable text, labels, GTOs, XEQs, or W instructions. Be cautioned about the W function, though. The effect it has is dependent upon what is plugged into Port 2, and to a lesser extent Port 1. Status registers T, X, N, or d may be cleared, altered, or contain garbage. The calculator may lock up for a couple of seconds, or until you remove the batteries for a second or two. Still, you should feel free to experiment, since none of the instructions or techniques described here should get you into serious trouble.

4E. FIX/ENG DISPLAY FORMAT

The FIX/ENG display format can be obtained by either normal or synthetic means. The advantage of this display setting is that the display appears the same as in the normal FIX mode until the display overflows or underflows (that is, until an exponent is required to display the number). Then instead of over- or underflowing to SCIentific notation, the calculator uses ENGineering notation, with exponents that are multiples of three.

The normal ENGineering mode can be very annoying as the exponent is always used, making familiar numbers more difficult to recognize. Yet when an exponent is necessary, displaying it as a multiple of three works very well with the metric system. The FIX mode offers numbers which are readily recognizable. The FIX/ENG format offers the best of both modes. Not only is this display mode handy, but it takes no more bytes to set in a program than the normal FIX or ENG mode.

All that is required to achieve the FIX/ENG display setting is to simultaneously set both flags 40 and 41. This can be done several ways. First, and without synthetic programming, you can use the Extended functions RCLFLAG and STOFLAG. RCLFLAG saves the flag 41 status as set, FIX mode is selected (setting flag 40), then STOFLAG selectively re-sets flag 41. See "HP-41 Extended Functions Made Easy", pages 69-70. The second way is to use the CCD Module's F/E function. You can also use the PPC ROM's "IF" routine to set flag 41 after a normal FIX instruction.

FIX/ENG can also be set using a synthetic key assignment, or with a synthetic FIX instruction (both of which are directly keyable with a ZENROM). The prefix, or first byte, of the assignment or instruction is decimal 156. This is the same as the normal FIX instruction. The postfix, or second byte, is taken from row 4 of the byte table. Postfix 64 corresponds to FIX/ENG 0, but it displays as FIX 4. Only the second decimal digit of the postfix is displayed. As an instruction or key assignment, FIX/ENG 0 is 156, 64. For FIX/ENG n, use 156, 64+n. This will display as FIX m, where m = (n+4) MOD 10.

The workings of this synthetic instruction deserve an explanation. Randy Cooper, in the January 1982 PPC Calculator Journal, analyzed what happens when the HP-41 executes a display setting instruction. The number of digits (flags 36-39) is set to match the last nybble of the postfix byte. Flags 40 and 41 (which set the mode type) are set to match the correct mode, then flags 40-43 are ORed with the first nybble of the postfix.

For FIX/ENG n, the postfix (in hexadecimal) is 4n. The second nybble sets n digits (flags 36-39), while the first nybble sets flag 41. The

FIX prefix (decimal 156) sets flag 40.

Several methods can be used to create the synthetic FIX instruction needed to change your display to FIX/ENG format. If you have a CCD Module, you can use the enhanced XEQ function to key in the two bytes directly. Or you can use a byte loading program like "LB" or "LBX" (from "HP-41 Synthetic Programming Made Easy" or the PPC ROM) to create the needed synthetic instruction. You can use a Text 0 prefix assignment (240, 156), or you can use the byte grabber key assignment (247, xxx). An example using the byte grabber is given below.

Position yourself to the place in memory where you want the instruction. In this case, we will create a FIX 70 (FIX/ENG 6) instruction. If there are no other program lines in memory there, key in ENTER↑ (in PRGM mode) for a filler. Now key in STO IND 28, X<Y?, BST, BST, and press the key assigned with the byte grabber. Backarrow once and SST to your new synthetic line. It displays as FIX 0, though this is really FIX 70. Single-step this line after returning to run mode (press PRGM, SST). The description of the effects of FIX 70 follow the next paragraph, which describes how you can use a synthetic key assignment to set FIX/ENG 6 from the keyboard.

Using a synthetic key assignment program like "MK", assign a key with prefix and postfix values 156 and 70, respectively. Holding that key should preview XROM 49,06 if the assignment was made properly. If it shows other numbers, clear the assignment using ASN ALPHA ALPHA, and try again. Once the assignment is made, you can use it in run mode to change the display setting to FIX/ENG 6 by pressing the key, or you can use it in PRGM mode to create the synthetic FIX 70 instruction.

To demonstrate how the FIX/ENG 6 display format behaves, begin by filling the stack with ten. Key in 10, then press ENTER↑ three times. Now press * repeatedly until the display overflows to engineering notation. You should see 10.00000 09. Further pressing of * will show how the decimal point progresses from 1. to 10. to 100., and then reverts back to 1, when the exponent increases by three. This demonstrates the overflow from FIX to ENG format.

To demonstrate the underflow to ENGineering notation, fill the stack with one-tenth by pressing .1, then ENTER↑ three times. Now press * repeatedly. After the sixth multiplication, the calculator underflows to ENG format. Subsequent pressings of * show the number decreasing from 100. to 10. to 1., and back to 100. at the time that the exponent decreases (becomes a larger negative number) by three.

If you have the PPC ROM or the ZENROM, you can use one of the programs listed below to set FIX/ENG mode. Of course, ZENROM owners can simply press SHIFT, FIX, and fill in the two digit prompt with a decimal value from 64 to 73. But the program below is useful because it alters only flags 40 and 41, and for demonstration purposes.

```
FOR ZENROM:                  FOR PPC ROM:

01 LBL "FE"                  01 LBL "FE"
02 40                        02 40
03 FC? IND X                 03 ENTER
04 TOGF                      04 FC? IND X
05 ISG X                     05 XROM "IF"
06 ""  (F0 NOP)              06 ISG X
07 FC? IND X                 07 ""  (F0 NOP)
08 TOGF                      08 FC? IND X
09 CLX                       09 XROM "IF"
10 RDN                       10 END
11 END
      24 bytes                     23 bytes
```

In the ZENROM program, lines 02 to 04 set flag 40 if it is clear. Lines 05 and 06 increase X from 40 to 41. Lines 07 and 08 will set flag 41 when it is clear. FIX/ENG display mode is now set. Lines 09 and 10 clear X and push it into T, leaving the original X, Y, and Z where they were.

In the PPC ROM version, lines 02 and 03 put 40 in X and Y. Lines 04 and 05 set flag 40 if it isn't set already. "IF" will drop the stack, which is the reason for line 03. Whether "IF" was executed or not, 40 will still be in X, where line 06 and 07 change it to 41. Lines 08 and 09 will invert the status of flag 41 when it is clear. The contents of the stack are somewhat variable, depending on the status of the two flags. However, Alpha will be clear and Z will contain a copy of the original contents of X in all cases at the termination of the program.

4F. CATALOG 1 RECOVERY

Sometimes in the course of synthetic programming, the continuity of Catalog 1 is disrupted. The first hint that something is amiss is often a subtle change in the usual "feel" of the HP-41. You may notice the calculator is a bit slow to leave or enter PRGM mode, or some similar deviation. (If the calculator locks up, see Appendix A for crash recovery techniques.)

Pressing CAT 1 will confirm whether the linked list of global instructions has been interrupted. Often, the catalog will list a single instruction, which isn't even a global function. But don't despair. Chances are good that all of your programs are intact, even though you cannot yet access them normally.

There are several things you should avoid doing before the continuity of Catalog is re-established. These include turning the HP-41 OFF and ON, executing GTO . . (or any operation which causes packing), or executing COPY, GETP, GETSUB, or PCLPS. Any of these can lead to MEMORY LOST.

The continuity of Catalog 1 may be lost if either the .END. itself is lost or modified, or if status register c no longer contains the correct pointer to the .END. . (The last three nybbles of register c contain the absolute address of the .END., which occupies bytes 2, 1, and 0 of that register.) If the .END. is missing, or nybbles 2, 3, and 4 of the .END. do not point correctly to the next global function upward in memory, access to Catalog 1 is lost.

In order to recover from this condition, we need to make status register c point to a legitimate .END. . Section 4G gives more information on the different END types. But all you really need to know right now is that any .END.s found above our new .END. will be converted to ENDs during packing. The best approach to take is to create a new .END. just above the key assignments, alarms, and any I/O buffers. Then you must modify register c to match this address, and PACK to recompute the global linkages.

Here is a Catalog 1 recovery procedure that you can use with the CCD Module:

1. Key in 12 and execute WSIZE. Then key in the number 204 (equals hex 0CC) for use as a starting address.
2. Execute PEEKR to recall register 204 to X without normalization.
3. Press shift ÷ (divide) to execute the X=0? function. If the result is YES, you have found a suitable empty register. Otherwise use CLX, 1, + to increase the address by land repeat steps 2 and 3 until you do find an empty register.
4. Now turn off USER mode and press ALPHA shift ENTER 192 shift ENTER 000 shift ENTER 045. This creates the 3-byte alpha string C0 00 2D.
5. Press X<> M to bring the bytes for the new .END. to X. Y should still contain the address of the empty register that you found.
6. Execute POKER to store the new .END. .
7. Press 13.1 and execute PEEKB to see byte 1 of register c (absolute address 13). If this number is not already a multiple of 16, press 16, /, INT, 16, *, then execute POKEB. This removes

the high bit of the .END. pointer, and this assumes that you found an empty register at an address below 256. Otherwise add 1 and POKEB again.

8. Press RDN three times to get the register address back in X, and press 256, MOD if it exceeds 256. Then press 13, X<>Y, and execute POKEB to store this number as byte 0 of register c.
9. GTO . . or PACK. This will probably take about ten or fifteen seconds to finish.

Now here is a Catalog 1 recovery procedure which makes use of the ZENROM's RAMED function:

1. Press ALPHA, space, SHIFT, ALPHA, C C, ALPHA. This places 20CC hex (byte 2 of address 0CC) in Alpha for use as a starting address.
2. Execute RAMED.
3. If the three visible bytes at register 0CC are all 00, key in C0 00 2D. Otherwise, press SHIFT, USER repeatedly until unused registers are found. Then key in C0 00 2D. The new .END. <u>must</u> start at byte 2 of the register. In particular, by pressing PRGM and/or USER, you should be able to align the .END. in the display and see 1:xxx C0,yy,2D, indicating that the yy is at byte position 1 in register xxx. Make a note of the address xxx of the new .END. you created.
4. Press ON to exit RAMED.
5. Press ALPHA, SHIFT, ALPHA, 0 D, ALPHA. This will select byte 0 of register c, address 00D.
6. Execute RAMED and press USER once.
7. We don't want to alter the first 3 nybbles visible, only the last three. Therefore, key in the same hex digit which appears in the left side of the middle byte. Then key in the three nybbles xxx of the address of our new .END. from step 3 (such as 0CC).
8. Press ON to exit RAMED.
9. GTO . . or PACK.

Catalog 1 should now be as it was before you lost access to it. If you believe that the first program in Catalog 1 is missing, perhaps because the data/program partition was moved downward, there is an easy way to recover that as well. You should know that normal register operations will normalize a program stored within the data registers, effectively destroying all or part of the program. So avoid using X<>, STO, RCL, VIEW, etc. Instead, you should check the position of the program/data partition (by looking at the contents of register c or using the PPC ROM program C?) and move it to the top of memory (normally 200 hex = 512 decimal) if you suspect alteration.

To check for program bytes within the data registers, use NRCLX followed by DECODE. Start with zero in X, and check more than one register. Look especially for Cx bytes, that are LBL and END prefixes and which are not likely to appear in data. Once you are satisfied that program bytes exist within the data registers, you can edit register c to move the program/data partition above the program bytes.

To accomplish this alteration of the program/data partition, begin by repeating steps 5 and 6 of the previous Catalog 1 recovery procedure. Press USER again. The left-hand byte should be hex 69. Altering the hex 169 "cold start constant" will cause MEMORY LOST when you exit RAMED, so be cautious. Make a mental note of the rightmost nybble shown in the display; it will be either 0 or 1. (This is the first digit of the 3-nybble .END. pointer.) Then key in the three-digit hexadecimal address listed below (according to the number of 64-register memory modules installed -- use 4 for a CX, CV or 41C with Quad):

Modules	Address
4	2 0 0
3	1 C 0
2	1 8 0
1	1 4 0
0	1 0 0

Complete the entry of byte 1 in register c by keying in the same value which existed before (0 or 1). Press ON to exit RAMED and check CATalog 1 immediately. PACKing may be needed to recompute global linkages, though this is not usually required. A side effect of this procedure is that any data will beome program steps before the first legitimate line of the first program. All data will be lost. In addition, you will need to reSIZE, since this procedure sets the SIZE to zero.

The Catalog 1 recovery procedures using the CCD Module or ZENROM to construct a new .END. and its pointer, then PACK, have a very high rate of success. However, if you are more concerned about the chance of losing Extended Memory than the loss of main memory, you can use the ZENROM's CLMM function. To quote the ZENROM manual: "CLMM restores the HP-41 Main Memory to Master Clear state by storing nulls into every register. In addition, all status registers and flags are restored to default states; all key assignments, timer alarms, and input/output buffers are eliminated; and the stack, LASTX and ALPHA are cleared. The size of the program memory will be set to 46 registers on the HP-41C and HP-41CV or 219 registers on an HP-41CX." The contents of Extended Memory will be unaffected. Using CLMM will spare Extended Memory in a situation when you are sure MEMORY LOST is inevitable. When you have to interrupt a protracted PACKING operation by pulling out the batteries, you have one such case.

If you understand the principles behind the CCD Module and ZENROM procedures just described, you can substitute routines in the PPC ROM. The method developed by Clifford Stern constructs an .END. at the address pointed to by register c:

```
    ALPHA  C 0 0 0 2 D  ALPHA
    XEQ "HN"
    XEQ "E?"  (Stop here if the result is not from 192 to 511)
    XEQ "SX"
    GTO . . or PACK immediately
```

This sequence creates a synthetic .END. at the location specified by the last three nybbles of status register c. In some cases, this procedure will not work. If the result of executing "E?" is outside the range of 192 to 511, you must not execute "SX". Instead, you should use one of the alternate procedures described later in this section.

If the number of free registers available (in run mode, press RTN and PRGM) does not match the number returned by the PPC ROM function XEQ "F?" within one-half, there are only two possible causes. Either there are time module alarms or an I/O buffer present (in which case the execution of "F?" has just made these unusable), or some of the free registers are not empty as they should be.

In the case of non-empty registers in the free register block, or if the result of "E?" is not between 192 and 511 inclusive, an alternate procedure can be used. Again, the PPC ROM is required:

Make sure the SIZE is at least 001.

XEQ "A?"

Add 193 to this number and take the INTeger part.

You now need to convert this decimal result to a three-digit hexadecimal number. The conversion is most easily performed in two steps. If the decimal number is greater than or equal to 256, subtract 256 and write down a 1 as the first digit of the hex equivalent. Otherwise, write down a 0 as the first digit. Now take the remainder (0 to 255) and use the QRC (byte table) to convert it to hex. This gives the last two digits of the hex equivalent (the row number is the first digit; the column number is the second digit). Now continue:

GTO "C?" and SST. Press SST again, holding it long enough to see 47 RCL c.

XEQ "NH"

Press append, then backarrow three times. This deletes the last three nybbles from the hexadecimal representation of status register c that we have in Alpha. Replace these three digits with the three hex digits that you calculated before. Just key them into the Alpha register, since you are already in the append mode. Press ALPHA twice and make sure there are 14 hex digits, with 1 6 9 in positions 6, 7, and 8. Press ALPHA again to exit ALPHA mode.

XEQ "HN" (X now contains the modified c register contents.)

GTO "PA" and BST twice.

Press SST, holding it long enough to see 150 X<> c.

Decrease the SIZE. It doesn't matter how much, as long as you make it smaller.

GTO . . (Do <u>not</u> XEQ "PACK")

The missing registers should reappear. Make a note of the number of free registers you have (hold SST and see .END. REG nn), and compare this to the number returned by XEQ "F?".

4G. MAKING PRIVATE PROGRAMS WITHOUT A CARD READER

Private status is established in a program by information contained with the END (or .END.). Nonsynthetic methods of making a program PRIVATE require the Card Reader or an HP-IL mass storage device such as the Digital Cassette Drive. The END recorded by WPRV and WRTPV on the magnetic medium is altered to private status. Thus, the program needs to be read back in to the calculator to establish a PRIVATE program in main memory.

There are advantages to making valuable programs PRIVATE. When you use synthetic programming techniques, there are times you can find yourself almost anywhere in program memory. Before realizing it, you can alter or delete part of your program, decompiling jump distances and generally messing things up. Private status provides protection against most cases of inadvertent alteration, and also a degree of security for your programs. (Methods of removing private status have never been published, but they are widely known by synthetic programmers.)

Using a mass storage device to make a PRIVATE copy of the program is time consuming. And because the program has to be read in again, it ends up at the bottom of Catalog 1. Several synthetic methods are possible to overcome these disadvantages, altering the END type to PRIVATE without moving the program. Not only are these methods fast, but they can also be used to avoid decompiling jump distances stored within GTO and XEQ instructions in the program. You can also use these techniques to create some rather strange .END.s and ENDs.

Before we continue, you need to know the placement of ENDs and the .END. in program memory, and their structure. An END may exist within any block of three bytes in program memory. However, the one and only permanent .END., which defines the border between the last program and the free registers, <u>must</u> occupy bytes 2, 1, and 0 (the rightmost three bytes) of the register it resides in. To better understand the structure of ENDs, it is necessary to break up the three bytes they are composed of, and consider them as six nybbles, or hexadecimal digits. The seven ENDs and .END.s listed below in Table 4.1 are a complete inventory of the types you will normally encounter.

TYPE OF END	NYBBLES (hex) 1 2 3 4 5 6	Decimal "LB" inputs
END, packed	C a b c 0 9	192, 0, 9
END, unpacked	C a b c 0 D	192, 0, 13
.END., right after GTO..	C a b c 2 0	192, 0, 32
.END., packed	C a b c 2 9	192, 0, 41
.END., unpacked	C a b c 2 D	192, 0, 45
END, PRIVATE, packed,	C a b c 4 9	192, 0, 73
END, PRIVATE, unpacked	C a b c 4 D	192, 0, 77

Table 4.1 -- Normal END types

Refer to the table while reading this. The first nybble of an END is, by definition, C. The second nybble (a) may be any value from 0 to D. This nybble, along with the next two nybbles (b and c), indicate the distance to the next global instruction (END or Alpha label) upwards in memory. Any number is acceptable, as PACKing will adjust these nybbles to their proper values. Refer to the end of this section for more on these three nybbles.

The third byte, composed of nybbles five and six, contains the most useful and interesting information in the END. Nybble five defines the type of END. For an ordinary END, the hex value is 0. The permanent .END. has a value of 2, and a PRIVATE END is 4. Other values are possible, but they are not normally used by the system. More on this later. Nybble six tells the HP-41 if the program needs to be packed, and whether to erase the compiled jump distance information when leaving the program mode.

Nybble six is best understood if you convert the hexadecimal digit to binary form. This can be done by using the byte table. The binary equivalents are along the bottom (hex at the top) of the table.

Nybble 6 (hex)	Binary	Type
9	1 0 0 1	packed, compiled
D	1 1 0 1	unpacked, compiled
F	1 1 1 1	unpacked, needs decompiling

The first and last bits are not actively used, and are generally ignored by the HP-41. You can set them to zero if it is more convenient. The second bit, when set to 1, indicates that the program needs to be packed. If the third bit is set to 1, the calculator will erase all of the compiled jump distance information (within GTO and XEQ instructions) in the program when leaving the program mode, packing, or turning off and on. These two bits are reset to zero when their respective functions are performed (packing and decompiling).

When an instruction is deleted from a program (or inserted where there is no room, so that all Catalog 1 programs are shifted down one register to make room), nybble six is changed to F. This indicates that both packing and decompiling are needed. If you don't have a ZENROM, you'll probably never see this nybble value in an END. When you exit PRGM mode, the HP-41 will decompile the branching instructions within the program and change nybble six from F to D.

It should be obvious why a nybble six value of 9 is the best to use when modifying an existing END where you do not want the HP-41 to rePACK or decompile your program. You must use 9 (or one of its equivalents: 0, 1, or 8) when creating a synthetic END to avoid decompiling. PACKing is still necessary to get the calculator to recognize the new END, but then, with the new END incorporated into Catalog 1, the decompiling procedure that follows packing is avoided if the nybble six value indicates that decompiling is not necessary.

Altering ENDs with the ZENROM is very easy if you use the RAMED function. It is much less risky to modify an existing END with RAMED than it is to create a new END with "LB", the Byte Grabber, Text 0 prefix, or other synthetic key assignments. (These other methods are more likely to cause disruption of Catalog 1 or MEMORY LOST.) With RAMED, the first two bytes, which properly link the END in the global chain of Catalog 1, can be left undisturbed, making changes to only the last byte. You can use the Byte Grabber to make a PRIVATE END or avoid decompiling without much worry. Just confine yourself to nybble five values 0 and 4, with nybble six values 9 or D. In any case, be sure to save vital data and programs outside the HP-41.

If you have a CCD Module, you can use the PHD (program head) function to find the absolute address of the first byte in the program following the END. Add one byte to get the address of the third byte of the END. Then you can use POKEB to modify that byte.

Whether you have a ZENROM, CCD Module, or just the Byte Grabber key assignment, work though the following example of creating a PRIVATE program. Begin by pressing GTO . . , PRGM, LBL "PRV" and GTO . . again.

If you are using the CCD Module, switch out of PRGM mode, execute PHD, and execute A+ to add one byte to this address. Then key in 73 (equals hex 49) and execute POKEB to replace the last byte of the END in the LBL "PRV" program. GTO "PRV" and switch into PRGM mode. You should see PRIVATE.

If you are using the ZENROM, return to the END following LBL "PRV" using CAT 1. Then execute RAMED and press PRGM twice. The right side of the display should show 01,09,00. Press 4 9 and the ON key. You should see PRIVATE immediately. It's that simple with the ZENROM!

If you are using the Byte Grabber, first return to the END following LBL "PRV" using CAT 1. Backarrow the END. Now key in STO IND 66, RCL 73, BST, BST and Byte Grab. Backarrow once and SST. You should see END. As soon as you PACK, you will see PRIVATE. The IND 66 postfix (hex C2) became the prefix for the END, and the decimal 73 postfix took its place as byte 3 of the END. PACKing adjusted the second nybble of byte 1 and all of byte 2 to point to the next higher global function in Catalog 1, incorporating the synthetically created END into Catalog 1.

Note that because we used RCL 73 (hex postfix 49) instead of RCL 77 (hex postfix 4D), any compiled jump information within the program would have been preserved. This procedure can be used to make any program PRIVATE while also avoiding decompiling.

To get rid of this PRIVATE program, use either CLP "PRV" or the Extended Function PCLPS. If you do not know how to clear Private status, you shouldn't ever make a program PRIVATE unless you have a backup (non-PRIVATE) copy as well.

You probably noticed that HP used only three of the sixteen possible values for nybble five (0, 2, and 4) for ENDs. In addition, a nybble five value of F is reserved for Alpha labels. These four values comprise all of the global functions normally found on the HP-41. But what of the twelve remaining values?

The other END types do indeed work, and can be useful if you are cautious. We recommend that only ZENROM and CCD Module users experiment with these non-standard ENDs. You should also note that there is no guarantee that every HP-41 ever made will act exactly the same in reaction to conditions that the designers of the operating system did not plan for. If your HP-41 does not perform in every respect as you think it should with ENDs other than type 0, 2, or 4, please do not contact Hewlett-Packard about these features. They are NOt MAnufacturer Supported (NOMAS).

Some explanation is needed before proceeding further. Just as nybble six is used by the HP-41 on a bit level, so is nybble five. The first bit of nybble five is not generally used, and most often set to zero. Changing this bit to a one usually has no effect, except that if nybble five is an F, the instruction becomes a Catalog 1 Alpha LBL.

The second bit of nybble five determines whether or not the program is PRIVATE. When this bit is a one, the program area immediately above this END or .END. (up to the next END) is PRIVATE. The third bit determines whether the END displays as an END or an .END. in Catalog 1 and in program mode. Bit four (which is normally 0) somehow alters the way that the END instruction functions when set to 1. See Table 4.2 below.

NYBBLE 5 BITS 3 4	FUNCTIONS AS	DISPLAYS AS	END TYPE
0 0	END	END	END
0 1	.END.	END	Pseudo-END
1 0	.END.	.END.	.END.
1 1	END or .END.	.END.	Pseudo-.END.

Table 4.2 -- More END Types

The exact reason for the odd behavior of bits three and four is buried within the HP-41's microcode (machine-level instructions), and beyond the scope of this book. But the external effects of these four bits in nybble five will be thoroughly described. To expand this table of four END types, we need to add the other two bits of nybble five. Since bit two determines PRIVATE status and bit one has no effect, this will double the number of distinctly different END types. These are listed below.

Type	Nybble 5	Decimal value	END	.END.
END	0 or 8	9 or 137	X	X
PRIVATE END	4 or C	73 or 201	X	X
Pseudo-.END.	3 or B	57 or 185	X	X
PRIVATE Pseudo-.END.	7	121	X	X
.END.	2 or A	41 or 169	No!	X
PRIVATE .END.	6 or E	1 05 or 233	No!	X
Pseudo-END	1 or 9	25 or 153	No!	X
PRIVATE Pseudo-END	5 or D	89 or 217	No!	X
LBL	F	Do not use		X

Table 4.3 -- Synthetic END's

The values underlined in the table are not necessarily stable. Without the correct values in nybbles 2 to 4 to link this END to the next higher global function, packing is likely to alter this END unless nybble six is 0 or 9. Often the END type will change to 0. The information in the last two columns will aid anyone willing to risk exploring these synthetic ENDs using the Byte Grabber. Detailed instructions describing how to create ENDs in Catalog 1 with the correct linkage values would take up too much space here. If you decide to try, use the Byte Jumper from Section 4I to decode the bytes of an existing END, backarrow it, and synthesize a new END of a different type with 9 or D for nybble six. Be prepared for several MEMORY LOSTs before you find a technique that works.

Nybble five values 0, 3, 4, 7, 8, B, and C give ENDs; these are listed in the top half of the table. The decimal value shown for each table entry is for a byte composed of nybble five as listed,combined with nybble six having a value of 9. ENDs should generally not be used in the place of the permanent .END., but if you want to experiment you can alter the .END. to one of these seven END types using RAMED, POKEB or the PPC ROM's "SX" routine. RAMED or POKEB are best since you can easily leave the first two bytes as they are. You should also use a value such as 0 or 9 for nybble six, so that the "packing needed" bit is clear (bit two). Otherwise, the END type may change during packing, often causing MEMORY LOST. With some END types, it is also possible to backarrow the .END.. This disrupts Catalog 1, leading to MEMORY LOST during packing.

Backarrow must be used with caution if you are using END types other than 0 and 4. Unless PRIVATE is displayed in PRGM mode, the use of backarrow will always change nybble six of the END or .END. to F. This is true even when used on an .END., and no changes are actually made! Whether nybble six is set to F by backarrow or by other changes in PRGM mode, using PACK will result in MEMORY LOST for .END. types 0, 1, 3, 4, 7, 8, B, and C.

The eight possible .END. types 2, A, 6, E, 1, 9, 5, and D are listed next in the table. The decimal values in Table 4.3 again correspond to nybble five combined with nybble six having value 9. Note that nybble six of an .END. will equal either 9 or 0 after using GTO . . or PACK.

Changing an existing END to one of these .END. types is easily done with the ZENROM's RAMED or the CCD Module's POKEB. There will be no ill effects of this change as long as the program remains undisturbed. However, if the END is moved within Catalog 1 (other than by SIZE changes or the deletion of programs in Catalog 1 above or below this program), several things may occur. After packing, you may discover that this END has become the .END., resulting in the loss of all programs below it (sometimes including your key assignments, I/O buffers, and alarms). MEMORY LOST is also possible. Experiment at your own risk.

The last listing in Table 4.3 is the Alpha label, in which nybble five has the value F. You can make labels with RAMED, POKEB, or "LB". But you should avoid using the Byte Grabber with labels, since this can disrupt Catalog 1 or cause instant MEMORY LOST. Labels are mentioned here for the sake of completeness.

Now that you are aware of the dangers involved, you are probably wondering what possible good these oddball ENDs can do. Some examples of the usefulness of synthetic END types follow, along with an explanation of the behavior associated with these ENDs.

If you use the ZENROM to change the .END. from type 2 to type 6, as long as the next global function upward in memory is an END, no one will be able to key in a program by hand at the end of memory. If the other ENDs are also PRIVATE, access to Catalog 1 will be restricted to someone expert enough to break Private status.

To change the .END. to type 6 with the ZENROM, use GTO . . to ensure there is an END above the .END., and to position to the .END. . Press PRGM and execute RAMED. You may find as many as six nulls before the .END. (which will appear as CC,01,20 or similar). Repeatedly press PRGM until byte 20 or 29 is shown in the middle of the display, and key in 6 0. Press ON to exit RAMED, and PRIVATE is shown immediately.

With a CCD Module, you first need to find the decimal address of the register that contains the .END. . This is easiest with the PPC ROM's "E?" routine. Otherwise you could RCL . c, DCD, and manually convert the last three hex digits in ALPHA to a decimal address. Once you have the address of the .END., you can use POKEB to replace its last byte with decimal 96 (hex 60).

Another way to achieve this result is to do GTO . . and WPRV or WRTPV to save the empty program as private on a magnetic card or mass storage file. Then read the empty private program back into the HP-41, and the .END. will be PRIVATE (type 6).

To resume normal operation, it is best to position yourself to the PRIVATE .END. by letting CAT 1 run to completion, then executing "CLP" ALPHA ALPHA. You may also use GETP (but not GETSUB) to load a program from Extended Memory to replace this PRIVATE .END. . Reading a program card will work the same way.

Using a type 5 .END. as an END is the ultimate protection for your programs. Since this type is also PRIVATE, access to your program is denied to anyone but those who can break PRIVATE. If someone makes a copy of your program on cards, cassette, or in Extended Memory (and swaps the module into another machine), they will have a copy which is very hazardous to use.

Type 5 is a PRIVATE Pseudo-END (literally a false END). While its usual function is as an .END., it displays as an END. It can function correctly as an END as long as nybble six remains 9 or 0 (no alterations should be made to this program) and nybbles 2 through 4 link it properly in Catalog 1.

If a program with hex 59 in nybbles five and six of the END is read into memory (from cards, IL devices or Extended Memory), using GTO . . will change this END into the .END. (type 7, still PRIVATE). Later breaking Private status and using the Byte Grabber to get rid of the troublesome END which is restricting access to the program will result in MEMORY LOST at the next packing. And if PACKing is done right after reading the program in, MEMORY LOST is immediate. Using type 5 for an END enhances the protection of Private status by creating pitfalls to trap all but the most knowledgeable programmers. Intentionally using type 5 for the .END. is also dangerous since this .END. appears as END in Catalog 1 or in PRGM mode after breaking Private status. Using backarrow to remove this innocent looking "END" has consequences you know well by now.

Another disadvantage of using a Pseudo-END (types 1, 9, 5, or D) for the .END. is that it leaves you with no ".END. REG nnn" as the last entry in Catalog 1 to tell you the number of free registers. Of course, you can always CAT 1 to a non-PRIVATE program, press RTN in run mode and press PRGM to see "00 REG nnn". But this is bothersome.

There is an easy alternative, which you might also like to use if your Catalog 1 is rather long. Take advantage of the fact that a type 3 Pseudo-.END. displays as ".END. REG nnn", though it functions as an END. Try the following example with the ZENROM: CAT 1 in run mode and R/S immediately. Press SHIFT, RTN, PRGM, XEQ "END", XEQ"RAMED",USER,30,ON,(BST,RAMED,PRGM,PRGM,09,ON to avoid decompiling the first program in Catalog 1) and press PRGM. Now start Catalog 1. You should see ".END. REG nnn" as the first entry in the catalog.

The advantage of having this pseudo-.END. at the top of Catalog 1 is that you don't have to execute the PPC ROM's "F?" routine or enter

PRGM mode to find out if you have enough free registers to read in a program without changing your SIZE. It's right there at the start of CAT 1. Unfortunately, END types 3, B, and 7 all have a nasty habit of changing into the .END. when read in from mass storage. After that, they usually become type 1 Pseudo-ENDs. You will avoid problems if you don't record programs containing these END types.

Changing the ENDs of your programs to Private status will protect your programs from all but an expert HP-41 programmer. Similarly, changing your .END. to type 4, 5, 6, 7, C, D, or E will prevent a non-expert from keying in a program. The use of both (along with removing your ZENROM, PPC ROM, and CCD Module) should protect program memory well enough so you can loan your HP-41 to an inexperienced user without fear that you will find program memory altered.

AVOIDING DECOMPILING

Whenever you read in a program from the Card Reader, Digital Cassette Drive, or Extended Memory, any information contained within the GTO or XEQ instructions of the program when it was recorded will be brought into memory. However, this jump distance information will be lost as soon as you GTO . . because the program lacks an END with the proper byte values (indicating that decompiling is not necessary) to prevent this.

If you have a ZENROM, it is extrememly easy to avoid decompiling. After reading in the program, BST or GTO .999 to position to the .END. . Enter PRGM mode, and be sure not to exit PRGM mode or PACK until you have finished modifying the END to prevent decompiling. Execute RAMED. At this point, there are at least two different courses you can take. The first method is the fastest.

In RAMED mode, press I for Insert mode. Flag 1 will be set and show in the display. Now press C 0. If the display shows C0,00,00 at this point, press ON and PACK, because you're done. Otherwise, press 0 four times (or 0 0 0 9) ON and PACK. PACK will compute the correct values for nybbles two to four without decompiling as long as nybble six is 0 or 9. Incidentally, you can also use hex 49 instead of 00 or 09 to make a PRIVATE END, or you can use any of the other END types you like.

Two other methods for avoiding decompiling have been devised by Clifford Stern. The first requires the ZENROM, while the second requires the Byte Grabber key assignment.

After reading in the program from a mass storage device, BST twice to position to the last program line. Enter PRGM mode, XEQ "END", execute RAMED, press USER 0 9 (or 0 0) and ON. Again, it's that simple.

For those using the Byte Grabber, here is Clifford Stern's other method, which also appeared in "HP-41 Synthetic Programming Made Easy". After reading the program into memory, switch to PRGM mode and BST. This puts you at the .END., which is the last line of the program. Make sure that there are at least 2 free registers (.END. REG 02 or greater). Press ENTER, STO IND 66, FIX 9, BST, BST, Byte Grab, backarrow twice, and PACK (not GTO . .). The IND 66 suffix becomes the first byte of a packed END, which prevents the processor from clearing the compiled branch information. No bytes are wasted because the PACK operation removes all packable nulls from the program. The presence of the new END eliminates the decompiling which would ordinarily follow.

To make a PRIVATE END with the Byte Grabber, use ENTER, STO IND 77, ENTER, +, and the same steps outlined above.

Byte Counting with ENDs

You can count the number of bytes and registers contained in a packed program by decoding the END itself. A typical END has the structure

C a b c 0 9 in hex (refer to Table 4.1, page 150). The three nybbles a b c point to the next global instruction upwards in memory. Alpha labels share the same structure C a b c followed by a Text byte from row F of the byte table. See SQRG p. 39. Together, these global labels form a linked list from the .END. up to the top global label in Catalog 1 (which has zero in nybbles a, b, and c).

For all practical purposes, nybbles b and c represent the number of registers in the distance to the next global function. These two nybbles can symbolize from 0 to 255 registers. Since main memory contains 319 registers plus 4 bytes, you could possibly have a program with a single Alpha label which contains more than 255 registers. Hewlett-Packard had to allow for this, so the first (rightmost) bit of nybble a was set aside for this possibility.

The remaining three bits of nybble a contain the number of bytes in excess of the register count. Since the first bit is seldom used, this nybble will usually be even, and will range from 0 to C for zero to six bytes.

The formula to compute the number of bytes (from the byte before the global being decoded back to and including the previous global) is given by:

$$(a / 2) + (7 * (16b + c))$$

If a program has a single Alpha label at its first line, the information within nybbles a, b and c of the END equals the byte count of the entire program. This does not include the END itself. ZENROM users can decode this information from global instructions very quickly using RAMED and the byte table. CCD Module users can execute PHD, A+, A+, to locate the middle byte of an END, then PEEKB, A+, PEEKB, to get the decimal values of the bc and Ca bytes. If you don't have a ZENROM or CCD Module, use the Byte Jumper presented in Section 4I.

4H. EXECUTING BYTES IN ALPHA

Often you need to execute a synthetic two-byte instruction from the keyboard. This can occur during your day-to-day use of the HP-41, or during development of a program. It's tedious to constantly create instructions with the byte grabber or to make synthetic key assignments and to have to clear them after one or two uses. If you don't need to put the instructions in a program, why should you have to bother with assigning a key or creating an instruction in PRGM mode?

Of course, if you have a CCD Module, you can use XEQ ENTER↑ to execute any two-byte function simply by specifying its decimal byte values. If you don't have a CCD Module, read the rest of this section to see how you can get a similar capability.

You can make a program that loads the bytes you want to execute in Alpha. It jumps into the Alpha register, executes the desired bytes, and then restores the program pointer (status register b) to jump back out of the Alpha register. The "2B" program below uses register M to hold your function, as well as several other instructions needed to manage the stack. The Extended Functions module or an HP-41CX is required.

```
01 LBL "2B"                     08  E6
02 "▒" (241,116)                09 X<> b
03 X<>Y                         10 CLX
04 XTOA                         11 RDN
05 X<>Y                         12 CLA
06 XTOA                         13 END     30 bytes
07 "⊦-▒▒▒▒ "                Line 07 is 245, 127, 240, 117, 145, 124
```

Lines 02 and 07 can be created using either RAMED, the byte grabber, Q loader, or one of the byte-loading programs ("LB", "LBX", etc.).

To use the program, enter the following into the stack: the operand, ENTER↑, the prefix (first byte), ENTER↑, and the postfix (second byte). The operand is the number you want in X at the time the instruction is performed, and is optional in many cases. Once the stack is loaded with the proper inputs, just XEQ "2B". Note that the stack is preserved, which allows you to repeat the same function by pressing R/S.

As an example, suppose you are trying to find a TONE that sounds right for a program on your machine. Key in:

159, ENTER↑, 30, XEQ "2B".

