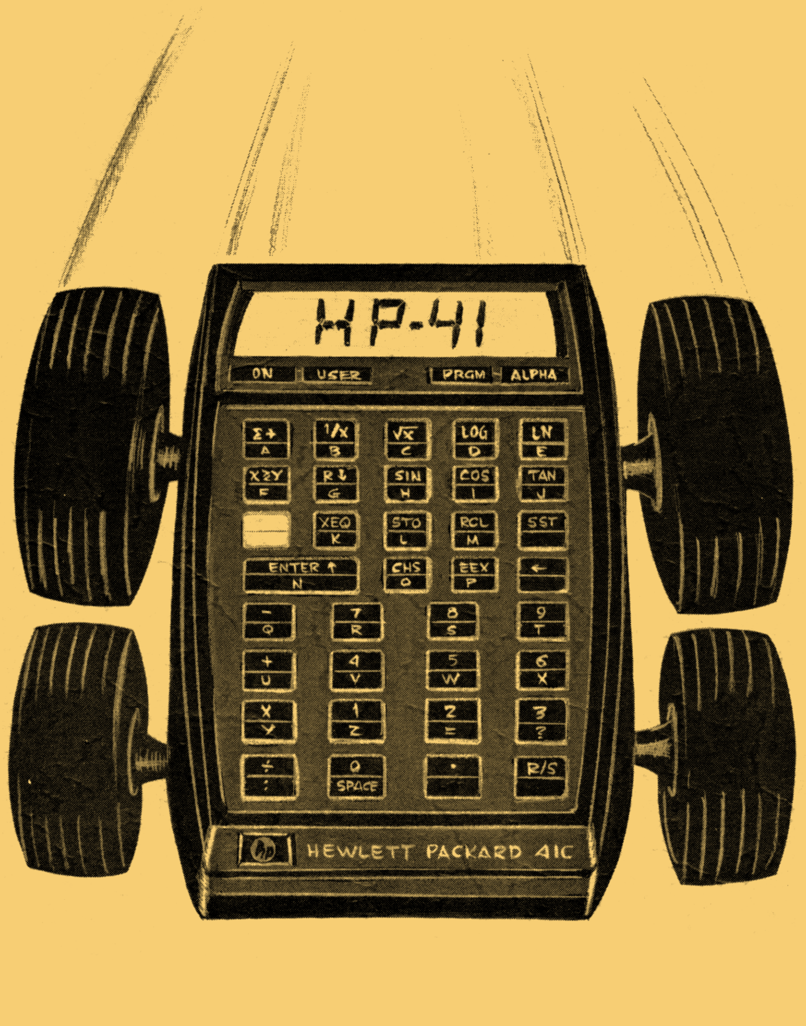
HP-41 SYNTHETIC PROGRAMMING MADE EASY

by Keith Jarett



To Richard J. Nelson,

Thanks for your help and support.

Keith Jarrett

HP-41 SYNTHETIC PROGRAMMING MADE EASY

By Keith Jarett

1540 Mathews Ave.
Manhattan Beach
CA 90266 USA

Printed in the United States of America.

Acknowledgement: This book would not have been possible without the existence of PPC, the users group that has fostered the development of synthetic programming since the 1979 introduction of the HP-41C. Several members of PPC have made direct contributions to the recently developed techniques in this book.

Most of these contributions were made by Clifford Stern, one of the handful of "grand masters" of synthetic programming. Clifford was the technical consultant for this book, developing several programs specifically for use here and spotting errors during several rounds of editing.

Many other members of PPC contributed indirectly through their own discoveries and developments that advanced synthetic programming over the last three years. Richard Nelson, the founder of PPC, deserves a large measure of recognition. He has single-handedly kept PPC alive for 8 years through untiring effort.

I dedicate this book to my wife, Catherine Van De Rosytne, who has patiently endured my HP-41 addiction, and who provided invaluable help throughout the preparation of this book.

Request for Errata: Errata and specific suggestions for improving this book are welcome. Mail them to: Keith Jarett, SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, CA 90266 USA. If your suggestion proves usable, I will mail you a plastic **Quick Reference Card for Synthetic Programming.** If you send me a complete set of corrections, I will mail you a new copy of **HP-41 Synthetic Programming Made Easy.**

The plastic **Quick Reference Card for Synthetic Programming** on the back cover is an indispensable tool for synthetic programming. Its use is described in Chapter 1. For further description see Appendix D and Appendix C, item 10.

For price information on this book, write to: SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, CA 90266 USA. Enclose an addressed return envelope for faster reply. Dealer and distributor inquiries are welcome.

TABLE OF CONTENTS

INTRODUCTION

WHAT IS SYNTHETIC PROGRAMMING?

Have you ever wondered why the HP-41 doesn't allow more than ten different TONEs? Or perhaps you have wondered why you can't store and recall numbers from the ALPHA register, or why parentheses are not available as display characters. **HP-41 SYNTHETIC PROGRAMMING MADE EASY** will teach you to overcome these limitations and add a whole new set of functions to your HP-41's vocabulary. Examples of added capability are:

- -Techniques you can use to make your programs faster, shorter, or to reduce their SIZE requirement
- -Three to six extra "scratchpad" stack-like registers for general use
- -21 additional display characters including parentheses, quotation marks, ampersand, and others
- -Over 100 additional TONEs
- -Enhanced alpha string editing ability
- -Suspension and reactivation of USER mode key assignments
- -Simultaneous setting of all 56 system and user flags to any desired state
- -Renumbering of data registers under program control to eliminate register usage conflicts between subroutines.

The creation and use of synthetic instructions is called synthetic programming. Synthetic instructions are those which cannot be entered from the keyboard by normal means. Thousands of synthetic instructions are possible. These range from non-standard TONEs to powerful instructions that access system scratch registers. Synthetic programming will not harm your HP-41 in any way, although the annoyance of occasional "crashes" (temporary keyboard lockup and/or MEMORY LOST) is to be expected as you are learning. Synthetic programming will work on all calculators in the HP-41 family, including the

HP-41C and CV, regardless of date of manufacture. It depends only on fundamental aspects of the calculator's internal operating system that are common to all HP-41's.

As a simple example of the beauty of synthetic programming, consider the two short programs listed below. The one on the left is a standard, nonsynthetic program to print out the message "Hewlett-Packard". It occupies 40 bytes of program memory (more about bytes in Chapter 1). The program on the right uses a synthetic instruction to do the same thing in only 20 bytes, exactly half the space. In this example, which you will encounter in more detail in Section 2E, synthetic programming overcomes the lack of direct access to lowercase printer characters on the HP-41.

```
NONSYNTHETIC:

01 "H"
02 ACA
03 SF 13
04 "EWLETT-"
05 ACA
06 CF 13
07 "P"
08 ACA
09 SF 13
10 "ACKARD"
11 ACA
12 PRBUF
13 CF 13
14 END
```

```
Programs to print
the message

Hewlet -Packard
```

```
SYNTHETIC:

01 "Hewlett-Packard"
02 AVIEW
03 END
```

```
                CAT 1
END      40 BYTES
END      20 BYTES
```

You need not become an expert to reap the benefits of synthetic programming. Armed with the knowledge and confidence provided by this book, you can quickly and easily create and run any synthetic program from the HP User's Library, the PPC Calculator Journal[1], or any other source. Also covered are the most frequent applications of synthetic programming, so that you may customize your own programs with synthetic instructions.

This book is designed to provide an easy, practical introduction to synthetic programming on the HP-41. It uses the latest simplified synthetic programming techniques in a "hands on" approach that makes it easy and fun to try the examples on your calculator as you read.

The scope of **HP-41 SYNTHETIC PROGRAMMING MADE EASY** is intentionally limited, in order to provide the most readable introduction to synthetic programming. Details are often bypassed, but references are given for those readers who wish to learn more about them. The casual synthetic programmer will be able to learn all he needs from this book. For others this book is a ticket of admission to the growing body of synthetic programming literature. It has all the framework you need to build your knowledge of synthetic programming.

If you own a PPC ROM[2], your progress through the book can be speeded up by using its advanced features such as synthetic key assignment and byte-loading programs. If you have just the calculator you will sometimes need to follow slightly more elaborate instructions to "bootstrap" your system to full synthetic capability. Either way it's fairly simple.

Hewlett-Packard does not support synthetic programming. Although many individuals in HP's Corvallis Division have some familiarity with synthetic programming, HP does not have the manpower to answer questions about synthetic programming from users. So please don't ask HP about synthetic programming. Just read this book and continue into the other sources of information (Appendix C) for answers to your questions.

The most important benefit you'll get from **HP-41 SYNTHETIC PROGRAMMING MADE EASY** is access to all published synthetic programs. Many synthetic programs, especially those in the PPC ROM, perform functions that can't be duplicated by any nonsynthetic program. After you have read this book, synthetic programs will no longer seem mysterious and forbidding. There are hundreds of powerful synthetic programs in the PPC Calculator Journal and elsewhere that will give your HP-41 capabilities you probably never dreamed of.

[1] The PPC Calculator Journal (PPC CJ) is a publication of Personal Programming Center (PPC), a non-profit public benefit California Corporation dedicated to personal computing. PPC has several thousand members, most of whom are fellow HP-41 enthusiasts. PPC members have been responsible for virtually every discovery in the field of synthetic programming, beginning with the first description of synthetic programming by William C. Wickes in the PPC CJ in 1979. The PPC Calculator Journal continues to be the primary source for the latest information on synthetic programming. To find out how you can get the PPC CJ, see Appendix C.

[2] The PPC ROM is a custom ROM plug-in module for the HP-41, designed by PPC members and manufactured by Hewlett-Packard. It contains 122 programs, most of which are usable as subroutines in your own programs, and most of which contain synthetic instructions. The manual is an astounding 492 pages long and has probably not been fully read by any one person. See Appendix C to find out how you can get the PPC ROM.

CHAPTER ONE

CREATING YOUR FIRST SYNTHETIC INSTRUCTION

A decimal (base 10) number xyz has the value $x \cdot 10^2 + y \cdot 10 + z \cdot 1$, where x, y, and z are any digits from 0 to 9. Similarly a binary (base 2) number $qrst_2$ (the subscript 2 indicates base 2) has the value $q \cdot 2^3 + r \cdot 2^2 + s \cdot 2 + t$, where q, r, s, and t are digits from 0 to 1. q is the "eights" digit, r is the "fours" digit, and so on. For example $1011_2 = 8+2+1 = 11$, and $11111111_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2 + 1 = 128+64+32+16+8+4+2+1 = 255$.

A hexadecimal (base 16) number uv_{16} has the value $u \cdot 16 + v$, where u and v are hexadecimal digits from zero to fifteen. Since there aren't any ordinary digits that correspond to the numbers ten through fifteen, it is standard notation to borrow them from the alphabet: $A_{16} = 10$, $B_{16} = 11$, $C_{16} = 12$, $D_{16} = 13$, $E_{16} = 14$, $F_{16} = 15$. For example $C5_{16} = 12 \cdot 16 + 5 = 197$, and $FF_{16} = 15 \cdot 16 + 15 = 255$. Incidentally, the shorthand "hex" will be used throughout this book. It means the same as hexadecimal or base 16.

If you are not familiar with base 2 and base 16 number systems, read the last two paragraphs again and give them a little thought. Like the rest of this chapter, it should all begin to fall together after a couple of readings. Hang in there, because we're going to start having some fun by the end of this chapter.

The basic unit of program memory in the HP-41 is called a byte. A byte is a collection of eight bits (binary digits) that can range in value from 00000000 base 2 to 1111111 base 2, or equivalently from 0 to 255 base 10. Although a byte can take on only 256 distinct values, there are thousands of distinct HP-41 instructions. The STO and RCL instructions alone have more than 400 variations. This variety is acheived by allocating more than one byte for some types of instructions. Simple instructions like +, LOG, and MOD occupy only one byte of program memory. Instructions like VIEW 14, RCL 99, and ΣREG IND X require two bytes -- one for the function name, or prefix, and the second one for the suffix. A few types of instructions require three bytes, while text lines require up to 16 bytes (for a 15 character text line).

Synthetic instructions can be created by removing prefix bytes from two-byte instructions, using a simple procedure described in this chapter and the next. As you shall see in the examples in this chapter and the next, the removal of a prefix frees the suffix byte, which can in turn become a prefix and attach itself to the following byte or bytes. By carefully selecting which instructions we start with, we can force a wide variety of synthetic instructions to appear after the original prefix byte is removed. To remove prefixes we use a workhorse key assignment called the "byte grabber", discovered by Erwin Gosteli after some pioneering work by Jack Baldrige. Incidentally, both Erwin and Jack are members of PPC, and their discoveries appeared in the PPC Calculator Journal (see Appendix C item 1). In fact, all the people mentioned in connection with discoveries or programs in this book are members of PPC.

Since the byte grabber is not a standard key assignment, a special procedure is required to create it. You are not expected to understand the procedure at this point, so just follow the required steps carefully. Turn your thinking cap back on after you have assigned the byte grabber.

Go get your HP-41 **now,** if you don't already have it in front of you. If you've got any ideas about reading this book first, then trying the examples later, **forget them!** The examples are an essential part of the learning process. Doing the examples will also make the text much easier to follow. When you read "go to line Ø5 and delete it", you won't have to ask yourself what line Ø5 is. Trying the examples as you go may seem to be slowing you down, but it will save you time in the long run because you won't have to read and re-read.

If you have a PPC ROM, skip to step 12.

If you do not have a PPC ROM, you can assign the byte grabber by carefully following an alternate procedure conceived by Keith Kendall. Follow these steps precisely or you'll have to start over from step 1. It may take a few tries to get it right, but be patient.

1. MASTER CLEAR to MEMORY LOST status. This is done by holding down the backarrow key while turning on the calculator, then releasing the backarrow key. There is a more complicated procedure for assigning the byte grabber that doesn't require a MASTER CLEAR, but you should consider this step a rite of initiation to synthetic programming. This certainly won't be the last time you get MEMORY LOST.
2. ASN "+" to the LN key (press: shift ASN ALPHA shift + ALPHA LN). This assignment will be replaced by the byte grabber assignment.
3. ASN "DEL" to the LOG key. (Press: shift ASN ALPHA D E L ALPHA LOG.)
4. Switch to PRGM mode. You should see ØØ REG 45.
5. Start CATalog 1 (still in PRGM mode) and press R/S immediately before the display blinks. Repeat this step if you didn't press R/S quickly enough.
6. Switch to ALPHA mode, then press the backarrow key with the .END. in the display.
7. You should see the program line 4Ø94 RCL Ø1. The origin of this mysterious line number will be explained in Section 6A. A "bug" in the HP-41's internal programming has just allowed you to escape the normal confines of program memory. You are now in the system scratch register area. More about this in Chapter 6, too. Now switch back out of ALPHA mode by pressing the ALPHA switch again.
8. GTO .ØØ5. You can press LN for ØØ5 to save

keystrokes. You should see Ø5 LBL Ø3. You are now in the key assignment area, which will also be covered in Section 6A. The next step is to remove the dummy "+" function assignment and replace it with the synthetic byte grabber assignment. Since the calculator thinks it is still in a program area, this replacement is accomplished by keying in program instructions that correspond to the data needed for a byte grabber assignment. This correspondence is not straightforward, so don't expect to understand it at this stage.

9. DEL ØØ3. You can save several keystrokes by pressing USER (to activate the DEL key assignment that you made to the LOG key), LOG, SQRT (the square root key). You should see Ø4 STO Ø1. You have now deleted the assignment of the + function. Next we replace it by the byte grabber.

1Ø. Key in the text line "?AAAAAA". If you don't have an Extended Functions module plugged in you will see Ø5 "?A‾‾‾‾‾". The last five A's went past the end of memory into what would be the first part of extended memory and appear only as "ghost" characters.

11. Switch out of PRGM mode and GTO.. or do CAT 1 to get out of the key assignment registers. Skip step 12 and go on to the following text.

12. If you have a PPC ROM, or if you are returning after reading Chapter 4 and you already have a copy of "MK" (Make Key assignments), assign the byte grabber using this abbreviated procedure instead of steps 1 through 11 above:

    a.) Clear any Time Module alarms that are present.
    b.) ASN ALPHA ALPHA LN (this clears the LN key of any assignment
    c.) XEQ **MK** or "MK"
    d.) When the PRE↑POST↑KEY message appears, supply

the inputs 247 ENTER↑ 63 ENTER↑ 15 and R/S. When the program stops again, you're done. You can backarrow the PRE↑POST↑KEY message, but it is not necessary.

If you have followed the above procedure carefully, the byte grabber should be assigned to the LN key. But don't try it yet; the byte grabber can be dangerous if you are not careful. If you press LN in USER mode and **hold it down,** you should see XROM 28,63, followed by the message NULL, indicating that the time limit for releasing the key has been exceeded. When the NULL message appears the byte grabber operation is cancelled, and it is safe to release the key. In a few pages you will using the byte grabber, so don't be impatient. A little knowledge now can save a lot of MEMORY LOST later.

If you have a card reader, write a status card (XEQ ALPHA W S T S ALPHA) to record this synthetic key assignment. Then, if you ever get MEMORY LOST, you can read in track 2 of the card to reinstate the byte grabber assignment. It is then OK to just backarrow the prompt for track 1.

NOTE: Whenever you see the notation BG, short for byte grabber, in the following discussion, it refers to the byte grabber assigned key, in this case LN. Unless the text specifies otherwise, the byte grabber key is to be pressed in USER mode and **in PRGM mode**.

WARNING: Don't press BG indiscriminately in PRGM mode. If you press it at or,just above an END, you may need to MASTER CLEAR to restore use of Catalog 1. (The first thing to try is to BST to the line that was displayed before you pressed BG the first time and BG again.) If your keyboard ever "locks up", simply remove the battery pack, and the printer if it is connected, for a couple of seconds and replace it. If that doesn't work, try turning the HP-41 off and on several times with the batteries removed. Pulling out any plug-in modules (especially QUAD MEMORY, XMEMORY, and XFUNCTION modules) may help. It is a very rare crash that requires overnight removal of the batteries.

Now switch into PGRM mode, GTO.., and key in these instructions, which we will be using shortly:

```
Ø1 ENTER↑
Ø2 X<> 88
Ø3 STO IND 31
Ø4 PI
```

Line Ø1 is a normal ENTER↑.

Line Ø2 is obtained by XEQ, ALPHA, X, shift COS, shift TAN, ALPHA, 8, 8. As you may know from reading the Owner's Manual, the HP-41 implements many more functions than could fit on the keyboard. Functions like X<> which are not on the keyboard must be accessed by XEQ, ALPHA, function name, ALPHA. The shifted ALPHA characters, like < and >, are unfortunately not shown on the keyboard. Instead you should look at the sticker on the bottom of your HP-41 to determine which shifted key corresponds to the desired ALPHA character.

In case you haven't used indirect instructions before, line Ø3 is STO, shift, 3, 1. The PI function can be accessed by shift, Ø.

Before using the byte grabber you need to know a little more about bytes. Put the calculator aside for a few minutes while you digest the next two pages.

For synthetic programming, it is often convenient to express the 256 possible values of a byte in hexadecimal (base 16). By splitting the eight bits of a byte into two four-bit groups and converting each four-bit group to a hexadecimal digit we get a two-digit shorthand for the value of a byte. In base 16 the letters A through F designate the numbers ten through fifteen. The equivalence of 4-bit groups to hexadecimal (base 16) digits is:

binary	hex	decimal
ØØØØ	Ø	Ø
ØØØ1	1	1
ØØ1Ø	2	2
ØØ11	3	3
Ø1ØØ	4	4
Ø1Ø1	5	5
Ø11Ø	6	6
Ø111	7	7
1ØØØ	8	8
1ØØ1	9	9
1Ø1Ø	A	1Ø
1Ø11	B	11
11ØØ	C	12
11Ø1	D	13
111Ø	E	14
1111	F	15
1 ØØØØ	1Ø	16

For example Ø1ØØ 11Ø1 base 2 = 4D base 16, and 1111 ØØØ1 base 2 = F1 base 16.

Take out your **HP-41 QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING** (the 2-7/8" by 6" plastic card that comes attached to the back cover of this book) or refer to the full-size byte table provided in Appendix D. The byte table contained in the Quick Reference Card ("QRC") is the Rosetta Stone of Synthetic Programming, illustrating the byte equivalences that are the key to creating synthetic instructions.

The byte is based on the hexadecimal representation rc_{16}, where r is the row number (Ø through F) and c is the column number. Rows Ø through 7 comprise the first half of the byte table; rows 8 through F comprise the second half. At the top of each box in the byte table part of the QRC is the primary function, or prefix, interpretation of that particular byte.

Immediately below is the suffix interpretation. At the bottom of the box is the decimal equivalent for that byte. On the right are display and printer character interpretations of the byte; these will be covered in Section 2E.

As an example consider the ENTER↑ instruction that you just keyed in as line Ø1. Since we find ENTER↑ in the prefix (top) portion of the box at row 8 column 3 of the QRC, we can conclude that ENTER↑ is represented internally as 83 hexadecimal. The bottom row of the box at row 8 column 3 tells you that 83 hexadecimal is equivalent to 131 decimal. You have no immediate use for this decimal equivalent, but you'll find it quite handy when you get to Chapter 3.

Next consider the X<> 88 on line Ø2. We find X<> at row C column E, and 88 in the suffix portion of the box at row 5 column 8. This means that X<> 88, a two byte instruction, represented internally as hexadecimal CE 58, occupying two consecutive bytes. Line Ø3 is STO IND 31. STO appears at row 9 column 1 while IND 31 appears at row 9 column F. Thus STO IND 31 consists of the two consecutive bytes 91 9F. Line Ø4, PI, is represented as hex 72 (row 7 column 2). Note that instruction line numbers are not stored in program memory. The HP-41 actually computes the line number by counting instructions from the top of the program.

Suppose we could somehow get rid of the X<> byte (the hex CE byte) in the X<> 88 instruction. The suffix 88 (hex 58) would be left to "fend for itself", becoming the instruction E↑X-1 (see row 5 column 8 of the QRC).

The byte grabber key assignment allows us to easily get rid of leading bytes in instructions. For this reason it is sometimes referred to as a "prefix masker". The byte grabber always operates on the program step <u>following</u> the one shown in the display, grabbing its leading byte.

Now get out your HP-41 again, turn it on, and verify that your program is still intact by switching to PRGM mode and pressing SST to step through it.

To illustrate the prefix masking behavior of the byte grabber on the X<> 88 instruction, first PACK (XEQ ALPHA P A C K ALPHA). Do not GTO.. , since you want to stay where you are in the program. GTO.. has the undesired effects of attaching an END to your program and "kicking you out" of it. Make sure you are in USER mode, then GTO .ØØ1 (the step before the X<> 88 instuction). Switch to PRGM mode if you are not already in PRGM mode, and BG (press the LN key). You'll see a strange looking text instruction

Ø2 ᵀ‾?‾‾‾‾▓ .

The starburst (all 14 segments lit) at the end of the text line is, or was, the X<> part of the X<> 88. This hex CE byte has been grabbed, leaving the suffix byte to become an instruction on its own. SST and you'll see

Ø3 E↑X-1 ,

precisely as predicted.

Review this example until you feel comfortable with it. Once you have conceptualized the byte structure of memory and the action of the byte grabber (see Figure 1.1), you are over the hump and on your way to some real synthetic programming.

What would happen if we grabbed the STO prefix from the STO IND 31 instruction? According to row 9 column F of the QRC, the IND 31 suffix byte would become a TONE instruction. But wait a minute. The TONE instruction needs a suffix of its own; after all, every TONE is a two-byte instruction. Where will this newly exposed TONE instruction get its suffix? Let's find out. BG at line ØØ3 (GTO .ØØ3 if you are not already there and press LN in PRGM mode) to grab the STO bye. SST to see

Ø5 TONE Y , a synthetic instruction!

A quick check of row 7 column 2 of the QRC reveals that the new TONE prefix captured the PI instruction, transforming it into the suffix Y (see Figure 1.1). It is certainly reasonable that the TONE instruction got its suffix from the next instruction in the program -- it had to get it from somewhere.

You can SST line Ø5 in RUN (non-PRGM) mode to hear your new synthetic tone. BST and SST to hear it again if you like it. There are more than 1ØØ other synthetic TONEs waiting to be explored.

hexadecimal byte value: row	column	program instructions	program instructions after byte grabbing
8	3	ENTER↑	ENTER↑
C	E	X<>	ᵀ‾?‾‾‾‾▓
5	8	88	E↑X-1
9	1	STO	ᵀ‾?‾‾‾‾▓
9	F	IND 31	TONE
7	2	PI	Y

Figure 1.1 Transformation of instructions by byte-grabbing.

CHAPTER TWO

FREQUENTLY USED SYNTHETIC INSTRUCTIONS

This chapter introduces the eight types of synthetic instructions that are most frequently used. Regardless of whether you get involved in writing exotic synthetic programs, you will want to use some of these easily understood instructions in your ordinary day-to-day programming. The types of instructions to be discussed in this chapter are:

A. Synthetic Tones, which personalize your programs;

B. Synthetic Exponential Data Entry Lines ("Short Form Exponents"), which save bytes;

C. Flag Register Control, used to preserve the display setting while constructing PROMPTs;

D. Program Pointer Control, which can freeze the "flying goose";

E. Synthetic Text Lines, used where synthetic characters such as parentheses or lower case letters are needed;

F. The TEXT Ø instruction, equivalent to an HP-25 NOP (No Operation) instruction;

G. Control of data registers "carved out of" the ALPHA register, which provides auxiliary storage for intermediate program results without disturbing the numbered data registers; and

H. Use of other operating system scratch registers for temporary data storage.

As examples of synthetic instructions are presented in this chapter, step-by-step procedures on how to create them will also be given. These procedures will use the byte grabber key assignment that was constructed in Chapter 1. Owners of the PPC ROM have the option of bypassing this procedure and creating the instructions directly using PPC ROM routine **LB** (Load Bytes). The appropriate **LB** inputs will be identified for each example. If the synthetic instruction consists of two bytes and is not a digit entry, PPC ROM routine **MK** can be used in lieu of **LB** if a key assignment of the function is also desired. It is recommended that PPC ROM owners try at least some of the examples in this chapter using the byte grabber instead of **MK** or **LB** .

For those of you without PPC ROMs, a short version of "LB" will be introduced in Chapter 3, along with instructions for using the byte grabber to key it up. You may do so now, but you will learn more about using the byte grabber by waiting until you get to Chapter 3 to key up and use "LB".

2A. Synthetic Tones

As mentioned at the end of Chapter 1, there are over 1ØØ possible synthetic tones of widely varying pitch and duration. Of the 16 distinct tone frequencies, the first ten are the frequencies of TONE Ø through TONE 9. The durations of synthetic tones vary from several milliseconds (tones audible only as a "click") to several seconds. For many prompting applications a relatively short, high-pitched tone is required. TONE 89 is one such tone. It can be created as follows. Delete any leftovers from the Chapter 1 examples and key in these program lines:

Ø1 ENTER↑	**LB** / **MK** inputs:
Ø2 STO IND 31	TONE 89 = 159, 89
Ø3 SIN	

Now, still in PRGM mode, GTO .ØØ1 and BG (press LN in USER mode). As usual, you'll see a text line like this: Ø2 ⊤‾?----▓ . SST to see your new synthetic instruction Ø3 TONE 9 . It may not look synthetic but you'll soon hear the proof that it is.

The IND 31 byte (hex 9F) became a TONE instruction after the STO byte was grabbed. The SIN byte (row 5 column 9 =

decimal 89) became the tone number. Synthetic tone numbers from 1Ø to 1Ø1 decimal are displayed in decimal with only the rightmost (ones) digit shown. Thus in this case TONE 89 displays as TONE 9. Other tones, whose second bytes are between row 6 column 6 and row 7 column F, carry a letter suffix as did TONE Y in the Chapter 1 example.

Switch to RUN mode and SST to hear TONE 89. It may become one of your favorites for prompting.

Table 2.1 summarizes the synthetic tones that are available to you. The frequency of a tone is determined by its column number in the table. The frequencies corresponding to column A,B,C,D,E, and F form an upward progression, with the highest synthetic frequency (column F) being just below that of TONE Ø, the lowest normal frequency.

The duration of each tone, in seconds, is listed in the table. This duration is the total time the HP-41 needs to execute the tone; therefore the actual audio output duration will be significantly shorter for the very brief tones. Durations may vary from those listed depending on when your HP-41 was produced. For example TONE Z is Ø.64 seconds long on newer HP-41's, versus only Ø.Ø61 seconds on the oldest HP-41's.

As you scan the tone table, you'll notice that TONEs 37 and 38 are the shortest, at .Ø2Ø seconds each. The following example illustrates a use for them. Clear the previous example and key in the program lines

```
Ø1 DEG                          LB / MK  inputs:
Ø2 CLX
Ø3 LBL Ø1
Ø4 STO IND 31                   TONE 37 = 159, 37
Ø5 RCL Ø5
Ø6 SIN
Ø7 SQRT
Ø8 STO IND 31                   TONE 38 = 159, 38
Ø9 RCL Ø6
```

10 SIN

11 SQRT

12 GTO 01

GTO .007, BG, and delete the text line. SST to see TONE 8 (actually TONE 38). GTO .003, BG, and delete the text line. SST to see TONE 7 (actually TONE 37). Now switch out of PRGM mode, RTN, and R/S. Although the HP-41's internal oscillator is not crystal controlled, this program makes a nice tick-tock imitation of a pendulum clock.

Synthetic tones have other applications as well. See Appendix B for a high-speed Morse code practice program that uses synthetic tones. You can use Figure 2.1 to help you choose the right synthetic tones for your applications. You can pick a tone frequency and duration, and look up which synthetic tone is the closest to what you need. Table 2.1 and Figure 2.1 are reprinted with permission from Robert E. Swanson, who compiled the data they contain for the HP-41/HP-IL SYSTEM DICTIONARY, which is unfortunately out of print.

2B. Synthetic Exponential Data Entry Lines

Pressing EEX CHS 3 in RUN mode gives you 1×10^{-3} in the X-register. But if you try to do the same thing in PRGM mode you'll get an instruction that looks like 1E-3 even though you only pressed E-3. The calculator insists on adding a superfluous 1, wasting a byte of program space. Now that we have a byte grabber I'll bet you can guess how we can get rid of that 1. Clear the previous example and key in

01 ENTER↑	LB inputs:
02 1E-3	E-3 = 27, 28, 19

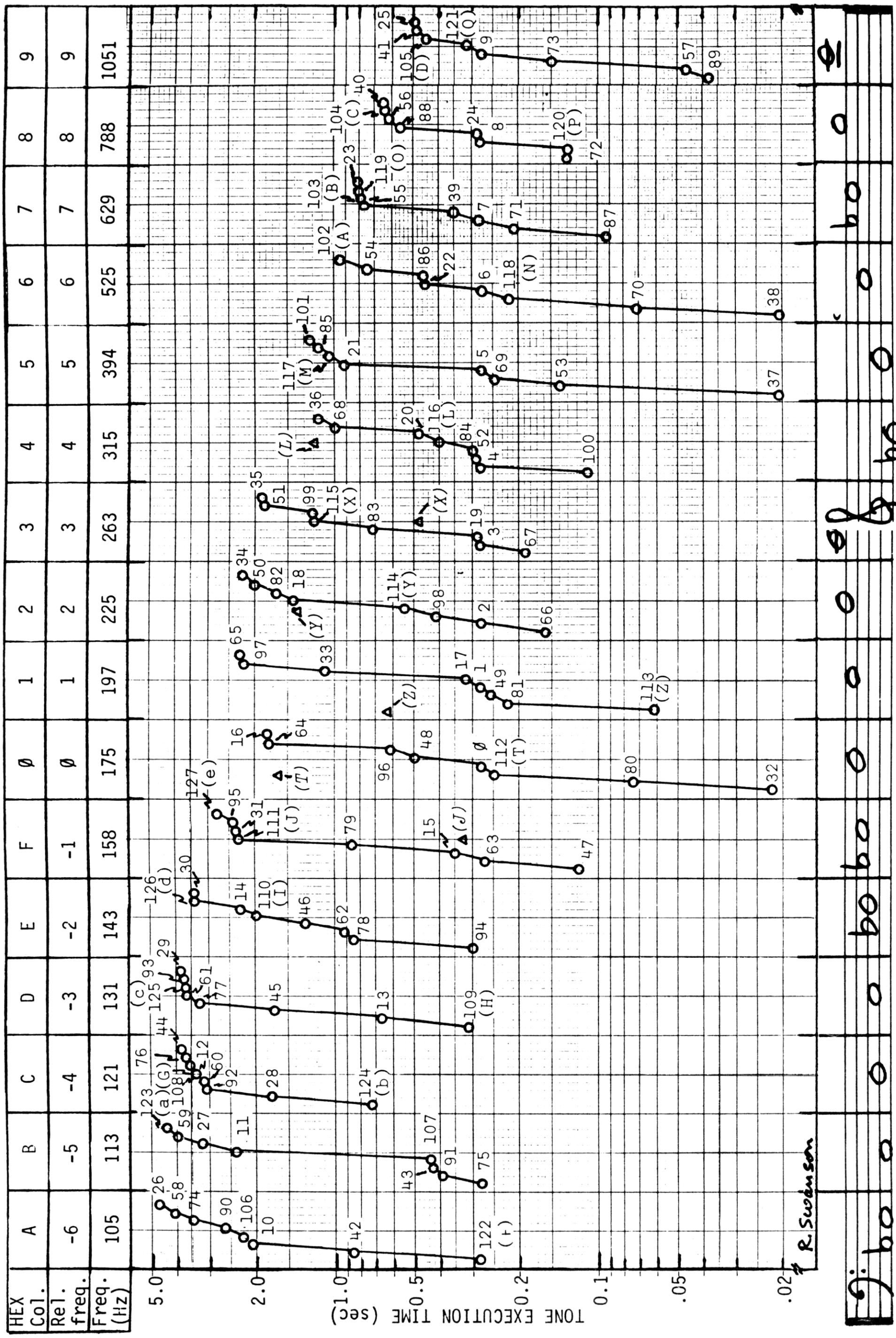
PACK (this is necessary this time). As in the Chapter 1 example, you must press XEQ ALPHA P A C K ALPHA, and <u>not</u> GTO.. , which would be easier to key in. The problem is that GTO.. leaves you "high and dry", requiring you to execute

HP-41C/CV TONE TABLE: Execution Times and XROM Numbers*

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	Ø 0.28 60,00	1 0.28 60,01	2 0.28 60,02	3 0.28 60,03	4 0.28 60,04	5 0.28 60,05	6 0.28 60,06	7 0.28 60,07	8 0.27 60,08	9 0.27 60,09	10 2.08 60,10	11 2.42 60,11	12 3.37 60,12	13 0.67 0.80 60,13	14 2.30 60,14	15 0.35 60,15
1	16 1.82 60,16	17 0.32 60,17	18 1.43 60,18	19 0.29 60,19	20 0.48 60,20	21 0.94 60,21	22 0.45 60,22	23 0.82 60,23	24 0.29 60,24	25 0.49 60,25	26 4.70 60,26	27 3.23 60,27	28 1.75 60,28	29 3.85 60,29	30 3.46 60,30	31 2.37 60,31
2	32 .022 60,32	33 1.10 60,33	34 2.25 60,34	35 1.90 60,35	36 1.17 60,36	37 .020 60,37	38 .020 60,38	39 0.35 60,39	40 0.65 60,40	41 0.49 60,41	42 0.83 60,42	43 0.43 60,43	44 3.80 60,44	45 1.71 60,45	46 1.29 60,46	47 0.12 60,47
3	48 0.50 60,48	49 0.26 60,49	50 2.04 60,50	51 1.85 60,51	52 0.29 60,52	53 0.14 60,53	54 0.75 60,54	55 0.77 60,55	56 0.62 60,56	57 .046 60,57	58 4.07 60,58	59 3.99 0.41 60,59	60 3.19 60,60	61 3.77 0.39 60,61	62 0.93 60,62	63 0.27 60,63
4	64 1.79 61,00	65 2.29 61,01	66 0.16 0.40 61,02	67 0.19 61,03	68 1.01 61,04	69 0.25 61,05	70 .072 .032 61,06	71 0.21 61,07	72 0.13 61,08	73 0.15 61,09	74 3.58 61,10	75 0.28 61,11	76 3.60 61,12	77 3.30 61,13	78 0.85 61,14	79 0.87 61,15
5	80 .075 61,16	81 0.22 61,17	82 1.68 61,18	83 0.72 61,19	84 0.30 61,20	85 1.16 61,21	86 0.46 61,22	87 .093 61,23	88 0.56 61,24	89 .038 61,25	90 2.61 61,26	91 0.39 61,27	92 3.12 61,28	93 3.78 61,29	94 0.30 61,30	95 2.45 61,31
6	96 0.62 61,32	97 2.21 61,33	98 0.41 61,34	99 1.21 61,35	100 0.11 61,36	101 1.27 61,37	102 A 0.96 61,38	103 B 0.80 61,39	104 C 0.64 61,40	105 D 0.45 61,41	106 E 2.26 0.23 61,42	107 F 0.43 61,43	108 G 3.54 61,44	109 H 0.31 61,45	110 I 2.00 61,46	111 J 2.33 *0.33* 61,47
7	112 T 0.25 *1.66* 61,48	113 Z .061 *0.64* 61,49	114 Y 0.55 *1.40* 61,50	115 X 1.19 *0.48* 61,51	116 L 0.40 *1.20* 61,52	117 M 1.07 [61,53	118 N 0.22 \ 61,54	119 O 0.78] 61,55	120 P 0.13 ↑ 61,56	121 Q 0.32 _ 61,57	122 ⊢ 0.29 ⊤ 61,58	123 a 4.38 61,59	124 b 0.73 61,60	125 c 3.77 61,61	126 d 3.45 61,62	127 e 2.84 61,63

Key: within each box is the decimal TONE number, execution time, and XROM numbers.

Actual tone duration is about .Ø15 seconds less, and may depend on the date of manufacture


HEX Col.
Rel. freq.
Freq. (Hz)
TONE EXECUTION TIME (sec)
A B C D E F Ø 1 2 3 4 5 6 7 8 9
-6 -5 -4 -3 -2 -1 Ø 1 2 3 4 5 6 7 8 9
105 113 121 131 143 158 175 197 225 263 315 394 525 629 788 1051
A-flat A B C D-flat E-flat F G A C E-flat G C E-flat G C
R. Swenson

Catalog 1 and interrupt it to get back into your program. You may save a little time in the long run by assigning PACK to a key; just ASN ALPHA P A C K ALPHA and press any key that doesn't already carry an assignment that you need.

Now GTO .001 and BG. Delete the text line -- the starburst at the end of the text line is the captured superfluous 1. SST to see 02 E-3 , a synthetic exponential data entry line, often called a "short-form exponent".

You can try this synthetic instruction by SSTing in RUN mode. You'll find that E-3 works just as well as 1E-3. It obviously saves a byte of program memory, but you should also be aware that it executes faster than 1E-3 to boot.

Execution time, but not memory, can also be saved by using the decimal point instead of the digit 0 for a zero entry, and E instead of the digit 1 for an entry of one. The lone decimal point is not a synthetic instruction, but the lone E is. To create it, just grab the STO prefix from a STO 27 instruction. Row 1 column B of the QRC shows that the 27 suffix will become an EEX instruction.

It was stated earlier that PACKing is necessary when you want to grab the leading 1 from an exponential data entry instruction. The reason is that all digit entry instructions are preceded by an invisible NULL byte (row 0 column 0) that serves solely to separate the new digit entry instruction from the previous instruction. Do not confuse NULL bytes with the NULL message that appears when you hold a key down for 2 seconds after the function preview appears. As its name implies, a NULL byte is a place holder that does nothing when executed (except when it is a suffix in an instruction like X<> 00 or ΣREG 00). NULL bytes, which are always invisible except when they are within text instructions, are created when instructions are deleted and are removed by PACKing. This behavior will be explained and illustrated in Chapter 5.