To try another TONE, simply press RDN (or CLX), key in the new postfix, and R/S. You can do this because the stack is preserved. Also, you can simply R/S to repeat the same TONE. For example, to now try TONEs 70, 114, and 106, just press RDN (see 159 in the display), 70, R/S, RDN, 114, R/S, RDN, 106, and R/S. These TONEs vary in duration between machines. Finishing with TONE 106 is a good idea, because it does not leave the HP-41CX in the "buzz" mode, which many synthetic TONEs do.

You may use "2B" to perform almost any two-byte instruction, <u>except</u> any RCL (prefix 144), or any other instruction that alters the stack (ENTER↑, RDN, ARCL, etc.; use X<> or VIEW instead). If the stack is altered, the STO b that is the last instruction to be executed in the Alpha register can fail to return the program pointer to line 10, or the result of the function may not be left in the X register.

Other restrictions on "2B" relate to the fact that the byte sequence in the Alpha register must not be altered by the execution of the specified function. You may not use "2B" to execute two-byte functions that specify M or IND M (postfixes 117 or 245), or to execute any of the ALPHA functions (ASHF, ATOX, etc.).

In general, if you stick with prefixes from 145 to 169 and 206, and avoid postfixes from rows 7 and F, you will dodge most potential sources of trouble. If the program stops with an error (NONEXISTENT) before execution is complete, just SF 25 and R/S. If the keyboard is locked up, press ON twice and GTO . .(not PACK). Resist the temptation to enter PRGM mode, as insertion of an instruction in the status registers can lead very quickly to MEMORY LOST.

The best way to get a clear understanding of how "2B" works is to temporarily insert a STOP instruction after line 08. Load the stack with the original 159, 30 (TONE 30) inputs and XEQ "2B". Now decode the contents of the Alpha register (M) using ATOX. Again load the stack with 159, 30 and XEQ "2B". This time, use SST to "walk through" from line 09 to the END instruction. The bytes you decoded from Alpha match the functions performed between the X<> b and CLX instructions listed in the program.

Line 02 of "2B" overwrites Alpha with a character corresponding to R↑ (decimal value 116). Lines 03 to 06 add two more characters to Alpha according to the decimal values of Y and X (prefix and postfix). Line 07 appends characters corresponding to Text 0, RDN, and STO b. The Text 0 serves as a NOP to allow the use of ISG and DSE functions without skipping the RDN instruction that follows. The RDN brings the pointer back into the stack and STO b replaces it.

Line 08 is the synthetic program pointer for Alpha register M. When line 09 puts E6 in register b, execution continues in byte 6 of M. (The four rightmost nybbles of register b actually point to byte 0 of register N at that point, but the HP-41 always displays and executes the instruction that follows the program pointer.) If you single-step the program, you'll see 10 R↑, 11 your function, 12 "" (Text 0 NOP), 13 RDN, 14 STO b, etc. as the bytes in Alpha are executed.

After the original pointer is restored, CLX, RDN, and CLA will clear the pointer (E6) in X and restore the stack. Alpha will also be empty. The line numbers you see as you SST will not match the program listing after line 09. But you can be sure nothing has actually changed in the program by single-stepping it in PRGM mode from line 01.

This program may be run in main memory and the memory of the Extended Functions (decimal address 192 to 64) only. It will not work in the ROM memory areas.

After you understand "2B", you may want to modify or completely rewrite it for another purpose. As an example, you may wish to modify the program to preserve the first 17 characters of Alpha. Or you could completely rewrite it to execute all of the possible TONEs (0 to 127), and, using the Time module, output the function times. To do this, you will need more than the seven bytes within Alpha register M. Therefore, the pointer at line 08 will have to be changed.

The direction of flow when using the status registers as program bytes proceeds from register e to T, that is, from higher to lower addresses. Therefore, changing the pointer line from E6 to E7 will allow you to use register N in addition to M. Fourteen bytes are then available for use instead of seven. Execution starts at byte 6 of register N and continues on into register M. Register M should contain a STO b instruction to restore the pointer and return to the original program. If not, execution will continue on into L and the other registers of the stack, until an error occurs, program execution halts, or the calculator locks up.

You can also use E8 or E9 for pointers to status register O and P, respectively. This will give you a maximum of 21 or 28 bytes. It's best to avoid using more registers than you need. Unused bytes in these registers will be nulls, except the four leftmost bytes in P. When using register P, follow the guidelines in Section 4A.

Non-Alpha status registers may also be used to execute program bytes. The bytes to be executed are assembled in Alpha, recalled using RCL M, and stored in or exchanged with the appropriate status register.

As an example, you might want to do this to perform an alpha function under program control. It's possible to assemble the bytes in register M and move them to another status register, such as register T, while preserving up to 17 of the original characters in Alpha. This approach allows you a maximum of 7 program bytes. Some variation of the sequence CLX, X<> M, STO status, CLX, 7, CHS, AROT should work well. See the chart below for the status registers you can use, along with a list of the pointers needed to access them.

			Decimal
Decimal			Character
Address	Register	Exponent	Value
015	e	E10	16
011	a		12
010	append (R)		11
009	Q		10
008	P	E9	9
007	O	E8	8
006	N	E7	7
005	M	E6	6
004	L	E5	5
000	T	E1	1

To construct the proper pointer for status registers a through Q, it is necessary to use Alpha. Using Q as an example, you could use the sequence CLA, 10, XTOA, CLX, X<> M to make the pointer. Due to the nature of register Q, this would have to be done before Q is loaded with bytes. This pointer could remain in the stack (or another status register), or it could be ASTOred as a single character and brought back to X when needed.

4I. HOPPING THROUGH RAM AND ROM

Synthetic programming provides the fun, and often useful, capability to make hops (or jumps) in RAM and ROM. This is somewhat similar to the jump that occurs when a GTO or XEQ instruction is encountered in a running program. Two methods can be used. The first method can only be used to move forward (downward) in program memory, either RAM or ROM, a distance of 0 to 15 bytes. The second method can be used to move in either direction, up to 110 bytes in RAM. In ROM, the maximum jump is 99 bytes (127 if you have a ZENROM).

Both methods make use of synthetic key assignments, and in both cases the assigned key must be pressed in RUN (non-PRGM) mode and USER mode.

The first method uses a class of key assignments called Byte Jumpers. Their usefulness was discovered by Bill Wickes, who made most of the important early discoveries in synthetic programming. Both the Byte Grabber and any Text 0 prefix assignment will byte jump in RUN mode. Prefix 0 assignments having a postfix from the last row of the byte table (0, 240 through 0, 255) will also byte jump. So do many other assignments with a prefix or postfix from row F of the byte table. But there is no need to bother with these other assignments, since one of the byte grabbers or a Text 0 prefix assignment will byte jump in RUN mode and serve other useful purposes in PRGM mode. If you do not already have one of these assignments made to a key, assign 240, 27 or 247, 63 to a key with a key assignment program like "MK" or "MKX", and then continue reading.

Now GTO . . and key in the following program lines:

```
01 LBL"bJ"
02 ENTER↑
03 "ABC"
04 LBL 01
05 "OPQRSTUVWXYZ"
```

Now return to run mode, GTO .003 and Byte Jump (press the key with the Text 0 prefix or byte grabber key assignment). Press ALPHA to examine the bytes in the alpha register. These bytes can be decoded using the Extended Function XTOA or the PPC ROM's "CD" routine.

If you used "CD", return to the program by doing XEQ "GE", GTO .003, and byte jumping again.

Now enter program mode and you should see

03 / .

Note that the line number has not increased. The line number is never updated during a byte jump. Next press backarrow and SST. Since we deleted the / byte, which was shown in the text line as the character C, we now have

03 "AB ¯ " .

The C has been removed, with a null in its place.

Now return to run mode, GTO .005, and Byte Jump. Switch into and out of ALPHA mode to make sure that the alpha register contains two characters. Enter PRGM mode and you should see

05 LN .

This is the byte that you keyed in as the character P in the text line.

Now key in ALPHA A B C D ALPHA, followed by BST and SST. Notice that the text line now appears as

"OP ABCD¯¯QRS" .

The text instruction "ABCD" opened up seven bytes within the "OPQRSTUVXWXYZ" text instruction. Including the F4 prefix byte, this instruction occupies 5 bytes, leaving 2 nulls. Of course, these 7 bytes are now characters rather than instructions, and the seven characters TUVWXYZ have been pushed outside the text line and are now separate instructions. This will be explained further in the paragraphs that follow. Meanwhile, you can restore the line number to its proper value by using GTO .005 or XEQ "PACK".

As you have seen, the Byte Jumper can be used to hop right into the middle of a text line and allow you to modify it. If you press the byte jumper accidentally or if you are just fooling around with it, don't be too surprised if you jump right into the middle of a text line or other multibyte instruction. Be sure not to alter any ENDs in main memory, or you may disrupt Catalog 1, leading to a possible MEMORY LOST.

The process that caused line 05 to change began when you keyed in an instruction with 05 LN (which was actually the character P) in the display. The HP-41 shifted all the bytes following LN downward one register (7 bytes) to make room for the insertion. The Text 4 byte plus the 4 characters A, B, C, and D use 5 of these 7 bytes. The remaining 2 bytes are left as nulls. The characters after Q, R, and S were pushed out of the original text line because of the one-register shift. You will find seven instructions corresponding to the characters T through Z in the form of program lines 06 through 12.

Now enter PRGM mode, GTO .004, backarrow that line, and key in LBL 14. Leave program mode, SST, and Byte Jump. Press ALPHA and you'll see the following 15 characters: a Text 12 byte (which displays as a starburst), O, P, Text 4 (starburst), A, B, C, D, two nulls, Q, R, S, T, and U. You can verify the identity of the starbursts by using the Extended Function ATOX or the PPC ROM routine "CD". These bytes were copied out of the program by the byte jumper as it performed the jump.

Exit ALPHA mode for a moment and byte-jump again. The alpha register now contains V, W, X, Y, and Z. These characters are program bytes following the ones copied on the last jump.

You have probably noticed that the number of program bytes copied into the ALPHA register (which is the same as the number of bytes in the forward jump) varies oddly. The length of the jump depends on the value of the second hexadecimal digit of the last byte in the line preceding the current line. So when we changed line 04 from LBL 01 to LBL 14 and used the Byte Jumper at line 05, fifteen bytes were copied into Alpha instead of two bytes as before. This is because LBL 14 appears in column F (fifteen) of the byte table, while LBL 01 appears in column 2.

Now you may clear the "bJ" program from memory as it is no longer needed.

Since the column number of the preceding byte is what determines the distance of the jump, you should realize that bytes from column 0 of the byte table (NULL, 0, RCL 00, STO 00, +, etc.) will not advance the program pointer or copy any bytes into the Alpha register. If you find yourself "stuck" when you want to continue to advance, you can try backing up to see if you can get a longer jump from an earlier point in the program. PACKing may also help, but not in ROM. If these tricks do not work, SST to the next line and continue.

The Alpha register will automatically be cleared when you use byte jumping to copy bytes into it, with one obscure exception. If the byte you are positioned to displays as CLD (hex 7F), it will behave like the append symbol (⊦), causing the following bytes to be appended to the current contents of Alpha. (As you know, any text instruction that starts with an append symbol causes the rest of the bytes in the text line to be appended to Alpha.) When byte jumping at a CLD instruction, no warning tone will be sounded if the Alpha limit of 24 characters is reached or exceeded.

If you think about this behavior carefully, you can see that the Byte Jumper behaves precisely like a Text instruction that is executed in a program. The line number is not updated, as indeed it would not in a running program. A Text n prefix causes the following n characters to be copied into Alpha, simultaneously jumping over these bytes to the next program instruction. The main difference is that when you byte jump, you are causing a Text instruction (hex Fn) to be executed, even though the byte that governs the jump may not be from row F. The operating system is forming the instruction by taking the first digit of the prefix (F) from the first digit of the key assignment and the second digit (n) from the second digit of the preceding byte.

Byte jumpers also work well in ROM. Again, line numbers are not updated as you jump. The bytes in Alpha after you byte jump are the same as the bytes you would get using the COPY function. They aren't the same as the actual ROM contents, since all ROMs use 10-bit rather than 8-bit words.

The fact that the line number doesn't change as you advance through memory can create some problems. If you BST when the line number is 01 in ROM, the line number will stay 01 and you will BST to the previous instruction. But in RAM, you will find yourself bounced to an END. This may be the END of the program you were in, the END of the preceding program, or the END of the first program in Catalog 1. This is very quirky and apparently unpredictable.

Another quirk of BST is encountered when you BST after byte jumping into the middle of a global instruction (Catalog 1 LBL or END) in RAM. After backstepping, you may find yourself almost anywhere in memory, from the data registers to the second Extended Memory module (even if you don't own one, the program pointer can still wind up there). An indication that this is happening is that the line number alone stays in the right side of the display for 5 to 10 seconds as the calculator searches empty registers for an instruction to display. For this reason, it is wise to SST once and BST twice, rather than using a single BST, if you think you might be in the middle of a global instruction. Another alternative is to use the register/byte jumper presented later in this section.

With a little practice, you can use the Byte Jumper to figure out whether a global label has been assigned (and if so, to which key), whether an END has been PACKed, or whether a GTO has been compiled. The crucial elements of this process are a knowledge of instruction structure and locating a suitable instruction one or more bytes before the instruction in which you are interested. The instruction that you locate must be from a column of the byte table such that the jump distance encompasses the instruction you want to investigate.

The second method of hopping through memory uses the Register/byte Jumper. This synthetic key assignment was first reported in the August 1982 issue (V9N5P7a) of the PPC Calculator Journal by Tapani Tarvainen. There are actually three fully functional Register/byte Jumpers. The only differences among them are the prompts that are displayed and the labels that are searched for when 00 is used to fill in the numeric prompt. Choose one of the three Register/Byte Jumpers listed below and assign it to a key. The discussion will be easiest to follow if you assign the first one:

"MK" inputs	Display	GTO label number
4, 178	⟩ __	01
4, 183	|Σ+ __	06
4, 188	2 __	11

The first Register/byte jumper listed is often called the "goose" assignment because it looks like the "flying goose" that jumps across the display as LBLs are encountered in a running program. It is the most commonly used of the three because of its unmistakable appearance, and because the prompt reminds you that the function has something to do with jumping. (What does one do when goosed?)

The second Register/byte Jumper is a bit strange in appearance. The third may look vaguely familiar, because the prompt it gives is the same as the one given by the 4, 31 (W') Q loader. It is also the same prompt as for the non-programmable assignment 0, 13 that locks up the keyboard temporarily (until you remove and replace the batteries). That key assignment can cause MEMORY LOST.

All these Register/byte Jumpers function like GTO instructions that are encountered in a running program. They do not alter the contents of the Alpha register. The two digits that you key in to fill the prompt form a byte that is used to represent the jump distance. The number of bytes jumped is different in RAM and ROM, precisely as the structure of compiled jump information in a GTO instruction is different in RAM and ROM. This will be explained further through examples. We will assume that you have assigned the "goose" key assignment, so you should press the Register/byte Jumper when a reference is made to executing the } function.

In ROM, the digits you supply in reponse to the } __ prompt are interpreted simply as the number of bytes to be jumped. Let's try an example. If you have a PPC ROM, GTO "SD". You are going to jump forward in ROM to LBL "SK". Press the Register/byte Jumper key and fill in the prompt with IND 20 (press shift 2 0). Enter PRGM mode and you should see

53 LBL "SK" .

Note that the line number is correct. This is always the case when using a Register/byte Jumper because part of the HP-41's programming makes sure that when a GTO is executed, the line number is replaced by FFF. This special code indicates that the line number is invalid. Then, when you enter PRGM mode or press the SST key, the operating system knows that it needs to recompute the correct line number. This recomputation of the line number also means that you cannot use a Register/byte jumper to get into the middle of an instruction in PRGM mode, as will be explained shortly.

Now switch out of PRGM mode and press } 20. You're back at LBL "SD"! Summarizing, to move forward nn bytes in ROM, fill in the prompt with IND nn, and to move backward use nn.

As another example, switch out of PRGM mode again and press } 40. This jumps you backwards 40 bytes to

26 LBL "LR" .

If you want to avoid switching into and out of PRGM mode, you can check your location in the program by pressing SST and holding it long enough for the instruction to be replaced by NULL.

Here's a final ROM example, illustrating that the goose assignment does not respect the placement of ENDs. In fact, it totally ignores them. First GTO "CV" and jump backwards 5 bytes (press } 05). If you enter PRGM mode you will find yourself at

131 END .

This END belongs to the preceding block of PPC ROM routines, "SR" through "BV" (XROM 20,00 through 20,07).

ZENROM users can use the register/byte jumper to hop up to 127 bytes in ROM. This can be done by pressing EEX to extend numeric input beyond two digits, or using inputs .T through .e for jumps of 112 through 127 bytes.

When you jump in either direction with this assignment, you may encounter certain cases for which the reverse jump will not bring you back to exactly where you started. This will occur if you jump into the middle of any multi-byte instruction and you look at that instruction by either holding SST or switching into program mode. In order to let you look at the instruction, the HP-41's operating system must compute a new line number (because the goose function caused the line number to be replaced by FFF, indicating the necessity of recomputation). The process of line number recomputation involves counting down from the top of the program to the current position. The calculator will not let you remain in the middle of a multibyte instruction at the end of this process. Instead, it will back you up one or more bytes to the beginning of the instruction you were in.

Also watch out when hopping over ENDs in ROM programs. They are 5 bytes long rather than 3, and you <u>can</u> offset the line number if you hop three or four bytes instead of five. This only happens in the forward direction over an END in ROM.

The use of the Register/byte Jumper in RAM is quite different. To demonstrate, using the PPC ROM again, make sure that 16 registers are free, XEQ "COPY" ALPHA A M ALPHA, then GTO "AM". Programs "NS", "NR", "PO", "Rb", "AM", and "MA" are now copied into RAM at the bottom of Catalog 1. If you do not have a PPC ROM from which to copy this program block, you should still be able to read through and understand the examples quite easily.

In RAM, the input number does not directly represent the number of bytes that are jumped. In fact, the numbers 01 through 15 will cause a jump of that many registers (7 to 105 bytes). The input value 16 will cause a jump of one byte, 32 will jump 2 bytes, 48 will jump 3 bytes, etc. From this behavior comes the name Register/byte Jumper.

Let's start with an example of how to use the Register/byte jumper in RAM. You should still be positioned to "AM" in run (non-PRGM) mode. Press

            ⦘ 18

then press and hold SST to see

            44 LBL "MA" .

The input value of 18 caused a jump forward in RAM. (In ROM the jump would have been backwards. This is due to the fact that ROM addresses increase as line numbers increase, whereas RAM addresses decrease as line numbers increase.) You fill in the prompt with IND nn to jump backwards in RAM.

The jump in this case is 15 bytes (two registers plus one byte). The general formula to compute the corect number to fill in the prompt is

            (16*B) + R ,

where R is the number of registers you want to jump (often zero, but no larger than 15) and B is the number of additional bytes to jump (zero to 6). Because of this nutty formula, larger numbers do not necessarily mean larger jumps. An input of 96 causes a jump of 6 bytes, but an input of 95 yields the maximum jump of 110 bytes.

Also, only 99 of the 111 theoretically possible jump distances are available for use with this function, unless you have a ZENROM. You cannot jump exactly 34, 41, 48, 55, 62, 69, 76, 83, 90, 97, 104, 111, or 112 bytes in one jump. This is because combinations involving 6 bytes and 4 or more registers require an input in excess of two digits. See the table below:

Inputs	RAM Jump distance
01 to 15	1 to 15 registers (7 to 105 bytes)
16 to 31	1 byte plus 0 to 15 registers
32 to 47	2 bytes plus 0 to 15 registers
48 to 63	3 bytes plus 0 to 15 registers
64 to 79	4 bytes plus 0 to 15 registers
80 to 95	5 bytes plus 0 to 15 registers
96 to 99	6 bytes plus 0 to 3 registers

Additional inputs possible with the ZENROM:

100 to 111	6 bytes plus 4 to 15 registers
112 to 127	1 to 16 registers (7 to 112 bytes)
T to e	1 to 16 registers (7 to 112 bytes)
129 to 199	Same as IND 01 to IND 71
128	Same as 0 - - USE WITH CAUTION

You may be wondering why an input of zero is not shown in this table. When zero is used for the input, a search is initiated for a label that matches the Register/byte Jumper being used (LBL 01 for the "goose" assignment, LBL 06 or LBL 11 for the others). It is just as if you single-stepped a GTO 01 in run mode. If no LBL 01 is found, NONEXISTENT is displayed. If IND 00 is used for an input, nothing happens. But if you use 00 for an input and LBL 01 is found, you are in trouble!

First, the program pointer is changed to match the location of the label. No surprise there. Second, if the distance to the label is less than 112 bytes, the operating system will compute the correct distance byte and put it in the byte before the current line, overwriting what was there before.

The value of the distance byte computed by the operating system is in the same format as the input number in the above table. If the label follows the GTO (that is, if it is at a higher line number), the formula for the decimal equivalent of the distance byte is

(16*B) + R .

If the label is above the GTO (a backward branch), the formula is

128 + (16*B) + R .

Be cautious when experimenting with this aspect of the Register/byte jumper. Though you can create a synthetic RCL instruction when LBL 01 immediately follows the instruction you are positioned to when you execute } 00, you shouldn't alter instructions haphazardly. You might accidentally create an END, or even alter the PRIVATE status of the preceding program. Be sure to count the byte that the label occupies if the jump is backwards. The jump is to the beginning of the label, so a forward jump does not include the label itself.

The Register/byte jumper can be used to hop over the .END. and into the assignment registers. The Byte Jumper also has this capability. From there, you can continue to SST through the Extended Functions module and into the status registers. But beware! If you press any instruction other than SST, BST, GTO .nnn, or backarrow in PRGM mode, you will cause MEMORY LOST. The reason is that this insertion of an instruction pushes down memory contents by one register, all the way into the status registers, where status register c is altered to a value that invariably causes MEMORY LOST. To experiment more safely, increase the SIZE to the maximum (so that you see .END. REG 00) before you jump beyond the bounds of program memory.

Moving in the other direction, the Register/byte Jumper will not cross the curtain into the data registers. It will stop at the top of the first program in Catalog 1. Any attempt to use the Register/byte jumper from a point within the data registers will move the pointer to the first line of the first program in Catalog 1, unless the jump is specified to go farther down into program memory.

The true nature of the Register/byte Jumper is revealed in PRGM mode. If you have the ZENROM or CCD Module, you can easily examine the GTO instructions created by this function. The first byte is set by the assignment, while the second byte is determined by the number keyed in. The short-form (two-byte) GTO created by these assignments will have the jump distance field cleared by the operating system while leaving PRGM mode in most cases. With some care to avoid decompiling, this assignment could be used for the GTO instructions within a label-less program (such as the one presented at the end of Section 3F). The Register/byte Jumper is actually a pre-compiled GTO which can be executed in run mode.

CHAPTER FIVE -- ZENROM UTILITIES AND M-CODE

5A. USING SELECTED ZENROM FUNCTIONS

The CCD Module's excellent Owner's Manual includes many useful utility routines that illustrate the use of the CCD Module functions. This section is intended to supplement the ZENROM manual, showing you how to use ZENROM functions to perform tasks on the HP-41 which have traditionally been done by programs.

The CCD Module and its Owner's Manual have equivalents to PPC ROM routines "GE", "HN", "NH", and "MK". This section is especially for ZENROM users. It will show you how to use the ZENROM in place of these and other PPC ROM functions, as well as their equivalents such as buffer-compatible "LBX" and "MKX" (SPME, pages 65 and 92).

LASTP

The ZENROM‘s LASTP function replaces the PPC ROM routine "GE". The one minor difference is that "GE" halts at line 00 of the the last program, eliminating all subroutine returns, whereas LASTP positions the user program counter to the first line of the last program in program memory, without eliminating pending returns. (The last program in main memory is the program containing the .END.) Thus, when you use LASTP in a running program the effect is equivalent to XEQ "last program".

CODE and DECODE

Just as the Extended Functions ATOX and XTOA are faster than their "CD" and "DC" PPC ROM counterparts, the ZENROM functions CODE and DECODE are much faster than the user code programs "HN" and "NH". Machine-code functions are generally ten to one hundred times as fast as their user code equivalents. Sometimes it is not even possible to construct an equivalent function in synthetic user code.

The ZENROM's DECODE function converts the contents of X into fourteen characters, each of which represents one nybble. This is equivalent to the behavior of the CCD Module's DCD function and the PPC ROM routine "NH" (with Flag 9 clear).

The "MT" (mantissa) program listed below this paragraph demonstrates the use of DECODE. This "MT" program is similar to the PPC ROM routine of the same name. The result of executing this "MT" is a ten digit number, except when the input was zero, in which case 0 is displayed. The result for non-zero inputs can range from 1,000,000,000 to 9,999,999,999, and is always positive. All ten digits are shown if you have selected any FIX display format. This "MT" program requires Extended Functions or an HP-41CX in addition to the ZENROM. With the CCD Module, use DCD at line 03.

```
01 LBL "MT"         06 RDN
02 ABS              07 ANUM
03 DECODE           08 CLA
04 RDN              09 END
05 ATOX                      19 bytes
```

Line 02 copies X into L, while not affecting the mantissa of the result left in X. Lines 03 and 04 decode this number. Lines 05 and 06 remove the leading zero (decimal character code 48), which represents a positive sign. Line 07 converts the contents of Alpha back into a number. Since the HP-41 is limited to ten digits, the exponent sign and exponent digits in Alpha will have no effect. Line 08 clears Alpha, and line 09 halts execution.

Only the LASTX register is lost from the stack. The original value of

X overwrites L. The other stack registers remain unchanged and Alpha is cleared. Non-Normalized Numbers will result in a value in X containing less than ten digits in most cases, depending on the effects of lines 02 and 07.

The ZENROM's CODE function is similar to the "CODE" programs (including the PPC ROM"S "HN" routine) which have been around for years, starting with the first "CODE" developed by synthetic programming pioneer Bill Wickes. The ZENROM's CODE converts the rightmost fourteen hexadecimal digits in Alpha into seven bytes. These seven bytes are brought into stack register X in the same way the RCL function would bring them in, raising the stack if the stack lift is not disabled. The fourteen bytes in status registers N and M are interpreted as the nybbles to be coded and brought into X. Bytes in Alpha register O and P are ignored. Leading nulls are coded the same as the character zero ("0"). CCD Module users can use the "CDE" program from page 7.15 of the Owner's Manual to do the same job as CODE.

To see how the ZENROM's CODE function interprets non-standard input characters (other than 0-9, A-F, or null), you need to understand the algorithm that it uses internally. The scheme used by CODE to convert the bytes (digits) in Alpha to nybbles in X is simple and elegant, relying on the bit-level structure of the input. CODE only pays attention to certain parts of each character byte. The second bit of the first nybble is used like a flag to determine how the second nybble is to be encoded. Three examples are given below, with this special bit underlined:

	Hex	Binary
null	0 0	0 $\underline{0}$ 0 0 0 0 0 0
"0"	3 0	0 $\underline{0}$ 1 1 0 0 0 0
"F"	4 6	0 $\underline{1}$ 0 0 0 1 1 0

When the underlined bit is zero, the value used for the encoded nybble is exactly the same as the rightmost four bits (nybble). This corresponds to the column that the byte occupies in the byte table. Nulls and characters "0" to "9" are interpreted in this way.

When the second bit is set, as it is for characters "A" through "F" from row 4 of the byte table, the second nybble must be encoded in a different way. Otherwise, "F" would be taken for a value of 6. When the second bit is set, the CODE function adds nine (binary 1 0 0 1) to the rightmost nybble. Any carry is disregarded. The resulting four bits are used for the value of the encoded nybble. For the character "F" (shown above):

$$\begin{array}{rr} & 0\ 1\ 1\ 0 \\ + & \underline{1\ 0\ 0\ 1} \\ & 1\ 1\ 1\ 1 \end{array}$$

Because of this scheme, CODE will interpret a null, decimal byte 16 (the "alpha identifier" byte added by ASTO), a space, "0", "G" and "W" as zero. Rows 0 to 3 and 8 to B of the byte table are translated according to the column number the byte is in. On the other hand, rows 4 to 7 and C to F are shifted nine places to make "A" through "F" equal nybble values A through F.

Because of the fact that no error message is generated for non-hexadecimal digits in Alpha, decimal values 0 to 15 can be used to represent nybbles 0 to F by simply using XTOA. There is no need to add 48 to values 0 through 9, or add 55 to values 10 to 15 in order to convert decimal nybble values to characters before using XTOA. CODE will correctly translate these decimal byte values as they are.

User-code versions of CODE generally use different methods to interpret the input, so their behavior for non-standard input characters will be different. For the characters from row 0 of the byte table, the PPC ROM's "HN" will work as CODE does, but the "CDE" program from the CCD Module Owner's Manual will code all row 0 characters as 0.

A program which takes advantage of the coding of row 0 characters is listed below. "CX" is a PPC ROM replacement which requires both Extended Functions and the ZENROM. Like the "CX" routine in the PPC ROM, this "CX" is to be used with **extreme caution**. You probably want to remove this program from Catalog 1 after you are done experimenting, because an accidental execution of "CX" is likely to result in MEMORY LOST.

This program moves the curtain to the absolute address specified by the number in X. With Extended Functions and a full complement of memory, the curtain can be moved to addresses 512 through 65, or 16 to 1. If the register below the one named in X does not exist, MEMORY LOST will occur when the program halts. If you simply want to experiment with this program without changing the curtain location, replace lines 28 and 29 with DECODE, AON.

```
01 LBL "CX"    12 FRC       23 RDN        Stack use
02 RCL c       13 X<>Y      24 3
03 DECODE      14 SQRT      25 AROT       T   Old c
04 RDN         15 ST*Y      26 RDN
05 8           16 X<>Y      27 CODE       Z   Z
06 AROT        17 XTOA      28 CLA
07 ST+ X       18 FRC       29 X<> c      Y   Z
08 X↑2         19 *         30 RDN
09 ST/ Y       20 XTOA      31 END        X   Y
10 X<>Y        21 CLX
11 XTOA        22 STO O     51 bytes      L   USED
```

Lines 02 to 04 decode the 14 hexadecimal digits (nybbles) from register c to Alpha. Lines 05 and 06 rotate the cold start constant (hex 169) to the right side of Alpha. Lines 07 and 08 change the number 8 into 256. The first digit of the new curtain address is put in Alpha by lines 09 to 11. (Remember that XTOA will ignore any fractional component in the decimal number in X.)

Line 12 retains the fractional portion (which contains the two remaining digits of the requested curtain value, modulo 256 and divided by 256) while discarding the integer part that has the first digit. Lines 13 and 14 turn 256 back into 16, which will be used to extract the two remaining digits through multiplication (lines 15 and 19). Lines 16 and 17 put the second digit of the address into Alpha. Line 18 discards the second hex digit, leaving the third nybble in the fractional remainder. Line 19 multiplies this fraction by 16, turning it into an integer from 0 to 15. Line 20 puts the corresponding character in Alpha.

Lines 21 through 23 eliminate the old curtain address from status register O before lines 24 to 26 rotate the address of the .END. back into the proper position. Line 27 translates the contents of Alpha into a Non-Normalized Number (NNN) in X. Line 29 exchanges this NNN with the contents of register c. It gets pushed down into register T by line 30. The curtain can be restored to its former location with the sequence R↑, STO c.

Although the "CX" program is not a utility that you will use frequently, it demonstrates how DECODE and CODE can be used together in a straightforward program to edit a NNN. The techniques used here may give you some ideas which can be applied to your programs. Step through the program using SST, switching in and out of ALPHA mode, to see how the program does its job.

RAMED and synthetic key assignments

The CCD Module's enhanced ASN function and the PPC ROM's "MK" program both provide convenient ways to create key assignments of synthetic functions. If you have a ZENROM, you can use RAMED, but you have to know how to proceed. Also, you will need to know the hexadecimal byte codes for the function you want to assign, rather than the decimal codes that you would use with the CCD Module or PPC ROM.

As you recall from Section 1B, the first byte of each key assignment register is hex F0. When the keycode (byte 3 or 0) of a key assignment register is zero, this indicates that the assignment was deleted using SHIFT ASN ALPHA ALPHA. Only when both bytes 3 and 0 of key assignment register are zero (deleted assignments in the same register), will the HP-41 allow the space to be freed by PACKING.

When you make a new key assignment, the HP-41 will check to see whether there is a deleted assignment. If so, the new assignment will overwrite the first available (lowest addressed) deleted assignment.

If no key assignments have been deleted, any new key assignments will take place in the register at address 0C0). If that register is already half filled, a new assignment is put in bytes 5,4, and 3. Otherwise, the other key assignments and buffers are raised one register and the new assignment is in bytes 2,1, and 0 (the rightmost bytes) of register 0C0. In either case, the value hex F0 occupies byte 6 (the leftmost byte).

You can avoid having to look up the hex keycode of a key while using RAMED to make one- or two-byte synthetic assignment. You don't need any programs or have to hunt around. Just use the following procedure:

1. Make a temporary "dummy" assignment to an unused key for each assignment previously deleted (not replaced), or execute the "PK" program in Section 5B (page 191) to pack the key assignment registers.

2. Press SHIFT ASN ALPHA C L P ALPHA and press the key you want to assign with a one or two byte function. (If you have already assigned CLP to a key, you should modify this procedure to use a different temporary assignment.)

3. Key in ALPHA SHIFT ALPHA C 1 ALPHA.

4. Execute RAMED.

5. Press USER twice. Press it three times more if you don't see xx,04,04 in the display (xx means any value is OK). The two bytes 04,04 are the prefix and suffix of the temporary CLP assignment. If you have assigned a function to the ENTER key (hex keycode 04), you should check that the address at the left of the display is 5:xxx or 2:xxx.

6. Key in the two hex bytes for the function (as it would be in a program instruction). Press PRGM once or key in 04 for the first byte if it is a one-byte assignment.

7. Press ON to exit the RAM EDitor.

8. Check the assignment by pressing and holding the key until NULL appears. Go back to step 2 if you wish to make more key assignments.

9. Clear any dummy assignments you made. You're done.

This procedure avoids the need to look up hex keycodes or fiddle with the bits in the assigned key bit maps of status registers e and R (⊢). After you use this technique for a while, you probably won't go back to using synthetic key assignment programs. By understanding the structure of the key assignment register, you can use RAMED to alter assignments to any desired byte values. All you really need is a QRC or byte table to look up the hex values for the function you are assigning. Non-programmable functions (like CLP) are shown in small print above row 0 of the QRC.

If you have difficulty using the RAMED procedure for making synthetic key assignments, you may want to use the fully automated program "MKZ" (Make Key Assignments with the Zenrom) on page 222.

Another popular tool for synthetic programming is the byte loading program. Thanks to RAMED, this type of program is now completely unnecessary. Unless you greatly prefer to work in decimal, you'll never want to sit there entering +, +, +, . . . as long as you have a ZENROM.

To edit existing bytes in a program, begin by executing RAMED in PRGM mode. Position the middle "window" to the byte you want to alter using the PRGM and USER keys. Key in the hex value of the byte which is replacing the one in the middle. RAMED moves to the next byte in memory automatically. Repeat this editing process to your heart's content, pressing ON to stop. How could it be any easier?

To insert bytes, press the "I" (COS) key after executing RAMED in PRGM mode. Be sure the "1" annunciator in the display is lit. The ZENROM uses the operating system of the HP-41 to open up one register at a time as needed, just as in normal entry of program instructions. The HP-41 also decrements the address of the .END. in status register c each time a register is opened up (because the .END. is simultaneously shifted downward). In addition, the HP-41 increments the register portion of the linkage information (nybbles b and c from Table 4.1) in the Alpha LBL, END, or .END. that most closely follows the opened register. This maintains the continuity of the global chain making up Catalog 1. Insert mode is not allowed within the data registers to avoid pushing garbage into the first program. Insertion is also not allowed below the .END. to prevent you from shifting the contents of the status registers. This would cause MEMORY LOST while exiting RAMED.

Many synthetic programs and techniques are replaced by the ZENROM. You really don't need Q loaders, Text 0 prefix assignments, or even the Byte Grabber! (They are still fun to use and useful as learning tools, however.) The ZENROM expands the synthetic programming capabilities of the HP-41 in a natural way. There is little you can't do with a general knowledge of the HP-41, a byte table, and a ZENROM.

5B. USING NRCLM AND NSTOM

The two ZENROM functions you will probably use most often in your own synthetic programs are NRCLM and NSTOM. Using these two functions, you can recall and store data within any existing RAM register of the HP-41, including Extended Memory. See Figure 1.2 on page 13 for the RAM Memory Map.

The ZENROM manual does not give examples of how you can use NRCLM and NSTOM. Since NRCLM is the ZENROM's most powerful programmable function, you should follow through the examples given here. Then you can try to implement some of your own ideas.

The first program presented here to demonstrate NRCLM and NSTOM is "PK" (P̲ack K̲ey) assignments. "PK" also includes a subprogram, "A?", which counts the actual number of key assignment registers used. Unlike the synthetic PPC ROM routines "PK" and "A?", this program does not destroy alarms or other data in the buffer registers above the key assignments. (At the time the PPC ROM programs were written, alarms and I/O buffers could not be foreseen.)

This program uses one flag, number 20. At present, this flag is not used by any peripherals or modules. Since it is cleared at turn-on, it should be safe to assume that it is clear at the beginning of the program. "A?" sets flag 20 at line 02, with execution continuing into "PK". Lines 28 and 29 will halt the program with the number of assignment registers currently used in the display (X). This will be a whole number. If flag 20 was set accidentally, halting "PK" at this point, or you want to pack the key assignment registers after counting the number used, just press R/S to restart the program.

CCD Module users note that your Owner's Manual contains "A?" and "PK" programs that use CCD Module functions.

```
01 LBL "A?"
02 SF 20
03 LBL "PK"
04 191
05 SIGN
06 70
07 LBL 10
08 ISG L
09 LBL 01
10 LASTX
11 CLA
12 XTOA
13 RDN
14 NRCLM
15 DECODE
16 RDN
17 ATOX
18 X=Y?
19 GTO 10
20 LASTX
21 LASTX
22 192
23 -
24 X≠0?
25 FS?C 20
26 RTN
27 ST+ X
28 ISG X
29 LBL 01
30 DECODE
31 CRFLD
32 RDN
33 DSE X
34 LASTX
35 X<>Y
36  E3
37 /
38 +
39 SAVEX
40 LBL 09
41 CLA
42 XTOA
43 NRCLM
44 XEQ 08
45 RDN
46 X<> O
47 XEQ 08
48 R↑
49 R↑
50 ISG X
51 GTO 09
52 RCLPT
53 FRC
54 SEEKPT
55 GETX
56 LBL 07
57 "▩"  (F1,F6)
58 GETX
59 GETX
60 X<> M
61 STO N
62 RDN
63 "⊦***"
64 STO M
65 RDN
66 "⊦¯¯"
67 XTOA          "A?"
68 RCL N
69 NSTOM         "PK"
70 RDN
71 ISG X         185
72 GTO 07        bytes
73 CLA
74 FLSIZE
75 DECODE
76 PURFL
77 CLA
78 LASTX
79  E
80 -
81 2
82 /
83 ENTER↑
84 EMDIR
85 RDN
86 CLD
87 RTN
88 LBL 08
89 STO M
90 CLX
91 "⊦¯¯¯¯¯¯"
92 RCL M
93 X=Y?
94 SF 20
95 "⊦*****"
96 CLX
97 X<> N
98 FC?C 20
99 SAVEX
100 END
```

Lines 04 and 05 of "PK" set up the stack to count the number of registers used for key assignments. The count starts at absolute address 0C0 (decimal 192). The loop that counts them runs from line 07 to line 19.