In the first example of this section we used PACK to remove the null that the HP-41 inserted between 01 ENTER↑ and 02 1E-3 . If line 01 had been a digit entry instruction, the null would not have been removed by PACKing. It would have been needed to maintain the separation of lines Ø1 and Ø2. Except for this special case, PACKing will always remove the null.

But there is another way to remove the null. One can simply key in a one-byte instruction to fill the space that is being held open by the null. Let's try this on the E-3 example. Clear line Ø2 and key in

Ø1 ENTER↑

Ø2 1E-3

There is now an invisible null between lines Ø1 and Ø2. Since we want to grab the 1 from 1E-3, not the null, we fill the null first. GTO .ØØ1, or just BST, and press RDN (roll down). This is a one-byte instruction that overwrites the NULL byte. Now BG and capture the leading 1. Backarrow twice and you'll have

Ø1 ENTER↑

Ø2 E-3

Thus the addition of two keystrokes to the procedure introduced at the beginning of this section eliminates the need for PACKing. This can be especially advantageous when you're adding a synthetic exponential data entry instruction to a long program which takes several seconds to PACK.

Chapter 5 will fully explain and illustrate the elusive behavior of nulls. It uses a synthetic technique to make them visible. Ambitious synthetic programmers who want to try fancy tricks like constructing a synthetic line -E should note that whenever you want to include a negative sign in a digit entry line the appropriate byte is row 1 column C, NEG, <u>not</u> row 5 column 4, CHS. The CHS key governs two different operations: negating a digit entry and negating an existing number.

2C. Flag Register Control

Normally when a program constructs an alpha message containing numbers, the display mode is altered. For example the sequence

```
Ø1 1.Ø1            Register number index -- 1 to 1Ø
Ø2 STO ØØ
Ø3 FIX Ø           These two steps are needed to make
Ø4 CF 29           the register number appear without
Ø5 LBL Ø1          a decimal point in the prompt
Ø6 "INPUT "      (Note there is a space following T)
Ø7 ARCL ØØ         Append the register number
Ø8 "⊦?"
Ø9 TONE 9
1Ø PROMPT
11 STO IND ØØ      Store the input in the current
12 ISG ØØ          register; add 1 to register index
13 GTO Ø1
```

Line Ø6 is XEQ ALPHA T O N E ALPHA 9. Line Ø7 is obtained by ALPHA shift RCL Ø Ø, while line Ø8 is ALPHA shift XEQ 3 ALPHA.

prompts for inputs numbered 1 to 1Ø and stores them in data registers 1 through 1Ø. It has the undesirable feature that lines Ø3 and Ø4 change the display mode to FIX Ø. Synthetic programming offers an easy way to avoid altering the display mode in cases like this one.

It's time for a brief digression about flags. Since a flag has only two possible states, set and clear, it makes sense for the calculator to use one bit (binary digit) to represent each flag. As it happens, the set state is represented by 1 and the clear state is represented by Ø. We saw in Chapter 1 that a byte consists of eight bits. The HP-41 Owner's Handbook reveals that a register consists of seven bytes. Thus there are 8x7 = 56 bits in a register. If the number 56 sounds familiar, perhaps it's because the HP-41 has 56 user and system flags, numbered Ø through 55. So it shouldn't be too surprising that all 56 flags occupy exactly one register in the HP-41.

The flag register is one of the sixteen HP-41 system scratch registers. You already know the first five: the stack registers T, Z, Y, X, and L. The names of the rest are found along row 7 of the QRC. The name of the flag register is **d** (row 7 column E).

Now to the case at hand. We want to preserve the display setting while constructing a numerical message. To do this we can RCL d before forming the message, saving the original flag register in X. After forming the message we STO d , transferring the original flag register contents from X back into the flag register. This restores all 56 original flag settings, including the display setting.

For the example given at the beginning of this section, this is accomplished as follows. Key in

```
Ø1 1.Ø1                     LB / MK  inputs:
Ø2 STO ØØ
Ø3 LBL Ø1
Ø4 "INPUT "
Ø5 STO IND 16               RCL d  = 144,126
Ø6 AVIEW
Ø7 FIX Ø
Ø8 CF 29
Ø9 ARCL ØØ
1Ø STO IND 17               STO d  = 145,126
11 AVIEW
12 "⊢?"
13 TONE 9
14 PROMPT
15 STO IND ØØ
16 ISG ØØ
17 GTO Ø1
```

GTO .ØØ9, BG, and delete the text line. SST to see STO d . GTO .ØØ4, BG, backarrow, and SST to see RCL d . The IND 17 byte (row 9 column 1) became STO, the IND 16 byte (row 9 column Ø) became RCL, and both AVIEW instructions (row 7 column E) became d suffixes. This version of the program will prompt for input for data registers 1 through 1Ø. When it is finished, the display mode will be unchanged, rather than the distinctly unfriendly FIX Ø.

```
01 1.01
02 STO 00

03♦LBL 01
04 "INPUT "
05 RCL d
06 FIX 0
07 CF 29
08 ARCL 00
09 STO d
10 "⊢?"
11 TONE 9
12 PROMPT
13 STO IND 00
14 ISG 00
15 GTO 01
```

The RCL d / STO d combination can be used anywhere you want to preserve the status of the display mode, trig mode, or other flags. The original flag register can be stored anywhere in the stack, but it should not be stored in a numbered data register. Data retrieved from a numbered data register is subject to <u>normalization</u>. If the 56 bits aren't in a configuration that the HP-41 recognizes as an alphabetic or numeric form, it will change bits as necessary to make it an alphabetic or numeric value.

The detailed specification of what bit patterns are recognized as alphabetic or numeric data is beyond the scope of this book but for our purposes here an abbreviated rule on normalization will suffice. Any 56-bit data pattern whose first four bits are ØØØ1 can be safely stored into and retrieved from a numbered data register. If the first four bits are other than ØØØ1 the data is subject to normalization (hence possible alteration) when retrieved. This is of course no problem if the data is actually numeric or alphabetic. Normalization is only a problem when dealing with non-standard bit patterns such as flag register contents.

If you wish to store a set of flag settings in a numbered register, you need to set the first four bits to ØØØ1 beforehand. This is easily done as the following example will illustrate. Clear the previous example except for its RCL d and STO d instructions. Then GTO .ØØØ and key in

```
Ø1 CF ØØ        These first four lines set the
Ø2 CF Ø1        first four bits of the flag
Ø3 CF Ø2        register to the pattern ØØØ1.
Ø4 SF Ø3
Ø5 RCL d
Ø6 STO Ø1
Ø7 GRAD
Ø8 SF Ø1
Ø9 CF Ø3
1Ø STOP
11 RCL Ø1
12 STO d
```

Switch out of PRGM mode, RTN, and R/S. Note that flag 1 and GRAD mode are set. R/S again to see the flags returned to their original state, with flags Ø, 1, and 2 clear and flag 3 set. If you don't mind an example that requires a little cleanup work with your flags you can change line Ø1 to SF ØØ and verify that many flags are changed when the program is executed. For a quicker cleanup you may wish to use the copy of the original flags that will be residing in stack register Y at the completion of the program. Since this copy wasn't stored in a numbered data register it's unchanged. Just RDN, GTO .Ø12, and SST to restore the flags.

2D. Program Pointer Control

The HP-41 maintains a program pointer in one of its operating system scratch registers. This pointer designates what part of memory will be displayed when PGRM mode is selected. The system scratch register that contains the program pointer (together with some of the return pointers -- these are discussed in Section 6A of this book and in the PPC ROM User's Manual under "Line by Line Analysis of LR ") is designated the "b" register by the HP-41 operating system.

To illustrate the ease of program pointer control on the HP-41 try the following example. Clear the previous example and key in

```
01 ENTER↑                  LB / MK inputs:
02 STO IND 16              RCL b  = 144, 124
03 MEAN
04 STO IND 31              TONE 89 = 159, 89
05 SIN
06 STO IND 17              STO b  = 145, 124
07 MEAN
```

GTO .005 , BG, backarrow, GTO .003 , BG, backarrow, GTO .001, BG , backarrow twice, and PACK (do not GTO..). Switch to RUN (non-PGRM) mode, RTN, and R/S. You'll hear the rapid staccato of repeated TONE 89's. The "flying goose" is frozen in place.

How does this work? The RCL b instruction copies the program pointer into the X register. The TONE 89 is executed, then the STO b puts the previously recalled value back into the program pointer. At the time the program pointer was originally recalled the next instruction to be executed was TONE 89. Therefore the STO b instruction causes execution to jump back to the TONE 89 instruction. If you RTN and SST this program you can verify that the sequence of execution is RCL b, TONE 89, STO b, TONE 89, etc.

The reason that the flying goose holds still when this program is run is quite simple. The goose is programmed to move one position each time a LBL is executed. But there are no labels in this program, despite the looping. Thus the goose is unable to move.

The next example provides the answer to an HP-41 trivia question: What is the shortest "infinite loop" on the HP-41? The answer is one program line, 2 bytes. Delete the TONE 89 from the previous example and PACK. You now have

```
Ø1 RCL b
Ø2 STO b
```

If you RTN and SST this program, you'll find that the execution sequence is RCL b, STO b, STO b, STO b, STO b, --- ad infinitum, although the line number keeps increasing. For SST execution the HP-41 always increments the line number unless it executes a GTO, XEQ, RTN or END instruction, in which case the line number is recomputed. The calculator does not recognize STO b as a "jump" instruction, so it doesn't bother to recompute the line number. If your SST finger were extremely durable, you would find out that the line number counts all the way up to 4Ø94 before starting over at Ø2. As you will learn in section 6A, the number 4Ø95 has a special meaning to the HP-41's internal programming. This number means that the line number needs to be recomputed.

For non-SST, free-running program execution, the calculator does not update the line number at each step. That would needlessly slow execution.

Advanced synthetic programming techniques are needed to fully utilize the power of the STO b instruction. The ultra-fast Morse code program in Appendix B illustrates precompiled indirect branching, a relatively straightforward application of program pointer control. Also, the sequence Ø, STO b, GTO .ØØ2 is an easy way to move the program pointer into the key assignment registers. Details of how information is stored in the key assignment registers can be found in the PPC ROM User's Manual, under "Background for MK ".

2E. Synthetic Text Lines

The HP-41 differs from its predecessors most notably in that it provides alphanumeric capability. This capability can be used to label outputs or prompt for inputs. However the set of display characters available seems to be rather limited. For example there are no parentheses or quotation marks.

Synthetic programming techniques permit 21 additional distinct display characters to be used in text instructions, including parentheses, quotation marks, apostrophe, ampersand, and others. These synthetic display characters can be edited into a text instruction in a way which we shall describe here. PPC ROM programs provide two alternate methods. The simplest is to use **LB** to create synthetic text instructions directly. The "Q-transfer" method, which requires a supportive program such as PPC ROM program **DC** , is also available. The first of these methods will be presented in Chapter 3. The second shall be introduced in Section 4B.

The byte-grabber method of creating synthetic text instructions, which is introduced in this section, is fairly simple and requires very little setup (just a byte grabber key assignment). Therefore regardless of the availability of other methods you should follow through the byte grabber examples of this section. You may find it the most convenient method for creating one or two synthetic text instructions.

Owners of a printer or an Extended Functions module may be acquainted (through the functions BLDSPEC and XTOA, respectively) with other, more cumbersome ways of creating synthetic display characters. In this section we will show that synthetic text lines can be used to save many bytes over the normal methods which use BLDSPEC or XTOA.

The structure of a n-character text instruction is quite simple. A hex Fn byte (row F column n) precedes n bytes, each of which represents a character. Thus n+1 bytes of program memory are needed to hold an n-character text instruction. The character-byte correspondence is illustrated in the byte table, which is part of the Quick Reference Card for Synthetic Programming. For example a row 5 column F byte displays and prints as _ . Certain synthetic characters appear substantially different on the printer compared with their displayed form. For example row Ø column 4 displays as X̄ but prints as α . A byte is only interpreted as a character when it is preceded by a row F byte that brings the byte in question into the scope of the text instruction. In the absence of a row F byte, program bytes are interpreted in the normal manner, as instructions or suffixes for previous instructions. Row F bytes can thus be regarded as TEXT instructions that require suffix bytes. The difference between TEXT instructions and most other instructions is that the number of suffix bytes is variable and that a TEXT instruction triggers a very different interpretation of suffix bytes, namely the character interpretation.

Synthetic text lines can be created using the byte grabber in a four-step procedure. First a text line of the desired length is created, with X's in the positions where synthetic characters are required. Then the TEXT instruction prefix is grabbed. This frees the suffix bytes to be instructions, rather than characters. In this form the X's can be replaced by instructions corresponding to synthetic characters. The final step is to release the grabbed TEXT prefix, which then captures the edited bytes and converts them to characters.

An example should make this procedure clear. Suppose we want to create the text line "HP'S #1" . Clear the previous example and key in

Ø1 ENTER↑	LB inputs:
Ø2 "HPXS X1"	247, 72, 8Ø, 39,
	83, 32, 35, 49 .

GTO .ØØ1 and BG but do not backarrow the text line. It contains the captured TEXT 7 prefix that you'll need later.

SST several times and you'll see that you now have:

```
01 ENTER↑
02 "‾?‾‾‾‾▩"
03 Σ-
04 LN
05 E↑X-1
06 Y↑X
07 RCL 00
08 E↑X-1
09 STO 01 .
```

Lines 03 through 09 each correspond to a character from the original text line. For instance, RCL 00 corresponds to the space. Row 2 column 0 of the QRC verifies this correspondence. What we'd like to do now is to replace the E↑X-1 instructions that correspond to the X's. GTO .008 and backarrow the E↑X-1 . We wanted a # symbol in this position. Checking row 2 column 3 of the QRC we find that the corresponding instruction is RCL 03 . Key in RCL 03 as the replacement for line 08. Now GTO .005 and backarrow the E↑X-1. Row 2 column 7 of the QRC tells us to key in RCL 07 as the new line 05 to get the apostrophe character.

If you have followed the instructions carefully you don't really need to PACK, but it can't hurt. You should have

```
01 ENTER↑
02 "‾?‾‾‾‾▩"
03 Σ-
04 LN
05 RCL 07
06 Y↑X
07 RCL 00
08 RCL 03
09 STO 01
```

Now GTO .001, and BG. You have grabbed the TEXT prefix from line 02. This released the question mark and the starburst to become instructions. SST and you'll see that the question mark became STO 15 (check row 3 column F). SST again and you'll see that the starburst has regained its former identity as a TEXT 7 instruction, in turn capturing the following 7 bytes as text characters. Thus we now have

Ø1 ENTER↑

Ø2 "‾?----▓"

Ø3 STO 15

Ø3 "HP'S #1" .

If you have a printer you may wish to compare the way these synthetic characters print with the way they display. (If you don't have a printer just look at the lower right corner of each box in the QRC to see the way that byte prints as a character.) You'll find that the apostrophe and the # symbol print as expected, but the starburst vanishes without a trace. This vanishing behavior is to be expected in program listings from any character in rows 8 through F. This point will be discussed further toward the end of this section.

The append instruction is unique among HP-41 instructions in its implementation. An append instruction is a text instruction whose first character is the append character ⊦ (row 7 column F). Since the append character takes up the first character byte of the text line and the text line cannot exceed fifteen characters, the maximum number of characters that can be appended is fourteen. If the append character is synthetically inserted into a text instruction in a position other than the first character byte, it loses its privileged "control character" status and becomes an ordinary character.

Let's edit some synthetic characters into an append instruction. Key in

Ø1 ENTER↑

Ø2 "⊦ABCDEFGHIJKL" .

GTO .ØØ1 and BG but do not backarrow. The byte grabber's text line will hold the TEXT 13 byte from the former line Ø2 until we are finished editing. SST through the program and you should see

Ø1 ENTER↑

Ø2 "‾?‾‾‾‾▓"

Ø3 CLD

Ø4 -

Ø5 *

Ø6 /

Ø7 X<Y?

Ø8 X>Y?

Ø9 X<=Y?

1Ø Σ+

11 Σ-

12 HMS+

13 HMS-

14 MOD

15 % .

Line Ø3 is the append control character (row 7 column F). Lines Ø4 through 15 correspond to the characters A through L. See row 4 of the QRC for the correspondence. Now GTO .ØØ4 and DEL Ø12 (XEQ ALPHA D E L ALPHA Ø 1 2). This deletes lines Ø4 through 15. We're going to replace all 12 characters by synthetic characters. We can simply key in the instructions corresponding to the characters we want. Try keying in these instructions:

instruction:	character:
Ø4 -	A
Ø5 LBL ØØ	*̄
Ø6 LBL 11	,µ
Ø7 RCL Ø2	"
Ø8 RCL Ø8	(
Ø9 RCL Ø9	)
1Ø STO 11	; (semicolon)
11 ASIN	\
12 DEC	_
13 CLD	⊦
14 1/X	ᵀ
15 +	@

Now PACK just to be sure there aren't any nulls present. Delete line Ø4 to create a NULL, then GTO .ØØ1, BG, and backarrow. You should see

Ø1 ENTER↑

Ø2 STO 15

Ø3 "⊦‾⊼µ"(),_⊦ᵀ@"

The LB inputs for this example are 253, 127, Ø, 1, 12, 34, 4Ø, 41, 59, 92, 95, 127, 96, and 64.

Put "ABC" in the ALPHA register and execute line Ø3. The ALPHA register will then contain "ABC‾⊼µ"(),_⊦ᵀ@". If you CLA and execute line Ø3 you'll get a surprise. The ALPHA register will contain "⊼µ"(),_⊦ᵀ@". The NULL (overline character) disappeared! The general rule is that NULL characters are visible only when they are interior or trailing characters in the ALPHA register.

If you execute ASTO X, even the interior and trailing nulls will be invisible in the X register, but they will still be present. This can be verified by trying the X=Y? test. The result will be NO if, for example, the X register contains an invisible null while Y does not, even if the two registers display the same way. This behavior is not useful enough to merit an example, but you should be aware that viewing an ASTOred string that contains nulls will not reveal the full structure. You should use ARCL and AVIEW when in doubt.

Printer owners may be aware that the printer function BLDSPEC can be used to generate any synthetic display character. For example the instruction sequence

Ø1 . (décimal point)

Ø2 X<>Y

Ø3 BLDSPEC

Ø4 PRX

will create a single display character corresponding to the decimal value (Ø to 127) in the X register. It will then print the character as well.

Try 38, GTO .ØØ1, R/S and you'll get the ampersand, a synthetic character. Row 2 column 6 of the QRC shows how the displayed version of the ampersand compares to the printed version. Try 5, R/S and you'll get the one-armed man ⊼ in the display and the Greek letter β on the printer. Row Ø column 5 of the QRC verifies this result. A large number of the 128 standard printer characters display as starbursts. Something like this must be expected since the 14 segment display does not have the flexibility of the printer's dot matrix output.

Owners of the Extended Functions module have available a powerful function, XTOA, that can be used to create synthetic display characters. XTOA is a much faster version of PPC ROM routine DC . Assign XTOA (or DC) to a convenient key and try CLA, 38, XTOA. Switch to ALPHA mode and you should see the synthetic display character &. If you now do ALPHA(off), 5, XTOA, ALPHA(on), you'll see &⊼. The one-armed man character (decimal equivalent 5) has been appended to the alpha register. To compare the printed versions you can execute PRA.

Printer owners will appreciate the byte savings that are possible by using synthetic text instructions to generate lower-case and mixed-case text. Consider the normal method of creating the printed output "Hewlett-Packard"

```
Ø1 "H"
Ø2 ACA      (load H into the print buffer from ALPHA)
Ø3 SF 13    (switch to lower case)
Ø4 "EWLETT-"
Ø5 ACA      (add lower case characters to the buffer)
Ø6 CF 13    (switch back to upper case)
Ø7 "P"
Ø8 ACA
Ø9 SF 13
1Ø "ACKARD"
11 ACA
12 PRBUF    (print the buffer contents)
13 CF 13    (back to upper case mode)
```

The byte count for this monstrosity is 37 bytes, compared with 18 bytes for the synthetic text line "Hewlett-Packard" followed by a PRA command. Moreover every mode change, between upper and lower case in this example, uses a valuable print buffer "register" (actually a byte). This is discussed in more detail on page 19 of the July 198Ø PPC Calculator Journal. The synthetic text line approach conserves print buffer space as well as program memory. Of course most of the lower case characters (all but a,b,c,d,e) in the synthetic text line appear only as starbursts in the display, although the text line prints properly in a program listing. If you can tolerate the somewhat messy SST display, you can achieve dramatic everyday byte savings by using synthetic text lines wherever you require lower-case or mixed-case printing.

Synthetic text instructions have much wider application than just generation of nonstandard display characters. They provide a simple, fast method to enter needed bytes under program control. Byte loader programs (Chapter 3), key assignment programs (Chapter 4), and other very powerful synthetic programs use synthetic text lines extensively. Using the first example from this section, we can illustrate the simplicity of synthetic text lines compared to the next best alternative, the XTOA function of the Extended Functions module.

```
Goal: Create the synthetic text "HP'S #1"
Best Method: synthetic instruction  Ø1 "HP'S #1"
          Total bytes used: 8  Execution speed: fast
Next Best:  use XTOA       Ø1 "HP"
            or DC          Ø2 39
                           Ø3 XTOA  (or XROM DC )
                           Ø4 "⊦S "
                           Ø5 35
                           Ø6 XTOA  (or XROM DC )
                           Ø7 "⊦1"
          Total bytes used: 18  Execution speed: slower.
```

Printer owners who like to use BLDSPEC to manufacture "custom" printer characters can save bytes and speed up their programs by using synthetic text instructions. The sequence: 7-character synthetic text instruction, RCL M, ACSPEC, substitutes for the normal sequence: number, BLDSPEC, number, BLDSPEC,...,number, BLDSPEC, ACSPEC. The RCL M instruction will be explained in section 2G. Details of the correspondence between the normal BLDSPEC numbers and the required 7-character synthetic text instruction can be found in the PPC ROM User's Manual under BL, or in the June 1980 PPC Calculator Journal.

For more exotic synthetic programming, synthetic text instructions often need to contain bytes from rows 9 through F of the QRC, which correspond to multi-byte instructions. The byte-grabber technique presented earlier in this section does not usually allow creation of such text instructions. The easiest way to create these instructions is to use a byte loader program, as you will see in Chapter 3. But beware! Synthetic text instructions containing bytes from rows 8 through F appear as expected in the display but print strangely. These row 8 to F bytes all display as starbursts. If they are printed via PRA, they will appear as shown on the QRC. For example a row C column D character displays as a starburst but prints as M. However if you list the program, all the row 8 to F characters in the text instructions will disappear, without even leaving spaces to hint at their presence. Certain of these characters, the ones that are shaded on the QRC, will cause additional strange behavior when listed (skipping spaces, switching to lower case, etc.) If this messes up your listing, manually GTO the following line and LIST the rest of the program. Incidentally, NORMAL mode listings give a slight hint of the presence of synthetic characters in that the statement number will usually be indented if an invisible character is present. If you're interested in learning more, consult the July 1980 PPC

Calculator Journal for an extensive, clearly written description of these printer control characters.

2F. The TEXT Ø instruction

The HP-41 allows text instructions up to 15 characters long, or 14 characters plus the append symbol. The first byte of a text instruction is taken from row F of the QRC, with the column number denoting the number of characters in the instruction.

But what about column zero? By logical extension, a row F column Ø byte would appear to denote a text line of length zero. One might therefore expect such a TEXT Ø instructions to be the equivalent of CLA. Let's find out. Key in

	LB input:	MK input:
Ø1 "ABC"		
Ø2 STO IND T	24Ø	24Ø, 24Ø

To key in line Ø2, press STO shift . (decimal point) 9 (T).

GTO .ØØ1, BG, and backarrow. The STO has been removed, and the IND T (row F column Ø) now assumes the identity of a TEXT Ø instruction. This instruction displays as a text symbol with nothing following. It prints as "" (nothing between quotation marks). Now run the program and switch to ALPHA mode. Surprise! The "ABC" that was loaded into the ALPHA register by line Ø1 is still there. The TEXT Ø instruction is not equivalent to CLA. Further experimentation will reveal that TEXT Ø has no effect on the ALPHA register or any other register (including the flag register). TEXT Ø will, like virtually all other program instructions, enable the stack lift. (See the Owner's Manual for a discussion of stack lift.)

What is an instruction like TEXT Ø good for if it doesn't do anything? Suppose we want to increment an unknown integer in the Y register without disturbing the stack. ISG Y does this but it will also skip a line if Y was non-negative.

Therefore we need to follow ISG Y by an instruction that will not affect the calculator's state whether it is executed or not. TEXT Ø is precisely the kind of instruction we want. Moreover it is the only such one-byte instruction on the HP-41. "Do nothing" instructions like TEXT Ø are called NOPs, short for no operation. NOP keys can be found on the HP-25, HP-33, HP-55, and some other calculators. Synthetic techniques have now given your HP-41 a similar capability. You'll see sequences like

```
Ø1 ISG X
Ø2 TEXT Ø
```

in many synthetic programs. You can use such a sequence anywhere you need an "increment but do not skip" capability. Of course TEXT Ø can also be used following a DSE instruction to decrement without skipping.

2G. Using the ALPHA register for data storage

We have seen that one byte of program memory is required to represent each character in a text instruction. We might therefore expect that the 24-character ALPHA register would require 24 bytes of non-program memory. This is equivalent to 24/7 = 3 registers plus 3 leftover bytes. These registers, together with the stack registers, the flag register, and others, are located in a separate section of memory called either system scratch or the status registers. The name status registers comes from the fact that the card reader's WSTS (write status) function records these registers on track 1 of a status card.

Since the flag register and the program pointer can be accessed directly by synthetic instructions, perhaps we can similarly access the 3+ registers that comprise the ALPHA register. The suffix bytes for the flag register and the program pointer register are from row 7, columns E and C respectively, of the QRC. You have probably begun to suspect that the other row 7 suffixes correspond to the other system scratch registers. But before you start experimenting, beware. You can safely RCL any of the status registers (the "normalization" of stored data mentioned in section 2C does not apply to status register operations), but don't alter their contents until you know what you're doing, unless you are prepared for the worst. For example if you clear status register c you'll get MEMORY LOST.

The ALPHA register occupies status registers M, N, O, and part of P. As long as you don't mind altering whatever was in the ALPHA register, you may use M, N, and O freely, just as you would use numbered data registers. From what you have learned about using the byte grabber you should be able to create the following program:

```
Ø1 LBL"RSHF"
Ø2 CLX
Ø3 X<> O
Ø4 X<> N
Ø5 X<> M .
```

If you need help, see the instructions at the end of this section.

For the moment let's concentrate on the X<> M instruction. Try the sequence CLA, 1.274Ø65ØØ2 E-4Ø, X<> M. For the X<> M you can GTO .ØØ5 and SST in RUN (non-PRGM) mode. Now switch into ALPHA mode and you'll see ⊼'@e¯)T . What's going on? Let's refer to the QRC to identify the 7 bytes that comprise this character string. Designated by row number r and column number c the 7 bytes are shown below.

BYTE IN HEXADECIMAL	01	27	40	65	00	29	60
BYTE IN CHARACTER FORM	⊼	'	@	e	¯	)	T
REGISTER IN NUMERIC FORM	+1.	27	40	65	00	2E-	40

SIGN (+1.) — **MANTISSA (10 DIGITS)** (27 40 65 00 2) — **SIGN** (E-) — **EXPONENT** (40)

The fourteen hexadecimal digits that comprise the seven bytes are 01274065002960. The ten digits of the original X-register contents are immediately recognizable as the second through the eleventh of these 14 digits. The first of the 14 digits is a sign digit. It is zero for positive numbers, 9 for negative numbers, and 1 for alpha data. The last three of the 14 digits represent the exponent and its sign. If the twelfth digit is zero the exponent is positive; if the twelfth digit is 9 the exponent is negative. The last two digits are the exponent digits if the exponent is positive. If the exponent is negative, the last two digits are 100 plus the negative exponent. In this case the exponent is -40, so the last two digits are 100+(-40) = 60. A simple rule that works for either positive or negative exponents is: add 1000 to the signed exponent (that is, add the exponent to 1000 if it's positive, subtract the exponent from 1000 if it's negative). Keep only the last three digits of the result. This gives the correct exponent digits for the HP-41 internal representation. In this case 1000-40 = 960.

If we execute GTO .005 and SST again to execute X<> M, the number 1.274065002 E-40 returns to the X-register and ALPHA is again clear. Now try another example. With the same number still in X, execute X<> M, switch to ALPHA mode, press append, backarrow, and A. You now have the string ⊼'@e¯)A . Switch out of ALPHA mode and execute X<> M again to get 1.274065002 E-59 . Since the character A is hexadecimal 41, the exponent became 41-100 = -59.

Feel free to explore further the equivalence of numbers and seven-character alpha strings using the X<> M instruction. Most numbers will consist primarily of starburst characters. You should be aware that if you bring an alpha string into the X register using X<> M, the result may behave strangely if the two sign digits are not zero or 9 or if there are digits other than 0-9 (that is, nondecimal digits) present.

When you're using M as a scratch register to store a number you probably won't care what the number looks like as a character string, but the character/number equivalence can be exploited in some advanced synthetic programming techniques. For example, if we wanted to enter the number 1.274Ø65ØØ2 x $1Ø^{-4Ø}$ in a program we could save 5 bytes of program memory by using "⊼'@e‾)⊤" followed by RCL M.

The X<> N and X<> O instructions behave similarly to X<> M. The difference is that X<> M places the number in the <u>rightmost</u> 7 positions of the ALPHA register. The instructions X<> N and X<> O access the next two groups of 7 characters, moving from right to left. Figure 2.2 should make this more clear. You may also wish to try this short example. Load "ABCDEFGHIJKLMNOPQRSTUV" into the ALPHA register. Execute CLX and X<> O (use GTO .ØØ2, SST, SST). The ALPHA register now contains "A‾‾‾‾‾‾‾IJKLMNOPQRSTUV". The seven characters that were occupying the O register (see Figure 2.2) have been replaced by the overline characters that result from null bytes (row Ø column Ø). The O register now contains the number zero. Execute X<> N and ALPHA will contain "A‾‾‾‾‾‾‾BCDEFGHPQRSTUV". Execute X<> O now and you'll get "AIJKLMNOBCDEFGHPQRSTUV". Thus, in addition to their utility as data storage instructions, the STO, RCL, and X<> instructions for status registers M, N, and O can be used to slice up and reassemble character strings in the ALPHA register. These character manipulation capabilities are used extensively in advanced synthetic programming to isolate bytes for decoding or to replace certain bytes of a string.

One easily understood string manipulation application is a 7-character right-handed alpha shift. The program "RSHF" performs such a shift for strings of up to 21 characters, removing the rightmost 7 characters.

```
Ø1 LBL"RSHF"
Ø2 CLX
Ø3 X<> O
Ø4 X<> N
Ø5 X<> M .
```

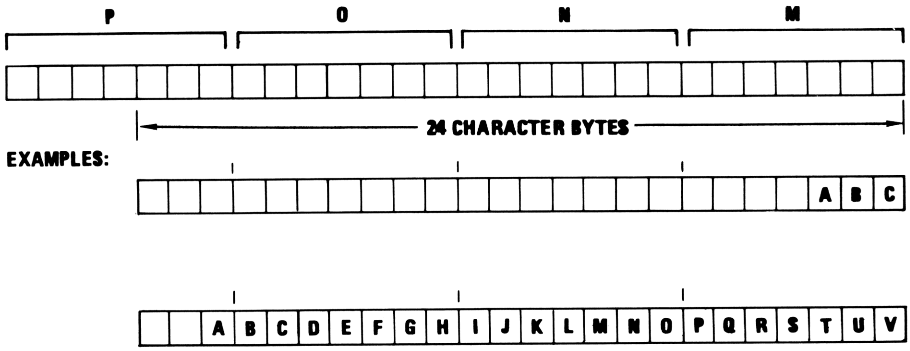


Figure 2.2 The ALPHA register. Character strings of length 1 to 24 are always right-justified. Leading positions are null (hexadecimal ØØ) and are invisible.

For example "ABCDEFGHIJKLMNOP", XEQ "RSHF", yields "ABCDEFGHI". You can SST in ALPHA mode to see how "RSHF" works.

Now let's see how access to status registers M, N, and O can help us in numeric programming. Having three extra registers "on the side" can greatly alleviate register usage conflicts. You can now write many of your subroutines so they don't use any numbered data registers. That makes them compatible with any program that only uses numbered registers. For example many of the routines in the PPC ROM use no numbered registers, so that programs that call these routines are free to use any and all numbered data registers. As a further aid to compatibility it is good programming practice not to rely on the contents of M, N, and O to remain the same when a subroutine is called.

Very short subroutines can often use part of the ALPHA register to avoid using either stack registers or numbered data registers. The ideal goal is operation equivalent to internal functions -- saving X in LASTX, saving the T register contents (in T), and providing the result in X.

As an example let's write a subroutine named "CNK" that will compute the statistical combination function,

$$C(n,k) = \frac{n!}{k!(n-k)!} = \frac{(n-k+1)(n-k+2)\ldots n}{k(k-1)\ldots 1}$$

the number of possible combinations of n items taken k at a time. This routine is to take the values of n and k from stack registers Y and X respectively and is to provide the result C(n,k) in X. The previous contents of Z and T are to end up in Y and Z as they would for a built-in function. The value k is to be saved in LASTX, while n is to be saved in T.

Due to the complexity of the calculation, "CNK" cannot preserve the contents of Z and T without using a scratch register. We will use status register M. This makes "CNK" compatible with any calling program that uses only numbered data registers. A sample "CNK" routine is listed below so you can key it up and try it out.

```
Ø1 LBL"CNK"                LB / MK inputs:
Ø2 -
Ø3 E                       27 or 27, Ø
Ø4 STO M                   145, 117
Ø5 RDN
Ø6 LASTX
Ø7 X>Y?
Ø8 X<>Y
Ø9 LBL Ø1
1Ø X<>Y
11 ISG X
12 TEXT Ø                  24Ø or 24Ø, 24Ø
13 ST* M                   148, 117
14 X<>Y
15 ST/ M                   149, 117
16 DSE X
17 GTO Ø1
18 X<>Y
19 RDN
```

```
20 X<> M                    206, 117
21 END
```

To create the synthetic lines use STO 27, STO IND 17, RDN, STO IND T, STO IND 20, RDN, STO IND 21, RDN, STO IND 78, RDN. For each of the five STO instructions grab the prefix byte by going to the preceding step in PRGM mode then pressing BG and backarrow.

Test "CNK" using 88 ENTER↑ 3 R/S, then 88 ENTER↑ 85 R/S. Both should give a result of 109,736. This is the number of possible three-note chords on an 88-key piano.

Here's how "CNK" works. At the beginning X contains k and Y contains n. "CNK" initializes status register M to 1 on line 04 so that the ST* M and ST/ M instructions in the LBL 01 loop will work as required the first time through the loop. After the execution of line 06, M contains 1, X contains k, and Y contains n-k. Then lines 07 and 08 interchange the roles of k and n-k if n-k is smaller. This makes use of the identity C(n,k) = C(n,n-k) to speed execution where possible. The LBL 01 loop increments n-k and multiplies the result into M. Then at line 14 k is brought back into X, after which it is divided into M and decremented. At this point (back at LBL 01 ready for the second pass through the loop), X contains k-1, Y contains n-k+1, and M contains (n-k+1)/k, the first factor in the expanded expression for C(n,k) that was given above. The loop is executed k times, after which X is zero and Y is n. The last three lines put Y in T, and bring the result from M to X, clearing M.

You may wish to change lines 04, 13, 15, and 20 of "CNK" to use status register O instead of M. This will allow alpha strings of up to 14 characters to remain undisturbed in N and M when "CNK" is used.

Here is the promised step-by-step procedure for creating ALPHA register access instructions. Key in

```
01 LBL"RSHF"              LB / MK inputs:
02 CLX
03 STO IND 78             X<> O = 206, 119
04 CLX
05 STO IND 78             X<> N = 206, 118
06 LASTX
07 STO IND 78             X<> M = 206, 117
08 RDN
```

GTO .006, BG, backarrow, GTO .004, BG, backarrow, GTO .002, BG, and backarrow. You now have the required synthetic intructions for "RSHF".

2H. Using other status registers for data storage

Status registers P, Q, and a can be used under limited conditions as temporary data storage. More details of how the HP-41 operating system uses these registers can be found in Section 6A of this book and on page 19 of the September 1979 PPC Calculator Journal, but we'll give a brief summary here.

Status register P can be used for storage in a program, but its contents will be altered if a digit entry line is executed, or if any operation is performed that causes a number to be displayed.

Status register Q can be used for storage as well, but its contents are also susceptible to alteration. If you execute a global ALPHA GTO or XEQ instruction (that is, a GTO or XEQ that refers to a Catalog 1 or 2 label), you'll lose whatever was in Q. This does not apply to ALPHA LBL instructions. Nor does it apply to XROM instructions, which are different in structure from ALPHA XEQ instructions, as we shall see in the next chapter. Q will also be altered if you spell out an alpha name from the keyboard for a GTO, XEQ, or LBL. Other instructions that alter Q are: any digit entry, SIN, COS, R-P, P-R, Y↑X, SDEV, and any instruction that causes the alpha register to be displayed (AVIEW, PROMPT, or PSE with AON). Status register Q is used extensively by the 82143A peripheral printer in its exchange of information with the 41 mainframe. If you plan to have the 82143A printer attached when you run your programs you should avoid using the Q register for data storage.

Status register a can be used by any program that will not cause the subroutine depth to exceed 2. This means that if the program contains no XEQ instructions it must not be called as more than a first level subroutine. If a routine that uses status register a is called as a second level subroutine, the END or RTN in the main calling program may not halt execution as it should. If register a wasn't empty (zero) a RTN will be attempted to an address given partially by the former contents of register a. You should also realize that any XEQ or RTN will disrupt the contents of the a register, shifting it by two bytes. Don't execute PSIZE (from the Extended Functions module) with anything in status register a either. The calculator will think that your data is a set of return addresses and it will adjust them as if they were return addresses to be revised according to the new SIZE. All this should be more clear after you read Chapter 6.

Problems (Solutions follow Chapter Six)

2.1 Using synthetic TONE P and normal TONE 8, construct a sequence of instructions to produce a Morse code "CQ" (dah-di-dah-dit, dah-dah-di-dah).

2.2 Using the byte grabber, make the synthetic instruction -E1. Hint: Make E1 first.

2.3 Using RCL d / STO d , write a short routine to view all ten digits of the number in the X register without altering the display mode. Hint: Modify the routine below so that the display mode is restored.

```
Ø1 LBL"VX"
Ø2 "  "      (2 spaces)
Ø3 SCI 9
Ø4 ARCL X
Ø5 AVIEW
Ø6 END
```

2.4 Using a RCL b / STO b loop, compute the Golden Ratio x = 1+1/x, displaying successive approximations.

2.5 a) Construct a sequence using synthetic text instructions that will generate a prompt "X(n)=?", where n is an integer from data register ØØ.

b) Modify this sequence to preserve the display mode.

2.6 Construct an output labeling sequence that will display "OUT=xμV" without altering the display setting, where x is to ARCLed in FIX 2 from the X register.

2.7 Construct a complete MOD function that operates like a built-in function. Registers Z and T are to be preserved, L replaced by x, Y by y mod x, and X by (y-y mod x)/x. You will need to use a scratch register such as M.