Label 10 counts the number of registers that start with hex value F (decimal character code 70). Lines 08 to 10 increment and recall the counter. The counter is converted to a character in Alpha by lines 11 to 13. Line 14 recalls the register at that address. Lines 15 to 17 overwrite Alpha with the decoded contents of this register and extract the decimal character code of the first nybble. Lines 18 and 19 will repeat this loop until a register is found which does not begin with F. This could be either a free (unused) register, or one which is used for a non-key-assignment buffer.

Line 20 recalls the counter for later use at line 33. At this point, the counter points to the first non-assignment register. Lines 21 to 23 subtracted 192, converting that absolute address into an accurate count of the number of registers used. If "PK" was executed, lines 24 and 25 will let the program continue. Line 25 halts execution if there are no key assignments.

Lines 27 to 29 convert this count into a number used to create a data file to hold both a counter and the assignments. Because each of the key assignments is held in a separate data register, the number of registers needed in the temporary file is twice the number of assignment registers. And because a control number is also stored in the file, the doubled assignment register count is incremented. Lines 09 and 29 (LBL 01) are NOPs.

The name of the temporary file is created using DECODE at line 30. For six assignment registers, a file containing thirteen registers is created. When line 31 opens this file, only the first 7 characters in Alpha are used for the filename. Thus, for six registers, the name of the file is "0130000".

The address of the first non-assignment register from line 20 is decremented at line 33 to get the address of the topmost key assignment register. Line 34 is never skipped. Lines 34 to 38 create a number that controls looping in two different sections of the program. This number is a control word in the format bbb.eeee (192.197 for six assignment registers). Line 39 saves this number in the data file.

The label 09 section uses this number to generate an address in Alpha for recalling the assignment registers. It controls how many times the loop (lines 40 to 51) is repeated. Labels 09 and 08 work together to disassemble and sort through the assignments, saving only those with a nonzero keycodes. Label 08 (lines 88 to 100) is called twice by label 09 to do most of the work.

Lines 41 to 43 convert the integer part of the control number in X to an address in Alpha and recall the contents of this register. Line 44 calls label 08 as a subroutine.

Line 89 overwrites the address in M with the contents of the recalled register. Line 90 clears X, so the zero value there can later be used for a comparison. Line 91 appends six nulls, and the next instruction recalls M. If the keycode of the right hand assignment is zero, the test at line 93 will be true and flag 20 will be set.

Line 95 appends five characters to Alpha. This shifts the right hand assignment over so that it occupies the three leftmost bytes of register N. Lines 96 and 97 recall N while clearing it. Lines 98 and 99 save the assignment if flag 20 is clear, which it will be if the keycode was not zero.

Execution continues at line 45. Line 45 returns zero to X, and line 46 exchanges this with register O. At this point, Alpha is clear. Line 48 calls subroutine 08, which will save the second assigment if the keycode is nonzero.

Lines 48 and 49 rotate the control number back into register X of the stack. Lines 50 and 51 repeat label 09 if incrementing X does not exceed the absolute address of the last key assignment register.

Line 52 recalls the file pointer value at the conclusion of the LBL 09 loop. This will be one greater than the number of legitimate key assignments. Line 53 puts this number in register L while clearing X. Line 54 resets the file pointer to zero. Line 55 recalls the original control number from the file, setting up the stack.

The label 07 section includes lines 56 through 72. This loop reassembles and stores the key assignments. For six assignment registers, this loop will repeat twelve times, no matter how many assignments were valid. This has the effect of storing an F6 byte and six nulls in the vacated (topmost) assignment registers. This empty key assignment register will be removed by the HP-41 by PACKing, or the next time you turn the machine off and on. Line 59 puts F6, rather than F0, in the key assignment register to help prevent accidental alterations in PRGM mode.

Lines 58 and 59 recall the two assignments to be assembled in a single register. This order of operation prevents this version of "PK" from changing the order of assignment pairs as the PPC ROM routine can. Line 60 trades the first assignment with the F6 byte in Alpha. Line 61 stores F6 in the rightmost byte of register N. This is adjacent to the three bytes of the assignment in register M. Line 62 puts the second assignment in X. Note that SAVEX and GETX do not normalize data.

Line 63 pushes the first assignment into register N. Line 64 stores the second assignment in Alpha right next to the first assignment in N. Line 66 shifts them two more places to the left. Line 67 also shifts the Alpha contents one place to the left while creating the needed storage address in M. Line 68 recalls the assembled key assignment register, which has been pushed into register N. Line 69 only uses the two rightmost bytes of M for the address where the reassembled register is to be stored. The other bytes in Alpha have no effect. Lines 71 and 72 repeat the loop until the control number is exceeded.

Line 73 is needed to clear Alpha before the file size is recalled. With Alpha clear, the size of the working file is recalled by line 74. Since the size of the file is the name, lines 75 and 76 purge this temporary file.

Lines 78 to 82 recall the file pointer value previously put in L by lines 52 and 53, subtract one from it, and divide it by two. This converts the pointer into the number of assignment registers used. The count is by half-registers (like the PPC ROM's "A?"), and is exact.

Lines 83 to 86 preserve the count in X while doing an Extended Memory directory. These lines are only needed if your Extended Functions module is revision 1B (Catalog 2 header -EXT FCN 1B). You can use R/S to interrupt the directory, and there is no need to press R/S to complete the program. EMDIR is needed after PURFL to re-establish a working file, or Extended Memory could be accidentally cleared. See page 19 of "HP-41 Extended Functions Made Easy".

"PK" does not actually purge the system of empty registers. After you execute "PK", the empty key assignment registers will exist above the legitimate key assignments, below any other buffer registers. When you PACK or turn the HP-41 off and on, the operating system will automatically get rid of these voided registers. In this way, there is no need for the program to shift buffer registers other than key assignments. These "empty" registers are quickly removed.

An F6 byte is used by "PK" instead of the F0 byte normally put in byte 6 of a key assignment register by the operating system. The F6 byte behaves the same as F0 in most circumstances. However, under some conditions, it will protect your key assignments from alteration. Assignments with byte values from row B through E of the byte table (some of which correspond to branching instructions) can change if they are executed as program instructions. This can happen accidentally after hopping over the .END. with the Byte Jumper, or while recovering from a crash that results in the pointer to the .END. in status register c being incorrect. The F6 byte works just like F0 in normal operation, and it is unlikely to be changed (except by the PPC ROM's "PK" routine). Assignments added after using this Pack Key program will have the F0 byte in the left of the register. If you key in ALPHA, SHIFT, ALPHA, C 1, ALPHA, execute RAMED, and see "F0" in the rightmost window, you can be sure that assignments have been added since you packed them.

The "RENFL" program presented next can be used either to rename a file in Extended Memory or to clear part of the Extended Memory directory. The only input needed by the program is in Alpha. This REName FiLe program will alter the Extended Memory register containing the name of the file.

With an input of "SS,SST" as an example, the program will search for an existing file named "SS", and replace the filename "SS" with the new name "SST". And if you input only the name of the file, that filename is replaced with seven FF bytes. This effectively removes the named file and all subsequent files from Extended Memory.

```
01 LBL "RENFL"
02 LBL 01
03 44
04 POSA
05 X<0?
06 "⊦-,▒▒▒▒▒▒▒"
07 X<0?
08 GTO 01
09 AROT
10 ATOX
11 RDN
12 XEQ 02
13 X<> N
14 STO a
15 X<> M
16 191
17 LBL 03
18 CLA
19 XTOA
20 X<>Y
21 NRCLM
22 X=Y?
23 GTO 04
24 RDN
25 X<>Y
26 CLA
27 DSE X
28 XTOA
29 X<>Y
30 NRCLM
31 STO M
32 ASHF
33 RDN
34 X<>Y
35 ATOX
36 -
37 DSE X
38 GTO 03
39 RTN
40 LBL 04
41 RCL a
42 X<> M
43 ALENG
44 XEQ 02
45 X<>Y
46 X<> M
47 NSTOM
48 RTN
49 LBL 02
50 7
51 X<>Y
52 -
53 LBL 05
54 DSE X
55 X≠0?
56 X<0?
57 RTN
58 "⊦ "
59 GTO 05
60 END
112 bytes
```

Line 06 = F9, 7F, 2C, FF, FF, FF, FF, FF, FF, FF.
Line 46 = F2, 7F, 20 (a nonsynthetic line to append one space).

Lines 03 and 04 determine the position of the comma in Alpha (if any). Lines 05 to 08 append a comma and seven FF bytes to Alpha if there is no comma in Alpha and loop back to line 02. Lines 09 to 11 rotate the comma to the leftmost position in Alpha and remove it.

Subroutine 02 is called by lines 12 and 44. This section adds the number of spaces required to fill register M with the filename and spaces. When the number of spaces are computed by label 02 (lines 49 to 52), label 05 (lines 53 to 59) appends them to the right of Alpha.

When execution continues at line 13, the new filename is recovered from register N and stored in status register a by line 14. Line 15 retrieves the old filename (with spaces) from register M to use in the label 03 loop for comparison. Line 16 is the beginning address used by the loop.

Label 03 performs two functions. This loop checks the header registers within Extended Memory (starting with the topmost register at decimal address 191) for a matching name. When a match is not found, label 03 decodes the rightmost byte of the second header register. This is used to compute the address of the next header register to be checked for a matching filename.

Lines 18 and 19 create the address in Alpha used by line 21 (NRCLM). Lines 22 and 23 skip to label 04 (line 40) when a match is found. Lines 24 to 29 throw away the contents of the non-matching header register and format the address of the next lower header register. Line 30 recalls it and the next five lines extract the rightmost byte value in decimal form. Line 36 subtracts the file length from the absolute address of the second header register, giving the address of the next header register to test.

Lines 37 and 38 will repeat the loop as long as the decimal address is a positive number. Zero or a negative address can result when you specify a nonexistent filename. Line 39 serves as a trap for just this condition.

Line 41 recovers the new filename from status register a. Line 42 swaps this with the address in Alpha. The number of characters are counted by line 43. The next line calls label 02, which again adds enough spaces to Alpha (0 to 7) to fill register M with the filename and spaces.

Line 45 brings the address back into X, while line 46 exchanges this with the filename. Line 47 stores the new filename (or seven FF bytes). Line 48 halts the program.

The seven FF bytes are interpreted by the operating system as the end of Extended Memory. If you accidentally execute "RENFL" without a new filename in Alpha (after a comma), don't despair. You can recover quite easily. The address "RENFL" altered is still in Alpha, and the old filename is in stack register Z. If you used EMDIR, the old filename may have been pushed into T. To repair the damage, press RDN twice (three times if EMDIR ran to completion) and manually execute NSTOM. Check the directory again to make sure it is now correct. Since Alpha contains the address in the form used by the ZENROM, you may also use RAMED to replace the seven FF bytes with a filename. Just be sure to add trailing spaces (hex 2 0) to overwrite all seven FF bytes.

"RENFL" searches through only the header registers of Extended Memory for a matching filename. In addition, because of the limited addressing scheme, only the registers within the Extended Functions module or HP-41 CX will be tested. If you are using Extended Memory modules and you specify a filename that resides within an Extended Memory module, the program will stop at line 30 with NONEXISTENT displayed. If you specify a nonexistent file and the memory within the Extended Functions module/ HP-41 CX is not filled, label 03 may loop infinitely. It is left to ambitious readers to augment this "RENFL" program to eliminate these two restrictions.

5C. MACHINE-CODE

The use of Machine-Code (M-Code) gives an HP-41 user virtually unlimited flexibility and control. Functions such as TIME and XTOA are written in M-Code. It's even possible to write your own Operating System (OS) in M-Code. The best way to get started, after you have the necessary hardware, is to read the book "HP-41 M-Code for

Beginners" by Ken Emery. Then you should write some simple M-Code functions which augment existing instructions. This section gives several M-Code examples, and user-code (HEPFOL) programs to demonstrate them. The examples here will give you a look at some of the capabilities of M-Code. Once you get your hands on the required hardware and read the manuals, you can come back to this section to pick up some programming practice and tips.

In order to write your own M-Code programs, you need a special device called a Quasi-ROM (or Q-ROM). This device contains RAM (Random Access read and write Memory) that is addressed as ROM (Read Only Memory), and that can store the ten-bit instructions needed for M-Code programs. Special M-Code software is required to write data to this memory. The ERAMCO MLDL (Machine Language Development Lab) Q-ROM plus EPROM box comes with the necessary software, or the ZENROM's MCED (Machine Code EDitor) program can be used with the MLDL, ProtoCODER 2, or 16K RAM storage unit. Refer to Appendix B under "EduCALC" as a source for these devices. Also see Appendix A of "HP-41 M-Code for Beginners".

Another useful tool for assembly language programming is an assembler, such as the DAVID Assembler, which is available from ERAMCO Systems, W. Van Alcmadestraat 54, 16785 LS Den Helder, The Netherlands. Their Netherlands telephone number is 02230-34777. Assemblers translate mnemonic-form instructions into M-Code.

A printer is not an absolute necessity for working with M-Code, but it certainly is a convenience. You will need a printer to use the ZENROM's disassembler. However, disassembly to HP-mnemonics is possible without a printer using the DISASM and MNEM functions of the ERAMCO MLDL (see page 13 of the manual). A block of 4K of ROM instructions can be saved on or recalled from an HP-IL mass storage device (Digital Cassette Drive or Disc Drive) with SAVEROM or GETROM from the ERAMCO software. Both ESMLDL-OS and ZENROM 3B allow ROM data to be formatted and stored/retrieved in main or Extended Memory. This works well for relatively small blocks of ROM code, but is not sufficient to store an entire 4K page. Serious M-Coders should have some sort of mass storage device.

To write your own custom functions in M-Code, several items are required. The Q-ROM hardware and software are first. If you have a ZENROM and a Q-ROM box or the MLDL (with or without ZENROM), this is sufficient. Next, you need a working knowledge of synthetic programming. This book and "Extend Your HP-41" are two good sources for intermediate-level synthetic programming information. Next, and most important, you need to know how ROM data is structured. The brief paragraph which follows is meant for review only. Consult your hardware manuals for more complete information (ZENROM manual pages 97 to 101, ProtoCODER2 manual page 36, ERAMCO MLDL manual pages 23 to 24, and "M-Code for Beginners" pages 20 to 23).

The first 10-bit word within a 4K page of ROM designates the device number, more commonly known as the XROM number. The second word of the 4K block tells the HP-41 OS how many functions (up to 040 hex, or 64 decimal) the ROM contains. The next section of the ROM contains the FAT, or Function Address Table. The FAT contains pairs of words giving the address of the first executable instruction of the function. The name of each function is held within the words immediately above the first instruction for that function. The last character of the function name is marked by having hex 080 added to it. At least two nulls must follow the FAT before the first function (or ROM header) name. Failure to construct the FAT correctly will result in garbage or lockup during CAT 2.

If you have used the ZENROM functions NRCLM and NSTOM, you are familiar with the fact that these functions use the three rightmost nybbles (2, 1, and 0) of status register M for the absolute address of the register to be acted upon. If you have ever written, or tried to write, a program to recall or store a block of registers with these functions, you know how much code is required to increment or decrement this address. The following pair of M-Code programs will increment M (INCRM) or decrement M (DECRM). Unlike ISG or DSE, these functions will not cause user-code instructions to be skipped.

```
      08D  M
      012  R
      003  C
      00E  N
      009  I
***   178  C=REG 5/M
      226  C=C+1   X
      2F0  WDATA
      3E0  RTN
      08D  M
      012  R
      003  C
      005  E
      004  D
***   178  C=REG 5/M
      266  C=C-1   X
      2F0  WDATA
      3E0  RTN
```

In the functions listed above, the ZENCODE mnemonics are used. The asterisks indicate the entry points; each is the location of the first executable instruction after the name. Each entry address should appear in your FAT.

These two routines are very simple, and make no use of OS subroutines. Yet they can be used to simplifiy almost every program you write that operates on a block of memory with ZENROM functions. They can even be used to increase/decrease the rightmost character in Alpha for comparison purposes.

The INCRM and DECRM functions are nearly identical in operation. The first instruction of INCRM (or DECRM) recalls status register M to the main register of the processor (C). At the same time, this selects status register M (similar to setting the working file in Extended Memory). The processor is normally in the hexadecimal mode for subsequent math operations. The next instruction adds (C=C+1) or subtracts (C=C-1) one from field X (the exponent field). This increases or decreases nybble 0 by one, with nybbles 1 and 2 possibly affected by a carry. WDATA stores this new value back in status register M. The M-Code RTN instruction acts like the user-code instruction of the same name.

These functions are so simple that the instructions doing the work use fewer ROM words than their names! No stack registers are affected. Both routines are fast, taking about the same amount of time as RCL 00 (23 milliseconds). In addition to being flexible, M-Code is <u>fast</u>.

M-Code routines like these can be written without detailed knowledge of HP-written software. However, most M-Code functions will benefit from the use of subroutines from internal operating system ROMs 0, 1, and 2. Not only will this shorten your M-Code routines, but it can save you a lot of tedious work! There are also certain cases when the last instruction of a function needs to return to the main operating system (OS) processing loop.

In order to correctly and effectively use the OS subroutines, you need to know what the subroutines do, their addresses, and what they do (their inputs and outputs). The VASM listing (which is HP's annotated assembly language listing for the operating system) is the most complete source for this information. In addition to the VASM (see Appendix B), it's a good idea to have CHHU Chronicle V1N2P15-18 and PPC Calculator Journal V12N9P10-16 for reference purposes. If you are a beginner, see PPC CJ V12N10P3-10. Some VASM listings will not necessarily agree completely with the internal ROMs of the HP-41CX. This is an example of why M-Code, like synthetic programming, is NOt MAnufacturer Supported (NOMAS). DO <u>NOT</u> contact HP for assistance with M-Code or synthetic programming. Refer to the Users' Groups in Appendix B, as they are the best information source for synthetic programmers and M-Coders.

The next M-Code routine can be used to generate tones in the same way the TONE function does. This function is similar to a TONE IND M instruction. However, the internal HP-written XTONE subroutine (hex address 16DE) can generate 128 previously unavailable tones, over and above those that you can generate using synthetic TONE instructions. The TONEM function below uses nybbles 1 and 0 of status register M for the hex value of the tone, which can range from 00 to FF (0 to 255).

```
     08D  M
     005  E
     00E  N              TONEM name
     00F  O
     014  T
***  178  C=REG 5/M
     358  ST=C
     379  *              XTONE subroutine does almost
     05A  NCGO 16DE      all of the real work
```

The first instruction executed (C=REG 5/M) recalls status register M. The rightmost byte (nybbles 1 and 0) are loaded in the status bits (flags 0 to 7) by ST=C. (These microprocessor status flags have no relationship with the user flags in status register d.) The last two instructions branch to the XTONE subroutine, where the contents of the status bits, or flags, are used to control switching the bender (the beeper disk) on and off. That is the whole program.

The following short program demonstrates a simple use of the M-Code functions DECRM and TONEM. It sounds a tone while VIEWing a decimal number in X corresponding to the tone. The program requires 26 bytes of user code.

```
01 LBL "TON"            06 TONEM
02 "▨" (F1, FF)         07 DECRM
03 255                  08 DSE X
04 LBL 10               09 GTO 10
05 VIEW X               10 END
```

Lines 02 and 03 put an FF byte in Alpha, and the corresponding decimal number in X. Line 05 shows the tone number while line 06 sounds the tone. Lines 08 and 09 cause the loop to be repeated, sounding tones 255, 254, 253 and so on down to 1. The program will stop with zero in X and Alpha clear (tone zero will not sound).

This example may seem trivial, or perhaps not very useful. Would you like a programmable PACK function? Or how about functions to convert a decimal address in X (0 to 4095) to and from the three rightmost nybbles of status register M (again for use with NRCLM/NSTOM)? M-Code makes it simple to program these functions.

```
      08B  K            REMARKS
      003  C
      001  A            PPACK name
      010  P
      010  P
***   1A0  ABC=0        Serves to clear carry flag only
      001  *            PACKN
      080  NCXQ 2000    Packs program memory
      351  *            NWGOOS: entry point for CLD
      01E  NCGO 07D4    Clears PACKING from the display
```

That's the entire Programmable PACK function. The same entry point is used for the non-programmable function, so all of the normal housekeeping (such as resuming in the correct location after packing is complete) is taken care of. NWGOOS clears the PACKING message from the display and exits through the Normal Function Return (NFRPU -- see VASM page 7, or "HP-41 MCode for Beginners", page 61). The NCGO instruction branches to the operating system ROMs and does not return here. PPACK clears all pending subroutine returns, so you should not use it in a subroutine.

The next M-Code routine, XTO3N, converts a number in X, ranging from 0 to 4095 (fractional part ignored) to three hexadecimal nybbles, appending these nybbles to Alpha. Operating system subroutines are called only when error conditions are detected. Except for these error traps, only relative jumps are used.

```
      08E  N
      033  3
      00F  O
      014  T                 XTO3N name
      018  X
***   1A0  ABC=0             Clear accumulators.
      0F8  C=REG 3/X         Recall X.
      0A6  A<>C    X         Separate exponent.
      2FE  ?C≠0   S
      03B  JNC +07h          Skip error traps if exponent sign is 0.
      27E  C=C-1   S
      27E  C=C-1   S
      389  * (ERRAD)         ALPHA DATA error for sign = 1.
      053  CGO 14E2
      0B5  * (ERRDE)         DATA ERROR is generated for other
      0A2  NCGO 282D         nonzero sign values.
      166  A=A+1  X          Add 1 to exponent value in A.
      2FC  RCR 13            Jump back here.
      1A6  A=A-1   X         Shift left until the exponent is zero.
      346  ?A#0    X         This puts the Least Significant Digit
      3EF  JC  -03           in nybble 13.
      05A  C=0     M         Clear any fractional remainders.
      23C  RCR  2            Rotate MSD to nybble 0.
      0DC  PT=10             Use the pointer as a counter.
      1EA  C=C+C  WPT        Loop returns here three times.
```

```
10E  A=C     ALL   Multiply just the rightmost digits by
1EA  C=C+C   WPT   ten.  Hex mode is correctly assumed.
1EA  C=C+C   WPT   The pointer also separates the active
14A  A=A+C   WPT   digits from those to remain unchanged.
04A  C=0     WPT   Clear the old digit.
2FC  RCR 13        Bring in the new digit.
20A  C=A+C   WPT   Add it to the total.
3D4  -PT           Decrement the pointer.
294  ?PT=7         Has the loop been repeated three times?
3B3  JNC -0A       If not yet, jump back to the first C=C+C.
2FA  ?C=0    M     Is the result more than 3 nybbles?
289  * (ERROF)     (Is X greater than 4095?)  Then branch to
003  CGO 00A2      OUT OF RANGE error generator.
0EE  B<>C    ALL   Save the result in B.
238  C=REG 8/P     Shift ALPHA left 3 nybbles to end of prgm.
05A  C=0     M
05E  C=0     S     Clear all but the 3 rightmost nybbles of P.
1BC  RCR 11        Rotate left 3 places.
10E  A=C     ALL   Save in A.
1F8  C=REG 7/O     Recall O.
1BC  RCR 11        Rotate left 3 places.
106  A=C     X     Shift 3 nybbles from O to P
228  REG=C 8/P     Store shifted nybbles in P.
1B8  C=REG 6/N     Recall N.
1BC  RCR 11
106  A=C     X     Extract 3 nybbles after rotating, and
0AE  A<>C    ALL   store in O.
1E8  REG=C 7/O
178  C=REG 5/M     Recall M.
1BC  RCR 11        Shift left 3 nybbles
106  A=C     X     Put the 3 rightmost nybbles in the
0AE  A<>C    ALL   copy of N and swap it to C.
1A8  REG=C 6/N     Store the result in N.
0AE  A<>C    ALL   Swap the shifted M copy back to C.
0C6  C=B     X     Put the conversion result in the three
```

```
168  REG=C 5/M     rightmost nybbles and store in M.
3E0  RTN           DONE.  62 ROM words total.
```

Like XTOA, this XTO3N function will ignore any fractional component within the number in X. However,there are also differences worth noting. Negative numbers will generate a DATA ERROR message. Having flag 25 set will suppress any of the error messages, clear flag 25 when any error is detected, and the function will not have any effect.

A short program shown below uses XTO3N to create a program pointer. This "DP" (Decimal to Pointer) program works like the PPC ROM's "DP" routine, only much faster.

```
01 LBL "DP"
02 CLA
03 RCL X
04 7
05 MOD
06 XTO3N          The byte number (0 to 6) is converted here to a
07 RDN            single nybble in Alpha.
08 LASTX
09 /
10 XTO3N          The 3 nybbles representing the absolute register
11 CLX            address are added here.
12 X<> M
13 END            24 bytes.
```

The next M-code function, 3NTOX, converts the three rightmost nybbles in Alpha to a floating point (decimal) number in X. The decoded nybbles are removed from Alpha. The result can range from 0 to 4095.

```
098  X             REMARKS
00F  O
014  T
00E  N             3NTOX name.
```

```
    033 3
*** 1A0 ABC=0            First executable instruction.
    178 C=REG 5/M        Recall M.
    05A C=0     M        Clear the mantissa.
    2E6 ?C#0    X
    0C3 JNC +18          If M.X is zero, skip ahead
    2DC PT=13
    110 LC 4             Prepare for conversion of 4 digits
    10E A=C     ALL      (3 nybbles may yield up to 4 digits)
    130 LDI
    010 CON 16           To prevent output going to display,
    270 RAMSLCT          Select a nonexistent register.
    3A1 * (GENNUM)       Convert hex to decimal. Output is in
    014 NCXQ 05E8        A.M, number of digits in B.S.
    04E C=0     ALL
    270 RAMSLCT          Enable Chip 0.
    0DE C=B     S        Recall the number of digits and
    2FC RCR 13           rotate to the exponent field.
    0AE A<>C    ALL      Swap this with the result in A.
    2FC RCR 13
    25C PT=9
    04A C=0     WPT      Clear all but the number itself.
    33C RCR 1
    1A6 A=A-1   X        Jump back here.  Decrease exponent and
    2FC RCR 13           rotate the number until the Most
    2FE ?C#0    S        Significant Digit is in C.S.
    3EB JNC -03
    33C RCR 1            Rotate once so the sign is zero (pos.)
    14E A=A+C  ALL       Combine exponent and number for result.
    18C ?FS     11       Is the stack lift disabled?
    03B JNC +07          If so, skip raising the stack.
    078 C=REG 1/Z
    028 REG=C 0/T
    0B8 C=REG 2/Y
    068 REG=C 1/Z        Raise the stack.
```

```
0F8  C=REG 3/X
0A8  REG=C 2/Y
0AE  A<>C   ALL
0E8  REG=C 3/X   Store the result in X.
1B8  C=REG 6/N   Here to end of routine shifts Alpha
10E  A=C    ALL  right three nybbles.
178  C=REG 5/M
046  C=0    X    Clear three rightmost nybbles.
0A6  A<>C   X    Swap in the rightmost nybbles of N.
03C  RCR 3       Rotate.
168  REG=C 5/M   Store new M.
1F8  C=REG 7/O
0AE  A<>C   ALL  Put O contents in A, bring N to C.
0A6  A<>C   X    Swap the rightmost nybbles.
03C  RCR 3       Rotate.
1A8  REG=C 6/N   Store new N.
238  C=REG 8/P
17C  RCR 6
29C  PT=7
04A  C=0    WPT  Clear the 7 leftmost nybbles of P.
13C  RCR 8       Rotate them back.
0AE  A<>C   ALL
0A6  A<>C   X    Shift the 3 rightmost nybbles of P
03C  RCR 3       into O, rotate and store it.
1E8  REG=C 7/O
0AE  A<>C   ALL  Recall modified P contents from A.
046  C=0    X    Clear the rightmost nybbles, rotate
03C  RCR 3       and store the result.
228  REG=C 8/P
3E0  RTN         DONE.  69 ROM words total.
```

The demonstration program listed below is similar to the PPC ROM's "PD" (Pointer to Decimal) routine, but does not alter Alpha.

```
01 LBL "PD"
02 X<> M
03 3NTOX
04 3NTOX
05 "├-***"      (append any three characters)
06 X<> Z
07 X<> M
08 CLX
09 RDN
10 7
11 *
12 X<>Y
13 8
14 MOD
15 +
16 END             32 bytes
```

Here is a program equivalent to the PPC ROM's "E?" (.END. finder) program that preserves the contents of the Alpha register.

```
01 LBL "E?"
02 RCL c
03 X<> M
04 3NTOX
05 XTO3N
06 X<>Y
07 X<> M
08 CLX
09 RDN
10 END             22 bytes
```

The final illustration for 3NTOX is an equivalent of the PPC ROM's "C?" (curtain finder) program. It also leaves Alpha unchanged.

```
01 LBL "C?"
02 RCL c
03 X<> M
04 3NTOX
05 RDN
06 3NTOX
07 X<>Y
08 "⊢***"      (append any three characters)
09 X<> M
10 CLX
11 RDN
12 END                28 bytes
```

Now for another two M-Code functions. First, the XTON function converts a decimal value in X to a single nybble and appends it to the right of Alpha. This shifts Alpha left one nybble. X may range from zero to fifteen, and values outside this are error-trapped in the usual fashion with respect to the flags (24 and 25).

```
      08E  N
      00F  O
      014  T
      018  X
***   0F8  C=REG 3/X    Recall decimal value from X.
      38D  * (BCDBIN)   Floating point number to binary
      008  NCXQ 02E3    conversion subroutine.
      10E  A=C    ALL   Save the result in A.
      33C  RCR    1     Rotate right one nybble,
      2E6  ?C#0   X     if the result is more than one nybble,
      0B5  * (ERRDE)    branch to DATA ERROR.
      0A3  CGO 282D
      39C  PT=    0     Set the pointer for the rightmost
```

```
               nybble.
178  C=REG 5/M  Recall M to merge the new nybble into
               the Alpha register.
2FC  RCR    13  Rotate left.
0A2  A<>C   PT  Swap in the new nybble while saving old
               leftmost nybble in A for later use.
2F0  WDATA      Store the new M.
1B8  C=REG 6/N  Recall N, rotate it left one place,
2FC  RCR    13  swap in the nybble that was once in the
0A2  A<>C   PT  leftmost position of M and store the
2F0  WDATA      result.
1F8  C=REG 7/O  Recall O and go through the same steps
2FC  RCR    13  as outlined for M and N.
0A2  A<>C   PT
2F0  WDATA
238  C=REG 8/P  Recall P.
2FC  RCR    13  Rotate left one nybble.
0A2  A<>C   PT  Bring to C the nybble that previously
               occupied the leftmost position in O.
2F0  WDATA      Store the new value of P.
3E0  RTN        DONE.  30 total words.
```

The following NTOX function converts the rightmost nybble of Alpha to a decimal number. The contents of Alpha are shifted right one nybble. The normal stack lift scheme is observed, and the output in X will range from zero to fifteen. NTOX does not use any operating system subroutines.

```
     098  X
     00F  O
     014  T            NTOX function name.
     00E  N
***  18C  ?FS     11   Is the stack lift disabled?
     03B  JNC     +O7  If so, don't raise the stack.
     078  C=REG 1/Z
```

```
028  REG=C O/T
0B8  C=REG 2/Y
068  REG=C 1/Z
0F8  C=REG 3/X
0A8  REG=C  2/Y   Complete stack raise.
1A0  ABC=0
238  C=REG 8/P    Recall P.  To prevent the left part of
15C  PT=    6     P from shifting right,
042  C=0    PT    clear the scratch area of P.
39C  PT=    0     Set the pointer to the rightmost nybble.
0A2  A<>C   PT    Save the rightmost nybble in A.
33C  RCR    1     Rotate right one nybble.
2F0  WDATA        Store modified P.
1F8  C=REG 7/O    Recall O.
0A2  A<>C   PT    Swap the rightmost nybble with the one
33C  RCR    1     saved from P and shift to leftmost
2F0  WDATA        position and store new O.
1B8  C=REG 6/N    Recall N.
0A2  A<>C   PT    Perform a swap and shift as before.
33C  RCR    1
2F0  WDATA        Store new N.
178  C=REG 5/M    Recall M.
0A2  A<>C   PT    Save the rightmost nybble in A while
33C  RCR    1     swapping it with the one from N,
2F0  WDATA        rotate and store.
2A0  SETDEC       Decimal mode for math to follow.
0AE  A<>C   ALL   Recall the nybble.
22E  C=C+1  ALL   Add and subtract one to effectively
26E  C=C-1  ALL   convert a single nybble to decimal.
33C  RCR    1
2E2  ?C#0   PT    Is the resulting decimal number one
023  JNC    +04   digit?  If so, skip.
21C  PT=    2
050  LC     1     Load 1 for the exponent digit.
33C  RCR    1
```

```
33C  RCR    1      Skip to here if exponent was zero.
0E8  REG=C 3/X     Store result in X.
3E0  RTN           Done.  45 total words.
```

These programs demonstrate the power and speed of M-Code quite well. The functions you can write with M-Code are only limited by your imagination and your knowledge of the HP-41 operating system. M-Code applications are being developed rapidly as more (and better) hardware is now available.

One thing which you may not immediately realize from a casual reading of the ZENROM manual is that owning a Quasi-ROM device will allow you to make copies of other ROMs (modules). Whether you have the MLDL or the 16K RAM Storage Unit, the copy (CPY on the overlay) command of the MCED mode will enable you to make a copy of all or part of any ROM. The advantages of doing this may not be obvious, but the examples which follow will explain.

I had a revision 1B Extended Functions module that I wanted to send back to Hewlett-Packard to exchange for an upgraded version (free of the PURFL and PCLPS bugs). But I just couldn't leave myself without the extended Alpha capabilities it gave me. After I got an MLDL, I made a copy of my Extended Functions in one of the two 4K pages of the MLDL and sent the module off to HP. This allowed me to continue to use ATOX, XTOA, and other Alpha functions. I couldn't use the functions involving Extended Memory files (like SAVEP), but this wasn't much of a hindrance for my programs. In less than two weeks, I received the revision 1C Extended Functions module free of charge. If you still have one of the 1B Extended Functions modules, you should check with HP to see if they will do the same for you.

The MLDL unit is quite flexible. In addition to 8K of Q-ROM, it provides 24K of EPROM (Eraseable Programmable Read Only Memory). In my machine I have used the second page of RAM to hold a copy of the ZENROM. This freed the port formarly occupied by the ZENROM module, making it easy to plug in the HP-IL module without having to sacrifice another peripheral.

Now for a brief discussion about copying. As the lawful owner of a certain piece of software, you have the right to make backup copies for your own personal use, provided that any copyright notice is retained. However, neither the author nor the publisher of this book assume any responsibility or legal liability for this practice. You should be warned that although recent U.S. court decisions have upheld the right to make backup copies, this protection does not extend to "pirates" who sell or give away copies of computer instructions to persons that have not purchased it from the original source. The practice of distributing illegal copies of software is both unethical and unfair to the authors.

The use of the copy command has a few restrictions you need to know about. If you copy any of the internal ROMs (0, 1 or 2), you will only succeed in locking up the HP-41 until the copy is cleared or switched off. Making a copy of the HP-IL or Printer modules is not very useful because the hardware needed for the instructions to do their jobs will be missing. Copies of the Extended Functions or Time modules will only allow some of the functions to be performed, again because of the missing hardware. For example, the Time module copy must be addressed to page 5, and it can only be used for the ADATE, ATIME, DATE+, DDAYS and DOW functions.

Another possible reason to make a copy of a ROM is to eliminate a conflict with another module. Only one module with a given ID number (XROM number) can be used by the HP-41 at a time. If you ignore this restriction, anything from a lockup to MEMORY LOST is possible.

So, to continue the previous discussion, let's assume that you want to use both the ZENROM and the Standard (STRD 1C) module. Both have XROM number 05. Making a copy of the Standard module and changing the first word (the ID number) would be easy. However, since this ROM is in user-code, all of the XROM 05,xx subroutine calls within the module as subroutines would have to be changed also. This would be tedious. It might be worthwhile to make these changes if you find the upper 4K page addressing feature of the ZENROM valuable.

Assuming that this is not the case, let's make a copy of the ZENROM and change the ID number to decimal sixteen. If the ZENROM is in port 2, it occupies page B. To make a copy of this module in a 4K bank of Quasi-ROM addressed to page E, begin by executing MCED. Press CPY (the * key), B 0 0 0 B F F F, R/S, E 0 0 0, R/S, key in 0 1 0 (sixteen in hex) and R/S. Backarrow and press ON. That's all there is to it. You can now remove the ZENROM after turning the HP-41 off. As long as your Quasi-ROM device is plugged in and set properly, the ZENROM operating modes and functions will be there for you to use.

The program which follows is designed to simplify the job of transferring blocks of ROM words to and from mass storage. It does much of the "footwork" normally associated with the SVE and GET commands of the ZENROM's MCED operating mode. The data registers are used to hold the specially formatted ROM data.

The ZENROM, Extended Functions (or HP-41 CX), HP-IL module (with Disc or Cassette Drive attached), and Extended I/O are all required. Of course, you also need a Quasi-ROM device unless you just want to make a copy of a particular section of the ROM within a module.

Before you use this program, you should refer back to pages 118 and 119 of the ZENROM manual to be sure you are familiar with using the SVE and GET commands of MCED. The SVE command has the potential of overwriting global instructions in main memory when supplied with improper inputs. This can lead to MEMORY LOST.

```
01 LBL "ROM↑MS"
02 "ROM WORDS?"
03 PROMPT
04 5
05 /
06 .8
07 +
08 INT
09 SF 20
10 XEQ 09
11 XEQ 10
12 AVIEW
13 SF 25
14 CREATE
15 FC?C 25
16 ZERO
17 WRTR
18 GTO 00
19 LBL 10
20 "FILENAME?"
21 AON
22 STOP
23 AOFF
24 RTN
25 LBL "MS↑ROM"
26 XEQ 10
27 AVIEW
28 FLLENG
29 ASTO L
30 ASHF
31 ASTO a
32 CF 20
33 LBL 09
34 CF 25
35 PSIZE
36 CLRG
37 RCL c
38 DECODE
39 RDN
40 8
41 AROT
42 "⊦***"
43 CLX
44 STO N
45 ASTO X
46 VIEW X
47 TONE 85
48 PSE
49 FS? 20
50 GTO 08
51 CLA
52 ARCL L
53 ARCL a
54 AVIEW
55 READR
56 LBL 08
57 "XEQ MCED "
58 RDN
59 FS? 20
60 "⊦SVE"
61 FC?C 20
62 "⊦GET"
63 PROMPT
64 LBL 00
65 END
        159 bytes
```

Line 14 = hex A7, 01. Line 16 = hex A7, 18.
Line 17 = hex A7, 15. Line 28 = hex A5, C3.
Line 55 = hex A7, 08.

Lines 02 and 03 of "ROM↑MS" (ROM to Mass Storage) prompt you for the number of ROM words you want to save in mass storage. Lines 04 to 08 compute the number of registers required to hold the ROM words. Note that you can specify more ROM words than you need to give yourself some overhead. Line 09 sets flag 20 before line 10 executes label 09. This will be used later to skip certain instructions.

Lines 34 to 36 set the number of data registers to the correct size and clears them. PACKING, TRY AGAIN will be shown if there are insufficient free registers to set the size higher. The address of the first data register (curtain) is decoded and isolated from register c by lines 37 to 44. Lines 45 to 48 store these three nybbles of the hexadecimal address in X as characters, and displays them while continuing to run the program. Since flag 20 was set earlier, lines 49 and 50 skip to label 08. All of the lines in label 08 except 62 will be executed. Line 63 will prompt with the message "XEQ MCED SVE" shown. This is instructing you to execute MCED and press SVE (the STO key).

When you do that, the display shows two 4-digit prompts separated by a comma. The first four hexadecimal digits are the starting address. The other four are the ending address. After you press R/S to continue, three prompts are shown. You need to enter the address that was displayed during the tone at line 47 here. The three digits of this address define the lowest register where data will be stored. If you aren't sure what the address was, use backarrow followed by ON to exit MCED mode. Then restart the program again.

After entering the correct start address, press R/S. The ROM data will be quickly formatted and stored. Press ON to exit MCED when "COMMAND?" is shown. Then use R/S to continue running the program. Label 10 will prompt you for the name of the file in Mass Storage. This can either be a new file, or an old one of equal or greater capacity. Lines 13 to 17 finish this routine, creating a new file if one by that name does not exist, clearing it if there is one already, and storing the contents of the data registers in the file. Line 18 stops execution at the last line of the program to prevent mishaps.

Line 26 of "MS↑ROM" (Mass Storage to ROM) calls subroutine 10 to input the filename. Line 27 shows this name while line 28 fetches the length of the file from the directory. Lines 29 to 31 save the filename in lastx and status register a for later use. Flag 20 is cleared by line 32. In most cases, flag 20 will be clear anyway. Line 34 clears the error ignore flag to be certain that line 35 will trap any errors generated by an inability to re-SIZE. Line 36 clears the data registers to ensure no old data remains. Lines 37 to 48 isolate the address of the first data register and display it while sounding a tone. Line 49 skips to line 51.

Lines 51 to 54 recall the filename and display it while line 55 recalls the data registers from mass storage. Only line 60 will be skipped within label 08. The resulting message, "XEQ MCED GET" tells you to execute MCED and press the GET command (RCL key). Before you do this, if you don't remember the number of registers in the file, press backarrow. The decimal number is in X. Then execute MCED and press GET. Fill in the first hexadecimal prompt with the three digits of the first data register address (curtain). Then press the decimal point and key in the number of registers. If the file contains unfilled registers, you can use less than the full count, especially if the Quasi-ROM device has data within that page beyond the area you are working with.

These programs can store/retrieve up to 1595 ROM words (319 data registers, with 5 words/register). This would require that you have no key assignments, and the skills needed to run the program in the Extended Memory of the Extended Functions module/HP-41CX. With no key assignments and only this program in main memory, up to 1485 ROM words can be used. But having MCED assigned to a key is handy.

If you don't have the Extended I/O, you can delete lines 27 and 28. Insert "RGS?", PROMPT between 25 LBL "MS/ROM" and 26 XEQ 10. Input the number of data registers needed by the file when the "RGS?" prompt is shown.

This program should be most useful to those who find that saving and recalling ROM words can only be accomplished with the ZENROM manual in hand. It should make the job easy, with a little practice.

CHAPTER SIX -- MORE UTILITY PROGRAMS

6A. ZENROM VERSIONS OF SPME PROGRAMS

The four programs that are presented in this section are functionally equivalent to those found in "HP-41 Synthetic Programming Made Easy" (SPME) by Keith Jarett. The ZENROM is required by all but the byte counting program. Extended Functions (or an HP-41CX) are used by all four programs. Each program has a smaller byte count than the original version in SPME. If you have a CCD Module, you have built-in equivalents to each of the programs in this section.

The "MKZ" program on the next page is equivalent to the "MKX" program on page 92 of SPME. If you have a CCD Module, you should use ASN ENTER↑, followed by two decimal byte values. This does the same job as "MKX" or "MKZ".

To use this program, first load the stack with the prefix, postfix, and keycode as is usual for key assignment programs. As an example, to assign TONE 89 to the LOG key, key in 159, ENTER↑, 89, ENTER↑, 14. Then run the program using XEQ "MKZ". When the program is finished, you can make another assignment by entering the correct decimal values and pressing R/S.

This program will not work properly if you single-step or stop the program between lines 03 and 04. Also, if an Alpha label exists within Catalog 1 that matches the name in line 02, the program will search unsuccessfully from the bottom of the key assignment registers upward. Eventually, the program will halt with a DATA ERROR at line 28 with 256 in X (unless it happens to find three bytes that match those of the search string, which is extremely unlikely).

Make Key Assignment Using ZENROM and Extended Functions

```
01 LBL "MKZ"
02 "AROT"
03 PASN
04 RCL R
05 DECODE
06 ASHF
07 ASHF
08 ASTO X
09 "A646"
10 ARCL X
11 ASTO L
12 CLA
13 R↑
14 XTOA
15 R↑
16 XTOA
17 X<> M
18  DECODE
19 ARCL Z
20 191
21 RCL M
22 STO a
23 LBL 01
24 X<>Y
25 ISG X
26 LBL 01
27 CLA
28 XTOA
29 NRCLM
30 DECODE
31 RDN
32 LASTX
33 POSA
34 X<0?
35 GTO 01
36 AROT
37 ASHF
38 CHS
39 AROT
40 X<> a
41 STO O
42 ATOX
43 6
44 AROT
45 R↑
46 CODE
47 X<>Y
48 CLA
49 XTOA
50 X<>Y
51 NSTOM
52 CLST
53 END
          99 bytes
```

Line 02 must match a function or program in Catalog 2 or 3 that does not have an identical label in Catalog 1. Line 03 will execute much faster if you use the name of a function that appears near the beginning of Catalog 2. For example, if you have a Time module or CX, change line 02 to "SW", and change line 10 to "A69A", which is the hex equivalent of the SW function.

Here is the line-by-line analysis of the "MKZ" program.

Lines 02 and 03 assign the AROT function to the key designated by the keycode in X. This sets the appropriate bit in the keycode map of status register R (append) or e. This dummy assignment will later be modified according to the decimal values in Z (prefix) and Y (postfix). Of course, the use of PASN assures that there will not be any wasted (half used) key assignment registers.

Line 04 recalls register R, which contains the hexadecimal assigned keycode in byte 0. Lines 05 through 07 decode the contents of R and discard all but the last two nybbles from Alpha. Lines 08 through 11 perform two functions. They append the keycode extracted from R to "A646" and store the 6-character result in L. This result will be used later in the Label 01 loop to find a matching assignment within the key assignment registers. In addition to this result, line 08 leaves the keycode in the stack for later use in assembling the new key assignment.

Lines 12 to 16 assemble the prefix and postfix bytes in Alpha using decimal values originally in Z and Y, respectively. Line 17 brings the two bytes from Alpha into the stack. Line 18 converts these bytes to hex, overwriting the contents of Alpha. Line 19 appends the two hex characters of the keycode (from line 08) to the decoded prefix and postfix in Alpha. This forms a 6-character hex equivalent of the new key assignment.

Line 20 provides the starting register address needed for the main LBL 01 loop. Since this address is incremented before anything else is done, the starting decimal value is one less than the first register recalled. The next two lines, 21 and 22, store the 6-character hex code for the new assignment (with a leading 0) in status register a for later use. An indirect effect of these two lines is to push the 191 starting address into register Y, which is where it needs to be in at the top of the LBL 01 loop.

The LBL 01 loop, line 23 to line 35, searches the assignment registers for the dummy AROT (A646xx) assignment. Line 24 brings the decimal address into X, where line 25 increases it by one. Line 26 is a NOP. The GTO 01 at line 35 does not branch here because it finds LBL 01 at line 23 first. Lines 27 and 28 convert the decimal address into a single byte in Alpha, where it is used by line 29 to recall that register. Line 31 makes room in the stack for line 32 to bring the target string (A646xx) into X from L. Line 33 checks for the string within the decoded register, overwriting X with -1 if it is not found. Lines 34 and 35 repeat the loop until the target assignment is found.

When the string being searched for is located, the position code in X will be either 2 or 8, depending on whether the target assignment occupied the left or right half of the assignment register. (Because neither its hex A6 prefix nor its hex 46 suffix is a valid keycode, the choice of AROT as the dummy assignment eliminates the remote possibility that a match could be found straddling the two halves of the assignment register.)

Line 36 rotates Alpha such that the target string occupies the leftmost six bytes. Line 37 removes these six characters from Alpha, leaving 8 characters. If X is 8, lines 38 and 39 have no effect. However if X is 2, lines 38 and 39 will rotate the header byte (usually F0) back into its normal position. Since "MKZ" does not test the value of the header byte, this program is compatible with the F6 key assignment header bytes that the "PK" (Pack Key assignments) program in Section 5B uses.

Line 40 recalls the 6 hex characters for the new assignment from status register a. Line 41 stores them in Alpha, to the left of the other eight characters. Line 42 removes the extra zero character. Lines 43 and 44 rotate the assignment characters into position, making a continuous block of fourteen characters. Line 45 brings the decimal address from T into X, so it does not get pushed off the top of the stack. Line 46 converts the characters in Alpha back into a Non-Normalized Number in X. Lines 47 to 50 manage the stack while converting the decimal address in Y to a byte in Alpha. Line 51 stores the coded bytes (NNN) in the correct register. The stack is cleared by line 52.

When the program ends, Alpha still contains the register where the new key assignment was stored. The assignment will always be in the three rightmost bytes of the register, so if you execute RAMED after "MKZ" finishes, you will immediately see the hex keycode in the middle window, and the postfix on the left. LASTX will still contain the target string.

RAM / ROM Byte Counting Programs

```
 01 LBL "RAMBC"        27 LBL 03         "RAMBC"
 02 LBL 02             28 "*"
 03 X<>Y               29 X<> M          "RAMBYT"
 04 XEQ 01             30 STO N
 05 X<>Y               31 RDN            "ROMBC"
 06 XEQ 01             32 ENTER↑
 07 -                  33 ASHF           "ROMBYT"
 08 RTN                34 ALENG
 09 LBL "RAMBYT"       35  E             106 bytes
 10 LBL 01             36 -
 11 XEQ 03             37 X≠0?
 12 7                  38 ATOX
 13 *                  39 ABS
 14 +                  40 ATOX
 15 INT                41 R↑
 16 RCL M              42 STO M
 17 X<>Y               43 RDN
 18 CLA                44 LASTX
 19 RTN                45 16
 20 LBL "ROMBC"        46 ST/ T
 21 XEQ 02             47 MOD
 22 7                  48 LASTX
 23 CHS                49 X↑2
 24 /                  50 *
 25 RTN                51 +
26 LBL "ROMBYT"        52 END
```

The programs listed on the preceding page on are based on the byte counting programs presented on page 86 of "HP-41 Synthetic Programming Made Easy". Extended Functions or an HP-41 CX are required. You just RCL b at two spots in program memory, then execute "RAMBC" to find the distance in bytes between those two spots. If you RCL b at the first and last lines of a program, you will have to add 3 bytes to get the total program length (this accounts for the END). "ROMBC" is the equivalent of "RAMBC" for dealing with ROM addresses.

If you have a CCD Module, you can use its PC>X and A-A functions in place of "RAMBC" as follows. You might want to assign PC>X to a key. Then just execute PC>X at each of the two spots in program memory. Execute A-A to compute the number of bytes between the two addresses. Unfortunately this method does not allow byte counting in ROM.

Here is the line-by-line analysis for the "RAMC"/"ROMBC" programs.

Line 02 of "RAMBC" gives "ROMBC" an entry point (see line 21), allowing "ROMBC" to execute this same series of instructions. Line 03 exchanges the two pointers, so that the one in Y will be decoded first. Line 04 calls label 01, which is the same as "RAMBYT", to decode the pointer into a decimal byte count from the bottom of memory. Lines 05 and 06 decode the second pointer. Line 07 subtracts the two results, stopping at line 08 with the difference in X.

The first thing "RAMBYT" (LBL 01) does is to branch to LBL 03. Label 03 is used as a subroutine by all four sections of this program. With the program pointer input in X containing four nybbles -- call them a, b, c and d -- the LBL 03 subroutine returns the following results:

To T, Y, and Z: Nybble a, plus a fractional remainder

To X: The decimal quantity ((b*256) + (16*c) + d)

Register M contains whatever was in Y when the LBL 03 subroutine was called (usually a program pointer).

Lines 12 and 13 multiply ((b*256) + (16*c) + d), which is the count of registers in a program pointer, by seven. This converts the register count to a byte count. Line 14 adds the decimal value of nybble a to this, giving the total number of bytes. Line 15 removes any fractional part remaining in the number. Lines 16 through 18 restore Y to its original position and clear Alpha.

Line 21 of "ROMBC" executes "RAMBC" in its entirety. The remaining lines, 22 to 24, divide the count by negative seven. Though RAM and ROM pointers differ in their format, this is all that is required to change a difference in bytes to correct for their nonuniformity. (This assumes that both ROM pointers came from the same page number.)

The first lines of the LBL 03 subroutine isolate the four nybbles (2 characters) of the program pointer from X. This is necessary since RCL b can recall pending returns along with the program pointer. Lines 28 to 30 and 33 remove any bytes to the left of the pointer. Lines 31 and 32 discard the pointer in X while making two copies of the value that was in Y. The ASHF at line 33 follows ENTER↑ solely to cancel the stack lift disable. If the leftmost nybbles of the pointer are zeros, line 34 will return an Alpha length of 1. Otherwise, the length will be 2. Lines 35 and 36 decrease this character count by 1. Lines 37 and 38 will then decode the left character into a decimal number as long as it was not a null. Line 39 saves the decimal equivalent of the left byte in L for later use at lines 44 and 47.

Line 40 converts the rightmost byte, nybbles c and d, to a decimal character code. Lines 41 to 43 save the value that was in Y in M, now that is has been emptied. Line 44 recovers a copy of the character code containing nybbles a and b. The quantity 16, line 45, is used three different ways. First, line 46 divides T (which contains the decimal value 16*a + b) by sixteen, isolating nybble a and usually leaving a fractional remainder. Second, it is used to separate nybble b and discard nybble a by taking the first decimal character code (16*a + b) modulo sixteen. Third, it is squared by line 49 after being recovered from L. Line 50 multiplies sixteen squared (256) by nybble b. Line 51 sums (256 * b) with (16*c + d), completing the decoding process.

"SA" / "RA" (Save / Recall Alarms)

```
01 LBL "SA"
02 191
03 SIGN
04 LBL 10
05 CLA
06 ISG L
07 BEEP
08 LASTX
09 XTOA
10 NRCLM
11 X<> M
12 ATOX
13 170
14 X≠Y?
15 GTO 10
16 LASTX
17 LASTX
18 ATOX
19 "ALM"
20 CRFLD
21 DSE X
22 +
23  E3
24 /
25 +
26 LBL 09
27 CLA
28 XTOA
29 NRCLM
30 SAVEX
31 CLX
32 NSTOM
33 RDN
34 ISG X
35 GTO 09
36 RTN
37 LBL "RA"
38 CLX
39 "ALM"
40 SEEKPTA
41 191
42 SIGN
43 LBL 08
44 ISG L
45 BEEP
46 LASTX
47 CLA
48 XTOA
49 CLST
50 NRCLM
51 X≠Y?
52 GTO 08
53 CLA
54 FLSIZE
55 LASTX
56 DSE Y
57 +
58 LASTX
59 X<>Y
60  E3
61 /
62 +
63 LBL 07
64 CLA
65 XTOA
66 GETX
67 NSTOM
68 RDN
69 ISG X
70 GTO 07
71 "ALM"
72 PURFL
73 CLST
74 CLA
75 END     132 byte
```

Important note for Extended Functions 1B (EXT FCN 1B) users: Insert an EMDIR instruction between PURFL (line 72) and the END.

The two programs listed on the preceding page, "SA" and "RA", are used to Save and Recall Alarms. This is useful in conjunction with two types of synthetic programs. First, some synthetic programs will cause MEMORY LOST if they are interrupted by an alarm. Second, some synthetic programs destroy buffers other than key assignments.

If you have a CCD Module, the SAVEB (save buffer) and GETB (get buffer) functions can be used to save and recall alarms. This application is described in the CCD Module Owner's Manual.

The "SA" program saves the alarms in a data file named "ALM" which it creates in Extended Memory. At the same time, "SA" clears the buffer registers containing the alarms. For this reason, buffers other than alarms should be cleared before using "SA". To do this, simply remove any modules which create buffers (after turning the HP-41 off) and cycle power on and off (press ON twice). You may then replace the modules. If you don't do this when buffers are present in memory above the alarms, they will be left to "float" between the key assignments and the .END.. They will take up free registers and the module that created them will not be able to use them.

"RA" recalls the alarm information stored in the data file and restores it to its original position above the key assignments. You may change your SIZE, add assignments, create buffers, and similar operations between executing "SA" and "RA". However, there must be sufficient free registers to restore the alarms. Compare the size of the data file "ALM" using EMDIR with the free register count seen after pressing SHIFT, RTN, PRGM while in run mode. If there are insufficient registers, "RA" will overwrite the .END., possibly causing MEMORY LOST. If this should occur, see Section 4F for recovery procedures.

Lines 02 and 03 of "SA" store 191 in L (LASTX), setting up the stack for LBL 10, which finds the first alarm buffer register. Line 06 increases the decimal register address in L. Lines 05, 08, and 09 convert the addresses from L into a character in Alpha. The BEEP is never executed. Line 10 recalls the designated register. Line 11 swaps the recalled NNN in X with the address in M. The first (leftmost) byte is converted to a decimal number by line 12. If this byte is hex AA, decimal 170, it will match line 13. When there isn't a match, lines 14 and 15 will repeat the LBL 10 sequence until an AA is found. If there are no alarms, you will get a DATA ERROR message at line 09 when the decimal address reaches 256.

Lines 16 and 17 bring two copies of the decimal address of the first alarm register into the stack. Line 18 decodes the second byte of the alarm register, which contains the number of registers in the buffer. Lines 19 and 20 create a data file "ALM" to hold the alarms. Lines 21 to 25 (line 22 is never skipped) combine the file size with the address to make an ISG register counter for the LBL 09 loop.

The LBL 09 portion of the program, lines 26 to 35, recalls the alarm data, saves it in Extended Memory, and stores zero over the alarms. Lines 27 and 28 convert the address in X to the format required by NRCLM (line 29). Line 30 saves the data in XM, and lines 31 and 32 overwrite the alarm register with nulls. Line 33 brings the counter back into X, where lines 34 and 35 increment the counter and repeat the loop until the counter exceeds its limit. When line 36 stops the program, all of the alarms have been saved and the registers have been cleared.

Lines 38 to 40 of "RA" reset the counter of the data file to zero. (This provision allows you to use other Extended Memory files between the execution of "SA" and "RA".) Lines 41 and 42 set up the stack for the LBL 08 loop, lines 43 to 52.

Lines 44 to 46 increment the register address counter and bring it into X. Line 45 is never executed. Lines 47 and 48 convert the address to a character in Alpha. Line 49 clears the entire stack except L, giving zero for testing at line 51. Line 50 recalls the register at the address in M. Lines 51 and 52 repeat the LBL 08 loop until an empty register is found.

Line 53 clears Alpha to make line 54 return the size of the working file (established at line 40). Line 55 recalls the address of the empty register. Line 56 decreases the value of the file size in Y. An instruction will never be skipped by line 56 (or line 21) because one-register alarm buffers are not possible. Lines 57 to 62 combine the address and filesize into an ISG control value for the loop which follows. This is like the one used for LBL 09 in "SA".

Label 07 restores the alarms in the buffer registers. Lines 64 and 65 convert the address into a character in Alpha. The data is retrieved from the file by line 66. Note that GETX does not normalize. Line 67 stores the register in X at the address in M. Lines 68 to 70 manage the stack, repeating the loop as needed. The "ALM" file is erased from Extended Memory by lines 72 and 73. (Add an EMDIR instruction here if you have revision 1B Extended Functions.) Lines 73 and 74 clear the stack and Alpha.

The next set of programs, "SK" and "RK", are used to Suspend and Reactivate Key Assignments. This can be helpful if you want to make temporary use of the execution of local labels A-J and a-e from the top two rows of keys in USER mode. "SK" will make sure that your key assingments do not interfere, until you use "RK" to reactivate them.

Versions of "SK" and "RK" appear in the PPC ROM and in "HP-41 Synthetic Programming Made Easy". To use "SK", you put a register number in X and execute "SK". The assigned key index, also called the assigned key bit map, is stored in two successive data registers. To restore the assigned key index and thus reactivate the key assignments, just put the same register number in X and execute "RK". A version of "SK"/"RK" for CCD Module users will also be presented in this section.

"SK/RK" Program

```
01 LBL "SK"              09 X<> e            17 ISG X
02 SIGN                  10 STO IND L        18 "" (hex F0 NOP)
03 CLX                   11 RDN              19 NRCLX
04 X<> R                 12 RTN              20 STO e
05 STO IND L             13 LBL "RK"         21 X<> L
06 ISG L                 14 NRCLX            22 RDN
07 "" (hex F0 NOP)       15 STO R            23 END
08 CLX                   16 X<> L                  47 bytes
```

Here is the line-by-line analysis for the "SK/RK" programs.

The first data register used by "SK" (Suspend Key Assignments) is specified by the number in X at the start of the program. Lines 02 and 03 store this value in L and clear X. Line 04 exchanges zero with register R (the Append register), effectively suspending all USER mode assignments to the unshifted keys. The bit map of assigned keys is saved in the register specified in L (which was in X). Lines 06 and 07 then increase this register number by one. The NNN remaining in X from register R is cleared by line 08. Line 09 swaps zero with the shifted key assignment bit map, while line 10 saves it in data register (x + 1). Line 11 rotates this NNN down to register T, putting the original Y, Z, and T in stack registers X, Y, and Z.

Line 14 of "RK" (Restore Key Assignments) recalls the register specified in X without normalizing. Be sure you don't merely use LASTX from "SK" as an input to "RK", because this value is actually x+1.

Line 15 of "RK" restores register R to its former value, using the contents of data register x. Line 16 recovers the data register number from L (put there by line 14). Lines 17 and 18 increment it. Line 19 recalls data register (x+1), and line 20 uses it to replace status register e. Lines 21 and 22 leave the stack with the contents originally in Y, Z, and T in X, Y, and Z.

An interesting side effect can be seen by single-stepping (SST) the entire program in run mode. Line 09 resets the program line counter within status register e to 000 as a by-product of clearing the key assignment bit map. After using SST at line 09, the next SST shows "01 STO IND L".

The operating system of the HP-41 does not recompute the line number with each SST. If it did, single-stepping a long program would be as slow as using back-step (BST). A running program sets the line number to hex FFF, indicating that the line number needs to be recomputed before it can be displayed again. Then, while a program is running, the HP-41 operating system saves time by not keeping track of the current line number.

Continue to SST the program until you get to LBL "RK". Then be sure to key in the correct value of X, indicating the lower register where the key assignment bit maps are stored. After you SST line 17, another interesting thing happens. Since an instruction was skipped, the line number gets recomputed and thus corrected.

Keep pressing SST until you see the instruction STO e. You know from the program listing that this is line 20. If you used SST throughout the program and line 18 (NOP) was not skipped, STO e will show as line 11! Otherwise, the line number will be right.

Now SST and hold it. The line displays as "10 X<> L"! As long as you SST line 09, the line after STO e will be shown as 10. The processor of the HP-41 always assumes that a line number is correct unless the three rightmost nybbles are FFF (hex). You can see similar behavior when using STO b to move around in memory.

Now clear X, press ENTER↑, and XEQ "SK". Zero will still be in X when "SK" finishes, because Y was cleared before executing "SK". Then press and hold R/S until NULL is shown. LBL "RK" is not line 01, but executing "SK" reset the line number to 000. Press R/S and let "RK" restore the key assignments.

You can use this program to figure out which operations cause recomputation of the line number, and which ones don't. A list of the latter would probably surprise you. It includes inserting instructions in program memory, turning the HP-41 off and on and running CATalogs other than 1. It is left as an exercise for the curious to compile this list.

If you have a CCD Module, your PEEKR function can be used in place of NRCLX in "RK", but you must account for the fact that NRCLX uses addresses relative to the curtain (that is, normal data register addresses), whereas PEEKR uses absolute addresses. There is a place for each kind of addressing, but here relative addressing is preferred. What is needed to bridge this gap is an equivalent to the PPC ROM's "C?" (curtain finder) routine.

Here is the full CCD Module equivalent program:

```
01 LBL "SK"              14 13.1         27 +
02 SIGN                  15 PEEKB        28 +
03 CLX                   16 X<>Y         29 PEEKR
04 X<>                   17 A+           30 STO
05 STO IND L             18 PEEKB        31 RDN
06 ISG L                 19 X<>Y         32 ISG X
07 "" (hex F0 NOP)       20 RDN          33 "" (hex F0 NOP)
08 CLX                   21 16           34 PEEKR
09 X<> e                 22 *            35 STO e
10 STO IND L             23 X<>Y         36 X<> L
11 RDN                   24 LASTX        37 RDN
12 RTN                   25 /            38 END
13 LBL "RK"              26 INT               68 bytes
```

Lines 14 through 27 provide the curtain-finding function for the CCD Module. You can use these steps as a separate routine if you like.

If you accidentally lose the contents of the key assignment bit map registers, there is an easy way to restore them. If you have Extended Functions or a HP-41CX, you can use GTO . . (GTO . . . for ZENROM users), CLA if it isn't empty, and XEQ "PCLPS". GETP and GETSUB also work well, but require some "cleanup" after reading in a program you don't want. Card Reader owners can read in a program card, interrupting after the first track is read. HP-IL (Cassette Drive or Disc Drive) mass storage users can employ READP.

All of these techniques cause the key assignment bit maps to be recomputed, using the contents of the key assignment registers and the assignment byte within each global label in Catalog 1. This approach is used in the "RK" program on page 197 of "HP-41 Extended Functions Made Easy". The "IN" program, on page 198 of XFME, creates a one-register data file and modifies it synthetically to be an empty program file. Another version of "IN" is given in the next section of this book.

6B. ZENROM VERSIONS OF XFME PROGRAMS

The five programs which follow are designed to replace programs from "HP-41 Extended Functions Made Easy", using ZENROM or CCD Module functions to save bytes. A ZENROM or CCD Module is needed for all but the last program, which can use either the PPC ROM or ZENROM. Extended Functions are mandatory for all except "VER" and "PRFL".

The "VER" program below is a fix for the verify function of the Card Reader, which alters the contents of the top register in Extended Memory module number 2 (when present). The absolute address of the affected register is decimal 1007, or 3EF in hexadecimal. Alteration of this register can disrupt a single extended memory file, or even destroy part of the extended memory directory.

VERIFY with Bug Fix

```
ZENROM version                 CCD Module version
01 LBL "VER"                   01 LBL "VER"
02 "▩▩" (F2, 03, EF)           02 1007
03 NRCLM                       03 PEEKR
04 "PRESS R/S"                 04 "PRESS R/S"
05 AON                         05 AON
06 VER  (A7, 85)               06 VER  (167, 133)
07 AOFF                        07 AOFF
08 "▩▩" (F2, 03, EF)           08 POKER
09 NSTOM                       09 R↑
10 RDN                         10 R↑
11 END                         11 END
      35 bytes                       34 bytes
```

Note: Because the Card Reader enforces the non-programmability of its VER function, line 06 must be entered either by a key assignment of VER with the Card Reader not present, by RAMED, or by byte grabbing.

Lines 02 and 03 of "VER" recall the contents of the affected register before line 06 has a chance to alter it. Lines 04 and 05 provide a reminder to restart the program after verifying the cards. Lines 08 and 09 restore the damaged register to its former value. The program then returns the original value to X before halting. The whole routine is only 35 bytes long (34 for the CCD Module version), illustrating once again the power of the ZENROM's NRCLM and NSTOM and the CCD Module's PEEK and POKE functions.

The next program, "PFF" (Purge File Fix), repairs the damage done by the PURFL function of the revision 1B Extended Functions module (-EXT FCN 1B). A bug within this function leaves no working file after PURFL. Unless a working file is established immediately (for example, by EMDIR), operations which use the working file (APPCHR, FLSIZE, GETX, RCLPT, etc.) will cause seven FF bytes to be stored in the first Extended Memory register, at location 0BF hex. This program replaces these FF bytes with the filename that you specify in the Alpha register, restoring your damaged Extended Memory directory.

```
                    "PFF" (Purge File Fix)
ZENROM version                  CCD Module version
01 LBL "PFF"                    01 LBL "PFF"
02 "⊦-      "                   02 "⊦-      "
03 7                            03 7
04 AROT                         04 AROT
05 X<> M                        05 RDN
06 "▓" (F1, BF hex)             06 191
07 NSTOM                        07 RCL M
08 STO M                        08 POKER
09 CLX                          09 RDN
10 EMDIR                        10 CLX
11 CLD                          11 EMDIR
12 END                          12 CLD
                                13 END
     34 bytes                        35 bytes
Line 02 = append six spaces (F7, 7F, 20, 20, 20, 20, 20, 20).
```

Line 02 appends six spaces to the filename in Alpha. Alpha must not be empty, or a null and six spaces will be used for the filename of the first file. If this happens to you, just put the correct filename in Alpha and execute "PFF" again.

Lines 03 and 04 of "PFF" rotate the filename and spaces so that the leftmost character of the filename is in byte 6 of status register M. Any leftover spaces will be in register N. In the ZENROM version, line 05 swaps the filename and spaces from M into X. Line 06 is hex address 0BF in character form. Line 07 stores the filename in register 0BF (decimal 191), restoring access to the extended memory directory. In the CCD Module version, lines 05 to 08 use POKER to accomplish the replacement of the filename in absolute location 191.

Lines 09 to the end clean up the stack, leaving the original X and Y in Y and Z. The ZENROM version also preserves the original Z in T. If the EMDIR at line 10 is allowed to run to completion, X will contain the amount of room left in X Memory. Otherwise it will contain zero.

The best way to avoid the PURFL bug in the revision 1B Extended Functions module is to use the "PRFL" program below in place of the PURFL function. This program prevents any damage to the contents of Extended Memory after PURFL by storing a Non-Normalized Number in hex address 040 (decimal 64) which specifies the first file in Extended Memory as the working file.

```
              "PRFL" (Purge File -- bug free)
ZENROM version                  CCD Module version
01 LBL "PRFL"                   01 LBL "PRFL"
02 PURFL                        02 PURFL
03 RCL M                        03 RCL M
04 "▩¯.▩▩"                      04 "▩¯.▩▩"
05 RCL M                        05 64
06 "@"  (F1, 40)                06 RCL M
07 NSTOM                        07 POKER
08 CLX                          08 CLX
09 RDN                          09 RDN
10 STO M                        10 CLX
11 RDN                          11 RDN
12 END                          12 STO M
                                13 RDN
                                14 END
      32 bytes                        34 bytes
```

Line 04 = F5, 10, 00, 2E, F0, BF in hex.

Line 02 purges the file from Extended Memory. Line 03 brings the filename into the stack to allow line 10 to restore it later. Line 04 is the NNN to be stored in hex address 040. See page 181 of "HP-41 of HP-41 Extended Functions Made Easy" or page 8 of the Synthetic Quick Reference Guide. This NNN contains 3-nybble fields specifying the top of this module (hex 0BF), the top of the next Extended Memory module (2EF), and the working file number (the first nybble, a 1). If you don't have any Extended Memory modules, the 2EF nybbles will have no effect, and will not be altered. Otherwise they prevent most of Extended Memory from disappearing. Line 05 to 07 bring this NNN into the stack and store it in address 040 (line 06).

Lines 08 and 09 clear this NNN and push it down into T. The CCD Module version has two more lines to clear the POKER address input. The last 3 lines restore the filename to Alpha, and push it into T. At the end of the program, X, is in its original state, Y and Z are clear, and T contains the filename as a NNN. The ZENROM version also preserves the former Y and L.

The next four programs are for ZENROM users. In conjunction with Extended Function CLKEYS, give you much more control of your key assignments. (CCD Module users already have this capability and a bit more, with the SAVEK, GETK, and MRGK functions.)

You can use the "SAVEK" program presented here to store multiple sets of key assignments as Extended Memory files, and recover them one at a time with "GETK". You can temporarily suspend key assignments with "SK". Both "GETK" and "RK" use a special file created in Extended Memory by the program "IN" (INitialize). This file is a one-register program file containing only nulls. Its only effect is to have the HP-41 recompute the contents of status registers e and R (append) when you use GETP to retrieve the file.

"SAVEK", "A?", "GETK", "RK", "SK" programs

```
01 LBL "SAVEK"
02 CLX
03 SF 25
04 SEEKPTA
05 XEQ 01
06 CRFLD
07 DSE L
08 LASTX
09 +
10 LASTX
11  E3
12 /
13 +
14 LBL 02
15 CLA
16 XTOA
17 NRCLM
18 SAVEX
19 RDN
20 DSE X
21 GTO 02
22 CLST
23 CLA
24 RTN
25 LBL "A?"
26 LBL 01
27 191
28 SIGN
29 ASTO a
30 LBL 03
31 ISG L
32    (F0 NOP)
33 LASTX
34 CLA
35 XTOA
36 NRCLM
37 DECODE
38 ATOX
39 70
40 X=Y?
41 GTO 03
42 LASTX
43 192
44 -
45 CLA
46 ARCL a
47 RTN
48 LBL "GETK"
49 CLX
50 SEEKPTA
51 FLSIZE
52 191
53 +
54 LASTX
55  E3
56 /
57 +
58 SIGN
59 CLKEYS
60 LBL 04
61 LASTX
62 CLA
63 XTOA
64 GETX
65 NSTOM
66 DSE L
67 GTO 04
68 CLST
69 LBL "RK"
70 "  " (F1, 20 hex)
71 GETP
72 RTN
73 LBL "SK"
74 .
75 STO R
76 STO e
77 RDN
78 END    150 bytes
```

The name of the file where the key assignments are to be stored by "SAVEK" should be in Alpha when the program is executed. The name must not exceed six characters. If Alpha is clear, and the working file is a data file, the assignments will be stored in (or for "GETK", recalled from) that file.

Lines 02 to 04 set the file pointer of the working file, or the file named in Alpha, to zero. If there is not a working file with Alpha clear, or a file matching the name in Alpha, flag 25 will be cleared. Line 05 uses the label 01 subroutine to count the key assignment registers. Up to six characters in Alpha are preserved by label 01 ("A?"). If line 06 is executed with flag 25 clear, it creates a data file big enough to hold the assignments. When flag 25 is set at this point, it is because SEEKPTA matched a file in Extended Memory, or Alpha was clear and a working file was already established. In either of these two cases the CRFLD at line 06 will fail, clearing flag 25.

Lines 07 to 13 combine the filesize with 191 from LASTX to make a control word. At label 02, X contains a number in the format bbb.eee, such as 198.191. Lines 14 to 21 form a loop which is executed for each key assignment register to be saved. Lines 15 and 16 convert the number in X to an address useable by NRCLM, line 17. Line 18 saves the recalled data in the file. Line 19 rotates the control word back into X, where line 20 decrements it and line 21 repeats label 02 if the limit is not reached. Lines 23 and 24 clear the stack and Alpha.

Line 26 of "A?" gives "SAVEK" an easy way to access it as a subroutine. LBL "A?" allows the subprogram to be used manually or by other programs. Lines 27 and 28 set up the stack for the label 03 loop. Line 29 preserves the first six Alpha characters. Lines 31 and 32 increment the decimal address in L. The address is recalled by line 33, and lines 34 and 35 convert it to a byte in Alpha. Line 36 recalls the register at this address. Line 37 decodes this register into fourteen characters in Alpha. Line 38 converts the leftmost nybble of the recalled register to a decimal character code in X. If this nybble is F, decimal 70, lines 39 to 41 repeat the loop. The loop is executed until a register is found which is not a key assignment register.

Line 42 recalls the decimal address of the first non key-assignment from LASTX. Lines 43 and 44 subtract 192 from it, changing this absolute address into the key assignment register count. Lines 45 and 46 restore the characters held in status register a to Alpha. At the termination of this subroutine, X contains the register count, and L contains 192.

The first two lines of "GETK" are similar to the beginning of "SAVEK". If there is a working file, lines 49 and 50 set the pointer to zero with no error, even though Alpha is clear. If you did something involving Extended Memory files between executing "SAVEK" and "GETK", the data file containing the key assignments needs to be named in Alpha to avoid an error at line 50. Otherwise, Alpha must be clear. Line 51 returns the size of the file, where lines 52 to 58 construct a control word as before, storing it in L. Line 59 clears the assignments prior to reading in new ones. If any buffers (alarms, etc.) are present, they will be overwritten fully or partially by the new assignments. Therefore you should not have any Time module alarms or other buffers present when you use "GETK".