2.8 Using the byte grabber, make the two-byte instruction hex F1 FØ (a single-character text instruction, where the character is hexadecimal FØ).

CHAPTER THREE

BYTE LOADING

If you constructed the examples of Chapter 2 by using the byte grabber, you will probably agree that the byte grabber is a powerful tool for rapidly creating many types of synthetic instructions. However, if you need to create several synthetic instructions at a time, another approach may be even faster. A special program, called a byte loader, can be used to create the desired instructions, loading them directly into program memory. You need only specify the decimal value (0 to 255) for each byte in the desired sequence.

The theory behind byte loaders is described in the PPC ROM User's Manual under LB and also in the December 1980 PPC Calculator Journal. Byte loading programs were pioneered by several PPC members, including William Cheeseman, Roger Hill, John McGechie, William Wickes, and the author. This book will confine itself to a discussion of how byte loaders are used.

There are three different byte loading programs that are available for your use in this chapter. The first of these is called "LB" (load bytes) and requires only a "bare" HP-41 to operate. This byte loader program, written by Clifford Stern, occupies 214 bytes and fits on a single magnetic card.

The second is the PPC ROM program LB , a superb byte loader written by Roger Hill. If you have a PPC ROM, familiarize yourself with the instructions for LB . They are similar, but not quite identical, to those for "LB".

The third byte loader, called "LBX", requires an Extended Functions Module. This program, also written by Clifford Stern, is a shorter, faster version of "LB" that makes extensive use of Extended Functions module functions like XTOA. If you decide to use "LBX", refer to problem 3.5 for the program listing.

Despite its compactness, "LB" does most of what the PPC ROM version **LB** does, lacking only such dispensable conveniences as interruptibility and cleanup messages. All the conveniences of the ROM version could not be incorporated without unduly enlarging the program. ROM programs are not constrained by length because they don't take up any of the user memory. In any case, what "LB" gives up in amenities, it gains in speed. If you have an Extended Functions Module, you should probably use "LBX" (see problem 3.5), since it is both shorter and faster than "LB".

If you have access to an HP-41 optical wand, you have the option of entering "LB" or "LBX" directly from barcode. Appendix E contains barcode for all the utility routines in this book, providing a fast, error-free method to enter these synthetic programs into your HP-41. Be sure to use a protective plastic sheet to avoid damaging the barcode. Of course if you would like more practice with the byte grabber, you can ignore the barcode for now.

If you do not have a PPC ROM or an Extended Functions Module, start with the following instructions to create the synthetic lines needed for Clifford Stern's "LB" :

```
01 ENTER↑
02 STO IND 16        (Press STO shift 1 6)
03 MEAN              (Press XEQ ALPHA M E A N ALPHA)
04 STO IND 17
05 RDN
06 STO IND L         (Press STO shift decimal L)
07 CLD               (Press XEQ ALPHA C L D ALPHA)
08 ENTER↑
09 ENTER↑
10 LBL 01
11 STO IND 78
12 RDN
13 STO IND 78
14 AVIEW             (Press ALPHA shift R/S ALPHA)
```

```
15 STO IND 78
16 AVIEW
17 STO IND 17
18 RDN
19 STO IND 78
2Ø AVIEW
21 STO IND 78
22 AVIEW
23 STO IND 78
24 RDN
25 STO IND 17
26 LASTX
27 STO IND 78
28 LASTX
29 STO IND 78
3Ø SDEV
31 STO IND 17
32 SDEV
33 STO IND Y          (Press STO shift decimal Y)
34 CLD
35 ENTER↑
36 STO IND 78
37 SDEV
38 STO IND 16
39 RDN
4Ø STO IND 17
41 SDEV
```

Now grab and delete the STO bytes from lines 4Ø, 38, and 36 (for example for line 4Ø GTO .Ø39, press the byte grabber key, and backarrow). Backarrow line 35 (do not PACK) then grab and delete the STO bytes from lines 33, 31, 29, 27, 25, 23, 21, 19, 17, 15, 13, and 11. Delete lines Ø8 and Ø9 (again, do not PACK), then grab and delete the STO bytes from lines Ø6, Ø4, and Ø2. Delete line Ø1 and key in the nonsynthetic lines that are required to complete the following listing of "LB". Line 61 is a text line containing a single space. Use 1E4 for line 72. If you like, the byte grabber can be used to remove the leading 1. In fact, if you're getting into the spirit of synthetic programming, you'll probably want to replace the "1" digit entries by "E" synthetic digit entry instructions.

If you're using the Extended Functions version of "LB", the above procedure gives you all the synthetic lines you need (plus a few extras to be deleted), except for line 34, STO N. To form this line, start with STO IND 17, LASTX, and grab and delete the STO byte.

Clifford Stern's byte loader "LB":

```
01♦LBL 01
02 CLST
03 BEEP
04 STOP
05 GTO "++"

06♦LBL "LB"
07 FS? 50
08 GTO 02
09 1
10 ENTER↑
11 ENTER↑
12 CLA
13 CF 21
14 AVIEW
15 -10
16 GTO "++"

17♦LBL 02
18 7
19 /
20 INT
21 FIX 0
22 CF 29
23 ARCL X
24 "⊦ REGS."
25 TONE 8
26 AVIEW
27 PSE
28 RCL b
29 STO [
30 "⊦••x̄"
31 X<> [
32 X<> d
33 CF 04
34 CF 05
35 CF 06
36 FS?C 07
37 SF 05
38 FS?C 08
39 SF 06
40 FS?C 09
41 SF 07
42 FS?C 10
43 SF 09
44 FS?C 11
45 SF 10
46 FS?C 12
47 SF 11
48 X<> d
49 INT
50 DEC
51 1
52 +
53 .1
54 %
55 +
56 +

57♦LBL 03
58 1.007
59 ENTER↑

60♦LBL 04
61 " "
62 ARCL Y
63 "⊦?"
64 AVIEW
65 STO [
66 RDN
67 STOP
68 FC?C 22
69 GTO 05
70 OCT
71  E4
72 +
73 X<> d
74 FS?C 19
75 SF 20
76 FS?C 18
77 SF 19
78 FS?C 17
79 SF 18
80 FS? 15
81 SF 17
82 FS? 14
83 SF 16
84 X<> d
85 X<> [
86 "⊦**"
87 STO \
88 ARCL Y
89 X<> \
90 ISG Y
91 GTO 04
92 SIGN
93 X<> c
94 LASTX
95 STO IND T
96 X<>Y
97 STO c
98 R↑
99 DSE X
100 GTO 03
101 GTO 01

102♦LBL 05
103 "⊦•"
104 ISG X
105 GTO 05
106 X<> c
107 RCL [
108 STO IND Z
109 X<>Y
110 STO c
111 GTO 01
112 END

LBL'LB
END            214 BYTES
```

Notes: suffix [means M — line 30 is hexadecimal F4 7F 00 00 02

suffix \ means N — line 62 is a single space

line 103 is hexadecimal F2 7F 00

Check your program very carefully against the listing. As with any program that uses status register c, any errors in it might be sufficient to cause MEMORY LOST when you run it. Therefore it is a good idea to record the program on a magnetic card so you will not have to start all over again because of a minor mistake. Note that some of the synthetic lines are displayed differently than they appear in the printed listing. For example line 30 displays as ᵀ⊦¯¯▒ and line 103 displays as ᵀ⊦¯ . The instructions that involve status registers M and N also appear differently in the listing than in the display. M is printed as [and N as \. This correspondence, which is important for several of the status registers, is illustrated in row 7 of the QRC. For example the suffix O prints as].

INSTRUCTIONS:

Here's the procedure for using Clifford Stern's "LB". The procedure for the PPC ROM's **LB** is substantially similar; details can be found in the PPC ROM user's manual.

At whatever location in program memory where you want to create a group of synthetic instructions, key in the sequence

LBL"++"
+
+
+
etc.

+
+
XEQ"LB" .

(If you're using the PPC ROM, this last instruction will change itself to XROM"LB".) The number of + instructions should exceed the number of bytes you want to create by 16.

If you didn't key up the above set of instructions in sequence, that is to say if you went back and inserted more +'s, you should PACK. If a multiple of 7 +'s was inserted then you don't need to PACK. The reason for this will be apparent after you read Chapter 5.

Since you'll be using "LB" frequently, it is a good idea to record the LBL"++" sequence on a card. If you key in 99 +'s (so that line 1Ø1 is XEQ"LB"), GTO.., and GTO"++", the sequence will fit on one side of a card. If you have an extended memory module you could key in "++", SAVEP, to create an extended memory file for the LBL"++" sequence. It could then be called up as necessary by GETP. The magnetic card approach has the advantage of being immune to MEMORY LOST.

At this point you can switch out of PRGM mode and XEQ "LB" from the keyboard or just press R/S if you're at the last line of the sequence. "LB" will first tell you how many registers are available for loading bytes, then it will prompt for each of the seven bytes that comprise each register. The number of registers available is INT((p-1Ø)/7), where p is the number of +'s that you keyed in. Table 3.1 is a handy quick reference to determine the number of +'s needed.

Number of +'s used	Number of registers available	Number of bytes available
Ø-16	Ø	Ø
17-23	1	7
24-3Ø	2	14
31-37	3	21
1Ø+7n	n	7n

Table 3.1. Number of +'s needed for "LB" setup.

In response to each prompt for a byte, you need merely key in the decimal equivalent (Ø through 255) of the desired byte and press R/S. **WARNING:** If you wish to correct a numeric entry before pressing R/S, you must press RDN (roll down) before keying in the correct entry. This is necessary because very important data is being held in the stack for use by "LB". This warning does not apply to the ROM version of **LB**.

When you have entered all the bytes that you need, just press R/S without a numeric entry. This terminates the byte loading process. If you run out of registers, "LB" will terminate automatically. Let's try an example.

Suppose you want to create a copy of the "CMOD" program from problem 2.6. Recall that the program listing (in the Solutions section that follows Chapter 6) included LB inputs:

	LB / **MK** inputs:
Ø1 LBL"CMOD"	
Ø2 X<>Y	
Ø3 STO M	145, 117
Ø4 X<>Y	
Ø5 MOD	
Ø6 ST- M	147, 117
Ø7 LASTX	
Ø8 ST/ M	149, 117
Ø9 CLX	
1Ø X<> M	2Ø6, 117

These decimal equivalents can be used to create the required 4 synthetic two-byte instructions.

Set up as described above with LBL"++", 24 +'s, and XEQ"LB". Switch out of PRGM mode and R/S. You'll see the message "2 REGS." followed by a prompt "1?". The "2 REGS." message means that you can create up to 14 bytes (2 registers times 7 bytes per register).

In response to the prompt "1?", key in the first decimal input, 145, and R/S. Key in responses to each of the prompts as shown below:

Prompt	Response
1?	145, R/S
2?	117, R/S
3?	147, R/S
4?	117, R/S
5?	149, R/S
6?	117, R/S
7?	2Ø6, R/S
1?	117, R/S
2?	R/S

The first seven inputs completed the construction of one register, which was then inserted into the LBL"++" area. This restarted the byte index at 1 (the first byte of the second register). Then pressing R/S without a digit entry in response to the prompt "2?" terminated the byte loading processing, completing the second register with NULL bytes and storing it in the LBL"++" area before halting. When "LB" halts you can press SST once to get to LBL"++". Then you can switch to PRGM mode and examine your new synthetic instructions. It is a simple matter to clean up the remaining +'s and key in the nonsynthetic part of the "CMOD" program.

As you can see, very little knowledge of synthetic programming is needed to operate the "LB" program. The only part of the process that requires such knowledge is the determination of what decimal inputs are needed to create the desired synthetic instructions. In Chapter 2 you gained much of this knowledge through using the QRC. For example you should be able to look at row 1 of the QRC to determine that -E1 can be created using LB inputs 28, 27, and 17.

There are still large areas of the QRC, particularly rows A through E, that have not been explained here. These areas are explained in some detail in Corvallis Division columns in the PPC Calculator Journal July, August, and September 1979 issues. This chapter will give an outline of these areas, together with specific references for more detailed information where appropriate.

What follows is a summary of how to determine which decimal inputs are needed to create a given instruction. In most cases you will also need to consult the QRC. Decimal values are found at the lower left corner of each box in the QRC. For example the decimal number 126 (row 7 column E) corresponds to either the AVIEW instruction, the suffix d, or the character Σ.

I. One-byte instructions

All these are nonsynthetic except for TEXT Ø (row F, column Ø, decimal 24Ø). Any decimal value from row Ø or rows 2 through 8 will create a nonsynthetic one-byte instruction unless it is preceded by another byte that requires a suffix.

Digit entry instructions will merge themselves into a single multi-digit numeric entry line unless they are separated by a null or some other type of instruction. Use decimal values from row 1, columns Ø through C, to make synthetic digit entry lines. For example -E-3 is decimal 28, 27, 28, 19.

II. Two byte instructions

Two-byte instructions have a prefix, or first, byte from the yellow shaded area of the QRC.

The first category of two-byte instructions is those in row 9, plus columns 8 through D of row A, and columns E and F of row C of the QRC. These take the first byte from the box containing the function name, plus a second byte from the box containing the desired suffix. Thus STO M is 145, 117; TONE C is 159, 1Ø4; RCL IND N is 144, 246; LBL X (local label) is 2Ø7, 115.

The second category of two-byte instructions contains the short form GTO instructions. These take the first byte from row B plus a second byte of zero. The zero is filled in by the HP-41 the first time the GTO is executed. The filled-in byte tells the processor the jump distance and direction.

The third category of two-byte instructions contains the GTO IND and XEQ IND instructions. These take a first byte of 174 (row A, column E). The second byte is Ø through 127 for GTO IND, or 128 through 255 for XEQ IND. Thus 174, 117 is GTO IND M, while 174, 245 is XEQ IND M.

The final category of two-byte instructions contains all XROM's. These are peripheral functions that reside in an external ROM (Read-Only Memory). When the peripheral is not plugged in, the function appears as XROM i,j , where i and j are two-digit decimal numbers from Ø to 63 (actually Ø to 31 for i). The number i designates the identity of the peripheral -- i is therefore called the ROM ID number. Certain peripherals contain two 4-kilobyte ROMs, each of which has its own ROM ID. The number j is a sequential number of the function (in Catalog 2 order) within the 4K ROM.

XROM instructions consist of a hexadecimal A (binary 1Ø1Ø) followed by two groups of six bits. The first group of six bits denotes, in standard binary, the identification number (Ø through 31) of the external ROM. For example, the printer is XROM 29, and the card reader is XROM 3Ø. The second group of six bits denotes, again in standard binary, the number (Ø through 63) of the function within the external ROM. For example, WSTS is the tenth function in the card reader. This can be checked by executing CAT 2 with the card reader in place and noting that WSTS is the tenth function name to appear after the CARD READER header. Thus WSTS is XROM 30, 10. In decimal byte numbers this is 167, 138 (See Figure 3.1) In general, the decimal byte number for XROM i, j are:

$$\text{byte } 1 = 160 + \text{INT}(i/4)$$
$$\text{byte } 2 = 64 * (i \bmod 4) + j$$

```
WSTS = XROM      30,    10
     = 1010 0111  1000 1010

        A    7    8    A

          167       138
```

FIGURE 3.1

A typical XROM instruction
and its decimal byte numbers.

III. Three-byte instructions

Three-byte instructions take a prefix, or first, byte from the green shaded area of the QRC.

The first category of three-byte instructions consists of the long-form GTO's. All GTO's that refer to labels other than 00 through 14 are three-byte GTO's. However with LB you can also create three-byte GTO's for labels 00 through 14. This valuable synthetic programming technique eliminates the 112-byte jump distance limitation normally associated with LBLs 00 through 14. It's not that you can't get to a LBL 00-14 with a normal two-byte GTO instruction; it's just that the GTO will be much slower. Jump distances of more than 111 bytes cannot be "remembered" by the GTO instruction as shorter ones can, because the binary form of the jump distance doesn't fit into the space allocated for it in the GTO instruction. The three-byte GTO instructions have a larger space for storing the jump distance, so there is no artificial constraint on jump distance.

Jumps to a short-form label (00 to 14) that are shorter than 112 bytes can use the normal two-byte GTO, while for longer jumps you should in most cases use a synthetic three-byte GTO. The difference between a three-byte GTO 14 and a three-byte GTO 99, other than the fact that the first is synthetic and the second is not, is that the first requires a one-byte label (LBL 14), while the second requires a two-byte label (LBL 99). Thus there is an overall savings of one byte by using the synthetic three-byte GTO instruction.

Three-byte GTOs require the following decimal inputs:

byte 1 = 208
byte 2 = 0
byte 3 = 0 to 127

Byte 3 designates the label number. For example 208, 0, 1 is a three-byte GTO 01, while 208, 0, 115 is GTO X (this requires a local LBL X -- decimal 207,115).

The second category of three-byte instructions consists of the non-alpha XEQ's. These are quite similar to the long form GTO's. The only difference is that the required byte 1 input is 224. Thus 224, 0, 98 is XEQ 98; 224, 0, 116 is XEQ L (which requires a LBL L -- decimal 207, 116).

To construct "compiled" GTOs and XEQs (that is, those for which the jump distance has already been filled in), refer to page 21 of the August 1979 PPC Calculator Journal for the detailed byte structure required.

The third type of three-byte instruction is the END instruction. The appropriate "LB" inputs to create an END are 192 and Ø followed by a third input that determine the type of END (see Table 3-2).

type of END	byte 3 LB input
packed END	9
unpacked END	13
packed .END.	41
unpacked .END.	45

TABLE 3-2
"LB" inputs for byte 3 of an END

Always pack immediately after creating an END or an alpha LBL in order to incorporate it into CAT 1.
The LBLs and ENDs in Catalog 1 form a linked list upward from the .END. , with the distance to the next higher LBL or END stored in the first and second bytes of the LBL or END. The encoding of the distance is the same as for a three-byte GTO or XEQ, except that the direction bit is not used. (The direction is always upward in program memory.) The instructions given here for creating ENDs simplify matters by allowing the calculator's PACK operation to fill in the correct distance for Catalog 1 linkage.

IV. Instructions involving ALPHA strings

Text strings require a leading byte from row F of the QRC (decimal 240 plus the number of characters in the string) as explained in section 2E. Each character then requires a single decimal input, usually between 0 and 127. For example "X(5)=?" is decimal 246 followed by the six character bytes 88, 40, 53, 41, 61, and 63.

Append instructions are text instructions which have an append symbol (row 7 column F = decimal 127) as the first character. The leading byte should be chosen to allow for the append symbol in the length of the string. For example "⊦@" is decimal 242, 127, 64.

Alpha GTO instructions are simply text lines preceded by a row 1 column D byte (decimal 29). Thus decimal 29, 243, 65, 66, 67 is GTO "ABC". Alpha XEQ instructions consist of a row 1 column E byte (decimal 30) followed by a text string. For example XEQ "FX" is decimal 30, 242, 70, 88. The mysterious W^{T} instruction found at row 1 column F is constructed much the same as an alpha GTO or XEQ, but it is only good for producing a crash condition that can be cleared by removing and replacing the battery pack.

Alpha labels are composed of 4 + n bytes, where n is the number of characters in the label. The appropriate LB inputs are 192, 0, 241 + n, 0, followed by the n character bytes. Thus LBL"A", a synthetic global (that is, CAT 1) label, is decimal 192, 0, 242, 0, 65. If you want the synthetic label to be assigned to a key, you'll need to use a nonzero value for the fourth decimal input. You'll also need to set a bit in status register ⊦ or e (see Section 6A). The correspondence of decimal byte codes and bit numbers to key locations is covered in the PPC ROM User's Manual under background for **MK** .

A much easier way to assign a synthetic global label to a key is to use the built-in function ASN. For any synthetic label that can't be assigned by ASN, you can use the Extended Functions module's PASN function. Only very strange labels like LBL ":" fall in the class that requires PASN.

<u>NOTE</u>: You should always PACK immediately after creating an alpha LBL or END in order to incorporate it into CATalog 1.

Practice with LB until you're familiar with creating the types of synthetic instructions that were introduced in Chapter 2.

Problems

3.1 Use LB to create the sequence of instructions

E
STO O
ST+ O
X<> O
STO M
ISG M
TEXT Ø
ΣREG IND M
VIEW O
FS? IND M
TONE E
"⊼⊼⊼⊤"

```
"⊦-T"
ASTO N
VIEW N
```

This set of instructions is not particularly useful, but it does illustrate a broad spectrum of synthetic instructions that can be individually quite useful.

3.2 Write a short nonsynthetic program to convert XROM numbers to the corresponding LB inputs. For an input of i ENTER↑ j the two outputs should be 160+INT(i/4) and 64*(i mod 4)+j as explained in the section on two-byte instructions. These two outputs are the decimal inputs required by LB to create XROM i,j.
Write a synthetic version of this program that replaces i and j by the two outputs without disturbing the contents of stack registers Z and T.

3.3 Illustrate the use of synthetic local labels by creating the sequence

```
LBL P       (not LBL"P")
TONE 37     (displays as TONE 7)
GTO P       (not GTO "P")
```

3.4 Create a synthetic CAT 1 alpha label longer than 7 characters, for example LBL"RPN CALCULATOR" .

3.5 If you do not have a PPC ROM, but you do have an Extended Functions module, here is a shorter, faster version of "LB", also written by Clifford Stern. The instructions for "LBX" are identical to "LB", and you can use "LB" to help key it up. The required LB inputs to create "LBX" can be found in the Solutions section following Chapter 6 if you're having trouble. If you plan to use "LBX" regularly, you should probably rename it "LB" and put away the original "LB".

```
01♦LBL 01
02 CLST
03 BEEP
04 STOP
05 GTO "++"

06♦LBL "LBX"
07 FS? 50
08 GTO 02
09 1
10 ENTER↑
11 ENTER↑
12 CLA
13 CF 21
14 AVIEW
15 -10
16 GTO "++"

17♦LBL 02
18 7
19 /
20 INT
21 FIX 0
22 CF 29
23 ARCL X
24 "├ REGS."
25 TONE 8
26 AVIEW
27 PSE
28 RCL b
29 "*"
30 X<> [
31 -2
32 AROT
33 RDN
34 STO \
35 ASHF
36 SIGN
37 ALENG
38 8
39 Y↑X
40 ATOX
41 *
42 512
43 MOD
44 ATOX
45 +
46 +
47 .1
48 %
49 +
50 +

51♦LBL 03
52 1.007
53 ENTER↑

54♦LBL 04
55 " "
56 ARCL Y
57 "├?"
58 AVIEW
59 STO [
60 RDN
61 STOP
62 FC?C 22
63 GTO 05
64 XTOA
65 X<> [
66 ISG Y
67 GTO 04
68 SIGN
69 X<> c
70 LASTX
71 STO IND T
72 X<>Y
73 STO c
74 R↑
75 DSE X
76 GTO 03
77 GTO 01

78♦LBL 05
79 "├♦"
80 ISG X
81 GTO 05
82 X<> c
83 RCL [
84 STO IND Z
85 X<>Y
86 STO c
87 GTO 01
88 END

LBL'LBX
END            160 BYTES
```

CHAPTER FOUR

SYNTHETIC KEY ASSIGNMENTS

4A. Key assignment programs

Byte loader programs are a big step forward in convenience from the byte grabber. Synthetic key assignment programs add even more convenience. A synthetic key assignment program can assign any one- or two-byte synthetic or nonsynthetic intruction to any key. For maximum convenience you can make a set of commonly used synthetic function key assignments and use LB to create any other synthetic functions that are needed in your programs.

Key assignment programs are similar to byte loaders in that decimal equivalents are used to construct bytes which are stored in the appropriate section of main memory. Rather than entering the decimal equivalents one at a time as with LB, you load the stack with two decimal byte numbers plus a row/column keycode.

The first key assignment programs were written by John McGechie in early 1980. They were a truly awesome achievement given the state of the synthetic programming art at that time.

Just as for LB, there are three different key assignment programs that are available for your use in this chapter. The first is called "MK" (Make Key assignments) and requires only the basic HP-41. This program occupies three tracks on two magnetic cards. It was written by Clifford Stern.

The second key assignment program is MK in the PPC ROM, written by Roger Hill. MK is a true masterpiece of synthetic programming and is virtually immune to user errors. If you have a PPC ROM, review the instructions for MK in the User's Manual.

The third program, called "MKX", requires an Extended Functions Module. Written by Tapani Tarvainen, it requires only one magnetic card. It is shorter and faster than "MK" or **MK** , and is more forgiving of user errors than either. The listing for "MKX" can be found at the end of this chapter under problem 4.4.

Although it is quite a short program, Clifford Stern's "MK" incorporates many of the desirable features of the PPC ROM's **MK** . As was the case for **LB** , all the conveniences and error traps of **MK** could not be incorporated in "MK" without unduly enlarging the program. However the most important error trap, KEY TAKEN, is implemented. A little error checking by the user instead of the program saves many bytes.

If you have an optical wand, you may enter "MK" or "MKX" directly into your HP-41 from the barcode in Appendix E. The first time, though, it might be better for you to practice using LB by keying up one of these programs.

"MK", which requires nothing but a "bare" HP-41, is listed below followed by the decimal inputs needed to create the synthetic instructions using LB. After you have used LB to create the synthetic instructions, fill in the nonsynthetic instructions in the normal way to complete the program. Once again the suffixes M, N, O, P, Q, and ⊢ appear as [, \,], ↑, _, and ⊤ respectively in a printed listing, although P and Q are not used in this program.

Note that lines 11, 2Ø, and 38 are not as they appear in the listing. Especially misleading is line 2Ø. Consult the list of "LB" inputs following the program listing to determine the composition of these and the other synthetic program lines.

```
01♦LBL "MK"
02 CLST
03 CF 02
04 CF 05
05 CF 06
06 CF 21
07 192
08 SIGN
09 X<> c
10 X<> Z
11 ""
12 RCL b
13 RDN
14 X<> IND L
15 X=Y?
16 GTO 02
17 X<> [
18 "├*"
19 STO \
20 "├*****"
21 X<> \
22 X<> IND L
23 R↑
24 ISG L
25 -
26 STO b

27♦LBL 01
28 "├****"
29 X<> ]
30 "KEY TAKEN"
31 TONE 0
32 AVIEW
33 PSE

34♦LBL 16
35 "PRE↑POST↑KEY"
36 TONE 8
37 AVIEW
38 ""
39 FS? 02
40 STO [
41 CLST
42 STOP
43 LASTX
44 XEQ 03
45 XEQ 03
46 R↑
47 X<0?
48 SF 05
49 ABS
50 STO \
51 R↑
52 X<> \
53  E1
54 MOD
55 X<>Y
56 LASTX
57 /
58 INT
59 4
60 DSE Z
61 X≠Y?
62 X=0?
63 ISG Z
64 ""
65 ST+ X
66 ENTER↑
67 R↑
68 *
69 ENTER↑
70 R↑
71 +
72 ST+ Y
73 RDN
74 FS? 05
75 +
76 R↑
77 RCL
78 R↑
79 XEQ 03
80 X<> T
81 X<Y?
82 SF 06
83 36
84 -
85 FS? 06
86 +
87 R↑
88 SIGN
89 FS? 05
90 RCL e
91 FC? 05
92 RCL '
93 STO \
94 FS? 06
95 "├*"
96 X<> \
97 X<> d
98 FS? IND Z
99 DSE Y
100 SF IND Z
101 X<> d
102 STO \
103 "├♦♦♦♦♦♦♦"
104 FC?C 06
105 "├*"
106 X<> ]
107 FS? 05
108 STO e
109 FC?C 05
110 STO '
111 X<>Y
112 X=0?
113 GTO 01
114 X<> c
115 RCL \
116 FC? 02
117 "├***"
118 RCL \
119 STO IND L
120 FS?C 02
121 ISG L
122 SF 02

123♦LBL 02
124 X<> Z
125 STO c
126 X<>Y
127 GTO 16

128♦LBL 03
129 R↑
130 OCT
131 STO \
132 CLX
133  E4
134 ST+ \
135 X<> \
136 X<> d
137 FS?C 19
138 SF 20
139 FS?C 18
140 SF 19
141 FS?C 17
142 SF 18
143 FS? 15
144 SF 17
145 FS? 14
146 SF 16
147 X<> d
148 X<> [
149 "├**"
150 STO \
151 "├*"
152 X<> \
153 STO [
154 END
LBL'MK
END         313 BYTES
```

LB inputs:

Line 09	206, 125	Line 11	241, 240*	Line 12	144, 124
Line 17	206, 117	Line 19	145, 118		
Line 20	247, 127, 42, 42, 42, 42, 42, 240*				
Line 21	206, 118	Line 26	145, 124	Line 29	206, 119
Line 38	241, 240*	Line 40	145, 117	Line 50	145, 118
Line 52	206, 118	Line 53	27, 17	Line 64	240
Line 77	144, 118	Line 90	144, 127	Line 92	144, 122

Line 93 145, 118	Line 96 206, 118	Line 97 206, 126
Line 101 206, 126	Line 102 145, 118	
Line 103 247, 127, 0, 0, 0, 42, 42, 42		
Line 106 206, 119	Line 108 145, 127	Line 110 145, 122
Line 114 206, 125	Line 115 144, 118	Line 118 144, 118
Line 125 145, 125	Line 131 145, 118	Line 133 27, 20
Line 134 146, 118	Line 135 206, 118	Line 136 206, 126
Line 147 206, 126	Line 148 206, 117	Line 150 145, 118
Line 152 206, 118	Line 153 145, 117	

*Indicates an invisible character from rows 8 through F in a text instruction.

Make very sure that you have keyed up "MK" correctly before you try to use it. As with "LB", MEMORY LOST is possible if this program is keyed up or used incorrectly. The theory behind "MK" is far too complex to discuss here. In fact, writing a SIZE 000 key assignment program (one that uses no numbered data registers) is the premier challenge in synthetic programming. In this book we shall confine ourselves to a discussion of how to use MK.

Instructions for using Clifford Stern's "MK"

1.) If you are using the time module, clear all alarms. Any alarms that are present when "MK" (or MK) is executed will be turned into garbage, rendered useless by normalization. You may replace the alarms after you've finished creating your synthetic key assignments. Section 4E presents a handy pair of routines that can automatically save all alarms in extended memory and bring them back from extended memory. Executing the "SA" (save alarms) routine before "MK" clears the alarms but saves them "off-line" for later restoration by "RA" (recall alarms). PPC ROM users should take note that alarms must be cleared before using PK or any routine that

calls LF (1K , +K , A? , or F?).
This restriction on alarms does not apply to "MKX" (see problem 4.4).

2.) Make sure that a sufficient number of key assignment registers is available before executing "MK". The number of free registers may be checked by executing GTO .ØØØ in PRGM mode. The number of key assignments that can be made using "MK" is twice the number of free registers, since each register can hold two key assignments. The PPC ROM's MK is more elaborate and can detect the absence of free registers, producing a "NO ROOM" error message.

3.) Execute "MK" to initialize the key assignment process. The program will find the first unused key assignment register so that previous key assignments are not disturbed. Never interrupt "MK" (or "MKX"). If you interrupt "MK", there is a small chance of getting MEMORY LOST. Restart "MK" immediately if you interrupt it. If you interrupt "MKX" you will not get MEMORY LOST, but you may lose access to Catalog 1. Therefore you should restart "MKX" immediately **without attempting to enter PRGM mode.** Your attempt to enter program mode may kick you out of the "MKX" program. This will force you to MASTER CLEAR to regain control unless you can find the former contents of status register c in the stack and execute a STO c. This will make more sense after Chapter 6.

4.) When the prompt "PRE↑POST↑KEY" appears, key in the three components of the key assignment -- decimal byte 1, ENTER↑, decimal byte 2, ENTER↑, user keycode (row/column), R/S. For example to assign RCL b to the 1/x key you would key in 144 ENTER↑ 124 ENTER↑ 12 R/S. The decimal equivalent of the RCL prefix is 144, the decimal equivalent of the suffix byte b is 124, and the row/column user keycode for the 1/x key is 12 (row 1 column 2 unshifted). The first two decimal numbers must be integers from Ø to 255, while the third input must be a valid user keycode. A user keycode is a decimal number of the form $\pm$rc, where r is the row number of the key, c is the column number of the key, and the sign is negative if the key is shifted. This is precisely the same form of keycode that is displayed momentarily when you execute ASN, or that is required as input for PASN (Extended Functions programmable assignment). Both **MK** and "MK" allow you to assign the shifted shift key (keycode -31), although "MKX" does not. If you do assign a function to the shifted shift key, a function that requires filling in a prompt is a good choice to prevent accidental execution.

Warning: Do not PACK, reSIZE, turn off, or use ASN when "MK" is halted for input, unless you are finished using it. Also do not disturb the alpha register or LASTX.

5.) When the prompt "PRE↑POST↑KEY" reappears (with the flag 2 annunciator set if you are using "MK"), you may enter the three inputs for a second key assignment. This will complete one key assignment register.

6.) The prompt "PRE↑POST↑KEY" will appear once again (without the flag 2 annunciator if you are using "MK"). requesting an input for the first key assignment of the next free register. Repeat steps 4 and 5 until you have made all the key assignments you want to make. Remember that you must not use more registers than the number if free registers that you observed before executing "MK".

7.) When you have made all the assignments you need, you may simply ignore the prompt for the next input. This is true even if your last assignment did not complete the register. However if you quit while flag 2 is set ("MK" only) you waste half a register unless you plan to fill it with a normal assignment using the built-in ASN function or its cousin, the Extended Functions module PASN function. Unlike "MK", ASN (or PASN) will always look for gaps in the key assignment registers before taking a new register.

8.) If you try to make an assignment to a key that is already assigned, the message "KEY TAKEN" will appear. At this point you have two choices. (But remember not to disturb ALPHA or LASTX.) Your first option is to clear the key of its assignment (ASN, ALPHA, ALPHA, key), re-enter the desired assignment information, and R/S. The second choice is to enter a new set of inputs specifying two decimal equivalents and a different user keycode.

As an example of the power of "MK", let's make the following synthetic function assignments:

STO b -11	STO d -12	STO M -13	STO N -14	STO O -15
RCL b 11	RCL d 12	RCL M 13	RCL N 14	RCL O 15
BG -21	X<> d -22	X<> M -23	X<> N -24	X<> O -25

The steps are as follows:

1) Manually clear any assignments from the top row, shifted and unshifted, and the second row, shifted only.
2) Check that at least 8 registers (15 assignments at two per register) are available by executing GTO .000 in PRGM mode.
3) Switch out of PRGM mode and XEQ "MK". Supply inputs as shown.

Flag 2 ("MK" only)	Input ("MK", MK , or "MKX")
clear	145, 124, -11, R/S
set	144, 124, 11, R/S
clear	145, 126, -12, R/S
set	144, 126, 12, R/S
clear	145, 117, -13, R/S
set	144, 117, 13, R/S
clear	145, 118, -14, R/S
set	144, 118, 14, R/S

clear	145, 119, -15, R/S
set	144, 119, 15, R/S
clear	247, 63, -21, R/S
set	2Ø6, 126, -22, R/S
clear	2Ø6, 117, -23, R/S
set	2Ø6, 118, -24, R/S
clear	2Ø6, 119, -25, R/S
set	backarrow or ignore.

These synthetic functions are sufficient for about two thirds of all synthetic program lines on average. For example only one third of the synthetic lines in "LB" and "MK" are outside this set of functions.

A few nonsynthetic functions are also handy to have assigned. Recommended are

ASN "X<>Y"	21	(press X<>Y key for 21)
ASN "RDN"	22	(R↓ key for 22)
ASN "SIZE"	23	(SIN key for 23)
ASN "PACK"	24	(COS key for 24)
ASN "DEL"	25	(TAN key for 25).

The first two of these assignments will eliminate the search for LBL F or LBL G when you press X<>Y or RDN in USER mode. This speeds response noticeably in many cases. The other functions are just handy to have immediately available, although the choice of key location is a matter of individual preference. PACK and DEL are useful with the Byte Grabber. The byte grabber or "LB" can be used to create any synthetic function that you don't have assigned to a key.

Although you would normally use ASN to assign nonsynthetic functions, as we did in this example, "MK" does allow assignment of nonsynthetic as well as synthetic functions. In response to the prompt "PRE↑POST↑KEY", simply key in a single decimal number from Ø to 255, followed by a keycode. For X<>Y the decimal equivalent is 113; for RDN it's 117. Check the QRC to verify the correspondence. For multibyte instructions, it's the same idea: DSE is 151, FC?C is 171, END is 192, GTO is 208, XEQ is 224, LBL is 207. Non-programmable functions use decimal byte numbers from row 0 of the QRC. For example to assign SIZE, PACK, and DEL using "MK", you would use the single decimal inputs 6, 10, and 2, respectively.

If you ever assign STO c or X<> c to a key you should either clear it as soon as you have finished keying up whatever program you're making or else plan to be <u>very</u> careful. Accidentally pressing STO c or X<> c gives a virtually certain MEMORY LOST.

For my own personal use, I find it convenient to have X<> c on the keyboard. To help prevent disaster I assign it to the relatively obscure location -21 (normally CLΣ). My complete synthetic keyboard looks like this:

column:	1	2	3	4	5
row 1 shifted			STO M	STO N	STO b
row 1 unshifted			RCL M	RCL N	RCL b
row 2 shifted	X<> c	X<> d	X<> M	X<> N	X<> O
row 2 unshifted	X<>Y	RDN	"EFT"	eGOBEEP	BG
row 3	no assignments				
row 4 shifted				DEL	
row 5 shifted	PACK				
row 6 shifted	SIZE	XROM T1	INT	XTOA	
row 7 shifted	STO Q			X<>__	
row 8 shifted	Q-LOAD				

I find that this arrangement of key assignments is easy to remember and requires very little switching in and out of USER mode when keying in synthetic programs, or even most other programs.

On row 1, 4 unused keys leave space for temporary program or function key assignments.

On row 2, "EFT" is a program described in problem 4.5. "EFT" allows you to execute Extended Functions or Time Module functions from the keyboard, calling them by number.

The eGOBEEP function is a synthetic one-byte key assignment that was discovered by Robert Edelen. Use the decimal inputs Ø ENTER↑ 167 ENTER↑ keycode R/S. When you press the key, the display shows eGOBEEP __ . If you fill in a decimal number k from Ø to 63, you'll get XROM 28,k , which includes the mass storage functions. If you fill in a k between 64 and 99, you'll get XROM 29,k-64 , which covers the full range of printer fuctions. For example PRKEYS is XROM 29,12, so eGOBEEP 76 will generate the PRKEYS command. The printer function PRP (print program) requires an ALPHA input. If you press eGOBEEP 77, you will not be prompted for the ALPHA input. Instead the byte-reversed contents of status register Q will be used, exactly as for the Q-loader, which is covered on the next few pages.

The "EFT" and eGOBEEP key assignments can be time savers after you've learned the numeric equivalents for the functions you use most often. A complete list of numeric equivalents for "EFT" and eGOBEEP is presented at the end of this chapter, accompanying the "EFT" program in problem 4.5.

Also on row 2 is the byte grabber, which requires decimal inputs 247 and 63 plus a keycode. On row 6, XROM T1 is a PPC ROM function that consists of a sequence of short synthetic tones. It provides a pleasant alternative to BEEP, at the cost of an additional byte in a program. XTOA is another assignment from the extended funtions module. Its usefulness will become apparent in the next section.

4B. The "poor man's byte loader"

The last two key assignments on the preceding synthetic function keyboard, STO Q and Q-LOAD, require additional explanation. Together with one of several byte-building programs, these assignments constitute a "poor man's byte loader". Assign these functions to convenient keys using "MK". The decimal byte values are 145, 121 for STO Q and 27, 0 for Q-LOAD. You'll also need the byte grabber and a RCL M key assignment which you should still have on the keyboard.