Label 04 recalls the assignments from the file and stores them in the key assignment registers. Like label 02, it works from top (highest numbered register) to bottom. Line 61 recalls the decimal address from L, while lines 62 and 63 convert that address into a character specifying the address in Alpha. Line 64 recalls the register from the data file, and line 65 stores it in the proper key assignment register. Lines 66 and 67 repeat the loop until the limit is reached. After the key assignments have all been restored, line 68 clears the stack and execution drops into "RK".

"RK" restores the key assignment bit maps in status registers e and R (append). Any execution of GETP would accomplish this. However, the special " " (space) program, invented by Tapani Tarvainen and created by an "IN" program, recomputes the bit maps without replacing the last program in memory. Lines 70 and 71 do the whole job.

"SK" merely stores zero in status registers R (append) and e. This effectively suspends the USER mode key assignments. This capability is useful when executing programs which utilize the local labels A to E, F to J, and a through e. By suspending your assignments, you can avoid having to clear them, yet still be able to use the local labels.

The "IN" (INitialize) program is needed to create the synthetic zero-byte program file used by the "RK" program. It is similar to Clifford Stern's original synthetic program appearing on page 198 of "HP-41 Extended Functions Made Easy". However, there are different restrictions which apply when using this program. Any Extended Memory modules should be removed before running it. This ensures that the file created by this program resides within the Extended Functions module or the built-in portion of extended memory in the HP-41CX (hex addresses 0BF to 041, decimal 191 to 65 inclusive). Two versions are listed, one for the ZENROM, the other for use with the PPC ROM.

Since both programs are the same up to and including line 16, they will be described up to that point as one program. Line 02 clears the error ignore flag, in case it was set. The number of registers available in Extended Memory is returned after EMDIR, line 03, runs to completion. Don't interrupt EMDIR, as this result is used to compute the address of the header register modified later. EMDIR can be replaced by EMROOM if you have an HP-41CX.

Lines 04 to 06 create a one-register data file using a single space for the name. Line 07 decreases the count of available registers by one using E from line 04. Lines 08 to 11 halt execution of the program if Extended Memory modules are inadvertently plugged in, though this check is not foolproof. Lines 12 and 13 convert the negative number from line 09 to the address in Extended Memory where the second register of the header for the " " program file should be (see pages 23 and 180-181 of "HP-41 Extended Functions Made Easy" or page 8 of the HP-41 Synthetic Quick Reference Guide).

```
ZENROM              PPC ROM

01 LBL "IN"         01 LBL "IN"
02 CF 25            02 CF 25
03 EMDIR            03 EMDIR
04  E               04  E
05 " "              05 " "
06 CRFLD            06 CRFLD
07 -                07 -
08 123              08 123
09 -                09 -
10 X>0?             10 X>0?
11 RTN              11 RTN
12 190              12 190
13 +                13 +
14 "▩-----⊼"        14 "▩-----⊼"
15 RCL M            15 RCL M
16 X<>Y             16 X<>Y
17 CLA              17 XROM "SX"
18 XTOA             18 CLST
19 RDN              19 CLD
20 NSTOM            20 END
21 CLX
22 RDN                 44 bytes
23 CLA
24 CLD
25 END
       50 bytes
```

Line 14 = hex F7, 10, 00, 00, 00, 00, 00, 01 (decimal 247, 16, 0, 0, 0, 0, 0, 1).

Line 14 forms a synthetic header register that will be used to replace the header register of the " " data file created at line 06. The first nybble of line 14 after the text byte is a 1. In a header register, this specifies a program type file (rather than the existing 2, which indicates a data file). The last nybble is also 1, denoting a file 1 register long. When this Non-Normalized Number is stored in the header register, it converts the data file to a program file. Other bytes within the second header register are overwritten. Lines 15 and 16 recall this NNN, and place it in Y. X contains the destination address in decimal form after line 16.

Experienced synthetic programmers will note that lines 14 and 15 could be replaced by F1 01, ASTO L, LASTX. This uses the ASTO function to put a 1 in the first nybble of the new header. Although this method saves bytes, it is not as clear to follow. The purpose of this chapter is to illustrate how ZENROM functions can be used in a straightforward manner to accomplish tasks that would be much more difficult using just synthetic programming.

In the ZENROM program, lines 17 and 18 convert the decimal address in X to the format used by NSTOM. Line 19 pushes it out of the way, putting the NNN in X. Line 20 stores the NNN in the second header register of the data file, converting it to a program file. Lines 21 and 22 clear X and rotate the stack down, restoring the original contents of X and Y to their respective positions. Line 23 clears the address in character form from Alpha. Line 24 clears the Extended Memory directory from the display.

In the PPC ROM version, line 17 uses the "SX" routine to store the NNN from stack register Y in the decimal address contained in X. Line 18 clears all of the NNNs from the stack and line 19 clears the display.

These programs create an artificial program file within Extended Memory which is useful in restoring the USER mode key assignments. They create a program, which when accessed by GETP, does not disturb the last program in memory. Actually, reading in any program from Extended Memory will cause the assigned key indexes in status registers R (append) and e to be recomputed. After being executed once, this program does not need to be used again so long as the synthetic program file remains in Extended Memory. EMDIR shows this file as "   P001".

To use this program to reactivate the key assignments after they were suspended (by storing zero in R and e) press ALPHA, space, ALPHA, XEQ, ALPHA, G E T P, ALPHA. Or use the "RK" routine above, which does the same thing.

6C. OTHER PROGRAMS

This section presents three utility programs that use ZENROM functions. "EX" (exponent) was written to demonstrate the DECODE functions of the ZENROM and CCD Module. The house-keeping routines complete this section.

The "EX" program gives the exponent of the number in X, from 99 to -99. Note that this is the exponent displayed in SCI (rather than ENG) mode. The ZENROM version is given here. With the CCD Module, use DCD at line 02.

```
                                     before  after
01 LBL "EX"       11 CLX               T       T
02 DECODE         12 MOD
03 X<> M          13 LASTX             Z       Z
04 STO P          14 -
05 RDN            15 RCL M             Y       Y
06 STO N          16 SIGN
07 RDN            17 X<> N             X       Exponent (X)
08 ANUM           18 X<>Y
09  E2            19 CLA               L       X
10 X>Y?           20 END
                            35 bytes
```

Here is the line-by-line analysis of "EX".

Line 02 converts the contents of X into fourteen Alpha characters representing the 14 hexadecimal (Binary Coded Decimal) digits in X. Line 03 swaps X with the seven rightmost characters in Alpha, while line 04 stores these characters in P. This seems odd, but this approach will make sense in a minute. Line 05 moves these characters to the top of the stack. The remaining originally decoded characters are overwritten with the original value of Y by line 06, with the next line moving this out of the way, to give room in the stack for computation of the exponent.

Line 08, ANUM, works rather ingeniously. Because the first four bytes of register P are not within the 24 character Alpha limitation, they are ignored. So, without bothering to remove the last four nybbles (characters) of the mantissa, they are effectively neutralized by putting them in register P. Assuming that Y contained a normal number or alpha string, the first byte of register N will not contain a digit character. The number returned by ANUM will then range from 0 to 99 for positive exponents, and from 999 to 901 for negative exponents.

Lines 09 to 14 have no effect on positive exponents (0-99). However, numbers greated than 100 (E2), indicating a negative exponent, are taken modulo 100 and then have 100 subtracted from them. As an example, 998 becomes -2.

Lines 15 and 16 put the original value of X into LASTX (L). Line 17 recovers the original value of Y. The exponent of X is swapped back into place by line 18. Line 19 clears Alpha.

Next is a set of programs for ZENROM users. These are shorter and faster versions of some PPC ROM utilities "F?", "E?", " Σ?" and "C?", plus a buffer register counter "B?" and a program register counter "P?".

```
01 LBL "F?"
02 XEQ 07
03 LASTX
04 +
05 XEQ 09
06 X<>Y
07 -
08 RTN
09 LBL "B?"
10 LBL 07
11 191
12 SIGN
13 LBL 08
14 ISG L
15 ""  (F0 NOP)
16 LASTX
17 CLA
18 XTOA
19 . (decimal point)
20 NRCLM
21 X≠Y?
22 GTO 08
23 LASTX
24 192
25 -
26 RTN
27 LBL "P?"
28 XEQ 10
29 XEQ 09
30 -
31 RTN
32 LBL "E?"
33 LBL 09
34 RCL c
35 "*"
36 X<> M
37 STO N
38 ASHF
39 RDN
40 ALENG
41 DSE X
42 ATOX
43 X≠0?
44 *
45 16
46 MOD
47 LASTX
48 X↑2
49 *
50 ATOX
51 +
52 RTN
53 LBL "Σ?"
54 RCL c
55 CLA
56 STO M
57 "⊦**"
58 CLX
59 STO M
60 ALENG
61 8
62 -
63 X≠0?
64 ATOX
65 16
66 *
67 ATOX
68 LASTX
69 /
70 +
71 INT
72 XEQ 10?"
73 -
74 RTN
75 LBL "C?"
76 LBL 10
77 RCL c
78 "**"
79 X<> M
80 STO N
81 ASHF
82 RDN
83 ALENG
84 2
85 -
86 X<0?
87 CLX
88 X≠0?
89 ATOX
90 X≠0?
91 *
92 16
93 *
94 ALENG
95 DSE X
96 ATOX
97 X≠0?
98 *
99 16
100 /
101 +
102 INT
103 CLA
104 END
    185 bytes
```

"F?" Returns the number of Free registers.
"B?" Counts the number of Buffer registers used.
"P?" Calculates the number of Program registers.
"E?" Decodes the absolute address of the .END..
"Σ?" Computes the lowest register in the Summation block.
"C?" Gives the absolute address of the Curtain (R_{00}).

Stack Contents after Execution:

	"F?"	"A?"	"P?"	"E?"	"Σ?"	"C?"
T	0	0	.END.	Y	addr.	X
Z	0	0	.END.	Y	addr.	X
X	0	0	.END.	X	addr.	X
X	Free rgs.	Buf. regs.	PRGM addr.	.END.	reg.	R_{00}
L	1st free register addr.	192	.END. addr.	Right-most byte	R_{00} addr.	R_{00} and frc.

Line-by-line analysis for "F?", "B?", "P?", "E?", "Σ?", "C?"

Line 02 of "F?" calls label 07 to count the number of buffer registers being used. Lines 03 and 04 recall 192 from L and add it to X, changing the count to an absolute address. Line 05 calls subroutine 09 to give the absolute address of the .END.. Lines 06 and 07 reverse these two addresses, and take their difference. The number of free registers is in X at the termination of this routine. L contains the decimal address of the first free register.

"B?", which is the same as label 08, counts the number of buffer registers currently being used. Warning: if there are no free registers (.END. REG 00), this routine will not work as it should.

Lines 11 and 12 set up the stack for the loop between lines 13 and 22 inclusive. Lines 14 and 15 increment the decimal address in L, recalled by line 16. Lines 17 and 18 convert this number into a character in Alpha which is used by line 20 as the address to recall. Line 19 brings zero into the stack. If the register recalled is not zero, lines 21 and 22 repeat label 08. Otherwise lines 23 to 25 subtract 192 from the decimal address, giving the number of registers used for buffers. Note that the X≠Y? instruction is used rather than X≠0? at line 21. This is necessary because non-numeric values can give an ALPHA DATA error message with the X≠0? instruction.

"P?" computes the number of registers between the first data register and the permanent .END., which is the area containing your programs (Catalog 1). Line 28 calls the label 10 subroutine, which computes the address of the first data register (curtain). Line 29 calls label 09 to calculate the address of the .END., and line 30 takes the difference between these two results.

"E?", which is the same as label 09, decodes status register c to give the absolute address of the .END. from the three rightmost nybbles. Lines 34 to 38 isolate the two rightmost bytes of register c in Alpha, containing the last four nybbles. Line 40 returns the number of characters in Alpha. If the two leftmost nybbles are zero, this will be 1, in which case line 41 will skip the ATOX instruction and make X zero, so that line 43 skips the multiplication at line 44. If the leftmost byte is not zero, lines 43 and 44 multiply the 1 in Y by the decimal equivalent of the character in X. This unusual approach gets rid of the 1, and allows the routine to both handle nulls properly and preserve the original X and Y.

Lines 45 and 46 save the right nybble of the left-hand byte while discarding the other nybble. Lines 47 to 49 multiply the value of the right nybble by 256 (computed as 16 squared). Line 50 decodes the other two nybbles, and line 51 sums all three. When this subroutine finishes, the original X and Y are in Y and Z.

" Σ?" returns the number of the lowest data register in the block of six registers used for statistical functions. This routine decodes the three leftmost nybbles of status register c. Lines 54 to 56 put the contents of register c in Alpha. Lines 57 to 59 push these characters into register N and clear M. Lines 60 to 64 convert the first character, containing the first two nybbles of the address, into a decimal character code. This byte may be zero if you have been using synthetic techniques, so provisions are made here for a zero value.

Lines 60 to 62 give a result of zero when a null character occupies the leftmost byte of register c. Otherwise, line 64 decodes the leftmost character which contains the two left nybbles of the summation register block address. Lines 65 and 66 multiply this value by sixteen, mathematically shifting their decimal value left one nybble. Line 67 decodes the remaining character, containing the least significant digit of the address in the left nybble. Line 68 recovers 16 from LASTX to enable line 69 to isolate just that nybble. Line 70 removes any fractional remainder from this division. Line 71 combines the three nybbles, in decimal form, to yield the absolute address of the summation block. Line 72 computes the curtain (first data register) address. Line 73 completes this routine by taking the difference between these absolute addresses. The result is the register number of the lowest register in the summation block.

"C?", which is also label 10, preserves only X while decoding the curtain address from status register c. Lines 77 to 81 use the Alpha register to isolate the rightmost three bytes of register c. Lines 82 to 89 decode the leftmost byte which contains the two most significant digits (nybbles) into a decimal value in X. As before, though nulls will not normally be found here, they are properly decoded.

Lines 92 and 93 multiply the decimal equivalent of this first byte by sixteen, so that it will be counted properly. Lines 94 to 98 return zero or the decimal character code to X. (If the ALENG at line 94 is 1, the second byte is zero, so the ATOX is skipped. The ALENG cannot be zero unless you have synthetically changed the .END. pointer to 000).

Lines 99 and 100 save the left nybble in this byte, discarding the right nybble at line 102. Line 101 sums the values of the three nybbles and line 103 clears the discarded nybbles from Alpha.

These utility routines allow you to compute the number of registers used in the various areas within the HP-41 main memory space. Since Extended Functions are required, there is no routine to give the number of data registers allocated (just use SIZE?). Most of these subroutines are available in the PPC ROM. Of course those synthetic routines do not require Extended Functions or the ZENROM, since the PPC ROM predates both of those developments. If you don't have a ZENROM, you may omit the first 26 lines of the program, leaving "P?", "E?", " ?", and "C?".

CCD Module users already have in their Owner's Manual an equivalent to "E?" (lines 02 to 13 of the "GE" and "CB" programs) and an equivalent to "B?", called "A?" (as is the similar routine in the PPC ROM). Lines 14 to 27 of the "RK" program on page 234 of this book provides a version of "C?". A version of " Σ?", listed below, can be obtained through a simple modification of the "C?" equivalent. As in the ZENROM version, "P?" is simply the difference between the "C?" and "E?" results. Similarly, "F?" is just the "E?" result, minus the "B?" result, minus 192.

Here is the "Σ?" program for the CCD Module:

```
01 LBL "Σ?"
02 13.5   (instead of 13.1)
03 PEEKB
04 X<>Y
05A+
06 PEEKB
07 X<>Y
08 RDN
09 16
10 *
11 X<>Y
12 LASTX
13 /
14INT
15 +
16 END
```

APPENDIX A -- CRASH RECOVERY

Whenever you lock up your machine, your objective is to restore normal operation with a minimum amount of information loss. The techniques described here should be tried in the order in which they are described.

There are two main types of crashes on the HP-41. The first (and usually least serious) type of crash involves the display freezing and the keyboard being disabled, so that keyboard input has no effect. There are many things that can cause this type of lockup. Often you can recover without MEMORY LOST. First try using the R/S key or the ON switch. If this doesn't help, use backarrow and R/S together, taking advantage of the two-key rollover. Press and hold the backarrow key, press the R/S key, and release the backarrow key. If this does not work, try the techniques described below.

The next technique is to reset the calculator by interrupting its power supply. But wait, there are precautions that must be followed first. There is no danger of harming the HP-41 by removing peripherals and modules from the ports while power is applied. But be sure not to plug anything into the HP-41 before normal operation is restored, as this can damage both the HP-41 and the accesory being plugged in (exception: you can safely plug in an application pack while the calculator is crashed if the batteries have already been removed). First, disconnect the AC adapter from the rechargeable battery pack (HP-82120A). If you have the HP-82143 Printer connected, it must be disconnected because it can supply power to the calculator.

If you have a second HP-41, you can transfer an Extended Functions module and Extended Memory modules into any available port. Be sure you make the transfer quickly, as most of these modules will lose their data in less than a minute with no power applied. By transferring the modules, you can save the contents if it is not already lost or damaged. Just be sure not to turn on the second machine, unless it does not have any of these devices. The modules can be returned to the original machine after you regain control.

Next, try to reset the calculator by removing the batteries for a few seconds and reinserting them immediately. This will often clear the time and date of the Time module or HP-41CX. However, that's a small price to pay to keep from losing the entire contents of the HP-41. Repeat pulling the batteries out for longer intervals if a few seconds fails to give back keyboard control (press ON repeatedly to check for the return of keyboard control). All but the most serious crashes will be cleared by this point.

Sometimes this first type of crash will progress to the second kind of lockup: failure to turn on. First make sure that you don't have a blank display. Press ON, ALPHA, ON, ALPHA. If the ALPHA annunciator fails to turn on, you hav the second, more serious kind of lockup. Something is preventing the HP-41 from successfully completing the normal checks it performs whenever it is turned on. Catalog 1 could have been disrupted, or the first registers above the buffers may contain garbage. Always check CATalog 1 after recovering from a lockup. See Section 4F if it is not as it should be.

Because this is the most difficult kind of lockup to recover from without MEMORY LOST, you may find it easier to perform a Master Clear. Hopefully, you have an up-to-date Write All set recorded on magnetic cards or tape. But if losing the contents of the HP-41 right now will cause a lot of hair pulling, keep on reading. With a little luck, you may still avoid disaster.

In addition to the techniques described in XFME, there is one more procedure you can try in order to regain control. Start by getting an AM radio (no, this isn't a joke). Tune to an empty spot on the dial at a fairly low volume. We are going to use the radio to pick up high frequency noise emitted by the HP-41. By listening to this noise, we can tell if the machine is on, and also make a pretty good guess as to its operating state.

Place the HP-41 as near to the radio as you can. Even on top of the radio is fine. You may already hear noise being picked up. If not, try pressing ON. If you know where the AM antenna (a small ferrite bar) is within the radio, move the HP-41 as close to it as you can. Turning the calculator ninety degrees may also help. You may have to experiment a little to find the right position.

The sound produced by the radio at this time might remind you of a motor boat. This is typical of a lockup. The sound made when the calculator is on but not active (waiting for a key to be depressed, also known as "light sleep") is similar to the "white noise" hiss heard between AM stations. This hiss is intensified while executing a function. When the radio is tuned right, you can also hear squeals or clicks during the execution of catalogs and programs.

What you want to do now is to somehow get this low "motorboat" clicking sound back to the normal high frequency hiss. This is a matter of trial and error. You should remove and replace the batteries while listening to the radio. At the same time, you should press and hold various keys. This should include the ON key. You can tell when you start to make progress, as the pitch of the noise (the frequency) will increase. In general, you should avoid the backarrow key, as this may lead to MEMORY LOST.

Persistence will eventually pay off. You may have to remove and replace the batteries several dozen times, though. Try holding the square root key down, removing and replacing the batteries, pressing and releasing the ON key, and continue holding the square root key for several seconds. Leave the calculator as it is (without pressing additional keys or releasing those already depressed) for several seconds when you hear the pitch rising. Don't give up until you regain control or you see MEMORY LOST.

If you want to lock up your HP-41 intentionally to experiment with this technique, be sure to write out all of your valuable programs, or save them in mass storage first. To cause a crash, try executing a program containing an instruction with the following byte values: 1F 80 (decimal 31, 128). If you don't own a ZENROM, you'll need to use a byte loading program or the 4, 31 Q loader.

If that doesn't lock up your machine, the following program is virtually guaranteed to work (Extended Functions or HP-41 CX plus the ZENROM or CCD Module are required). It creates a synthetic zero-length buffer. Just execute the program, write down the number in X, then turn the calculator off.

```
ZENROM version                  CCD Module version
01 LBL "LOCKUP"                 01 LBL "LOCKUP"
02 "▓" (F1, BF)                 02 191
03 LBL 14                       03 RDN
04 ATOX                         04 LBL 14
05 ISG X                        05 R↑
06 ""  (Text 0 NOP)             06 ISG X
07 XTOA                         07 ""  (Text 0 NOP)
08 .                            08 RCL X
09 NRCLM                        09 PEEKR
10 X≠Y?                         10 .
11 GTO 14                       11 X≠Y?
12 RCL M                        12 GTO 14
13 "▓¯LOCKU"                    13 R↑
14 X<>M                         14 "▓¯LOCKU"
15 NSTOM                        15 RCL M
16 END                          16 POKER
   43 bytes                     17 END          41 bytes
```

Line 13 = hex F7, EE, 00, 4C, 4F, 43, 4B, 55.

After you recover control (which may be very tedious), you will probably find CATalog 1 messed up. Be sure you don't turn the HP-41 off again or cause packing. Either will hang you up again. To clear the garbage bytes, use POKER or NSTOM to clear the register that "LOCKUP" stored. ZENROM users have the option of executing RAMED (the correct address may still be in Alpha) and changing byte 5 of the register (the buffer length field) from 00 to 01. If the address has been lost from Alpha, edit line 13 of the "LOCKUP" program, making the second byte 1 instead of 0. Then run the program again to undo the damage.

Here is a line-by-line analysis of the ZENROM version of "LOCKUP". Line 02 gives the starting address in Alpha where the search for an empty register begins. Lines 04 to 07 increment this address before anything else is done, so the starting address is the lowest key assignment register, 0C0 in hex, or 192 decimal. Line 08 brings zero into the stack for the test at line 10. Line 09 recalls the contents of the register at the address in Alpha. Lines 10 and 11 will repeat label 14 until the first empty register above the buffers is found. This part of the program is similar to buffer or assignment register counting programs.

Line 12 recalls the address from Alpha to prevent line 13 from overwriting it. Line 13 is what locks up the HP-41. It is a type "E" buffer register specifying a length of zero. Before the HP-41 can check to see if there is such a device supporting this buffer, it hangs up in an infinite loop because of the length of zero. Of course, at present, there is no device using type E. It wouldn't matter if there were. Line 14 trades the address in X with the lockup register in M. The register is stored above any other buffers at line 15.

The CCD Module version works much the same way. The main difference is that the POKER function requires the register number to be in X rather than in M. A little bit of stack management ensures that the R↑ instruction at line 05 will bring the current register number into X.

APPENDIX B -- REFERENCES

This section is intended to give a partial list of sources for additional information on the programming and use of the HP-41. A large base of programs and accessories exist for the HP-41, both from Hewlett-Packard and other sources. This section does not attempt to provide a detailed, exhaustive list. However, this general guide provides a good starting point to find more information on the programs and accessories that are available now.

USERS' GROUPS

1. Handheld Programming Exchange (HPX) currently supports the HP-41, HP-71, and the new HP-28. This group contains many of the former members of CHHU (Club of HP Handheld Users), which ceased operation in March 1987. Plans for a member publication are not yet firm as of Spring 1987. To obtain HPX membership information and an application, send a 9" by 12" self-addressed stamped envelope with first-class postage for 2 ounces to:

HPX Attention: Membership request APT
P.O. Box 566727
Atlanta, GA 30356
U.S.A.
Phone (404) 391-0367, from 6 to 8 PM Eastern time only.

Some back issues of the CHHU Chronicle (October 1984 to March 1987) are still available. Write for prices and availability to:

CHHU Back Issues
P.O. Box 10758
Santa Ana, CA 92711-0758
U.S.A.
Phone bulletin, updated biweekly: (714) 472-9580

2. PPC, the Personal Programming Center, also publishes ten times per year. The PPC Calculator Journal contains a mixture of programs and applications for the HP-41 and HP-71, with articles contributed by members. PPC also sells some items, such as the PPC ROM, HP-IL System Dictionary and VASM listings. To receive a membership application and a backissue price list, send a stamped self-addressed 9" by 12" envelope with postage for 3 ounces to:

PPC, Dept. APT
P.O. Box 90579
Long Beach, CA 90809-0579
U.S.A.

3. HP Users' Library: The programs within the Users' Library are catalogued by application and encompass a wide range of uses. HP maintains high standards for the documentation of submitted programs, though the quality of individual programs may vary. A number of third part software packages are sold through their catalog. Members receive the catalog and several programs free.

The current fee for a one-year membership is $25.00 in the USA or Canada, and $40.00 elsewhere. Mail your payment in the form of a check drawn on a U.S. bank to:

HP Users' Library
1000 N.E. Circle Blvd.
Corvallis, OR 97330
U.S.A.

BOOKS AND ACCESSORIES

1. EduCALC Mail Store, 27953 Cabot Road, Laguna Niguel, CA 92677, U.S.A. is the most complete mail-order source of HP calculators, accessories and books. Their prices are competitive and their catalog is free. They carry a complete line of HP calculators, peripherals and accessories. Non-HP items such as the MLDL, 16K RAM Storage Unit, Port Extender, ZENROM and CCD modules are also available.

2. SYNTHETIX, P.O. Box 1080, Berkeley, CA 94701-1080, U.S.A., publishes this and several other books on HP calculators.

RECOMMENDED READING

Back issues of the CHHU Chronicle or PPC Calculator Journal are an excellent source of information for most areas of interest. They contain a wealth of information. The Users' Groups, listed earlier, are a valuable resource for anyone interested in learning more about the HP-41. Joining one or both of the groups and reading the backissues are highly recommended.

If you are a synthetic programmer (or would like to become one), there are several good books available. Suggested books include "HP-41 Synthetic Programming Made Easy", "Extend Your HP-41", and "HP-41 Extended Functions Made Easy". If you want to learn the inner secrets of using synthetic techniques in programs, get a copy of the PPC ROM manual and figure out how the synthetic programs work. The "HP-41/ HP-IL System Dictionary" is an excellent reference. Even if you don't yet own any HP-IL peripherals, having all of the normal HP-41 functions plus those of the Extended Functions, Time module, Printer, Card Reader, Wand, PPC ROM, HP-67/97 functions, HP-IL, Video Interface, plus key terms and error messages in dictionary form is a real time saver. It also contains much of the information found in the SQRG, though it is not as compact. It is available from EduCALC and PPC.

If you want to get into Machine-Code (M-Code) programming, and you have the necessary equipment (see Section 5C), you should start by reading Ken Emery's "HP-41 M-Code for Beginners." Advanced techniques are discussed in addition to the basics, making this book of interest to M-Code programmers at any skill level.

APPENDIX C -- ANALYSIS OF EARLIER SYNTHETIC PROGRAMS

Analysis of programs from "HP-41 Synthetic Programming Made Easy"

The book "HP-41 Synthetic Programming Made Easy" was written to help a novice learn how to use synthetic programming. Some of the programs provided there as tools are themselves excellent examples of advanced synthetic programming techniques. Line-by-line analysis of these programs would not be easily understood by a beginner, so it was not included in "HP-41 Synthetic Programming Made Easy". Now that you are becoming an expert in synthetic programming, the line-by-line analysis of these programs can be a powerful learning tool for you. It is good to learn about synthetic programming techniques in isolation, but next to writing your own synthetic programs, nothing is as helpful as understanding how advanced programs work.

"LB" by Clifford Stern (SPME, page 52)

The LBL 01 section halts execution when "LB" finishes. The GTO "++" instruction is provided to allow you to SST to the program area containing the + instructions for cleanup.

Line 06 is actually encountered twice. The first time is when you execute "LB" from the keyboard. The second time is after the XEQ "LB" instruction in your LBL "++" area of program memory is encountered. Flag 50, the message flag, is set by "LB", so that execution jumps to LBL 02 the second time through.

Lines 09-16 set up the stack and flags for a count of the +'s. The stack is loaded with 1's, flag 21 is cleared so that a turned-off printer will not halt execution, the AVIEW instruction sets flag 50, and the initial count is set to -10. (At least 17 +'s are needed in the worst case before a register is available for storage of newly created bytes.) The GTO "++" instruction starts the count.

Lines 17-27 convert the count of +'s into a count of registers. The count (call it ccc) is displayed momentarily. Now the real action starts.

Lines 28-50 decode the return address (pushed onto the return stack by the XEQ "LB" instruction). The return address is fully contained within nybbles 6, 5, and 4 of register b.

Lines 29-32 put nybbles 4, 5, and 6 of register b into flags 04-15 in register d. The 02 byte in line 30 provides the correct exponent for later use at line 49, so that data to the right of these three nybbles is discarded by the INT instruction.

Lines 33-35 clear the three bits containing the byte number within the register, so that only the three nybbles carrying the register number will be decoded. Lines 36-47 shift the 9 bits of the register number from binary into octal format.

Octal numbers consist of 3-bit digits. The HP-41 represents numbers as consisting of a series of 4-bit digits. Therefore, to convert from binary to octal, we distribute three bits at a time from a binary number into the successive 4-bit fields of an HP-41 register. This technique is illustrated by the diagram on page 87 of SPME.

Lines 48-50 give the decimal value of the register address. Call it rrr. This is used in lines 51-56 to form the DSE counter value rrr+ccc.(rrr+1)

LBL 03 starts the register formation loop. An ISG byte counter counter is pushed into Y. (The DSE register counter is pushed into Z.)

The LBL 04 loop is repeated for each byte that is created. The partially completed register to be stored is held in X at the top of the LBL 04 loop. Its initial code of 1.007, a duplicate of the original ISG counter, is actually irrelevant, since these 7 bytes will be shifted left one by one and lost.

Lines 61-64 form a message " n?", where n is the current byte number within the register. Lines 65-67 put the partially completed register in M, prepare the stack by rolling it down, then halt for input.

Line 68 checks whether data was entered and clears flag 22 for its next use. Note that flag 22 was not cleared at the start of the program, because it is safe to presume that at least one byte will be entered. If data was not entered, the LBL 05 routine (described below) fills out the register with nulls and stores it.

Lines 70-84 convert the decimal value entered into a byte. The conversion to OCTal starts the process. Adding 10,000 and putting the result in register d establishes the position of the 8 bits in flags 14, 15, 17, 18, 19, 21, 22, and 23. Lines 74-83 eliminate the gaps, putting the 8 bits in flags 16-23.

Lines 85-86 put the new byte at the left end of M. Line 87 puts the partially completed register next to this in N. Line 88 cleverly shifts everything left 6 bytes, and line 89 extracts the partially completed register with its new byte added.

Lines 90-91 increment the byte counter and return to LBL 04 to add another byte if necessary. Line 92 puts the completed register in LASTX. Line 93 sets the curtain (Register 00 pointer) to 000. Then the completed register is recovered from LASTX and stored in the current register (designated by the DSE register counter). Then the correct contents of register c are restored. Line 98 puts the DSE register counter back in X, so that the LBL 03 can be restarted if necessary.

When all the available registers have been filled, the LBL 03 loop cannot be repeated. The GTO 01 instruction causes the program to halt as explained above.

The LBL 05 section is used when you stop before you run out of registers. Lines 102-105 fill out the current register with nulls. Lines 106-108 store the register. Lines 109-110 restore the correct contents of register c. Then a GTO 01 instruction halts the program.

"LBX" by Clifford Stern (SPME, page 65)

"LBX" works just like "LB", except that both the conversion of the return address to a register number and the conversion of the entered decimal value to a byte are performed using extended functions. The conversion of the entered decimal value to a byte is done quite simply in "LBX", using just the XTOA instruction (line 64). The conversion of the return address to a register number is more complicated, but still shorter than in "LB".

Lines 29-44 decode nybbles 4, 5, and 6 from register b. Lines 29-34 put nybbles 3, 4, 5, and 6 at the right end of M, with a lone 2A byte at the right end of N. The ASHF instruction removes all data from the alpha register except nybbles 3, 4, 5, and 6. The purpose of the SIGN instruction at line 36 is to provide a 1 in the stack for use at line 46 in adding 1 to the limit of the DSE counter.

Lines 37-44 convert these bytes into decimal as follows. Lines 37-39 result in 256 if nybbles 3 and 4 are not both zero (that is, if the resulting character is not a null, leading to an ALENG of 1 or even 0). For now, let's assume that nybbles 3 and 4 are not both zero.

Lines 40-41 form 256 times the combined decimal value of nybbles 3 and 4. Lines 42-43 discard all but the last bit of nybble 4, the 9th bit of the register address. The effect is the same as if the byte number portion of the address were cleared (as it was in "LB").

Lines 44-45 convert nybbles 5 and 6 to decimal and add it to the decimal value (either 256 or 0) for the 9th bit of the register address.

Now look at what happens if nybbles 3 and 4 are zero. Line 39 gives 1, line 41 gives the decimal value of nybbles of 5 and 6. Lines 42-45 leave this result unchanged. If all 4 nybbles are zero, line 39 gives zero, and this result is maintained through line 45.

"MK" by Clifford Stern (SPME, page 69)

Lines 01-08 initialize the flags and stack. The value 192, in LASTX, is used as an absolute register address. Lines 09-10 move the curtain to 000 and put the former contents of register c in Z. It will be restored at line 125. The rest of the stack is clear, because of the CLST at line 02. Line 11 puts an F0 byte in alpha for use in restoring a key assignment register.

Lines 12-26 are a RCL b/STO b loop that searches for an empty key assignment register. The RCL b instruction pushes the former register c to the top of the stack. After the RDN, X and Y are both zero. Line 14-15 recall the assignment register without raising the stack, then compare it to the zero value in Y. An X=0? instruction cannot be used, because that would give an ALPHA DATA error on a nonzero key assignment register.

When an empty register is found, the program jumps to LBL 02, which is the bottom of the assignment register storage loop. This restores the correct value of register c and jumps to LBL 16 for entry of the first synthetic assignment.

If the register is not empty, the former contents must be replaced. Recalling a key assignment register changes the first byte from F0 to 10, so this byte must be restored to its correct F0 value. An F0 byte is already in M for this purpose.

Lines 17-18 put the recalled register in M and shift it left one byte. Line 19 attaches the F0 byte, as the rightmost character of N. Lines 20-22 shift the repaired register into N, bring it back to X, then store it where it came from.

Line 23 brings the recalled program pointer back to X. Line 24 increments the register counter, skipping line 25. Line 26 restores the former program pointer, jumping back to line 13 and repeating the loop.

When the first empty key assignment register is found, the GTO 02 leads back to LBL 16. At the top of the LBL 16 loop, flag 02 is clear if this is the first assignment for a register, or set if this is the second assignment. In the case of the second assignment, X contains the partially formed key assignment register (F0 plus three bytes).

Lines 35-42 sound TONE 8, place the prompt for input in the display, put either F0 or the partially formed key assignment in the alpha register, and halt for decimal key assignment inputs. The stack is cleared so that only one function byte is needed to assign a one-byte function. LASTX still contains the current register number.

Line 43 brings the register number into the stack for safekeeping. The LBL 03 subroutine takes the decimal number from T and converts it into a byte. The procedure is virtually identical to that used in Clifford Stern's "LB". The 10,000 is added by a ST+ N rather than +, to limit stack usage to one register. Lines 148-149 exchange the result with M and shift it to the leftmost byte. Lines 150-151 append the new byte to the former contents of M, and lines 152-153 put the partially formed key assigment register back in M.

After the prefix and postfix bytes are appended to the contents of M, it is time to process the row/column keycode. Line 46 brings the row/column keycode to X. If the keycode is negative, indicating a shifted key, flag 05 is set. Then lines 49-50 remove the sign of the keycode and store this value in N. Lines 51-52 put the current register number in N and bring the unsigned keycode back to X. (The unsigned keycode is also in Y at this point.)

Lines 53-58 break the keycode into the column number in Y, and the row number in X. Lines 59-60 put a 4 in X and decrement the column number. If the column number was 1, line 61 is skipped, and line 62 causes line 63 to be skipped. Line 64 is a NOP.

If the column number was 2 or more, and (line 61) if the row number is 4, the ISG Z at line 63 adds 1 to the column number. Otherwise the ISG Z is skipped. At line 64, we have 4 in X, the row number in Y, and the column number minus 1 in Y, corrected to account for the wide ENTER↑ key in row 4. Based on these row and column values, call them R and C, the internal keycode equivalent (for storage in key assignment registers) is R + 16*C, plus 8 if the key is shifted. The flag number for the assigned key index is 36 - (R + 8*C).

Line 65 changes the 4 in X to 8. C is in Z, R in Y. Lines 66-68 put 8*C in X. Lines 69-71 put 8*C in Y and 8*C + R in X. Lines 72-73 put 16*C + R in X, 8*C + R in T. Y and Z contain 8. Lines 74-75 add 8 if a shifted key is being assigned.

After line 78, T contains 16*C + R, plus 8 for a shifted key. Z contains 8*C + R, Y contains the current register number, and X contains 8. The register number needs to be held in the stack so that the LBL 03 routine can be used. Line 79 attaches the keycode byte to the alpha register. After line 80, X has 8*C + R, Y has 8, and Z has the current register number.

Lines 81-82 set flag 06 if 8*C + R is less than 8, indicating that a one-byte shift will later be required to set the appropriate flag. (Flags above 29, corresponding to 8*C + R less than 7, are not directly settable at line 100.)

Lines 83-84 subtract 36 from 8*C + R. The result is negative, but the sign is ignored at line 100. If flag 06 is set, 8 is added to this number, which reduces the flag number by 8. Lines 87-88 put the register number in LASTX, and 1 in X.

Lines 89-93 recall the appropriate assigned key index register and store it in N. (M still contains the full or partial key assignment register.) If flag 06 is set, the key index is shifted left one byte. Then the key index is put in the flag register for testing and setting of the appropriate flag. Line 98 tests the flag. If the flag is already set, indicating that this key is taken, Y is decremented from 1 to 0. Line 100 sets the designated flag if it was originally clear. Lines 102-106 put the key index back in N, shift it left 6 bytes if flag 06 is set, 7 bytes if flag 06 is clear, clear flag 06, and extract the new key index from register O. Line 103 contains 3 nulls so that these bytes can complete a partial assignment register in N. Lines 107-110 store the key index back in the appropriate register. At this point a fully or half-composed key assignment register is in N.