If you are fortunate enough to have an extended functions module, its XTOA function will serve very well as a byte builder. If you have a PPC ROM, its DC function will work. These functions take a decimal input between 0 and 255 from the X register and create the corresponding byte, which is then appended to the ALPHA register (meaning that it becomes the last byte in status register M). If you don't have an extended functions module or a PPC ROM, create this short synthetic routine to do the same job.

```
01♦LBL "DC"      10 FS?C 17       19 STO \
02 OCT           11 SF 18         20 "├*"
03  E4           12 FS? 15        21 CLX
04 +             13 SF 17         22 X<> \
05 X<> d         14 FS? 14        23 STO [
06 FS?C 19       15 SF 16         24 RDN
07 SF 20         16 X<> d         25 END
08 FS?C 18       17 X<> [       LBL'DC
09 SF 19         18 "├**"       END          54 BYTES
```

LB inputs:

Line 03	27, 20	Line 05	206, 126	Line 16	206, 126
Line 17	206, 117	Line 19	145, 118	Line 22	206, 118
Line 23	145, 117				

Note that this is the basic byte-building routine that Clifford Stern wrote for his "MK" and "LB" programs.

Use ASN to assign XTOA, **DC** , or "DC", whichever you are using, to a convenient key. Now we're ready to start. The Q-LOAD function creates a text instruction of up to 7 characters from the reversed contents of status register Q. For instance to create the string "HP'S #1", we would first create the string "1# S'PH" in the ALPHA register, perform a RCL M to extract it from the ALPHA register to X, then transfer it to status register Q and press the Q-LOAD key. Let's try it:

CLA		
49	XTOA	(Use **DC** or "DC" if you don't have
35	XTOA	XTOA. Some of these characters are
32	XTOA	nonsynthetic and can be appended
83	XTOA	directly, but it's probably not
39	XTOA	worth the bother.)
80	XTOA	
72	XTOA	

At this point you have the string "1# S'PH" in the ALPHA register. Now find a suitable place in program memory where you'd like to insert the text instruction "HP'S #1". If you don't already have such a place, just GTO .. and use the bottom of program memory. When you're at the right spot in PRGM mode, switch back to RUN mode and use key assignments to do RCL M, STO Q. Now switch back to PRGM mode and press the Q-LOAD key. You'll see the synthetic digit entry instruction E, which comes from the decimal value 27 of the Q-LOAD key assignment (see row 1 column B of the byte table). SST once to see the text instruction "HP'S #1". Press Q-LOAD again and you'll get the two synthetic instructions E and TEXT Ø. The first use of the Q-loader cleared status register Q. The second use therefore produced a text instruction with no characters. So in addition to its ability to create synthetic text instructions, the Q-LOAD key assignment provides a quick and easy way to get both the synthetic digit entry E and the TEXT Ø NOP instruction.

But the real power of the Q-loader is unleashed by using it in combination with the byte grabber. First you use the Q-loader to create a text instruction of up to seven characters, then you grab and delete the text prefix, releasing the character bytes to become instructions. The following rather lengthy example will illustrate the power of this "poor man's byte loader" technique. Follow through it very carefully a couple of times until you understand the techniques that are being used.

In this example we will create the synthetic instructions needed for the "CMOD" routine of problem 2.6. The four instructions are STO M, ST- M, ST/ M, and X<> M. The decimal equivalents are 145, 117, 147, 117, 149, 117, 206, and 117. We proceed from the last byte to the first one:

CLA
2Ø6 XTOA
117 XTOA
149 XTOA
117 XTOA
147 XTOA
117 XTOA

The first group of 7 bytes is now ready to be loaded into program memory. GTO .. and key in LBL "CMOD" as a place holder. Switch out of PRGM mode, RCL M, and STO Q. Now switch back into PRGM mode and press the Q-LOAD key. You'll see the familiar E instruction. Do not SST yet; instead press the BG key. This removes the text prefix from the Q-loaded text instruction. Backarrow twice to remove the grabbed byte and the E instruction. You now have

Ø1 LBL "CMOD"
Ø2 RDN

```
Ø3 ST- M
Ø4 ST/ M
Ø5 X<> M
.END.
```

It remains to load the STO byte. Switch out of PRGM mode and

```
CLA
145 XTOA .
```

Now GTO "CMOD", RCL M, STO Q, switch to PRGM mode, and Q-LOAD. PACK to remove the invisible nulls between this new Q-loaded text instruction and the seven bytes we loaded before. Still at the E instruction in PRGM mode, press BG and backarrow twice. SST through the program and you should see

```
Ø1 LBL "CMOD"
Ø2 STO M
Ø3 ST- M
Ø4 ST/ M
Ø5 X<> M
.END.
```

The STO byte was loaded in the text line. As soon as it was released from the text line, it absorbed the RDN byte, which became the suffix M.

With a little practice, this "poor man's byte loader" can be used to quickly create synthetic instructions with a minimal amount of setup. All that is required are key assignments for RCL M, STO Q, Q-LOAD, and BG, plus an extended functions module or a PPC ROM or the "DC" program, and of course, the QRC.

It is good practice not to create pieces of instructions with the Q-loader as we did in the first group of seven bytes in the above example. It would have been better to stop at the sixth byte, creating three instructions, then pick up the remaining two bytes on the second loading. This eliminates the need for time-consuming PACKing. The PACKing procedure was shown here because it is necessary when creating synthetic instructions that are more than 7 bytes long.

The only limitation of Q-loading is that trailing nulls are suppressed. Thus for example if you want to create the instruction hex F2 7F ØØ (append one null), you'll need to add a dummy "filler" instruction such as ENTER↑. For this example the full procedure is CLA, 131 (the ENTER↑ instruction), XTOA, Ø (null), XTOA, 127 (append), XTOA, 242 (TEXT 2 prefix), XTOA, move to desired location, RUN mode, RCL M, STO Q, PRGM mode, Q-LOAD, BG, and backarrow twice. You'll also have to get rid of the ENTER↑ following your new synthetic instruction. If the dummy 131 byte were not included, the steps Ø, XTOA, would not do anything and you'd end up loading only the two decimal bytes 242, 127.

Further discussion of Q-loading appears on page 27 of the October 198Ø PPC Calculator Journal.

4C. Pseudo-XROM previews

The only two-byte functions that are nonsynthetically assignable to keys are peripheral functions. When the corresponding peripheral is not plugged in, the function appears as XROM i,j when the key is held down, where i and j are two-digit decimal numbers from ØØ to 63. The notation XROM means that the assigned function resides in an external ROM (Read-Only Memory). The number i designates the identity of the peripheral -- i is therefore called the ROM ID number. Certain peripherals contain two 4-kilobyte ROMs, each of which has its own ROM ID. The number j is a sequential number of the function (in Catalog 2 order) within the 4K ROM.

When a key that carries a synthetic two-byte function assignment is depressed, the HP-41 assumes for purposes of displaying the fuction preview that the key assignment is a normal XROM function. If the two decimal bytes of the key assignment are x and y, the XROM numbers i and j that are displayed in the XROM i,j preview are

$$i = 4(x \bmod 16) + \mathrm{int}(y/64)\ ,\ \text{and}$$

$$j = y \bmod 64\ ,$$

where mod signifies the modulo function (see MOD in your HP-41 Owner's Handbook). For example ST+ IND M = 146,245 appears as XROM 11,53 while TONE Y = 159,114 appears as XROM 61,50. This correspondence can be visualized on the QRC. The column number of the first byte x is, in fact, x mod 16. This pins down i to four possible values, which are shown in row A of the QRC, at least for columns 0 through 7. For example, ST+ is in column 2. Checking column 2 of row A we see the notation XR8-11, indicating that the first of the two XROM numbers displayed will be 8, 9, 10, or 11.

The exact value of i is determined by which block of 4 rows the second byte y is in. The heavier horizontal lines on the QRC help you to visualize the block boundaries. Rows 0 to 3 correspond to the first value of i, rows 4 through 7 to the second, rows 8 through B to the third, and rows C through F to the fourth. If you then visually move the second byte up to a corresponding box in rows 0 to 3 (this is equivalent to taking y mod 64), you can read off the value of j from the second line of the box.

Let's continue with the ST+ IND M example. Since the IND M suffix is in the fourth group of 4 rows, the value of i is 11. Next we visually translate the IND M suffix from row F column 5 up to row 3 column 5, which is the corresponding position in the first block of 4 rows. Checking the decimal value at the bottom of the row 3 column 5 box, we see that the value of j is 53. So ST+ IND M previews as XROM 11, 53.

The XROM preview numbers reveal much about the assigned synthetic fuction, but they do not quite uniquely determine it. For example an assignment of DSE IND 10 previews as XROM 30,10, or as WSTS if the card reader is attached. This assignment is indistinguishable from the WSTS function until the key is released. If you're ever in doubt about the identity of a particular assignment, try it in PRGM mode first. But just in case it's a byte grabber, don't press it when you're in the vicinity of the .END. or any nonpermanent END. Remember the byte grabbing constraints from Chapter 1!

For more details on XROM preview correspondence see page 47 of the March 1981 PPC Calculator Journal. Page 45 of the August 1981 PPC CJ contains a fascinating article by Roger Hill on how the XROM correspondence can affect the behavior of synthetic key assignments in PRGM mode.

4D. The RCL b key assignment

Unique among assignable synthetic functions is RCL b. Unlike other key assignments, which aren't essential if one uses "LB", the RCL b key assignment is much more powerful than a RCL b instruction located in program memory. Executed from the keyboard, RCL b brings the current program pointer to the X register. Executed in a program the result would always be the same, namely the location of the RCL b instruction in program memory.

The result of a RCL b instruction is a program pointer encoded in the last two bytes of X, expressible in four hexadecimal digits. In the encoded form the pointer is not especially useful. Two routines are presented here that convert the RCL b program pointer to a decimal number of bytes. Two more routines provide a convenient way to determine the number of bytes between two locations in program memory.

The RAMBYT routine performs exactly the same function as PPC ROM routine **PD** . To use the RAMBYT routine, just go to any point in Catalog 1 program memory and press the RCL b assigned key in RUN mode. The result is a program pointer for that location. Execute RAMBYT (or **PD**) to convert this pointer to a decimal value.

The ROMBYT routine is similar to RAMBYT, except that it expects as input a program pointer from a ROM location. If you have a PPC ROM or any application ROM, you can try out ROMBYT. Just go to a label or any other location in the ROM, execute RCL b from the keyboard in RUN mode, and XEQ "ROMBYT" to see the decimal byte number corresponding to the program pointer.

The most common application of program pointer decoding is counting the number of bytes between two locations in a program. For instance you may wish to know the total byte count of a program. The RAMBC program determines the distance between two program pointers by using RAMBYT to decode each pointer, and subtracting the resulting decimal numbers. RAMBC is functionally equivalent to the PPC ROM routine CB (count bytes).

To illustrate RAMBC, let's find out now many bytes long the RAMBC/RAMBYT/ROMBC/ROMBYT group of routines is. PACK program memory if it isn't already packed. Go to LBL "RAMBC", RCL b in RUN mode, BST (to the END), RCL b in RUN mode, and XEQ "RAMBC". The result should be 156, indicating that the program is 156 bytes long, from the beginning of LBL "RAMBC" to the beginning of the END. If you want to include the END in your byte count, add 3 bytes to get 159. If the last line of the RAMBC program group is .END., your byte count will be up to 6 bytes more. In this case you can GTO.. and repeat the above RCL b procedure to get the true byte count.

Divide by 112 to find out how many tracks the program will require when recorded on magnetic cards. The END is recorded on the cards, but if you have a program that is 112 bytes without the END, you don't have to read in track 2. In a case like this the prompt for the last track can be backarrowed for both recording and reading in. The only thing on the last track will be the END, which carries no information.

A more advanced use of RAMBC is to determine whether a long-form (three-byte) GTO is required, or whether a short-form (two-byte) GTO will suffice. Short-form GTO's (GTO

ØØ through GTO 14) should only be used where the jump distance is less than 112 bytes. This allows the jump distance to be compiled, or stored in the instruction itself, the first time the GTO is executed. Subsequent executions will be much faster because the search for the LBL is avoided. Only long-form GTO's can store jump distances longer than 112 bytes, so that if you use a short-form GTO where the jump distance is too long, your program will be slowed down noticeably by the continual label searching.

To determine whether a two-byte GTO, and its corresponding one-byte label, can be used without losing the advantage of the compiled branch, first key in the GTO and LBL in their desired positions in the program. Use GTO nn and LBL nn, where nn is between ØØ and 14, inclusive. PACK to remove any superfluous nulls. Go to the line following the GTO instruction (if it happens to be the .END. insert a dummy instruction and PACK again) and RCL b in RUN mode. Then go to the corresponding LBL instruction (you can use BST, SST) and RCL b again. XEQ "RAMBC" to see the jump distance in bytes. If this jump distance is between -111 and +111 bytes, inclusive, then the two-byte GTO is sufficient. Otherwise you'll need a three-byte GTO.

An alternative procedure is to RCL b at the GTO instruction, SST to get to the LBL, RCL b, and XEQ "RAMBC". The result should be between -1Ø9 and +113, inclusive.

If you need a three-byte GTO, you can construct a synthetic one using LB inputs 2Ø8, Ø, nn, where nn is between ØØ and 14. Or you can key in the sequence STO IND 8Ø, ISG nn, BST twice, BG and backarrow to remove the STO byte. Either way, this allows you to use the one-byte LBL nn, saving one byte over the standard instructions GTO xx, LBL xx, for xx from 15 to 99. Once created, a synthetic three-byte GTO will never change to a two-byte GTO, and it will always compile the branch distance properly. It can be distinguished from a two-byte GTO by using RAMBC to determine its length in bytes.

Here are the listings for RAMBC, RAMBYT, ROMBC, and

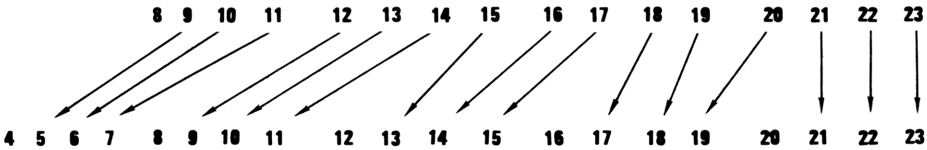
ROMBYT. ROMBC is of course analogous to RAMBC, except that it operates on ROM program pointers.

```
01♦LBL "RAMBC"
  02 X<>Y
  03 XEQ 01
  04 X<>Y
  05 XEQ 01
  06 -
  07 RTN

08♦LBL "RAMBYT"
  09♦LBL 01
  10 XEQ 03
  11  E41
  12 /
  13 INT
  14 LASTX
  15 FRC
  16  E4
  17 *
  18 DEC
  19 7
  20 *
  21 +
  22 RTN

23♦LBL "ROMBC"
  24 XEQ 02
  25 X<>Y
  26 XEQ 02
  27 -
  28 RTN

29♦LBL "ROMBYT"
  30♦LBL 02
  31 XEQ 03
  32  E37
  33 /
  34 DEC
  35 RTN

  36♦LBL 03
  37 "*"
  38 X<> [
  39 STO \
  40 ASHF
  41 "├♦♦♦A"
  42 X<> [
  43 X<> d
  44 CF 08
  45 FS?C 09
  46 SF 05
  47 FS?C 10
  48 SF 06
  49 FS?C 11
  50 SF 07
  51 FS?C 12
  52 SF 09
  53 FS?C 13
  54 SF 10
  55 FS?C 14
  56 SF 11
  57 FS?C 15
  58 SF 13
  59 FS?C 16
  60 SF 14
  61 FS?C 17
  62 SF 15
  63 FS?C 18
  64 SF 17
  65 FS?C 19
  66 SF 18
  67 FS?C 20
  68 SF 19
  69 X<> d
  70 END
LBL'RAMBC
LBL'RAMBYT
LBL'ROMBC
LBL'ROMBYT
END          159 BYTES
```

LB inputs:

Line 11	27, 20, 17	Line 16	27, 20	Line 32	27, 19, 23
Line 38	206, 117	Line 39	145, 118		
Line 41	245, 127, 0, 0, 0, 65				
Line 42	206, 117	Line 43	206, 126	Line 69	206, 126

The core of this group of routines is the LBL 03 subroutine, which uses a couple of tricks of the advanced synthetic programming trade. Its first four steps isolate the last two bytes of X in the ALPHA register. These bytes are then shifted left (line 41) and transferred to the flag register. At this point the 15 program pointer bits (the leftmost bit is not needed here) reside in flags 9 through 23. Flag operations are used to shift the bits into octal (base 8) format, with three bits per digit (see below). This leaves five octal digits in flags 4 through 23, with flags 4, 8, 12, 16, and 2Ø clear. These five octal digits are extracted from the flag register in the form a.bcde x $1Ø^{41}$. Regular arithmetic operations can then be used to separate the digits if necessary, after which the DEC function converts the digits to decimal. This trick of shifting bits into octal format and converting to decimal was pioneered by Roger Hill, the author of many routines for the PPC ROM.



You'll have to read the discussion of program pointer formats in Chapter 6 to understand the manipulation of the octal digits in the "RAMBYT" and "ROMBYT" routines.

4E. Saving and Recalling Timer Alarms **PPC ROM REQUIRED**

Most key assignment programs (except "MKX" -- see problem 4.4) have one feature in common: they will not work properly if any alarms are present, and they will disrupt the alarms as well. One solution is to manually clear the alarms using the time module's ALMCAT function. This is tedious and it requires writing down the alarm information and re-entering it later.

If you have an extended functions module and a PPC ROM, you can use Clifford Stern's "SA" (save alarms) and "RA" (recall alarms) to automatically transfer the alarms to extended memory, then back to main memory when you're done using the key assignment programs. "SA" uses the extended function module's SAVERX function, which, unlike RCL, permits extraction of data from main memory without normalization (Section 2C discussed normalization). Actually the first and last registers of the alarm block are normalized, but this damage is repaired by "RA".

Here are the instructions for using "SA" and "RA" :

1) Make sure there is at least one END somewhere above LBL "SA" in Catalog 1. This is necessary to permit the backwards GTO (line 66) to work properly with the curtain lowered. This will be explained in Section 6C.
2) After you have verified that there is at least one END above LBL "SA", XEQ "SA" to save the alarms in extended memory in a file named "ALM" and to clear the alarm data out of main memory. DATA ERROR at line 86 means there are no alarms to be stored. DUP FL at line 86 indicates that a file named "ALM" already exists in extended memory. Execute PURFL, then press R/S to complete program execution. NO ROOM at line 86 signifies that there aren't enough unused registers remaining in extended memory to store the alarms. At your option you may continue execution after purging one or more files and re-loading "ALM" into the ALPHA register.
3) Use any key assignment program you like. When you have your synthetic key assignments set up the way you want them, XEQ "RA" to restore the alarms and purge the "ALM" file from extended memory. The "RA" routine uses the Extended Functions module's programmable SIZE function if needed to open enough free registers below the .END. for the alarms. If the current total of free registers and SIZE is insufficient to accomodate the alarms, you'll get a DATA ERROR message at line 15. If this happens, PACK and/or clear a program and XEQ "RA" again. "RA" terminates with an OFF instruction, requiring you to turn the HP-41 back on. This OFF instruction is required to take care of the case in which you turn the calculator off after executing "SA" but before executing "RA". The Time Module saw no alarms the last time the calculator was turned off, so its countdown timer is not

active. The OFF instruction starts the Time Module counting down for the nearest alarm immediately, and enables it to advise you of any past-due alarms. A CLOCK instruction would serve the same purpose. For subroutine use, you may replace the OFF instruction by RTN, as long as you keep in mind the fact that if the calculator is turned off while the alarms are saved the Time Module's countdown timer will not be accurate until the next time you turn the HP-41 off.