Lines 111-113 check the value of Y. If it is zero, indicating that the key was taken, execution continues at LBL 01. Lines 27-29 bring the leftmost 4 bytes of N, which may contain the previous synthetic key assignment, into X. The current assignment bytes are discarded. A "KEY TAKEN" message, TONE 0, and pause precede the normal input prompt.

If Y is not zero at line 112, meaning that the key was not already taken, execution continues normally. The fully or half-composed key assignment register is recalled from N to X. If flag 02 is clear, the register has only one assignment, and line 117 shifts the 4 bytes to the left end of N. Then lines 118-119 store the assignment in the current register. If flag 02 is set, indicating that there are two assignments in the register, flag 02 is cleared and the register counter is incremented. The SF 02 instruction is executed if and only if flag 02 was clear at line 122. This indicates the next time through the LBL 16 loop that the second assignment of the register is being processed.

Lines 123-127 restore the correct value of register c, put what may be a partly composed key assignment register in X, and return to the top of the LBL 16 loop. This procedure of storing each key assignment as it is entered, without waiting for the second assignment to fill a register, allows you to just stop making assignments at any time without using a special termination procedure.

"SA" and "RA" by Clifford Stern (SPME, page 89)

Line 40 uses the PPC ROM routine "OM" to move the curtain to absolute location 16 (decimal). The previous value of register c, which will be restored later, is put in X. Line 40 provides a starting register number, which with the curtain at 16 will access absolute address 192, the lowest possible key assignment or alarm register. Lines 42-44 compute the address of the register immediately below the .END., relative to the curtain at 16. This is the highest possible alarm register. If this number is less than 176, the .END. must be at absolute address 192, and there are no key assignments or alarms. LBL 03 is used to restore the correct value of c. The program will halt at line 86 with a DATA ERROR, due to trying to create a file of zero registers.

If the .END. is not at absolute address 192, execution continues. The 176 and the relative address of the highest available register are used to form an ISG counter for the alarm search. Line 50 puts this counter in LASTX. At this point the old value of c is in Y, Z, and T. Lines 51-52 put a hex 10 byte at the right end of the X register. This value will be used repeatedly at line 59 to detect key assignment registers.

Now the alarm searching loop, LBL 01, begins. Line 54 puts a hex F0 byte in alpha. Next the current register is recalled and placed in M, with the F0 byte coming into the X register. Line 57 appends a hex AA byte, which can be used later to replace the first byte of alarm register, damaged by the RCL operation.

Line 58 extracts the first byte of the recalled register from alpha, and puts the F0 byte in its place. Line 59 checks whether the recalled register was a key assignment register, with a first byte of F0 that recalls as hex 10. If not, the first alarm register has presumably been found, and execution skips to LBL 02. Otherwise we have an alarm register, and alpha already contains the repaired contents, ready to be shifted, extracted, and restored. Line 61 performs a 6-character shift. (The contents of c are fixed by "OM", so this is guaranteed to work as intended.) Lines 62-63 extract and restore the repaired key assignment register. After line 64, X contains the hex 10 test byte, Y contains the old c, and L still contains the current relative register number. Lines 65-66 repeat the alarm search loop until a non-key assignment register is found or until the last register below the .END. is tested. In that case, the LBL 03 routine is used to clean up, and the program will halt with a DATA ERROR at line 86. The CLA at line 67 is needed to ensure a zero file size input for the CRFLD at line 86.

Assuming a non-key assignment register is found, execution picks up at LBL 02. The old c is in Z, the hex 10 byte is in Y, the leading byte of the recalled register is in X, and the trailing 6 bytes of the recalled register are left-justified in M. If the recalled register was empty, line 71 will test true, and there are no alarms. Alpha is cleared, and the program will halt with a DATA ERROR at line 86.

Otherwise the CLA is skipped. Assuming that the register was indeed the bottom register of an alarm buffer, line 70 will have appended 6 characters, all but the leftmost byte of the register, to alpha. Since the leading AA byte was already present at the right end of M, the M register now contains the repaired alarm register. Lines 73-74 extract and store it. Register N contains the alarm register with a leading F0 byte. Lines 75-77 extract and discard this byte, bringing the old c value to X. Line 78 restores the original value of c, and bring the temporary "OM" value of c into X for later use. This is necessary to avoid trouble with a DATA ERROR halt at line 86.

Lines 80-82 put the relative address of the first alarm register in Y, and 1 less than this number, which is the address of the topmost key assignment register (191 if no key assignments) in X. Line 83, which is never skipped, decodes the second byte of the alarm register, which gives the total number of registers in the alarm buffer. This is used as the file size input for creating of the "ALM" data file. Line 85 makes sure that flag 25 is clear to guarantee a halt if there were no alarms, or if there is insufficient extended memory. Otherwise the program could halt at line 94 with the curtain at 16.

Once the data file is created, a SAVERX counter is constructed, using the relative address of the first alarm register as the starting point and the relative address of the topmost key assignment register plus the file size as the last register to be saved. Lines 91-93 lower the curtain to 16, and line 94 saves the alarm registers in extended memory. Line 85 clears the block of registers that was just saved. Lines 96-99 restore the correct value of c, beep to signal completion, and halt.

Lines 01-06 start the "RA" routine by finding the number of free registers (destructive of alarms, so you'd better not have any present when you execute "RA"), then computing the .END. location minus the number of free registers. This is the absolute address of the first available register for alarms. After line 11, X contains the current

SIZE plus the number of free registers, Y contains the SIZE, and Z contains the absolute address of the first available register for alarms. After line 13, X contains the maximum SIZE that can remain after accommodating the stored alarms and leaving no free registers.

Lines 14-15 give the DATA ERROR message at line 15 if the number of free registers, even accounting for a reduction of the SIZE to zero, is insufficient. Otherwise (lines 16-17) if the maximum usable size is less than the current size, PSIZE reduces the SIZE as required.

Lines 18-19 bring the absolute address of the first available register for alarms to X and position the curtain there. The old c value (to be restored later) is left in T. Line 20 brings the alarm data file into the correct position in main memory. Now all that remains is some repair of the normalization of the top and bottom registers caused by SAVERX.

Line 21 brings the old c value to X to prevent its loss. The FLSIZE from line 22 will be decremented (line 31), then used as a relative address of the top register of the alarm buffer. Lines 24-26 put an F0 byte at the right end of N, and an AA byte at the right end of M. Line 27 appends the rightmost 6 bytes of the first alarm register to alpha. M now contains the repaired first alarm register. Lines 28-29 extract and restore the value. N now contains a left-justified F0 byte, which is exactly what is needed for the topmost register of the buffer. Lines 30-32 extract and restore it. Lines 33-34 bring the old value fo c to X and restore it. Finally, the "ALM" file is purged, and a beep signals completion. The OFF instruction is explained at the top of page 89.

"MKX" by Tapani Tarvainen (SPME, page 92)

Lines 01-04 use PASN to assign the "ANUM" function. "ANUM" was chosen because it has a short name, and should be close to the top of

Catalog 1. Line 03 makes sure that flag 25 is clear, because if the PASN fails the program cannot complete its task. Line 05 loads alpha with a dual-purpose character string. It will serve as a temporary c register, to move the curtain to hex 0C0 (decimal 192), the first possible key assignment register. The F0 byte will also serve as the first byte of a key assignment register.

After line 06, X contains the temporary c code, Y contains the keycode (no longer needed), Z contains the desired postfix byte in decimal, and T contains the prefix byte input. Lines 07-10 construct and append to alpha the desired prefix and postfix bytes. Lines 11-13 put the append (R) register in both O and M, so that the rightmost byte, which contains the internal keycode for the assigned key, can be extracted.

Line 13 also extracts the partially completed new assignment (F0, prefix, postfix) from M. Line 14 shifts everything left 6 bytes, leaving the last 3 bytes of M as F0 A6 42. The "ANUM" key assignment has prefix and postfix bytes A6 42. This value will be used to help locate and replace that assignment. Line 14 also has the effect of moving the keycode byte to the left end of both M and O.

Lines 15-17 put the partially assembled new assignment in O, put the keycode byte at the left end of M, and store the F0 A6 42 bytes next to it at the right end of N. The left end of N has the keycode byte.

Line 18 appends an F0 byte to push the keycode byte into N from M, and into O from N. At this point the right end of O contains F0 plus the three bytes of the new assignment, and the right end of N contains F0 plus the three bytes of the "ANUM" assignment which is to be located and replaced. T contains the temporary c value, and the rest of the stack is not needed. Line 19 switches the new assignment data into the stack for safekeeping.

Lines 20-21 move the curtain to decimal absolute address 192, saving the old c in X. Line 22 recalls the right-justified bytes F0 A6 42 kk (where kk is the keycode) for use in finding the "ANUM" assignment. Lines 23-24 establish a register counter (starting at 0) in LASTX. The T register now contains the right-justified new assignment, and Y contains the "ANUM" assigment.

The LBL 01 loop locates the "ANUM" assignment. Line 26 recalls the current key assignment register. Line 27 puts the register in M, bringing a right-justified F0 byte to X. Lines 28-29 use the F0 byte to replace the leading 10 byte of the recalled register. (Given that the PASN succeeded, we cannot run into an empty register or an alarm buffer register before finding the "ANUM" assignment.) Lines 30-31 push the first 3 assignment bytes into N and bring them to X. Line 32 pushes the second 3 assignment bytes into N, with an F0 prefix supplied by line 31, for later use.

Line 33 checks whether the first assignment of the register is "ANUM". If not, line 34 switches the first assignment with the second. (Both now have F0 prefix bytes attached, so they are fully interchangeable.) If the "ANUM" assignment is found (line 35), then flag 25 is set as an indicator for later use. In this case (line 38), the new assignment is brought into X.

Now the key assignment in X, either the old non-ANUM assignment or the new replacement assignment, must be combined with the old assignment in N and restored in the current key assignment register. Line 39 shifts N left 4 bytes, left-justifying the 3 assignment bytes in N. Line 40 attaches the F0 plus 3 assignment bytes from X, and lines 41-43 shift the full assignment into O, extract it, and restore it. Line 41 also provides an F0 byte at the right end of M, which is needed at the start of the LBL 01 loop.

If flag 25 is still clear, the ISG L instruction is executed, and skips to the GTO 01 instruction. If flag 25 was set, indicating completion of the replacement, the flag is cleared, line 46 tests false, and the cleanup procedure at line 48 is begun. The old c is recovered from T and restored, the stack is cleared, and the program halts. Actually, you might want to add a CLA for good measure, to wipe out the remains of some considerable activity there.

"EFT" by Clifford Stern (SPME, page 94)

Lines 01-04 remove all but the rightmost 7 characters from alpha. Lines 05-07 allow this input to be replaced if you so desire. Then the real action starts. Lines 08-10 add 64 to the input to get the decimal equivalent of the correct suffix byte. You may want to change line 08 to RDN on general principles, to avoid an accidental enabling of the stack lift if you interrupt the program with R/S right after CLX. But CLX is faster, and you shouldn't be interrupting "EFT" anyway.

Lines 11-12 save the alpha input in the stack and load a string that will be executed as program instructions. These instructions are STO T, B0 54 (a fast 2-byte NOP), LBL 11, RDN, and the A6 Extended Functions/Time Module prefix. Lines 13-14 attach the desired suffix byte.

Flag 25 is set at line 15 to avoid halting with the program pointer set to the status registers. Line 16 appends bytes that will be executed as the instructions CLD, R↑, STO b.

Lines 18-19 move the contents of M to a, while putting the desired alpha register input in M. Lines 20-21 clear N and move its former contents to b. The program pointer is set to 54 0C. The high-order bit in the register number (nybble 3) is ignored, so the pointer is effectively 50 0C, byte 5 of status register b. Setting the register number to 40C rather than 00C is necessary in case you are using GETP or PCLPS. With a program pointer of 00C, the calculator would assume that you are positioned to the last program in Catalog 1. GETP would then have the undesired effect of causing the newly retrieved program to be executed immediately. And PCLPS would execute the .END. as a return instruction, which cannot be permitted because there is garbage remaining in the return stack. This is also why the routine is halted with a STOP instruction at step 24.

After the X<> b at line 21, execution continues at byte 4 of register b. The instructions are STO T, B0 54 (a NOP), LBL 11 (another NOP), RDN, the specified Extended Functions/Time Module function, CLD, R↑, STO b. The STO T and RDN put the previous program pointer in both Z and T, so that it will be available after R↑ even if the specified function lifted the stack.

The STO b instruction returns execution to line 22, which rolls down the program pointer. If flag 25 is still set, it is cleared and the program stops. Otherwise the SF 30 instruction gives a DATA ERROR message to indicate to the user that some sort of error has occurred.

"SOLVE" by Keith Jarett (SPME, page 127)

The register usage of "SOLVE" is:

Register 00	Function name
Register 01	$f(X_{n-1})$
Register 02	$X_n - X_{n-1}$
Register 03	X_n

Lines 01-17 set up these initial conditions, for n=2. Actually, XGUESS1 is X_2, and XGUESS2 is X_1, but this should have no effect on the program's result. LBL 14 evaluates f(x) by raising the curtain to protect registers 00 to 03, then calling the named program (XEQ IND Y), then lowering the curtain back to its original location. Note that lines 46-47 4, CHS, are faster than a single -4 line.

Lines 18-38 perform Newton's method, also called the secant method. The next root estimate X_{n+1} is computed as

$$X_{n+1} = X_n - (X_n - X_{n-1}) * f(X_n)/[f(X_n) - f(X_{n-1})]$$

Lines 18-21 compute $f(X_n)$. Next lines 22-25 compute $f(X_n)/[f(X_n) - f(X_{n-1})]$. Lines 27-30 compute and store $X_{n+1} - X_n = (X_n - X_{n-1}) * f(X_n)/[f(X_n) - f(X_{n-1})]$. Lines 31-33 replace X_n with X_{n+1} in register 03. Then if the absolute value of $f(X_n)$ is sufficiently small (here, less than 1 E-6), X_n is displayed as the solution. Otherwise LBL 10 is repeated.

"CU" by Tapani Tarvainen (SPME, page 127)

"CU" was explained on pages 131-133 of "HP-41 Synthetic Prorgamming Made Easy". Although that explanation is above a beginner's level, you should be able to understand it now and appreciate Tapani Tarvainen's outstanding effort.

Instruction Timer Program by Clifford Stern (SPME, page 149)

As mentioned on page 150, the data registers used by this program are:

Register 00:	number of instruction groups
Register 01:	curtain lowering code (sets curtain at 16)
Register 02:	return pointer for the stored byte sequence
Register 03:	number of instruction storage registers

The input to "IN" specifies how many registers are to be used for instruction storage. The instructions are stored in the free registers below the .END., and PSIZE is used to reduce the SIZE if necessary to accommodate the requested numeber of registers. The desired instruction sequence is replicated as many times as possible within the requested number of registers.

Execution starts at line 31, LBL "IN". The number of instruction storage registers requested is saved in data register 03. Then the number of free registers is computed, then the absolute address of the first (lowest addressed) free register is found at line 38. This number is saved temporarily in data register 01.

Lines 40-41 put the current SIZE in Y. Lines 42-47 compute the SIZE plus the number of free registers, minus (R+7), where R is the number of registers of instructions to be stored. In order to run, the program needs 4 data registers plus R+3 free registers: R registers of repeated instructions, one register of instructions to start the timing, one register of instructions to end the timing, and one empty register to form a protective gap between the stored data and the key assignments. Thus, between the SIZE and the free registers, a total of R+7 registers is needed. If the value S+F-R-7 at line 48 is negative, the total number of registers available is insufficient. In that case, the SQRT function gives an eror message.

Otherwise the program continues, computing S+F-R-3. If this is less than S (line 52), then the number of free registers F is less than the required R+3. The SIZE must be reduced by the amount of the deficiency. This is done at line 53.

Line 54 lowers the curtain to 16 absolute, and rolls up the old value of c into Y. This only purpose of this step is to make sure that at line 58 the curtain lowering code will be in ASTOrable form. Line 55 brings the address of the first free register, computed at line 38, into X. This first free register must be left empty to form a gap between the key assignments and the program instructions to be stored. Therefore the first register address that can be used is one greater than this value. Lines 56-58 add 1 and set the curtain to the this address. The value of c set by "OM" is left in T. The old value of c drops from Y to X.

Line 59 restores the correct value of c, bringing the "CX" value of c to X. This allows register 03, the number of instruction storage registers, to be recalled. Adding 1 gives the relative address of the register in which will be stored the starting instructions that precede the repeated instructions being timed. (In relative addresses, register R+1 has the starting code, registers 1 to R have the repeated instructions, register 0 has the ending code, and register -1 is the empty gap register.)

Lines 63-67 store the ending instructions: STOPSW, RCLSW, HR, NEG, RTN. The NEG instruction is simply a slightly faster way to enter the value 0 in X. Lines 68-69 store the starting instructions: FS? 02, STOP, RUNSW. Lines 70-72 restore the correct value of c, and put the "CX" value of c in register 01, where it will be needed later to store the repeated instructions to be timed. Normalization will not be a problem, because the use of "OM" at line 54 ensures that the "CX" value of c is valid alpha data.

Line 72 also extracts the absolute address of the first (lowest addressed) free register from register 01. Y contains R+1. Lines 73-78 use these two numbers to construct a return address that will be used later to jump in at byte 4 of the starting register (register R+1 with the curtain in the "CX" position). To start at byte 4, we need an address of 101x xxxx xxxx xxxx. The byte portion of this address is equivalent to a register count of 2560. An extra 1 is added because there are R+2 total registers of stored instructions.

The pointer is stored in register 02 in alpha data form. (Note that this alpha data return pointer allows the PPC ROM's "XE" program to be used at line 177 in an unconventional manner. Here "XE" is used to jump into the status registers, rather than into a ROM program.) Line 80 signals completion of the initialization.

Now the instruction storage section begins. LBL 10 stops the stopwatch and sets it to zero. Lines 85-89 set the display mode and clear alpha and X. Lines 90-102, the LBL 11 loop, prompt for the decimal equivalents of each byte in the group to be timed. The byte sequence is built in M, and a count of the number of bytes in the group is kept in register a.

Line 91 appends the last byte entered to alpha. (The first time through the LBL 11 loop, X is 0 and alpha is clear, so that alpha remains empty.) Line 92 increments the group size counter in register a. Line 93 is always skipped. Line 94 saves the byte group in X, so that the prompt for input can be formed. After the input prompt is put in the display, line 99 puts the byte group back in M. If a decimal input was provided, the GTO 11 causes the byte to be appended and the next prompt to be generated. If no input was provided, lines 103-106 clear alpha, blank the display, put the byte group back in M, and correct the group size count in register a.

Lines 107-113 calculate and store the number of instruction groups, and store it in register 00. If this number is not an integer, line 114 will cause a halt with an error message. Line 115 jumps to LBL n, where n is the number of bytes in the instruction group. The different values of n are explained separately below.

First, the n=7 case. Lines 117-120 move the 7-byte group from M to O, clear the X and a registers, and jump to LBL 12. In the LBL 12 section, lines 144-146 make a DSE counter by adding a fractional part, normally .0000n, but zero in this case. Line 147 copies this DSE counter into LASTX. Lines 148-151 lower the curtain to the previously set up "CX" position, recall the 7-byte group from O to X, and jump to LBL 09.

In the LBL 09 section, the 7-byte group is stored in relative addressed registers R down to 1. After register 1 is stored, the DSE at line 163 skips to LBL 13. The LBL 13 cleanup section clears the blank display (so that X will be visible when the program halts), restores the curtain, and clears the stack. A tone is sounded if Flag 02 is set or Flag 01 is clear. The program halts if Flag 01 is clear.

Now let's look at the n=6 case. At line 125, M contains the 6-byte group and register a contains 6. Line 129 puts M in X as alpha data. Lines 130-133 compute the integer part of 17/n, 2 in this case. The short RCL b/STO b loop appends 2 copies of the 6-byte sequence. There are now 18 characters in alpha. Lines 138-140 shift alpha left 3 characters, left-justifying the string in register O. Lines 141-143 compute .0000n, which is .00006 in this case.

The LBL 12 section works as in the n=7 case. The DSE counter is computed as R.0000n, so that every nth register from R down to 1, will be stored in the LBL 09 loop. After this pass, the DSE a instruction does not skip as it did in the n=7 case. The GTO 00 causes the program to set up for the next pass. The R.0000n counter in LASTX is changed to (R-1).0000n and brought into the stack.

Line 156 appends one byte to alpha, so that register O now contains the seven bytes 2345612, rather than the previous 1234561. The RCL O at line 158 puts the sequence 2345612 in X where it can be stored in every 6th register down from (R-1). After line 158, the byte sequence for the current register is in X, the old value of c is in Y, and the new DSE counter is in LASTX and Z.

After this pass through the LBL 09 loop, the LBL 00 section bumps alpha one more byte to the left (by appending the character "1" without the decimal point). then it changes the DSE counter to (R-2).0000n, so that the sequence 3456123 can be stored in every 6th register down from (R-2).

After a total of 6 passes, the DSE a at line 153 skips to the LBL 13 section for cleanup.

The n=3, 2, and 1 cases use the same instructions and work the same as the n=6 case. The RCL b/STO b loop ends with a total of 18 characters in alpha. In each of these cases, every nth register is the same, and the n-byte group must be shifted one byte to the left between adjacent registers. This is the same as for the n=6 case because 3, 2, and 1 are all divisors of 6.

The n=4 and n=5 cases have to be handled a bit differently. In both of these cases, after the STO b/RCL b loop there are 20 characters in alpha. Therefore only 1 character needs to be appended to left-justify the string in O. Thus, flag 29 is set at line 124, so that line 140 can be skipped later, leaving only the single-character shift at line 138.

Now an explanation of the choice of flag 29. For the n=5 case, the alpha register must be shifted 2 characters to the left after each pass. Thus the first set of registers gets the sequence 1234512, the second set gets 3451234, the third gets 5123451, the fourth gets 2345123, the fifth 4512345. With flag 29 set, line 155 appends two characters, the 1 and the radix (either decimal point or comma).

For the n=4 case, the alpha register must be shifted 3 characters to the left after each pass. Line 122 sets FIX 1 rather than FIX 0, so that line 155 will append 1.0, a 3-character string.

The LBL "T" portion of the program does the timing. PPC ROM routine "XE" executes as a subroutine the free register address computed at line 78 and stored in register 02, setting up a return to line 178. The instructions executed in the free register area are: FS? 02, STOP, RUNSW, followed by the repeated byte group being timed, then STOPSW, RCLSW, HR, 0 (actually NEG, which enters a zero in X), and RTN. Then line 178 resets the stopwatch for the next pass. Lines 179-181 converts the hours to milliseconds. Lines 182-185 compute the number of milliseconds per instruction group, set FIX 9 to see the result, sound a tone, and halt.

If you press R/S at this point, you enter the cleanup routine at line 01. The flags are reset to default values, the display is blanked, and "LF" locates the free registers relative to a curtain value of 16. Note that the presence of the gap register between the stored instructions and the key assignments ensures that "LF" will work properly. The "OM" routine sets the curtain to 16. Line 05 puts the ISG counter for the free register block (from "LF") in X. Line 06 increments this count because the gap register is already clear. Then the "BC" routine clears the rest of the registers that were used for instruction storage. The LBL 13 section restores the curtain and cleans up.

The LBL "1", "2", and "3" sections are non-prompting entry points. They all finish by jumping to LBL 16. The conditions needed at LBL 16 are: the number of bytes per group (1, 2, or 3) in register a, the byte sequence in M, FIX 0 mode, flag 29 clear, and the display blanked.

At LBL 01 the number of bytes per group is in X and the suffix byte or bytes are in alpha. Line 21 puts the group size in register a. Lines 22-26 put the suffix bytes in the stack, blank the display, and set the flags correctly. Lines 27-30 put the first byte in M (with the rest of alpha clear), append the suffix bytes, and jump to LBL 16, continuing as if the data had been entered manually.

"MC" by Clifford Stern (SPME, pages 156-157)

Lines 01-05 set flag 26 for later use, then exchange the contents of the flag register with a code that sets flags 26, 28, 29, 39, 40, 48 (ALPHA mode), and 55. Line 06 recalls the program pointer. Line 07 tests whether this is the first time through (flag 26 set) or whether this point was reached by the ASTO b at line 16. Flag 26 is cleared, so that the second time through line 07, execution will jump to LBL 01.

The first time through, lines 09-16 are executed. Lines 09-10 put blank alpha data in Z. Y still has the old flag register, and X has the program pointer from line 06. Line 11 puts the program pointer in M and 0 in X. Line 12 puts 0 in LASTX. Line 13 puts the program pointer back in X in the form of a character string. Lines 14-16 add a return address of 00 1D to the program pointer and put it back in register b, jumping back to line 07. Two more return addresses will be added later, with the effect of pushing the 1D portion into the most significant byte of register b. Then the resulting values (from RCL b) can be stored as alpha data. Since flag 26 is now clear, the GTO 01 instruction at line 08 is executed.

The LBL 01 section starts by prompting for input characters. If flag 23 is set, indicating that characters have been entered, the line 22 is skipped. If flag 23 is clear, indicating that no characters have been entered, execution continues at LBL 06. That case will be analyzed below. First, let's see what happens to the entered text.

The VIEW Z instruction blanks the display. Flag 26 is cleared so that the TONEs for the message input will not sound during the compilation process.

Lines 25 and 26 clear X and Y. Then lines 27-28 clear the P register and check whether it was empty. If so, we skip to LBL 02 with flag 05 clear. Otherwise flag 05 is set and the contents of alpha are "bumped" 7 bytes to the right. The previous contents of register M, which will be needed later, are saved in Y. The LBL 02 loop appends a null to push a single character into P. Line 38 tests whether a character was present. (Note that with the first six bytes of P being 0, this is one of the few cases where an X=0? instruction can be used to test a non-normalized number without the risk of an ALPHA DATA error message.) If the character was a null, the LBL 02 loop continues until a non-null character is found. The character is put back in P (line 40) and the stack is rolled down, leaving the previous contents of M in X if flag 05 was set, 0 otherwise. LASTX still contains 0, Y contains 0, and Z contains the old flag register.

The 0 at line 42 has the effect of altering some of the bits in register P. The significant feature here is that the first nybble is set to 1. This allows the single character at the right end of register P to be interpreted as valid alpha data for indirect execution (lines 55-58). At this point, the old register d is in T, where it stays for most of the rest of the program.

Line 43 clears flag 29 and tests whether this is at least the second time through the LBL 05 section. The LBL 05 section is first entered by means of an XEQ instruction in order to load a return to line 46 onto the return stack for use during playback. Subsequent passes must use GTOs to preserve the correct return stack arrangement.

The LBL 05 section removes the first character from the P register, putting 0 in P. Because of line 42, this register has a first byte of 1x, making this a valid single-character alpha string. Line 58 executes the tone sequence corresponding to the character and loads another address onto the return stack. This serves the dual purpose of providing a return to line 59 in both the setup and playback phases, since each RCL b instruction captures the return address for use during playback. Since flag 26 is clear during the setup phase, the TONEs act as NOP instructions, executing quickly and silently.

Line 59 increments the register count in LASTX, initially 0, and skips line 60. Line 61 checks whether the specified tone sequence was found. If not, the program assumes that this is because it has run out of input and encountered a null character. If you were unfortunate enough to enter a character outside the program's rather complete Morse Code set, the rest of the current set of input characters will be lost.

Assuming that the specified tone sequence was found, execution continues at LBL 03. The RCL b results from just ahead of the tone sequence are stored in the current register. In the playback mode, this value will be stored in register b to jump directly to the tone sequence. The RDN at line 50 prevents the old flag register contents from being pushed off the top of the stack. Note that the stored RCL b value includes a pending return to line 59.

Line 52 places a zero in X and stes the third nybble of register P to 1, so that when step 53 pushes the next character into P, P is again treated as valid alpha data. The LBL 05 section executes the tone sequence to pick up the correct RCL b value.

After the last character in alpha is processed, line 58 will clear flag 25. Since flag 26 is still clear (we are in the compilation mode, rather than the playback mode), the DSE L at line 65 will decrement the register counter to restore its correct value. (The counter was incremented before we learned that the current character was a null.) Line 66, which is never skipped, tests whether we still have another 7 characters in X (from line 33) waiting to be processed. If so, we store them in M and go back to LBL 04 to process them. If not, or after we finish processing them, we go back to LBL 01 to get more input.

This time, everything works the same up to line 43. Here, since flag 29 is now clear, we will GTO 05 rather than doing the XEQ 05. This works much the same, except that the return to line 46 is already on the return stack.

When the input is finished, you press R/S without any alpha input, and execution jumps to LBL 06. The counter in LASTX has the number hhh of the highest register used. Lines 70-76 form the ISG counter 1.hhh and store it in register 00, which was cleverly bypassed by incrementing the counter at line 59 before using it to store the recalled program pointer at line 49. Lines 77-79 put the ISG counter in LASTX, then restore the old flag register. This has the important effect of setting flag 26 to enable playback.

Playback begins immediately, starting with register 01. Storing this value in b takes care of those messy pending returns, and executes the first tone sequence. The RTN that follows the tone sequence causes a jump to line 59. LASTX is incremented, and the RTN at line 60 is <u>not</u> skipped. That RTN causes a jump to line 46, which recalls the next register and puts it in b, starting the next tone sequence. This process continues until the ISG L at line 59 reaches a skip condition. At this point flag 25 is clear and flag 26 set, so we jump to LBL 07.

At LBL 07, the 1.hhh ISG counter is put back in LASTX, so that you can just press R/S to hear the Morse Code message again.

Analysis of programs from "HP-41 Extended Functions Made Easy"

The book "HP-41 Extended Functions Made Easy" included a chapter of synthetic utility programs. These are very useful programs, but no line-by-line analysis could be provided within the scope of "HP-41 Extended Functions Made Easy". These programs represent the fully mature state of the art in synthetic programming, with the extended functions. The line-by-line analysis presented here will help you understand some of the most advanced synthetic programming techniques.

"XF" by Clifford Stern (XFME, page 173)

"XF" has a similar purpose to the "EFT" program presented in "HP-41 Synthetic Programming Made Easy". The method used to achieve this purpose is quite different, however. "EFT" constructed the byte sequence to be executed and stored it in register a for execution. "XF" stores the sequence at the end of the "XF" program itself. This allows more complex functions to be used, and it allows error conditions to be indicated by the normal error messages. It also permits error recovery without requiring a second execution of the program or function. Simply fix the the incorrect input and press

R/S, since the program has stopped at the desired function.

Lines 01-03 save the X input in LASTX, freeing a stack register for temporary use. Lines 04-05 save the register N portion of the alpha input in register a. Lines 06-08 add 64 to the function number. This provides the correct decimal input for the suffix byte that accompanies the A6 prefix. Note that ST+ Y is used instead of + so that LASTX will be preserved.

Line 09 appends 13 bytes to alpha. The 14th byte is the computed suffix, appended at line 11. At this point, O contains the register M portion of the alpha input, and N contains a temporary code for register c to set the curtain to 001. M contains 7 bytes corresponding to instructions that will be stored as lines 27 to 30 of "XF". Actually only the hex 7B suffix byte for line 27 is stored. The prefix byte is in the register above the modified register.

The register to be modified occupies bytes 64-70, the 10th register, of the "XF" program. Since "XF" is the top program in Catalog 1, this register is 10 below data register 00. With the curtain at 001, and with full memory (quad module, CV, or CX), the relative address of the register to be modified is 512 - SIZE - 10 - 1 = 501 - SIZE.

Lines 12-16 compute the correct relative address without disturbing LASTX. After line 16 the relative address is in X, the Y and Z inputs are in Z, the X input is in LASTX, the N input is in register a, and the M input is in O. The temporary c code is in N, and the register to be stored is in M. The T register is available for use.

Lines 17-18 lower the curtain, putting the relative address in N and the old c register value in X. Line 19 saves the old c register value in register a, bringing the N input into the stack. Line 20-21 extract the 7 instruction bytes from M and store them in the 10th register of the "XF" program. Lines 22-25 extract the M input from O and place it properly in a clear alpha register. The N input is rolled down into X. Line 26 puts the N input into alpha and clears X. Line 27 brings the old c register value back to X and clears register a. Lines 28-29 restore the correct value of c and bring the X input back into X. At this point the Y and Z inputs are in Y and Z, and T is clear.

Line 30 is the specified instruction from the Extended Functions, Time Module, or Wand. Line 31 clears the display in case you executed EMDIR with a revision 1B or 1C Extended Functions module. These modules leave the last directory entry in the display, rather than clearing it as the HP-41CX does. Clearing the display allows you to see the number of available extended memory registers in X.

"EFTW" by Clifford Stern (XFME, page 177)

Lines 01-04 remove all but the rightmost 7 characters from alpha. Lines 05-07 allow this input to be replaced if you so desire. Then the real action starts. Lines 08-10 add 64 to the input to get the decimal equivalent of the correct suffix byte. You may want to change line 08 to RDN on general principles, to avoid an accidental enabling of the stack lift if you interrupt the program with R/S right after CLX. But CLX is faster, and you shouldn't be interrupting "EFTW" anyway since it is not used as a subroutine.

Lines 11-12 append 14 characters to the alpha register, leaving the original alpha input in O. The first 7 appended characters form a temporary c register code for setting the curtain to 001. The last 7 appended characters correspond to instructions that will be stored as lines 23 to 27 of "EFTW". Only the hex 75 suffix byte for line 23 is stored.

The register to be modified occupies the 9th register of the "EFTW" program. Like "XF", "EFTW" must be the top program in Catalog 1. With the curtain at 001, and with full memory (quad module, CV, or

CX), the relative address of the register to be modified is 512 - SIZE - 9 - 1 = 502 - SIZE. Lines 13-16 compute the correct relative address. Again, you may want to change line 13 from CLX to RDN.

After line 16 the relative address is in X, and the X and Y inputs are in Y and Z. The M input is in O, the temporary c code is in N, and the register to be stored is in M. The T register is available for use.

Lines 17-18 lower the curtain to 001, putting the relative address in N and the old c register value in X. Lines 19-20 extract the 7 instruction bytes from M and store them in the 9th register of the "EFTW" program. Lines 21-23 extract the M input from O and place it properly in a clear alpha register. X is cleared. This zero value will end up in Z.

Lines 24-25 brings the old c register value back to X and restore it. Lines 28-29 restore the correct value of c and bring the X input back into X. Line 24 brings the X and Y inputs into X and Y. After line 26, Z is clear.

Line 27 is the specified instruction from the Extended Functions, Time Module, or Wand. Line 28 clears the display to let you see the number of available extended memory registers in case you executed EMDIR with a revision 1B or 1C Extended Functions module.

"VER" by Clifford Stern (XFME, page 185)

Lines 01-03 make sure that an Extended Functions/Memory module (or HP-41CX) is present. If this were not confirmed, line 10 could cause a halt with the curtain lowered.

Line 04 puts a temporary register c code in N and the number 189.003003 (hex 01 89 00 30 03 00 02) in M. Lines 05-07 put the temporary register c code into both the stack and register c, where it sets the curtain to 001. The old register c is brought into the stack. Line 08 recalls the value 189.003003 to X.

Lines 09-10 put the hex code 10 00 10 00 2E F0 BF in absolute address 64, the pointer register at the bottom of the Extended Functions/Memory module. As explained on page 181 of XFME, the format of this register is 000 WW 000 NNN TTT. Therefore, lines 09-10 set the working file WW to the first file in the extended memory directory, the address NNN of the top register of the next extended memory block to 2EF, and the address TTT of the top register of this block to 0BF. Of these three effects, the important one is selecting the first extended memory file as the working file.

With the curtain set at 001, line 11 moves absolute registers 190, 191, and 192 to LASTX, M, and N (register 3, 4, and 5 relative to 001). As explained on page 182 of XFME, this normalizes register 192. The "VER" program will replace the leading byte of the recalled register with F0 to repair the damage.

Line 12 shifts alpha to the left 1 byte. This moves the normalized 10 byte from the recalled register 192 into O. Lines 13-14 put 191 in register a. This value will later be used as an indirect address for the repair of register 192.

Lines 15-16 bring the temporary c register code (from line 06) to X and store it in O. The trailing F0 byte of this code replaces the normalized 10 byte of the register 192 contents. Line 17 shifts alpha to the left 6 bytes, pushing the repaired register 192 contents into O, and the register 191 contents into N. The value in M becomes hex 2A 00 00 00 16 BF FF. This value will be used temporarily as the second header register of the first file in extended memory.

The structure of this register, 2AAA 0000 RRR SSS, was explained on page 181 of XFME. The header address nybbles AAA are optional, normally supplied by the HP-41 but not needed by it. The SSS nybbles here specify a file size of 4095!

This particular file size is required to circumvent an internal memory check that the HP-41 will make when you try to use the file. Before a file is accepted as valid by the extended memory operating system, the system checks whether any part of the file is contained in a module whose contents have been lost due to its temporary or permanent removal. If this is found to be the case when the user attempts to access such a file, the partition code is written into its name header register and a FL NOT FOUND message is displayed.

The main test to establish the validity of a file is to check the location one register past the end of that file. This would contain either the first header register of the next file or the partition code. Its location is computed by attempting to jump forward from the second register of the named file a distance of one plus its FLSIZE. If the resulting register is found to be in a properly connected module, as judged by the various pointer registers of the XM modules, the named file is accepted as valid. When one is added to a file size of FFF, the sum is 000 in the three-digit register field used by extended memory. This provides a jump of zero from the second header of the named file. Since its register has been previously verified, the file passes the test.

The RRR nybbles set the record pointer to 16B, which is 363 in decimal. This is 124 to get to the last register in the Extended Functions/Memory module, 238 to get to the last register in the first Extended Memory module, 1 to get to the first register in the second Extended Memory module.

Lines 18-19 repair register 192. Line 20-21 store the code from M in register 190 (relative address 189). This changes the working file into a 4095-register data file as explained above. Lines 22-23 store the same code in register 191 as the name of the first extended memory file. Actually any code other than all F's would do the job; the idea is just to make sure that the first extended memory file has a name. This is necessary in case extended memory was empty when "VER" was executed.

Line 24 brings the old register c value to X, so that it is not lost from the top of the stack. Line 25 brings the former register 191 contents into the stack, so that alpha can be used. Line 26 brings the record pointer (363) to X for later use at line 35. Line 27 puts the contents of location 1007 in the stack, without normalization. Lines 28-29 construct the restart prompt and turn on ALPHA mode so that the prompt will be seen after the magnetic card operations are completed.

Lines 32-35 reset the record pointer to 363 and restore the value in location 1007. Lines 36-37 restore the value in location 191. Lines 38-39 check whether the former contents of register 190 matches the the former contents of register 191. If so, normally because both registers were zero, the program assumes that extended memory was empty, and line 40 clears the pointer register to correspond with the empty extended memory.

Lines 42-43 restore the contents of register 190. Lines 44-45 restore the old value of c.

"PFF" by Clifford Stern (XFME, page 188)

Lines 01-04 add trailing spaces if needed, then rotate the first 7 characters of alpha into M. After the current value of c and the new file name from M are saved in the stack, a temporary code is stored in c to put the curtain at hex 0BF = 191.

Line 09 stores the file name in the correct format in location 191. Lines 10-11 restore the correct value of c. Lines 12-13 clear out the stack, and finish with EMDIR to set a working file in case you have revision 1B extended functions. Line 14 clears the display to let you see the number of available extended memory registers in case you executed EMDIR with a revision 1B or 1C Extended Functions module.

"EXM" (XFME, page 191)

Lines 01-05 put the byte 190 = hex BE in alpha. Line 06 sets the program pointer to byte 0 of register 0BE. Byte 6, the leftmost byte, of register 0BD is the first byte executed. This is where the program starts, provided that the first file in extended memory is a program file.

"SAVEK"/"GETK"/"RK"/"SK" by Tapani Tarvainen (XFME, page 197)

Line 01 terminates execution if you execute "RK" when this is the last program in Catalog 1, as explained on page 196 of XFME.

The LBL 04 section of the program translates the .END. pointer in the rightmost 3 nybbles of register c into a decimal number. The routine starts by recalling register c. Lines 48-51 isolate the last 2 bytes of c in M. Lines 53-61 are identical to lines 37-45 of "LBX" from SPME. See the analysis of that program if you have any trouble following the logic.

The LBL 10 section of the program brings the old value of c into X and lowers the curtain to hex 0C0 = decimal 192. The .END. pointer is set to hex 200 to avoid clearing the global label assignments at line 78, and to make sure that even if the program is interrupted the Catalog 1 chain will not be altered before it is restored later.

Line 03 of "SAVEK" recalls the file name from M to X. Then the LBL 04 subroutine decodes the .END. pointer. If the .END. is below location

193, there are no key assignments and the RTN at line 07 halts execution. Otherwise flag 10 is set to indicate that counting of the key assignment registers is in progress. The second time through LBLs 01, 02, and 03, the registers are saved (line 30).

The LBL 01 section forms an ISG counter for the key assignment area, relative to a curtain of 192. Lines 13-14 put a hex 10 byte in X for the comparison at line 24, which looks for a hex 10 byte in the recalled key assignment register. Lines 15-16 set the curtain to decimal 192 and save the old c register value in LASTX.

The LBL 02 section starts by rolling down the stack, putting the hex 10 byte in X and the ISG counter in Y. Line 19 recalls the current register. Lines 20-23 replace the first byte with hex F0, bringing the recalled first byte to X. If that byte was not hex 10, meaning that the register does not contain key assignments, the GTO 03 instruction is executed. (Note that if the register was an alarm register, its first byte will change from AA to 1A. This will be fixed later by the HP-41 itself.) If the byte was hex 10, indicating a key assignment register, line 26 appends 6 characters, shifting the repaired key assignment register into N.

Lines 27-28 restore the key assignment register. Line 30 is skipped. Lines 31-32 continue searching for the first empty or alarm register in the key assignment area, skipping to LBL 03 if the register immediately below .END. has just been checked.

The LBL 03 section begins by restoring the old contents of register c. Then lines 36-38 put the file name in an empty alpha register. Lines 39-40 recover the relative address of the first free register. This is also the number of key assignment registers. Line 41 clears flag 10 and skips line 42.

Line 43 makes a data file with the proper name and size. Note that an error stop at this point is not hazardous. Line 44 is necessary so that the right ISG pointer will be formed at line 12 for the next pass through the key assignment registers.

The next pass saves each repaired key assignment register in the new data file. The LBL 03 section finishes as before, except that the RTN at line 42 halts the program.

Lines 68-74 of "GETK" increase the SIZE by 1, then decrease it to the original size. This procedure ensures that there is at least one free register. Line 75 puts the SIZE+1 to X, and the SIZE in Y. Line 76 recalls the size of the specified file. Line 77 sets the curtain to decimal 192, bringing the old c value to X. Line 78 clears the key assignments, except for the global label assignments. Lines 79-80 then restore the original value of c, and put the temporary c value in T for later use. Lines 81-84 set the SIZE to SIZE+1+FLSIZE, then restore the original SIZE. This ensures that there are at least FLSIZE+1 free registers present after clearing of the key assignments. After line 84, Y contains the old c value.

Line 85 recovers the file size from LASTX. Line 86 gives the absolute address of the .END. . Lines 87-90 subtract 192 and the file size to get the current number of free registers minus the key assignment file size. This is one greater than the number of registers that could be occupied by Time Module alarms. This number of registers will be moved upward in memory to make room for the key assignments. The extra register is specified because the REGMOVE at line 99 will normalize the top register moved, and we can be sure that this extra register will be empty.

Lines 91-99 move all alarms upward by F registers, where F is the file size. The old value of c is left in Y. Line 100 moves the key assignment data from the key assignment data file to the key assignment registers, without normalization. Lines 101-102 restore the old value of c.

Lines 103-106 use GETP on a special empty program file to reconstruct the assigned key indexes in registers R (append) and e. The program then halts. This reconstruction is also performed as the sole function of "RK", which reactivates suspended key assignments. The use of the synthetic zero-byte program file for this purpose was invented by Tapani Tarvainen. The advantage of the empty program file is that GETP does not disturb the last program in memory.

Lines 107-111, the "SK" routine, suspend all key assignments by clearing the assigned key indexes in registers R (append) and e.

"IN" by Clifford Stern (XFME, page 198)

Line 02 uses EMDIR to find the number of usable extended memory registers. This number will be used later to find and alter the second header register of a new data file, making it into a synthetic zero-byte program file. If you have an HP-41CX, you can replace both EMDIR's in this program by EMROOM.

Lines 03-05 create a the 1-register data file that will be converted into the zero-byte program file. The file name is a blank character (note the comma in the line 04 text). Actually the file name is 7 blank characters after trailing blanks are added.

After line 04, register N contains hex 20 2C 01 69 00 13 F0, which will be used as a temporary register c code to put the curtain at 001. M contains 01 89 00 30 03 00 02, which is the number 189.003003 to be used as a REGMOVE input. Line 06 adds 1 to the number of usable extended memory registers.

Lines 09-10 lower the curtain to 001, leaving the old value of register c in X. Lines 09-10 move registers registers 190, 191, and 192 to LASTX, M, and N (register 3, 4, and 5 relative to 001). As explained on page 182 of XFME, this normalizes register 192. The "IN" program will replace the leading byte of the recalled register with F0 to repair the damage.

Line 11 shifts alpha to the left 1 byte. This moves the normalized 10 byte of the recalled register 192 from N into O. Lines 12-14 recall the temporary code from c (changing the first nybble from 2 to 1) and put it in O. The trailing F0 byte of this code replaces the normalized 10 byte of the register 192 contents. Line 15 shifts alpha to the left 6 bytes, pushing the repaired register 192 contents into O, and the register 191 contents into N.

Lines 16-17 recall the hex code 2A 00 01 69 0B DF FF from M and store it in c. This puts the curtain at hex 0BD = decimal 189. Lines 18-19 store the hex code 20 2C 01 69 00 13 F0 from line 14 in location 190. This code fools the calculator into thinking that the first file in extended memory is a data file of length 3F0 = decimal 1008. This is an illegal (too large) file size.

Lines 20-21 restore the repaired contents of register 192. Note that if this register was empty, the lone F0 byte will be eliminated by the HP-41 the next time it is turned on. The EMDIR instruction at line 23 causes the HP-41 to recognize that the size of the first file is illegal. The calculator puts the partition code (all F's) in register 191, effectively clearing the extended memory directory. The number returned to X is the number of available registers in all of extended memory.

It is at this point that a bug of "IN" surfaces. If you have the full complement of extended memory, and if the third block of extended memory (the second Extended Memory module) has not been used since the last MEMORY LOST, line 23 of "IN" will cause the HP-41 to clear the middle block of extended memory (by disrupting the pointer register at the bottom of the first Extended Memory module). This is not a problem if there are no files which extend into the middle block of extended memory.

There are two ways to prevent this bug from destroying part of your extended memory. The first is to use the alternate, bug-free version of "IN" on page 305. The second is to check whether the above conditions apply, then create a dummy data or text file of the maximum possible file size, EMROOM. (EMROOM is the number of available extended memory registers shown after execution of EMDIR or EMROOM.) Purge the dummy file immediately, then execute "IN".

A third option is also available for the adventurous ZENROM, PPC ROM, or CCD Module owners. You can repair the damage after the fact. Just put the right value, hex 00 00 00 40 3E F2 EF, back in register 513 (hex 201).

Let r denote the original number of available extended memory registers (from line 02), and let t denote the total number of usable extended memory registers (from line 23). At line 24, the T register contains r+1 (from line 06), and X contains t. Line 25 computes r+1-t for use after line 32.

Lines 26-27 put the hex code 2A 00 01 69 0B DF FF in 190, transforming the first file in extended memory into a synthetic 4095-register data file. As explained in the analysis of the "VER" program, this file size is accepted as valid by the HP-41 and allows access to all of extended memory.

Lines 27-28 restore the original value to register 191. This is the name of the first file in extended memory. (If extended memory was originally empty, this will be the 1-register "blank" named file.)

Lines 30-31 put the first file name in an otherwise clear alpha register. This will be used at line 35. Lines 32-34 test the value of r+1-t. If it is 1 (the only possible positive value, signifying empty extended memory at the start of "IN"), line 34 converts the integer part to 2. This avoids destruction of the partition register at line 40, as will be explained shortly.

At line 35, X contains r+1-t, normally a negative value. Since SEEPKTA ignores both the sign and the fractional part of the number in X, the number in X is effectively t-r-1. Line 35 selects the register number, within the 4095-register data file, of the second header register of the file created at line 05.

There is an exception if extended memory was empty at the start of "IN". In that case, the second record in the 4095-register data file, register 188, is selected. This avoids selecting register 189, which contains the partition code, so that line 40 does not have to be skipped.

Line 36 puts a code in M for the trailing bytes of the synthetic zero-byte program file. The byte count is zero and the file size is 1. (Normal program files have at least 3 bytes, for the END instruction.)

Lines 37-38 test whether extended memory was empty at the start of "IN". In this case, register 190 must not be restored to its original (zero) value. Instead, register 190 must be left with the header for the synthetic zero-byte program file. Line 38 accomplishes this by replacing the former contents of register 190, which are currently held in LASTX, with the required header information, so that the restoration of this register at line 42 does not need to be skipped.

Lines 39-40 construct the second header register, hex code is 10 00 00 00 000 001, for the synthetic program file, then write it into extended memory at the correct position. If extended memory was empty at the start of "IN", location 188 (register 2) is written, which is harmless because it lies beyond the new file and its partition code.

Lines 41-42 restore the contents of location 190, which has now changed 3 times. Lines 43-44 restore the former contents of c. The X<> c instruction is used instead of STO c so that valid alpha data is left in X at the conclusion of the program. This avoids the remote possibility of a crash due to execution of an arithmetic function on a non-normalized number with leading zero nybbles.

Now here is the bug-free version of "IN". The original "IN" used an illegal file size to cause the HP-41 to store the partition code in register 191 and return the number of usable extended memory registers at line 23. This version takes a more straightforward and less hazardous approach. It stores the partition code in register 191 rather than setting up an illegal file size. Then EMDIR returns the number of usable extended memory registers without any unpleasant side effects.

```
01 LBL "IN"
02 EMDIR or EMROOM
03  E
04 hex FE 20 2C 01 69 00 13 F0 01 89 00 30 03 00 02
05 CRFLD
06 +
07 RCL N
08 X<> c
09 RCL 04
10 REGMOVE
11 hex F2 7F 2A (append one asterisk)
12 RDN
13 RCL 12
14 STO 06
15 hex F7 7F 00 01 69 0B DF FF
16 X<> M
17 STO 12
18 STO 01
19 X<> O
20 STO 03
21 X<> N
22 ENTER↑
```

```
23 hex F7 FF FF FF FF FF FF FF
24 X<> M
25 STO 02
26 RDN
27 EMDIR or EMROOM
28 CLD
29 ST- T
30 X<>Y
31 STO 02
32 R↑
33 X>0?
34 E↑X
35 SEEKPTA
36 hex F1 01
37 X>0?
38 ASTO L
39 ASTO X
40 SAVEX
41 X<> L
42 STO 01
43 R↑
44 X<> c
45 END
          100 bytes
```

Lines 01-15 of this version of "IN" are the same as the original.

Lines 16-18 recall the hex code 2A 00 01 69 0B DF FF from M and store it in both register c and location 190. This puts the curtain at hex 0BD = decimal 189 and transforms the first file in extended memory into a synthetic 4095-register data file. The hex code 10 2C 01 69 00 13 F0 is left in M.

Lines 19-20 restore the repaired contents of register 192. Again, if this register was empty, the lone F0 byte will be eliminated by the

HP-41 the next time it is turned on. Lines 21-22 move the name of the first file (from register 191) onto X and Y.

Lines 23-25 store the partition code, FF FF FF FF FF FF FF, in register 191. This is to fool the calculator into thinking that extended memory is empty. Lines 26-29 of this version of "IN" work the same as lines 22-25 of the original, storing r+1-t in T.

Lines 26-27 put the hex code 2A 00 01 69 0B DF FF in 190, transforming the first file in extended memory into a synthetic 4095-register data file. As explained in the analysis of the "VER" program, this file size is accepted as valid by the HP-41 and allows access to all of extended memory.

Lines 30-31 restore the original value to register 191. This is the name of the first file in extended memory. (If extended memory was empty, this will be the 1-register "blank" named file.) Lines 32-45 are identical to the first version of "IN".

"WFL"/"RFL"/"RPF" by Clifford Stern (XFME, page 205)

Lines 01-12 set up the flags to indicate which program is being executed. The flag settings for each program are:

	Flag 01	Flag 05	Flag 06
"WFL"	either	clear	clear
"RFL"	either	clear	set
"RPF"	clear	set	don't care

For "WFL" and "RFL", Flag 01 is set for compact ASCII file storage mode, clear for normal (complete file: ASCII, program, or data) storage mode.

The LBL 02 section first checks that there is a file name in alpha.

Lines 13-15 generate an error message if alpha is empty, and lines 16-19 generate an error message if alpha contains a comma. Then line 20 gets the file size.

Lines 21-23 left-justify the file name, with trailing blanks as needed, in M. After line 25, the file name is in Y (in non-normalized form) and the file size is in X.

Lines 26-27 branch to LBL 04 if the normal mode (Flag 01 clear) was selected, indicating that the entire file is to be transferred. This also includes "RPF", which transfers the entire program file to main program memory.

If the compact mode of "WFL" or "RFL" was selected (ASCII files only), flag 25 is set. It will be used to detect the end of file. For "RFL", lines 29-30 branch to LBL 04 with flag 25 set, bypassing the LBL 03 section. For "WFL", X and the file pointer are zeroed and the LBL 03 section is executed.

The LBL 03 section counts the characters in the text file, keeping a running total in X. The loop is repeated (lines 38-39) until the end of file is reached. Lines 40-41 account for the start-of-record bytes (one for each record), except for record zero. The record 0 start byte and the end of file byte (FF) are still not counted.

So why do lines 42-43 add 8 rather than 2? Simple. The extra 6 ensures that after division by 7, the integer part of the result is the same as if we had rounded up the fractional part of the actual byte count. Thus the line 45 result is the minimum number of registers needed to hold the entire contents of the text file, at 7 bytes per register.

In the LBL 04 section, line 47 sets the SIZE to the file size, except in the case of "WFL" with flag 01 set for compact storage. In that case, the SIZE is set to the computed number of registers needed to store the text file contents. At this point, flag 25 is clear unless the compact mode of "RFL" was selected.

Line 48 loads hex 00 00 01 69 00 13 F0 in N, and 01 89 00 30 03 00 02 (the number 189.003003) in M. Lines 49-51 lower the curtain to 001 and save the former register c contents in status register a. This is the reason why you can't call these programs from a subroutine. Lines 52-53 move absolute registers 190, 191, and 192 to LASTX, M, and N, respectively. Register 192 is normalized.

Lines 54-56 shift the first byte of register 192 into O and replace it with F0. Line 57 shifts alpha left 6 bytes, leaving the repaired register 192 in O, register 191 in N, and the hex code 2A 00 01 69 0B DF FF in M.

Lines 58-60 clear the P register, recall the former register 191 contents to the stack. Line 61 brings the 7-byte file name (from line 24) into X. Lines 62-64 compare the specified file name with the contents of register 191, then set flag 07 if the named file is the first file in extended memory, and clear flag 07 otherwise.

Lines 65-67 first store the hex code 2A 00 01 69 0B DF FF in c, setting the curtain to 0BD = decimal 189, then store the same code in register 190, temporarily transforming the first file in extended memory into a 4095-register data file so that all of extended memory can be accessed.

Lines 68-69 restore the repaired contents of register 192. At this point the stack contents are entirely disposable. The old value of register c is kept in register a, the SIZE is set to match the number of registers required to store the file, M contains the name of the desired file, and flag 07 specifies whether that file is the first file in extended memory. Register N contains a copy of register 191, the name of the first file in extended memory, and LASTX contains a copy of register 190.

The LBL 05 section moves through the chain of file headers in extended memory, to locate the desired file name. Lines 70-72 put -2 in Y and the former contents of register 190 (from line 53) in X. Lines 73-74 skip to LBL 06 if the desired file is the first file in extended memory.

Otherwise line 75 puts 0 in X. The -2 in Z is the initial value for the record number of the name register within the 4095-register data file. To this value will be added the sizes of the intervening files until the desired file is found. Now the LBL 05 loop is started.

Line 77 stores the current file name in register O. (Note that this is the first 7 characters of alpha, since line 59 cleared P.) The first time through the LBL 05 loop, O is cleared and the name of the first file of extended memory is used from N.

Lines 78-79 take the correct value of register 190 from LASTX and restore it in register 190. This allows the FLSIZE instruction at line 81 to find the current file named in alpha. The R↑ instruction at line 80 brings the current record number count (initially -2) to X. Line 82 adds the current file size without disturbing LASTX. Lines 83-84 add 2 to account for the header registers of the file. Z now designates the register number, within the 4095-register data file, of the first header register (the name register) of the next file in extended memory.

Lines 85-86 clear the current file name from O, exposing the name of the first file in extended memory in register N as the leading characters in alpha. Lines 87-88 store the hex code 2A 00 01 69 0B DF FF in register 190, again temporarily transforming the first file in extended memory into a 4095-register data file so that all of extended memory can be accessed.

Line 89 brings the current register number back to X. Lines 90-94 select this register, recall the first header register without normalization, and compare it with the name of the desired file. If the names do not match, the desired file has not been found and the LBL 05 loop is repeated. Note that the register number is in Z again, and the current file name is in X at this point.

The LBL 05 loop moves down the extended memory directory, file by file, accumulating a record count until the name register for the desired file is found. When a match is found at line 93, X and Y contain the name of the desired file, lines 95-96 put the desired file name in N and the name of the first file in extended memory in M.

Lines 97-98 bring a copy of the second header register of the desired file into X and put the current register number (corresponding to that second header register) in Y. The 4095-register data file is still the working file.

Note that if flag 07 was set, M and N are the same (the first file is the desired file), the second header register of that file is in X, and the register number -2 is in Y. The 4095-register data file is still the working file. Thus, the register contents are consistent whether or not the desired file is the first file in extended memory.

Line 100 stores the second header register of the desired file in O. Lines 101-103 shift this value one byte to the left and replace the leading byte (containing the file type code) with hex 20. This change, when stored, will effectively transform the designated file into a data file. Lines 104-106 rotate the altered second header register back into O. Lines 107-108 re-select the register within the 4095-register data file corresponding to the second header register of the desired file.

Line 109 brings the second header register of the desired file into X. This value is kept in the stack so that it can be restored later. The restoration is done for "RFL" and "WFL" only. "RPF" leaves the file type as Data.

Lines 110-111 clear register O and bring the modified (data type) second header register of the desired file into X. If flag 07 is clear, indicating that the desired file was not the first file in extended memory, the SAVEX at line 113 replaces its second header register with the modified header register. If flag 07 is set, indicating that the desired file was the first file in extended memory, the SIGN at line 115 replaces the copy of register 190 in LASTX with the modified second header register.

Lines 116-118 restore the contents of register 190 from LASTX, replacing the 4095-register data file header with the correct final value. Lines 119-120 restore the correct value of c. Line 121 sets flag 25 for two reasons. First, if "WFL" or "RFL" is used only for the purpose of transferring the file to or from main memory, a card reader is not needed. Setting flag 25 prevents an error stop at line 132 or 126 when no card reader is present. Second, flag 25 prevents an error stop in case the value for the SEEKPT at line 201 is not allowable.

Lines 122-123 skip to LBL 07 if flag 05 is set, indicating that "RPF" is being used. Otherwise TONE 8 sounds. For "RFL", flag 06 is set and lines 125-128 read the data from magnetic cards into the data registers, then save the data registers into the designated file. For "WFL", flag 06 is clear and lines 129-132 transfer the data from the extended memory file into the data registers, then write the data registers onto magnetic cards (if a card reader is plugged in).

Line 133 clears flag 25 in case it remains set after steps 126 or 132. (As good programming practice, we would have to clear flag 25 at the end of the program anyway, so we might as well do it at the earliest possible opportunity.)

Line 135 again sets the curtain to decimal 189, and also stores the hex code 2A 00 01 69 0B DF FF in location 190, changing the first file in extended memory to a 4095-register data file. Lines 136-137 clear register N, exposing the name of the first file in extended memory in M as the leading characters in alpha. Lines 138-139 select the register number corresponding to the second header register of the desired file. This pointer value was last used at line 108. Just as at line 109, line 140 brings the original second header register of the desired file into X.

If flag 07 is clear, indicating that the desired file is not the first file in extended memory, the SAVEX at line 142 restores its original second header register, and the LASTX at line 144 puts a copy of the contents of register 190 in X. (Note that lines 117-118 copied LASTX into register 190.)

Line 145 restores the original value to register 190, regardless of whether flag 07 is set or clear. Lines 146-149 restore the original value of register c, put the current SIZE in X, and halt.

The LBL 07 section is used by "RPF". At this point the program file has been converted to a data file, so that its registers can be easily retrieved. Register c contains its original value, flag 25 is set, and the desired file name is in N as the leading 7 characters of alpha.

Lines 151-152 get the program byte count and store it in data register 00. Lines 153-157 isolate the last 2 bytes of register c in M. Lines 158-166 convert the .END. pointer (the rightmost 9 bits in M) to a decimal value. These lines are the same as lines 37-45 of "LBX" from "HP-41 Synthetic Programming Made Easy".

Line 167 copies the decoded .END. location eee into Y. Line 168 loads hex 10 69 into M. To this will be attached a usable curtain pointer and .END. pointer, so that the final value can be put in register c for temporary use.

The curtain will be set to the .END. location. Because of step 2 of the "RPF" instructions, the .END. is alone in bytes 2, 1, and 0 (the rightmost three bytes) of its register.

Lines 169-172 compute 16*eee for use as a temporary curtain pointer, and add 2 for use as the most significant digit of the temporary .END. pointer. Lines 173-174 convert the computed decimal value into two bytes, which are appended to the hex 01 69 code in alpha. The LBL 10 subroutine does most of this work, in a simple and straightforward manner.

Now that the needed computations are finished, lines 175-176 move the program's byte count from data register 00 to LASTX for later use. Lines 177-179 append a null to alpha and zero the record pointer in what is now a data file. Line 180 changes the SIZE to 000. Since the SIZE was set at line 47 to the file size, which is the number of registers needed to hold the program, the new .END. location is exactly this number of registers above the old .END. location. These registers below the new .END., which is also alone in its register, will be used to store the retrieved program. The retrieved program will start in the register containing the new .END., and a replacement .END. will be stored at the old location.

Lines 181-182 store the hex code 00 00 01 69 ee e2 00 in register c, setting the curtain to the old .END. location. Line 183 gets the file size in registers. This number is the relative register address of the new .END. location, and the register number in which the first retrieved program bytes will be stored.

Line 184 puts the hex code C0 00 2D in M. Thus M contains the bytes for a permanent .END. (right-justified in its register). The status of this .END. is non-private, compiled, but not packed. Line 185 copies the initial relative register number into O.

Lines 186-188 store the intermediate value of c (after resizing to 000 at line 180) in the stack and in register N, for later changing of its .END. pointer to match the original value. The original .END. location eee is brought into X. Lines 189-190 store the new .END. at the old .END. location, and leave the old .END. location eee in M.

The LBL 08 section uses the relative register number in O as a DSE counter, retrieving each program register and storing it in program memory, until the end of file and the location of the newly stored .END. are simultaneously reached.

At the conclusion of this loop the entire program file has been stored in program memory. One detail remains to be taken care of. The program file includes a trailing checksum byte plus additional possibly non-zero bytes following the stored END instruction, that are not part of the program. These bytes need to be zeroed.

The last program register has already been stored, but a copy is still in X. The byte count of the program can be used to identify the location of the checksum byte so that it can be cleared, along with any remaining bytes in the register. Then the corrected register can be stored in relative address 01 as the last register of the program.

Lines 196-199 put a copy of the program register from the file in M, the old .END. location in T, the intermediate (SIZE 000) c register contents in Z, the program's byte count in Y, and 7 in X. Lines 200-202 set the data file pointer to the byte count modulo 7, so that "RPF" can be used again on the file, which is now a data file. ("RPF" only needs the modulo 7 remainder of the byte count to locate the trailing checksum byte.) Flag 25 is still set at line 201, in case you had a very short program for which the byte count modulo 7 equals or exceeds the file size. In that case the SEEKPT fails, which would cause the program to halt prematurely if flag 25 were not set. After this step, flag 25 is no longer needed and it is cleared.

Lines 203-205 subtract 7, specifying a rightward rotation by 7 - (byte count modulo 7) bytes. Since the byte count does not include the checksum byte, this rotation shifts the checksum byte and any trailing bytes of the register off the right end of M. Lines 206-210 append nulls to replace the bytes that were shifted off the end of M.

Lines 211-213 bring the intermediate c register contents into X, swap it into M and put the corrected final program register in the proper location (relative address 01). Now all that remains is correction of the .END. pointer value in the intermediate c register contents.

Lines 214-216 remove the 2 rightmost bytes of the intermediate c register value. This includes the least significant nybble of the curtain pointer, which must be 0 because the SIZE is 000. Line 217 brings the old .END. location into X. Lines 218-219 convert this value into two bytes and appends those bytes to alpha.

The final c register value is then extracted from M and stored. The program concludes with a BEEP.

"ASG"/"PASG"/"MKX" by Tapani Tarvainen and Gerard Westen (XFME, page 212)

Lines 01-43 of "ASG" prompt for the function name, then the key to which the function is to be assigned. This procedure is designed to appear much like the built-in ASG function. After line 43, the function (prefix and postfix, separated by a space) are in alpha, and the row/column keycode is in X.

Lines 44-154 of "PASG" decodes the alpha representation of the function to be assigned into decimal prefix and postfix byte values. This is done by using PASN to assign the prefix function, then taking the hex code for the function out of the R (append) register. The suffix is decoded directly by the program.

Once the prefix and postfix bytes are in decimal form, the "MKX" portion of the program makes the assignment, using essentially the same approach as the "MKX" program in "HP-41 Synthetic Programming Made Easy". PASN makes a dummy assignment, then the program locates this assignment in the key assignment registers and replaces it. Since PASN cannot assign the shifted shift key, neither can these programs.

Lines 01-05 save the current flag settings in LASTX, and use a synthetic text instruction to set flags 3, 40, 45, and 48, clearing all others. This sets FIX 0, ALPHA mode, and the system data entry flag (45). Setting flag 45 allows input to be directly appended to the alpha register when the program halts at line 07. (Unfortunately, no underscore prompt is provided.)

When the program is restarted after line 07, the alpha register contains "ASN prefix postfix". Line 08 is necessary because if a printer is present, flags 55 and 21 will be set when the program halts. Line 09 appends another space, to separate the prefix and the postfix from the keycode that will follow (just as for the normal ASN display).

The LBL 01 loop displays the assignment information and waits for the user to press a key to which the function will be assigned. If X=0 at line 14, no key was pressed and the loop is repeated. If X=Y at line 16, the shift key was pressed. In that case a hyphen is appended to alpha, flag 03 is cleared to indicate a shifted key, and the LBL 01 loop is repeated to identify the shifted key to which the function will be assigned.

If the shift key is pressed a second time, the two hyphens must be removed from alpha. Lines 21-25 accomplish this. Then flag 03 is set to indicate an unshifted key.

When the key for the assignment is pressed, the GTO 02 at line 17 is executed. Lines 29-30 append the keycode to alpha and display the assignment information. Lines 31-32 make the keycode in X negative if a shifted key was specified.

If an unshifted key is being assigned, lines 33-34 append a byte to alpha. This makes the keycode portion of the assignment information in alpha 4 characters long, regardless of whether a shifted or unshifted key was assigned.

Lines 35-36 restore the original flags. Lines 37-38 remove the two leading characters "AS" from alpha, leaving 83 in X. Lines 39-41 rotate alpha to the right 4 bytes, putting a space and the keycode portion of the assignment information at the front of alpha. Line 42 removes these 4 bytes, the "N" of ASN, and the space that follows the N. Alpha now contains just the prefix, a space, and the postfix. Line 43 brings the keycode back into X.

Lines 44-47 locate the first space (the one that follows the prefix). If a space is found, the program skips to LBL 03. If no space is present, then only a prefix was specified, and the normal PASN function suffices to make the assignment.

The LBL 03 section appends a null as a separator, then line 56 rotates the prefix to the right end of alpha. Line 57 appends two characters to alpha. First, the character @ is used as another separator. As a character that is not normally permitted in labels or funtion names, it marks the end of the prefix name. The character 0 is added to provide a numeric value for the ANUM at line 84, in case both the postfix and prefix are non-numeric.

Line 58 removes the space at the front of alpha. Lines 59-60 check whether another space is present. A second space is taken to mean that an INDirect function has been specified. In that case, a flag must be set or cleared as an indicator and the "IND " characters must be removed.

If a second space is present (POSA >0) the AON step at line 61 is skipped. Thus, flag 48 (ALPHA mode) clear indicates an INDirect function, and flag 48 set indicates a normal direct function.

Line 62 has no effect if a second space is not present, because X was incremented from -1 to 0 at line 60. If a second space is present, line 62 rotates the characters "IND " to the right end of alpha. For example, if we are assigning STO IND N, alpha will now contain "N¯STO@0IND ".

Lines 63-64 attempt to locate a comma in alpha. If a comma is present, the program assumes that the function was of the form "XROM xx,yy". Lines 67-80 then extract xx and yy, converting these values to the decimal byte equivalents for the XROM xx,yy function.

Line 67 rotates xx to the right end of alpha. Line 68 brings the keycode back into X. Lines 69-70 put the value of yy in X and remove it from alpha. Line 71 puts the value of xx in X, pushing yy into Y. Lines 72-76 compute 64*(640+xx)+yy, which is the decimal equivalent for the 2-byte function XROM xx,yy. Lines 77-81 break this value into decimal equivalents for the prefix and postfix bytes, then skip to LBL 06. As a side effect, the keycode is duplicated into T.

The LBL 04 section handles two-byte functions other than XROM functions. Line 83 brings the keycode back into X. If a numeric postfix is present, line 84 puts the postfix value in X. Note that a numeric prefix with a non-numeric postfix is not allowed. So after line 84 we have either the postfix value or 0 (from line 57) in X.

Line 85 decodes the first character from alpha. In case the postfix was non-numeric, this decoded postfix character will be converted into the correct decimal byte value.

Lines 86-87 subtract an offset of 84, so that T corresponds to 0. If the result in X is less than or equal to zero, the postfix is numeric, L, M, N, O, P, Q, R, or T. In this case, we skip to LBL 04. If X is negative (all postfixes but T), lines 100-103 add 9. If the result, at line 104, is greater than zero, the postfix is L, M, N, O, P, Q, or R. The + at line 107 adds the ANUM result from Y, which we now know to be zero.

If the result at line 104 is not greater than zero, the postfix is either numeric (negative number in X) or T (zero in X). Lines 109-110 add another offset of 17, so that for a numeric potfix X will still be negative. For a postfix of T, X will be 17. If the postfix is T, lines 111-114 add 95, obtaining the correct decimal equivalent of the T potfix byte, 112. Lines 115-116 put the decimal equivalent for the postfix in Y. If the postfix is numeric, lines 115-116 leave the correct decimal postfix equivalent (the ANUM result from line 84) in Y.

Now let's go back and consider the case in which the number in X at line 88 is greater than zero. In this case, the postfix is X, Y, Z, a, b, c, d, or e, or append. The number in X ranges from 4 (X) to 43 (append). Lines 90-92 subtract this number from 7, giving a result ranging from -36 (append) to 3 (X). The current values for postfixes Z, Y, and X are 1, 2, and 3, repectively, adjacent to the 0 value for the postfix T at this point. Lines 93-94 skip to LBL 05 for postfixes X, Y, and Z, handling these the same as T and getting the decimal postfix equivalents 113, 114, and 115.

For postfixes a, b, c, d, or e, or append, lines 95-97 give results 5, 6, 7, 8, 9, or 10 for append, a, b, c, d, e. The append code is correctly adjacent to the postfix a code. Lines 98-107 add 2, then 3, to get 10 for append and 15 for e. Lines 108-114 add 17 and 95, to get 122 for append and 127 for e. These are the correct decimal postfix equivalents. Lines 115-116 leave the postfix equivalent in Y.

The section from line 84 to line 116 is probably the trickiest in this program, and it's totally nonsynthetic! To write such compact, multipurpose sets of instructions requires practice and persistence. Often you need to solve the problem several times from different approaches, then see which solution is the shortest.

After line 117, X is zero, the decimal postfix equivalent is in Y, and the keycode is in Z. Lines 117-120 locate the null that separates the remains of the postfix in alpha from the prefix, then rotate the prefix to the front of alpha. (Actually the null is rotated to the front of alpha, but then in that position it disappears.)

Lines 121-122 clear all flags, copying the old flag register into the stack for later restoration. Line 123 brings the keycode to X, then lines 124-125 attempt to assign the prefix to the designated key. This will succeed if the prefix was a valid function name, and not a numeric input.

If the PASN succeeded, byte 2 of Q (the fourth byte from the left) will be the assigned function prefix. Line 127 stores this value in P, so that the function byte becomes the first character in alpha.

Lines 128-130 put the keycode in Y and restore the old flag register. The most recent flag register value is brought into X. If this value is zero, the PASN at line 125 must have failed, clearing flag 25. If not, the PASN worked and line 132 gets the correct decimal prefix code for the function from alpha.

The ASHF at line 133 removes the P register portion of alpha, leaving the prefix followed by "@" as the leading characters in alpha. If the PASN failed, line 135 brings the numeric prefix vaule into X.

After line 135, X contains the decimal prefix code, Z contains the keycode, and T contains the decimal postfix code. Line 136 rotates the postfix code into X. If flag 48 is set, indicating a normal direct function, the program jumps to LBL 06.

If flag 48 is clear, indicating an indirect function, more work is needed. For most indirect functions, 128 needs to be added to the postfix code. However, if the user specified GTO IND or XEQ IND, things are more complicated. For GTO IND, the prefix GTO was assigned, with a decimal code of 208. In that case the prefix needs to be changed to 174 and the direct postfix can be left unchanged. For XEQ IND, the prefix XEQ was assigned, with a decimal code of 224. (Note that 224 is the only possible PASN-derived prefix code greater than 208.) In this case the prefix is 174 and 128 must be added to the postfix.

Line 139 swaps the postfix code with flags 0-7. Line 140 brings the keycode into X, and lines 141-142 put 208 in Y and the prefix code in X. If these do not match, that is for all indirect functions except GTO, the high bit (flag 07) of the postfix must be set. For GTO IND, the postfix is left unchanged. All this happens without disturbing the stack. This is quite a creative use of the capabilities of X<>F.

Lines 145-146 test whether X (the prefix) is greater than or equal to Y (208). If so, we have the XEQ IND case. Then lines 147-149 replace the prefix in X with 174. Lines 150-151 bring flags 0-7 back to X and restore them. The postfix value is brought into X.

The LBL 06 section clears flag 48 and brings the keycode into X, pushing the prefix and postfix values into Z and Y, respectively.

The "MKX" section works much like the "MKX" program in "HP-41 Synthetic Programming Made Easy". First the function "ANUM" is assigned to the key. This assignment will replace the assignment that may have been made at line 125. Lines 158-160 get the keycode from the rightmost byte of R (append) and store it in P. Line 161 appends A6 42, which are the two bytes of the ANUM function. After this step, the keycode byte kk is the leftmost character of alpha.

Line 162 stores hex 10 kk 00 00 00 00 00 in N, and line 163 appends hex kk 00 00 00 00 00 to alpha. Lines 164-167 append the prefix byte and the postfix byte to alpha, pushing hex A6 42 kk into N. These three bytes are the same as those in the temporary ANUM assignment, and they will be used to locate that assignment. When the assignment is located, the A6 and 42 bytes will be replaced by the just-formed prefix and postfix bytes.

After line 167, register P is empty and register O contains hex 10 kk 00 00 00 00 00. Line 169 recalls hex 00 00 00 00 A6 42 kk from N. Line 169 removes the hex 10 from O. Lines 170-171 rotate the keycode to the right end of alpha, appending it to the prefix and postfix bytes in M. The number 16 is put in LASTX. Line 172 puts the assembled 3 bytes for the new assignment in X. The three bytes for the ANUM assignment are in Z.

Lines 173-175 bring the old value of c into the stack, and store hex 00 40 01 69 0B 02 00 in M and c. This puts the curtain at hex 0B0 = decimal 176. Line 176 brings the three bytes for the ANUM assignment into X.

The LBL 07 loop searches for the ANUM assignment. Line 178 recalls the current register (starting with decimal 192, the first key assignment register). Lines 179-182 put hex 2A 2A at the right end of O, and the current register in both N and M. Line 183 puts hex 10 2A 2A F0 aa bb cc in M, where aa bb cc are the three assignment bytes in the left half of the key assignment register. Lines 184-185 bring hex 00 00 00 00 dd ee ff, the second set of assignment bytes from the current register, into X. Hex 2A 2A is placed in N, right-justified. Line 186 leaves just aa bb cc in alpha.

If dd ee ff does not match the ANUM assignment, it is switched with aa bb cc from M. If there is a match at line 189, line 190 brings the new assignment to X, replacing the ANUM assignment bytes.

Line 191 left-justifies the non-matching assignment in M, and line 192 stores the assignment from X next to it. Line 193 appends hex 84 84 84 F0, left-justifying the 6 assignment bytes in N. Lines 194-196 put the F0 byte in O next to the 6 assignment bytes. Then lines 196-198 shift the reconstructed assignment register left 6 bytes into O, and store it in the current register. If the ANUM assignment was not found in this register, lines 199-203 increment the register number (relative to 182) in LASTX, then go back to LBL 07. Otherwise lines 204-209 restore the original value of c, clear the stack, alpha, and the display, and halt.

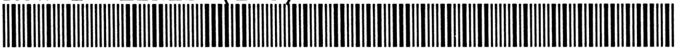
If you have followed through the line-by-line analysis of all these synthetic programs, you are well equipped to write your own programs using these powerful techniques. Good luck!

APPENDIX D -- BARCODE FOR PROGRAMS

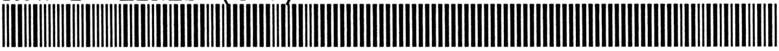
This barcode was created on an HP Laserjet^{tm} printer using Ken Emery's Barcode Generating ROM, available mid-June 1987 from SYNTHETIX. If you have access to either a Laserjet^{tm} or a Thinkjet^{tm} printer, you can now instantly and economically produce beautiful barcode for your programs in any size you want. With barcode, you can swap programs by mail with anyone that has an optical wand. You can even use barcode as the ultimate backup memory system! Check the store where you bought this book, or write to SYNTHETIX, P.O. Box 1080, Berkeley, CA 94701-1080, U.S.A. for price information on the new Barcode Generating ROM.

PROGRAM:ReNFL 16 REGISTERS PROGRAM USES 9 ROWS

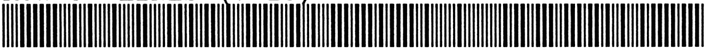
ROW 1 LINES (1-3)



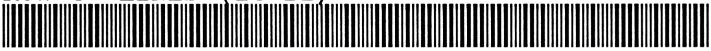
ROW 2 LINES (4-7)



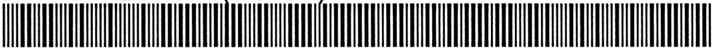
ROW 3 LINES (7-13)



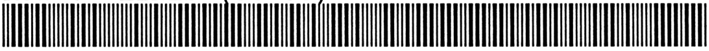
ROW 4 LINES (14-21)



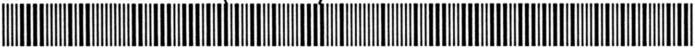
ROW 5 LINES (21-29)



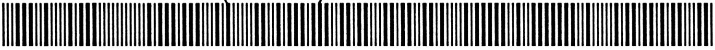
ROW 6 LINES (30-38)



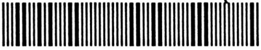
ROW 7 LINES (39-45)



ROW 8 LINES (46-54)

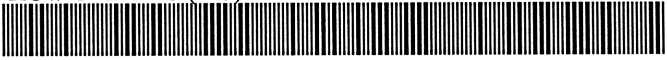


ROW 9 LINES (55-55)

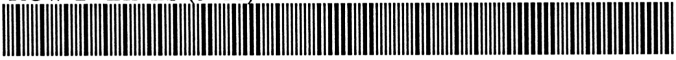


PROGRAM:CB 10 REGISTERS PROGRAM USES 6 ROWS

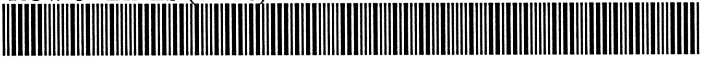
ROW 1 LINES (1-5)



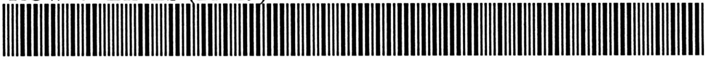
ROW 2 LINES (5-11)



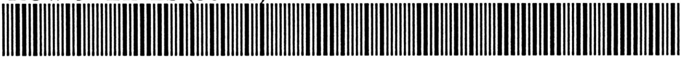
ROW 3 LINES (11-20)



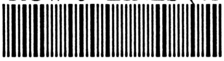
ROW 4 LINES (21-29)



ROW 5 LINES (30-41)

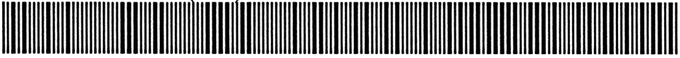


ROW 6 LINES (41-41)

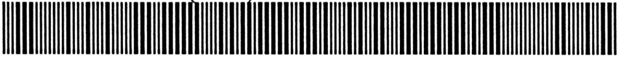


PROGRAM:FE 4 REGISTERS PROGRAM USES 2 ROWS

ROW 1 LINES (1-5)

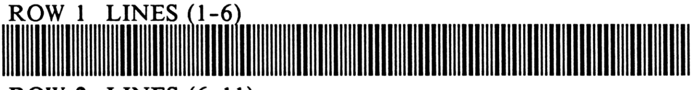


ROW 2 LINES (5-11)

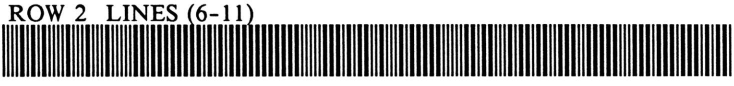


PROGRAM:2B 5 REGISTERS PROGRAM USES 3 ROWS

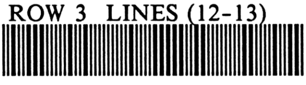
ROW 1 LINES (1-6)



ROW 2 LINES (6-11)

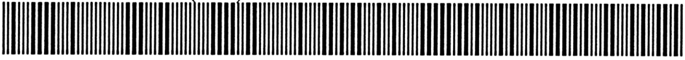


ROW 3 LINES (12-13)

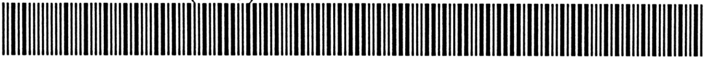


PROGRAM:CX 8 REGISTERS PROGRAM USES 4 ROWS

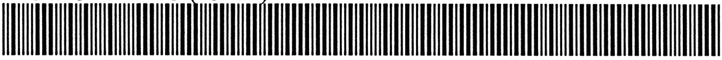
ROW 1 LINES (1-6)



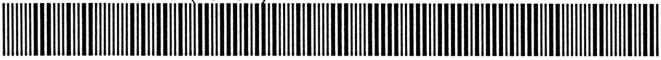
ROW 2 LINES (6-15)



ROW 3 LINES (15-24)

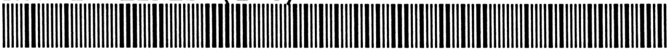


ROW 4 LINES (25-31)

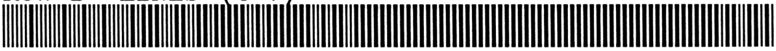


PROGRAM:RENFL 16 REGISTERS PROGRAM USES 9 ROWS

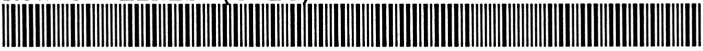
ROW 1 LINES (1-4)



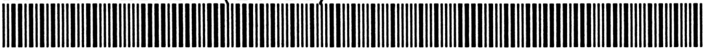
ROW 2 LINES (4-7)



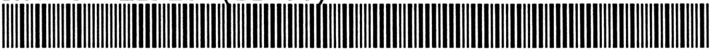
ROW 3 LINES (8-14)



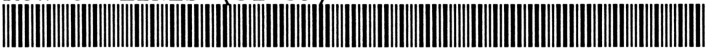
ROW 4 LINES (14-21)



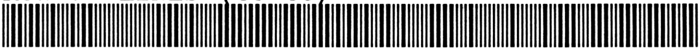
ROW 5 LINES (22-30)



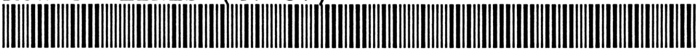
ROW 6 LINES (31-39)



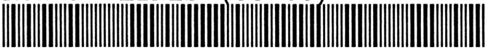
ROW 7 LINES (40-46)



ROW 8 LINES (47-57)

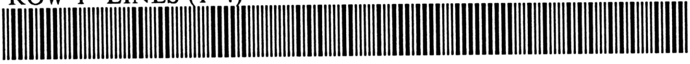


ROW 9 LINES (58-60)

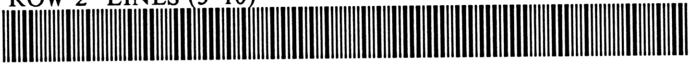


PROGRAM:T0N 4 REGISTERS PROGRAM USES 2 ROWS

ROW 1 LINES (1-4)

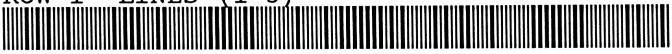


ROW 2 LINES (5-10)

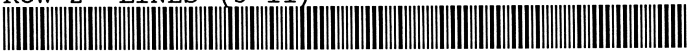


PROGRAM:PK 27 REGISTERS PROGRAM USES 15 ROWS

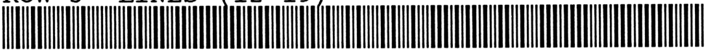
ROW 1 LINES (1-3)



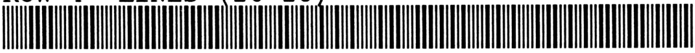
ROW 2 LINES (3-11)



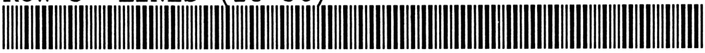
ROW 3 LINES (12-19)



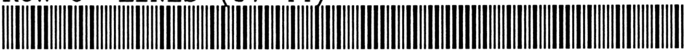
ROW 4 LINES (20-28)



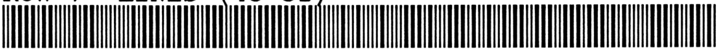
ROW 5 LINES (28-36)



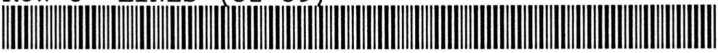
ROW 6 LINES (37-44)



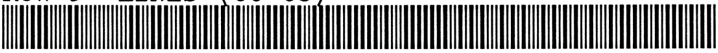
ROW 7 LINES (45-52)



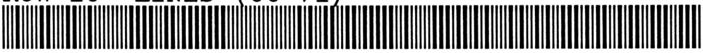
ROW 8 LINES (52-59)



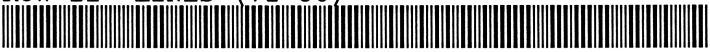
ROW 9 LINES (60-65)



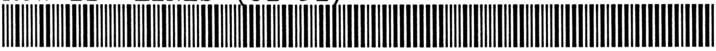
ROW 10 LINES (66-71)



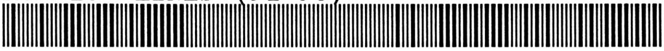
ROW 11 LINES (72-80)



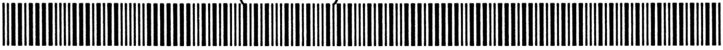
ROW 12 LINES (81-91)



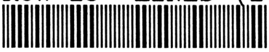
ROW 13 LINES (91-95)



ROW 14 LINES (95-99)

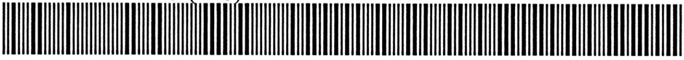


ROW 15 LINES (100-100)

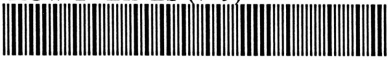


PROGRAM:MT 3 REGISTERS PROGRAM USES 2 ROWS

ROW 1 LINES (1-6)

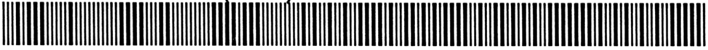


ROW 2 LINES (7-9)

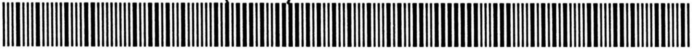


PROGRAM:ROM^MS 23 REGISTERS PROGRAM USES 13 ROWS

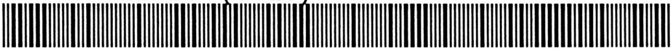
ROW 1 LINES (1-2)



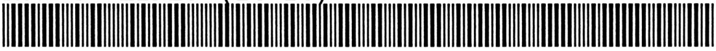
ROW 2 LINES (2-6)



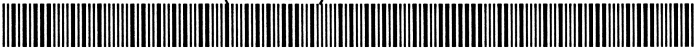
ROW 3 LINES (7-13)



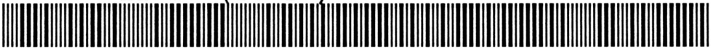
ROW 4 LINES (14-20)



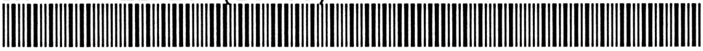
ROW 5 LINES (20-25)



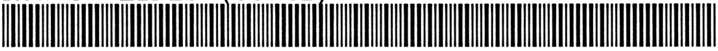
ROW 6 LINES (25-27)



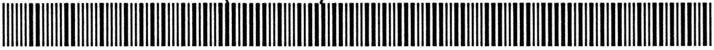
ROW 7 LINES (28-35)



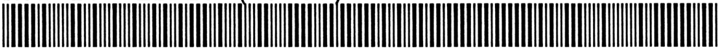
ROW 8 LINES (35-42)



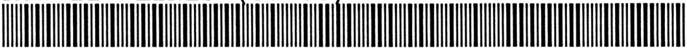
ROW 9 LINES (42-49)



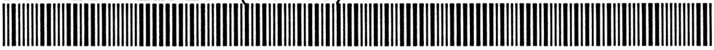
ROW 10 LINES (49-57)



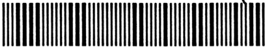
ROW 11 LINES (57-60)



ROW 12 LINES (60-64)

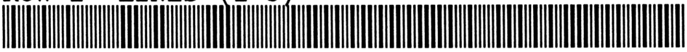


ROW 13 LINES (65-65)

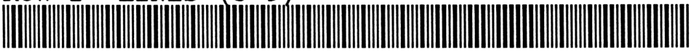


PROGRAM:MKZ 15 REGISTERS PROGRAM USES 8 ROWS

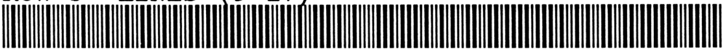
ROW 1 LINES (1-3)



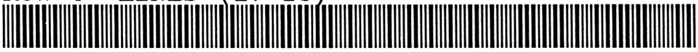
ROW 2 LINES (3-9)



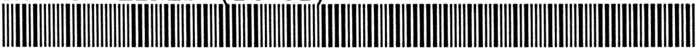
ROW 3 LINES (9-17)



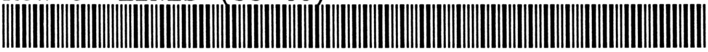
ROW 4 LINES (17-23)



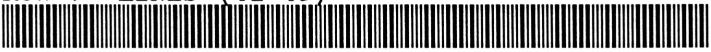
ROW 5 LINES (24-32)



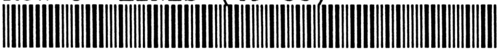
ROW 6 LINES (33-40)



ROW 7 LINES (41-49)



ROW 8 LINES (49-53)



PROGRAM:RAMBC 16 REGISTERS PROGRAM USES 9 ROWS

ROW 1 LINES (1-4)

ROW 2 LINES (4-9)

ROW 3 LINES (9-16)

ROW 4 LINES (16-20)

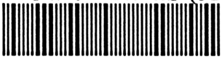
ROW 5 LINES (21-26)

ROW 6 LINES (26-32)

ROW 7 LINES (33-42)

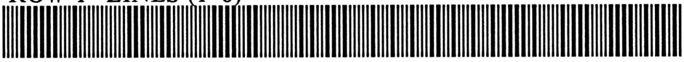
ROW 8 LINES (42-52)

ROW 9 LINES (52-52)

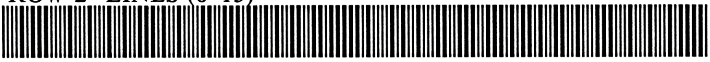


PROGRAM:SA 19 REGISTERS PROGRAM USES 11 ROWS

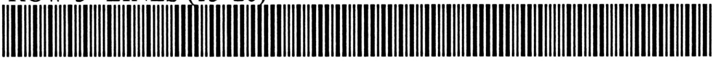
ROW 1 LINES (1-6)



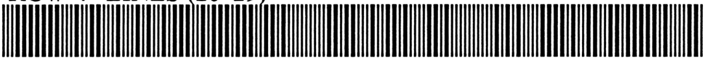
ROW 2 LINES (6-13)



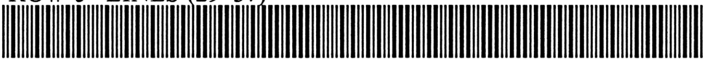
ROW 3 LINES (13-20)



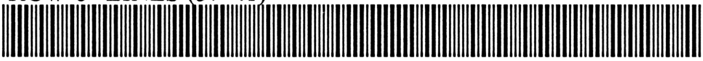
ROW 4 LINES (20-29)



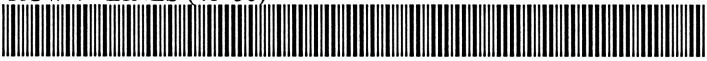
ROW 5 LINES (29-37)



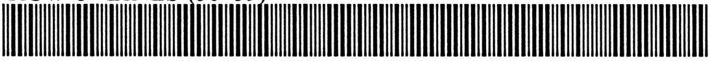
ROW 6 LINES (37-41)



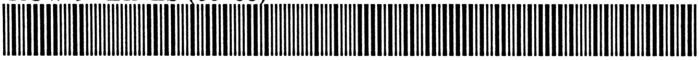
ROW 7 LINES (41-50)



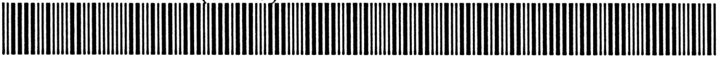
ROW 8 LINES (50-59)



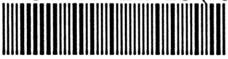
ROW 9 LINES (60-68)



ROW 10 LINES (69-75)

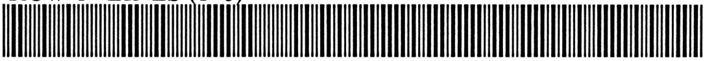


ROW 11 LINES (75-75)

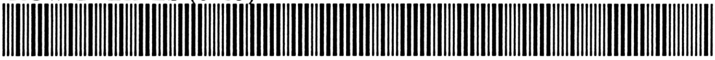


PROGRAM:SK 7 REGISTERS PROGRAM USES 4 ROWS

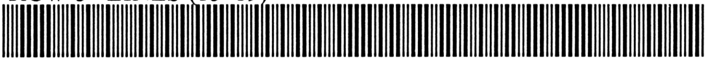
ROW 1 LINES (1-6)



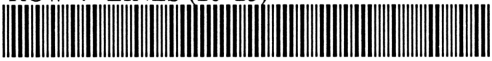
ROW 2 LINES (6-13)



ROW 3 LINES (13-19)

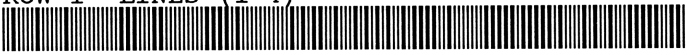


ROW 4 LINES (20-23)

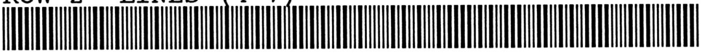


PROGRAM:VER 5 REGISTERS PROGRAM USES 3 ROWS

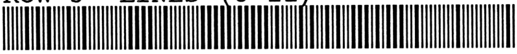
ROW 1 LINES (1-4)



ROW 2 LINES (4-7)

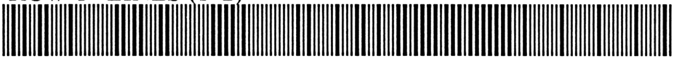


ROW 3 LINES (8-11)

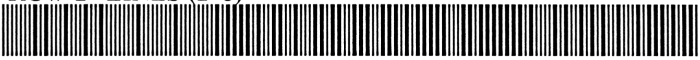


PROGRAM:PFF 5 REGISTERS PROGRAM USES 3 ROWS

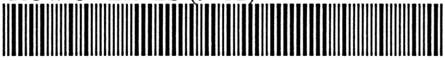
ROW 1 LINES (1-2)



ROW 2 LINES (2-8)

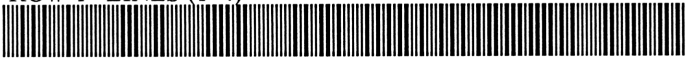


ROW 3 LINES (9-12)

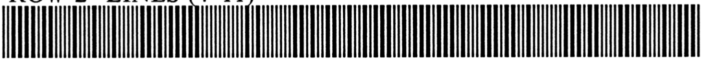


PROGRAM:SAVEK 22 REGISTERS PROGRAM USES 12 ROWS

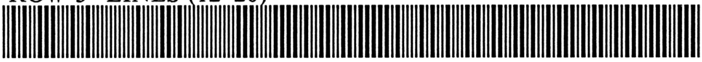
ROW 1 LINES (1-4)



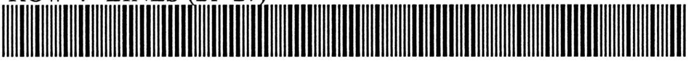
ROW 2 LINES (4-11)



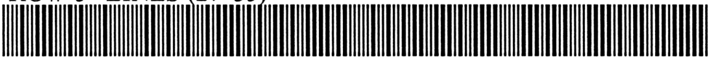
ROW 3 LINES (12-20)



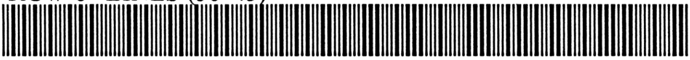
ROW 4 LINES (21-27)



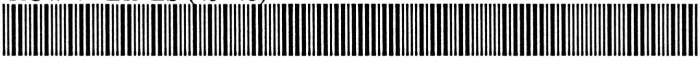
ROW 5 LINES (27-35)



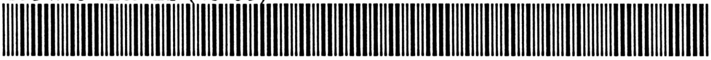
ROW 6 LINES (36-43)



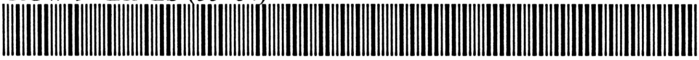
ROW 7 LINES (43-48)



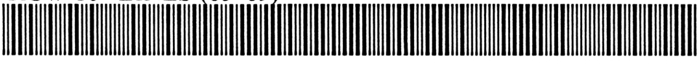
ROW 8 LINES (48-55)



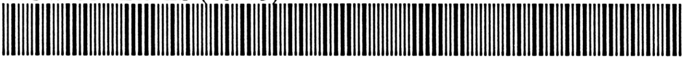
ROW 9 LINES (55-64)



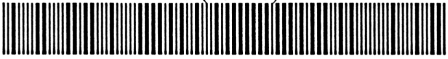
ROW 10 LINES (65-69)



ROW 11 LINES (70-75)

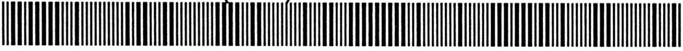


ROW 12 LINES (75-78)

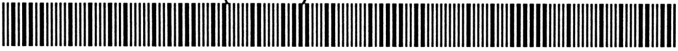


PROGRAM:F? 27 REGISTERS PROGRAM USES 15 ROWS

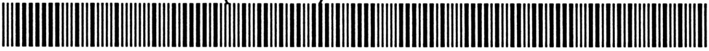
ROW 1 LINES (1-5)



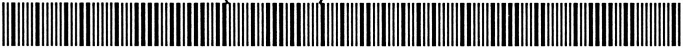
ROW 2 LINES (5-11)



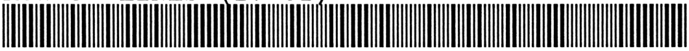
ROW 3 LINES (11-20)



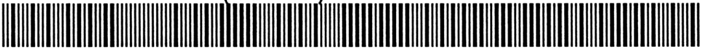
ROW 4 LINES (21-27)



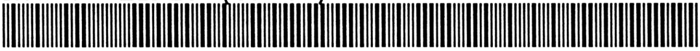
ROW 5 LINES (27-32)



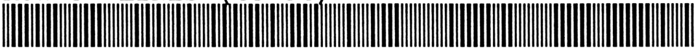
ROW 6 LINES (32-38)



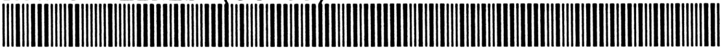
ROW 7 LINES (39-47)



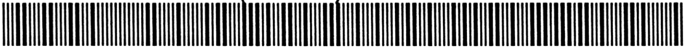
ROW 8 LINES (48-54)



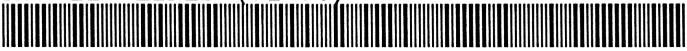
ROW 9 LINES (54-60)



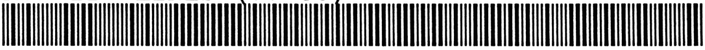
ROW 10 LINES (61-70)



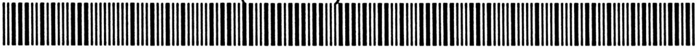
ROW 11 LINES (71-76)



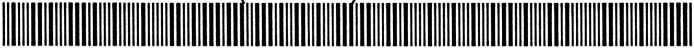
ROW 12 LINES (77-83)



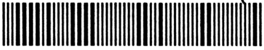
ROW 13 LINES (84-94)



ROW 14 LINES (94-103)

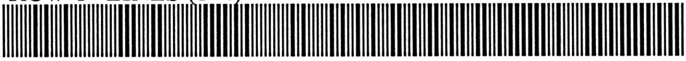


ROW 15 LINES (104-104)

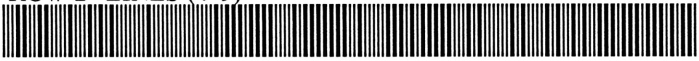


PROGRAM:PRFL 5 REGISTERS PROGRAM USES 3 ROWS

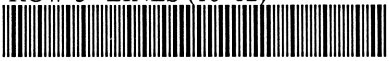
ROW 1 LINES (1-4)



ROW 2 LINES (4-9)

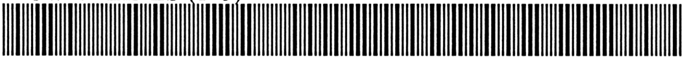


ROW 3 LINES (10-12)

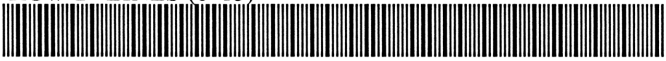


PROGRAM:IN 8 REGISTERS PROGRAM USES 4 ROWS

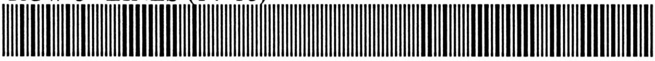
ROW 1 LINES (1-5)



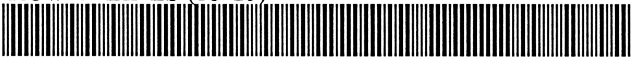
ROW 2 LINES (6-13)



ROW 3 LINES (14-18)

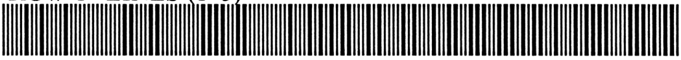


ROW 4 LINES (18-25)

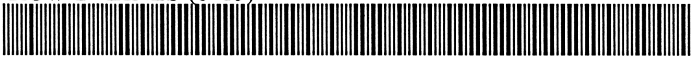


PROGRAM:EX 5 REGISTERS PROGRAM USES 3 ROWS

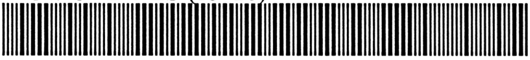
ROW 1 LINES (1-5)



ROW 2 LINES (6-15)

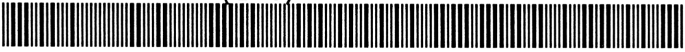


ROW 3 LINES (15-20)

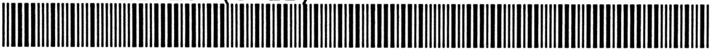


PROGRAM:TRI∢ 65 REGISTERS PROGRAM USES 35 ROWS

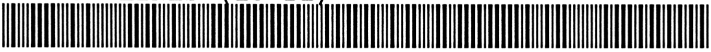
ROW 1 LINES (1-4)



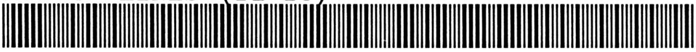
ROW 2 LINES (5-12)



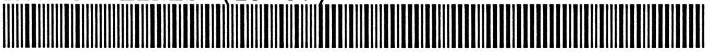
ROW 3 LINES (13-21)



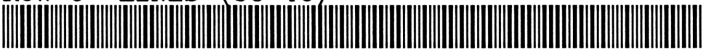
ROW 4 LINES (21-28)



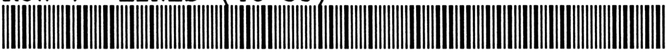
ROW 5 LINES (29-37)



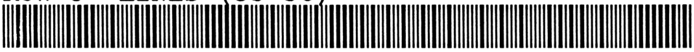
ROW 6 LINES (38-46)



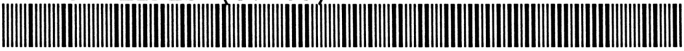
ROW 7 LINES (46-53)



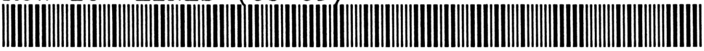
ROW 8 LINES (53-56)



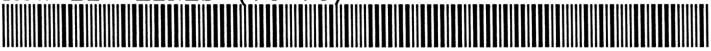
ROW 9 LINES (57-63)



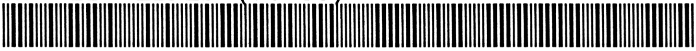
ROW 10 LINES (63-69)



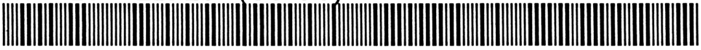
ROW 11 LINES (70-76)



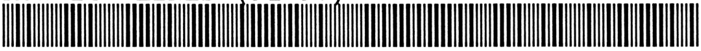
ROW 12 LINES (76-83)



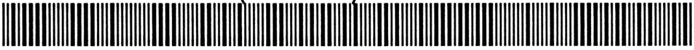
ROW 13 LINES (84-90)



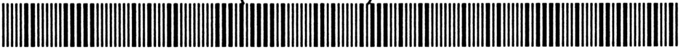
ROW 14 LINES (91-97)



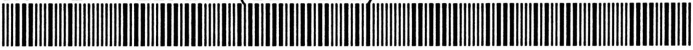
ROW 15 LINES (97-105)



ROW 16 LINES (106-112)

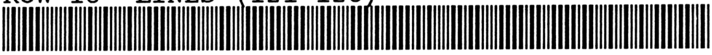


ROW 17 LINES (113-121)

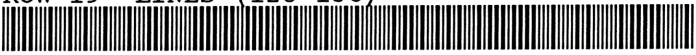


PROGRAM:TRI∡

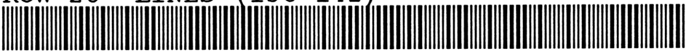
ROW 18 LINES (121-128)



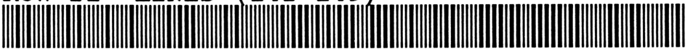
ROW 19 LINES (128-136)



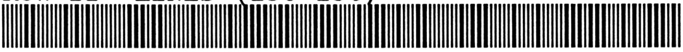
ROW 20 LINES (136-142)



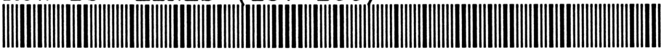
ROW 21 LINES (142-149)



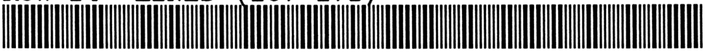
ROW 22 LINES (150-156)



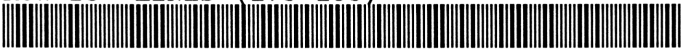
ROW 23 LINES (157-166)



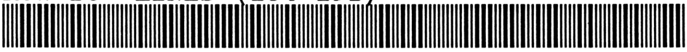
ROW 24 LINES (167-172)



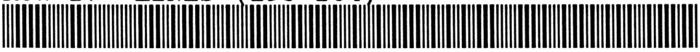
ROW 25 LINES (173-183)



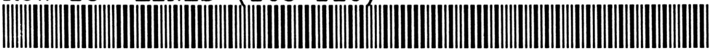
ROW 26 LINES (184-192)



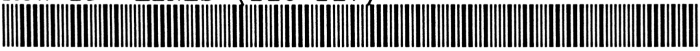
ROW 27 LINES (193-204)



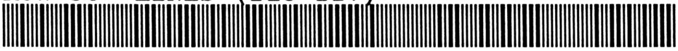
ROW 28 LINES (205-210)



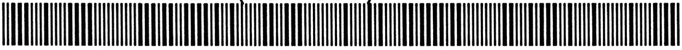
ROW 29 LINES (210-217)



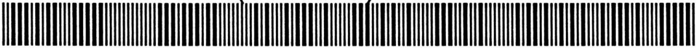
ROW 30 LINES (218-227)



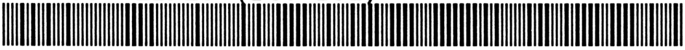
ROW 31 LINES (228-236)



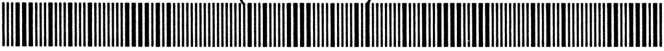
ROW 32 LINES (237-245)



ROW 33 LINES (246-251)



ROW 34 LINES (252-259)



ROW 35 LINES (259-262)

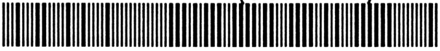


FIGURE AND TABLE INDEX

STATUS REGISTER INDEX

INDEX

PROGRAM INDEX

NAME	BYTES	SECTION	PAGE	BARCODE
TRI⊦	448	2E	51-65	337
ReNFL	107	3F	100	325
CB	67	3G	106	326
SHR	14	4B	128	----
FE	24	4E	142	326
2B	30	4H	163	326
MT	19	5A	182	328
CX	51	5A	185	327
PK	185	5B	191	328
RENFL	112	5B	197	327
T0N	26	5C	205	327
DP	24	5C	208	----
PD	32	5C	211	----
E?	22	5C	211	----
C?	28	5C	212	----
ROM↑MS	159	5C	218	329
MKZ	99	6A	222	330
RAMBC	106	6A	225	331
SA	132	6A	228	332
SK	47	6A	232	333
VER	35	6B	236	333
PFF	33	6B	237	333
PRFL	32	6B	238	336
SAVEK	150	6B	240	334
IN	50	6B	244	336
EX	35	6C	246	336
F?	185	6C	248	335

ORDER BLANK

	Price per copy	Qty	Amount
For HP-71'S			
HP-71 Basic Made Easy, by Joseph Horn	$18.95	_____	__________
For HP-71'S & HP-41'S			
Control the World with HP-IL, by Gary Friedman	$24.95	_____	__________
For HP-41'S			
HP-41 Advanced Programming Tips, by A. McCornack & K. Jarett	$20.95	_____	__________
HP-41 M-Code for Beginners, by Ken Emery	$24.95	_____	__________
Inside the HP-41, by Jean-Daniel Dodin	$12.95	_____	__________
Extend Your HP-41, by W. Mier-Jędrzejowicz	$29.95	_____	__________
HP-41 Extended Functions Made Easy, by Keith Jarett	$16.95	_____	__________
HP-41 Synthetic Programming Made Easy, by Keith Jarett (Includes one Quick Reference Card)	$16.95	_____	__________
Quick Reference Card for Synthetic Programming	$2.00	_____	__________
Synthetic Quick Reference Guide (SQRG)	$5.95	_____	__________
For HP-10C, 11C, 15C, AND 16C			
ENTER (Reverse Polish Notation Made Easy), by J.Dodin	$4.95	_____	__________
Humor			
It's Amazing How These Things Can Simplify Your Life: The Harold Guide to Computer Literacy	$4.95	_____	__________
ROM's			
Barcode Generating ROM by Ken Emery	$199.95	_____	__________
AECROM by Redshift Software	$ 99.00	_____	__________
Sales tax (California orders only, 6 or 7%)			__________

Shipping	1st book	Add'l books
within USA, book rate (4th class)	$1.50	$0.50
USA 48 states, United Parcel Service	$2.50	$1.00
USA, Canada, air mail	$3.00	$1.50
elsewhere, book rate (6 to 8 week wait)	$2.00	$1.00

elsewhere, air mail $12.05 for **Extend Your HP-41**, $6.05 for others

Free shipping for **ENTER** and **It's Amazing...** with purchase of any other book

Free shipping for QRC plastic cards or SQRG (any number)

Free shipping for ROM's

Enter shipping total here $__________

Total due $__________

Checks must be in U.S. funds, and payable through a U.S. bank.

Name____
Address____
____
City______________________State__________Zipcode__________
Country____

Mail to:
SYNTHETIX, P.O.Box 1080, Berkeley, CA 94701-1080, USA Phone (415) 339-0601

CAN'T STOP PROGRAMMING?

This book introduces you to the tools available for advanced programming and gives you a detailed tutorial on programming technique. Next are some important Synthetic Programming techniques, several of which are not described in other books. These include TEXT 0 prefix key assignments, Catalog 1 crash recovery techniques, and making programs PRIVATE without a magnetic card reader or HP-IL.

Users of the ZENROM or CCD utility modules will especially enjoy the application programs for these powerful modules. In addition to being useful programs in themselves, these examples also illustrate how to harness some of the most powerful programming features of both the ZENROM and the CCD Module. The book includes an overview of HP-41 Machine code (M-code), so you can get an idea of whether this exciting area is for you.

Those of you who have read Keith Jarett's earlier books "HP-41 Synthetic Programming Made Easy" and "HP-41 Extended Functions Made Easy" will find a special treat in the appendix: Complete line-by-line analysis of all the synthetic programs from both books! Now you can see how Clifford Stern and the other synthetic programming masters do it.

If you like to push your HP-41 to the limit, or perhaps a little beyond, this is the book for you.

ISBN 0-9612174-6-4