Here's the listing of Clifford Stern's "SA" and "RA" :

```
01◆LBL "RA"
02 XROM "F?"
03 INT
04 XROM "E?" 25 "
05 X<>Y
06 -
07 SIZE?
08 ENTER↑
09 "ALM"
10 LASTX
11 +
12 FLSIZE
13 -
14 X<0?
15 SQRT
16 X<Y?
17 PSIZE
18 R↑
19 XROM "CX"
20 GETR
21 R↑

22 FLSIZE
23 ""
24 RCL [
       "
26 STO \
27 ARCL 00
28 RCL [
29 STO 00
30 X<> \
31 DSE Z
32 STO IND Z
33 R↑
34 STO c
35 "ALM"
36 PURFL
37 BEEP
38 OFF 57 "├

39◆LBL "SA"
40 XROM "OM"
41 176

42 XROM "E?"
43 17
44 -
45 X<Y?
46 GTO 03
47  E3
48 /
49 +
50 SIGN
51 "0"
52 X<> [

53◆LBL 01
54 ""
55 RCL IND L
56 X<> [
       "
58 X<> \
59 X≠Y?
60 GTO 02
61 ARCL c

62 X<> \
63 STO IND L
64 RDN
65 ISG L
66 GTO 01
67 CLA
68 GTO 03

69◆LBL 02
70 ARCL IND L
71 X=0?
72 CLA
73 X<> [
74 STO IND L

75◆LBL 03
76 ATOX
77 R↑
78 X<> c
79 LASTX
80 INT

81 ENTER↑
82 DSE X
83 ATOX
84 "ALM"
85 CF 25
86 CRFLD
87 +
88  E3
89 /
90 +
91 X<>Y
92 X<> c
93 X<>Y
94 SAVERX
95 XROM "BC"
96 X<>Y
97 STO c
98 BEEP
99 END

LBL'RA
LBL'SA
END        175 BYTES
```

LB inputs:

Line 23	241, 240*	Line 24	144, 117	Line 25	241, 170*
Line 26	145, 118	Line 28	144, 117	Line 30	206, 118
Line 34	145, 125	Line 47	27, 19	Line 51	241, 16
Line 52	206, 117	Line 54	241, 240*	Line 56	206, 117
Line 57	242, 127, 170*				
Line 58	206, 117	Line 61	155, 125	Line 62	206, 118
Line 73	206, 117	Line 78	206, 125	Line 88	27, 19
Line 92	206, 125	Line 97	145, 125		

*Indicates an invisible character from rows 8 through F of the QRC (decimal values 128 through 255).

Note that lines 25 and 57 contain the character AA_{16} (decimal 170), which is a printer control character that causes 10 spaces to be skipped. Printer control characters, discussed at the end of Section 2E, can cause even stranger behavior in program listings. The shaded characters in rows A through E of the QRC are printer control characters.

Problems

4.1 Review the solutions to the Chapter 2 problems and consider how synthetic key assignments could speed up keying in those programs.

4.2 Try keying up Clifford Stern's "LB" program by first using the "poor man's byte loader" technique to create the following instructions

hex F4 7F 00 00 02
E4
X<> c
STO c
hex F2 7F 00

X<> c

STO c

Fill in the rest of the synthetic instructions using your "working" keyboard of synthetic function assignments. You can then fill in the nonsynthetic instructions to complete the "LB" program.

4.3 Predict and verify the XROM number previews for the following synthetic key assignments:

a) TONE 89

b) X<> P

c) ISG IND N

4.4 Here is a new key assignment program that uses the Extended Functions Module. Called "MKX", it was conceived and written by Tapani Tarvainen, and revised and optimized by Clifford Stern. It uses a totally different approach, made possible by the capabilities of the PASN (programmable key assignment) function. Essentially, "MKX" uses PASN to make a dummy assignment to the designated key, then it finds and replaces that dummy assignment in the key assignment registers. "MKX" is sufficiently different from "MK" and **MK** that a separate set of instructions is called for:

1) Make sure you don't have a global label "ANUM" in any of your Catalog 1 (user) programs. If you do have a LBL "ANUM", executing "MKX" will place an FØ byte in the leftmost position of every register of user memory, including all programs and data. This is virtually equivalent to causing MEMORY LOST, since you'll probably decide to MASTER CLEAR rather than try to clean up the mess.

2) Load the stack with three inputs and execute "MKX". The three inputs required for "MKX" are the same as you would use for "MK" or **MK** . The difference is that you load the stack with the two decimal inputs and the

keycode (in Z, Y, and X, respectively, as for MK) before executing "MKX".

3) Alarms need not be saved or cleared. They will not be disrupted.
4) If you don't have enough free registers, you'll get PACKING, TRY AGAIN at line Ø4. This is much more forgiving than "MK".
5) Like "MK", "MKX" is not interruptible.
6) If you try to assign a key that is already taken, the new assignment will replace the old one, with no indication that this has occurred. If this isn't what you want to happen, check the key before executing "MKX".
7) To assign another key, simply load the stack with the three required inputs and execute "MKX" again or simply R/S since the last assignment left you at the top of the "MKX" program anyway.
8) There are no wasted half-registers with "MKX". Each new assignment is treated identically, and a new register is opened only if there are no existing "holes" to be filled in the assignment registers.

```
01♦LBL "MKX"
02 "ANUM"
03 CF 25
04 PASN
05 "ᵡiµ♦"
06 RCL [
07 R↑
08 XTOA
09 R↑
10 XTOA
11 RCL ᵀ
12 STO ]
13 X<> [
14 "├♦♦♦        B"
15 X<> ]
16 X<> [
17 STO \
18 "├"
19 X<> ]
20 R↑
21 X<> c
22 RCL \
23 .
24 SIGN
25♦LBL 01
26 X<> IND L
27 X<> [
28 "├*"
29 STO \
30 "├***"
31 X<> \
32 "├***"
33 X≠Y?
34 X<> \
35 X=Y?
36 SF 25
37 X=Y?
38 R↑
39 "├****♦"
40 STO ]
41 "├♦♦"
42 X<> ]
43 STO IND L
44 FC?C 25
45 ISG L
46 X=Y?
47 GTO 01
48 R↑
49 STO c
50 CLST
51 END

LBLᵀMKX
END            123 BYTES
```

LB inputs:

Line Ø5   245, 1, 1Ø5, 12, Ø, 24Ø*

Line Ø6   144, 117   Line 11   144, 122   Line 12   145, 119

Line 13   2Ø6, 117

Line 14 247, 127, Ø, Ø, Ø, 24Ø*, 166*, 66

Line 15 2Ø6, 119 Line 16 2Ø6, 117 Line 17 145, 118

Line 18 242, 127, 24Ø*

Line 19 2Ø6, 119 Line 21 2Ø6, 125 Line 22 144, 118

Line 27 2Ø6, 117 Line 29 145, 118 Line 31 2Ø6, 118

Line 34 2Ø6, 118

Line 39 245, 127, 42, 42, 42, Ø

Line 4Ø 145, 119

Line 41 244, 127, Ø, Ø, 24Ø*

Line 42 2Ø6, 119 Line 49 145, 125

*Indicates a character from the second half of the QRC, normally invisible in printed listings, but visible as a starburst in the display.

4.5 If you like the eGOBEEP key assignment that provides fast access to all the printer and mass storage functions, you may wish to try this short routine by Clifford Stern. It provides a capability similar to eGOBEEP for the Extended Functions and Time Modules.

Just key in the required stack input if any, ENTER↑, then key in the number of the desired function and XEQ "EFT". The "EFT" program will PAUSE for about a second to allow you to key in an ALPHA argument such as a file name. If the ALPHA argument you want was already in the ALPHA register, you won't have to key anything in. ALPHA inputs are limited to seven characters or less. "EFT" builds a short sequence of bytes containing the requested XROM instruction, then it executes the sequence. The byte sequence is actually contained in status registers b and a.

There are two notable constraints on "EFT". The first is that unlike eGOBEEP, "EFT" works only in RUN (non-PRGM) mode, so it cannot be used to enter program lines for Extended Function Module or Time Module

Functions. The second is that you must not use "EFT" to execute PSIZE (function number 3Ø), or to execute XYZALM (function number 93) where a nonzero Z input is needed. PSIZE will alter the byte sequence in status registers b and a that "EFT" is executing there. The XYZALM constraint is due to the fact that the Z register contents are altered to a value that is effectively zero by the time the XYZALM instruction is executed from the status registers. You should also avoid using "EFT" to execute PCLPS (function number 27) if this would clear "EFT" itself, because you would then begin executing the key assignment registers.

Incidentally, the reason for lines 15 and 23 is to defer any error stop until after the return to program memory. If you halt in the status registers, the processor takes a very long time to compute a line number.

```
01♦LBL "EFT"     08 CLX           15 SF 25       22 RDN
 02 RCL [        09 64             16 "├├t|"     23 FS?C 25
 03 CLA          10 +             17 RDN         24 STOP
 04 STO [        11 RCL [         18 X<> [       25 SF 30
 05 AON      12 "pTµu       "     19 X<> a       26 END
 06 PSE          13 X<>Y          20 X<> \     LBL'EFT
 07 AOFF         14 XTOA          21 X<> b     END          58 BYTES
```

Barcode for "EFT" can be found in Appendix E.

LB inputs:

Line Ø2 144, 117   Line Ø4 145, 117   Line 11 144, 117
Line 12 247, 145*, 112, 176*, 84, 12, 117, 166*
Line 16 245, 127, 127, 116, 145*, 124
Line 18 2Ø6, 117   Line 19 2Ø6, 123   Line 2Ø 2Ø6, 118
Line 21 2Ø6, 124

*Indicates an invisible printer character. The hex A6 (decimal 166) character in line 12 causes 6 spaces to be skipped.

Numeric function codes for "EFT" and eGOBEEP
(XROM numbers are also included for reference)

"EFT"
(XFUNCTIONS, TIME, WAND)

-EXT FCN 1B

1	ALENG	25,01
2	ANUM	25,02
3	APPCHR	25,03
4	APPREC	25,04
5	ARCLREC	25,05
6	AROT	25,06
7	ATOX	25,07
8	CLFL	25,08
9	CLKEYS	25,09
10	CRFLAS	25,10
11	CRFLD	25,11
12	DELCHR	25,12
13	DELREC	25,13
14	EMDIR	25,14
15	FLSIZE	25,15
16	GETAS	25,16
17	GETKEY	25,17
18	GETP	25,18
19	GETR	25,19
20	GETREC	25,20
21	GETRX	25,21
22	GETSUB	25,22
23	GETX	25,23
24	INSCHR	25,24
25	INSREC	25,25
26	PASN	25,26
27	PCLPS	25,27
28	POSA	25,28
29	POSFL	25,29
30	PSIZE	25,30
31	PURFL	25,31
32	RCLFLAG	25,32
33	RCLPT	25,33
34	RCLPTA	25,34
35	REGMOVE	25,35
36	REGSWAP	25,36
37	SAVEAS	25,37
38	SAVEP	25,38
39	SAVER	25,39
40	SAVERX	25,40
41	SAVEX	25,41
42	SEEKPT	25,42
43	SEEKPTA	25,43
44	SIZE?	25,44
45	STOFLAG	25,45
46	X<>F	25,46
47	XTOA	25,47

-TIME- C

65	ADATE	26,01
66	ALMCAT	26,02
67	ALMNOW	26,03
68	ATIME	26,04
69	ATIME24	26,05
70	CLK12	26,06
71	CLK24	26,07
72	CLKT	26,08
73	CLKTD	26,09
74	CLOCK	26,10
75	CORRECT	26,11
76	DATE	26,12
77	DATE+	26,13
78	DDAYS	26,14
79	DMY	26,15
80	DOW	26,16
81	MDY	26,17
82	RCLAF	26,18
83	RCLSW	26,19
84	RUNSW	26,20
85	SETAF	26,21
86	SETDATE	26,22
87	SETIME	26,23
88	SETSW	26,24
89	STOPSW	26,25
90	SW	26,26
91	T+X	26,27
92	TIME	26,28
93	XYZALM	26,29

- WAND 1F -

129	WNDDTA	27,01
130	WNDDTX	27,02
131	WNDLNK	27,03
132	WNDSUB	27,04
133	WNDSCN	27,05
134	'WNDTST	27,06

eGOBEEP
(HP-IL, PRINTER)

-MASS ST 1H

1	CREATE	28,01
2	DIR	28,02
3	NEWM	28,03
4	PURGE	28,04
5	READA	28,05
6	READK	28,06
7	READP	28,07
8	READR	28,08
9	READRX	28,09
10	READS	28,10
11	READSUB	28,11
12	RENAME	28,12
13	SEC	28,13
14	SEEKR	28,14
15	UNSEC	28,15
16	VERIFY	28,16
17	WRTA	28,17
18	WRTK	28,18
19	WRTP	28,19
20	WRTPV	28,20
21	WRTR	28,21
22	WRTRX	28,22
23	WRTS	28,23
24	ZERO	28,24
25	--	28,25
26	-CTL FNS	28,26
27	AUTOIO	28,27
28	FINDID	28,28
29	INA	28,29
30	IND	28,30
31	INSTAT	28,31
32	LISTEN	28,32
33	LOCAL	28,33
34	MANIO	28,34
35	OUTA	28,35
36	PWRDN	28,36
37	PWRUP	28,37
38	REMOTE	28,38
39	SELECT	28,39
40	STOPIO	28,40
41	TRIGGER	28,41

-PRINTER 2D

65	ACA	29,01
66	ACCHR	29,02
67	ACCOL	29,03
68	ACSPEC	29,04
69	ACX	29,05
70	BLDSPEC	29,06
71	LIST	29,07
72	PRA	29,08
73	'PRAXIS	29,09
74	PRBUF	29,10
75	PRFLAGS	29,11
76	PRKEYS	29,12
77	PRP	29,13
78	'PRPLOT	29,14
79	'PRPLOTP	29,15
80	PRREG	29,16
81	PRREGX	29,17
82	PRΣ	29,18
83	PRSTK	29,19
84	PRX	29,20
85	REGPLOT	29,21
86	SKPCHR	29,22
87	SKPCOL	29,23
88	STKPLOT	29,24
89	FMT	29,25
	--	

CARD READER (XROM 30) is not accessible through eGOBEEP.

CHAPTER FIVE

UNDERSTANDING PROGRAM EDITING ON THE HP-41

In Section 2B you were promised an explanation of how nulls are created when programs are keyed up and edited and under what conditions they can be removed by PACKing. This explanation is simplified by the construction of a very special synthetic instruction called an FØ label. The FØ label is capable of displaying several following instructions as text characters without actually absorbing them as the byte grabber does.

First construct this special synthetic instruction using "LB", with inputs 192, Ø, 24Ø. Alternatively, if you have the byte grabber assigned to a key, you may key in the instructions STO IND 64, RCL IND T, BST twice, BG, and backarrow to remove the STO byte. With either method you should PACK immediately so that the calculator can incorporate this synthetically-created LBL into Catalog 1. You now have a synthetic global label instruction. It is synthetic since its third byte is 24Ø decimal = FØ hexadecimal (hence the name FØ label). Normally the third byte of a Catalog 1 LBL instruction is 241 + n, where n is the number of characters in the label name. A third byte of 24Ø gives a name length of -1. It turns out that the calculator interprets this highly nonstandard length parameter in contradictory ways. For displaying the FØ label in PRGM mode, the processor uses n = 15, which is -1 modulo 16. So you see LBL^{T} followed by 15 characters. The processor skips one byte (which is normally the byte containing the key assignment information for the label), and displays the following 15 bytes as characters. However if you SST in PRGM mode you'll see that these character bytes have not really been absorbed into the FØ LBL instruction.

An example should make this point clear. But first a word of caution. Do not SST the FØ label in non-PRGM mode or run a program containing an FØ label. That will "crash" the HP-41, locking out the keyboard until the battery pack is removed and replaced to clear the crash. Removing the batteries halts an internal "infinite loop" condition, in this case without disturbing the memory contents. Executing an FØ label is one of the friendliest crashes. Others (such as byte-grabbing the .END. and deleting it) cause an almost almost-unavoidable MEMORY LOST.

Starting with your FØ label in the display (PRGM mode), key in the sequence of instructions -, *, /, X<Y? (Press XEQ ALPHA X shift COS Y ? ALPHA), X>Y?, X<=Y?, Σ+, Σ-, HMS+, HMS-, MOD, %, %CH, P-R, R-P, LN, X↑2, SQRT, Y↑X, CHS, E↑X, LOG, 1Ø↑X, E↑X-1, SIN, and COS. Now go back to the FØ label and you'll see

LBL "BCDEFGHIJKLMNOP"

(If you don't see this display, PACK and you should get it.)

The characters B through P are actually the instructions *, /, through LN, that follow the FØ label. Rows 4 and 5 of the QRC show the correspondence of instructions to these characters. To further illustrate this correspondence, locate and backarrow the / instruction and go back to the FØ label. You'll see

LBL "B¯DEFGHIJKLMNOP" .

This illustrates that when instructions are deleted, they are replaced by nulls, which are normally invisible. The overline character is the character representation of a null (hexadecimal ØØ = decimal Ø) byte. Now PACK and you'll see

LBL "BDEFGHIJKLMNOPQ",

which shows the removal of nulls by packing.

The FØ label enables us to see a striking demonstration of the operation of the processor when instructions are inserted in a program. Single step to the X<Y? instruction, corresponding to the character D, and insert a + instruction. Go back to the FØ label and you'll see

LBL "BD[illegible]¯¯¯¯¯¯EFGHIJ"

The @ character corresponds to the + instruction. But you probably didn't expect the six nulls (overline characters). This example illustrates that whenever an instruction is inserted where there is no room (that is, where an insufficient number of nulls are present), seven null bytes are opened for the new instruction, even though only one null may actually be used. The rest of program memory, down to and including the final .END. , is shifted down one register (seven bytes), decreasing the number of free registers by one. (Refer to Chapter 6 for a description of how program memory is organized and where the free registers are.) Because of the register operations available to the processor, this one-register shift is much faster than a one-byte shift would be.

Insertions where sufficient nulls are already present will not disturb the rest of program memory. For example, single step to the + instruction and key in the instructions STO Ø1, STO Ø2, STO Ø3, STO Ø4, STO Ø5, and STO Ø6. Go back to the FØ label and you'll see

LBL "BD@123456EFGHIJ" .

The six new instruction bytes exactly filled the available space. Any additional insertion would open another seven bytes.

Now that you have seen how insertion of instructions is accomplished by the processor, you can understand why the byte grabber works. When pressed in PRGM mode, the byte grabber creates a TEXT 7 prefix, followed by a null byte and a third byte that has always been decimal 63 in this book (MK can make it any value you like). A TEXT 7 instruction occupies 8 bytes of program memory, consisting of a one-byte TEXT 7 prefix followed by 7 character bytes. But the processor only knows that it has to make room for the three bytes that are being inserted. In the usual case there are no nulls present for the insertion, so 7 new ones are created. Therefore the eighth byte -- that is, the seventh character -- is taken from the existing program. Figure 5.1 illustrates the capture of this byte from program memory for the example of Chapter 1.

BEFORE

Instructions:	ENTER↑	STO	IND 31	PI
Hex equivalent:	83	91	9F	72
Decimal equivalent:	131	145	159	114

AFTER

Instructions:	ENTER↑	"-?-----▓"									TONE Y	
Hex:	83	F7	Ø	3F	Ø	Ø	Ø	Ø	Ø	91	9F	72
Decimal:	131	247	Ø	63	Ø	Ø	Ø	Ø	Ø	145	159	114

Figure 5.1 Creation of TONE Y using the Byte Grabber

The byte grabber can be used to grab up to 5 bytes if you like. Simply PACK or otherwise make sure there are no nulls ahead of the bytes you want to grab, just as you would for using the byte grabber normally. Then, before pressing the BG key, insert one to four bytes of "filler" instructions. For example, to grab two bytes you could insert a "filler" X<>Y before pressing BG. We did this in Chapter 2 to grab the 1 from exponential entry instructions without packing. To grab three bytes, you could insert the digit 9 and BG. To grab four bytes, insert EEX and BG. To grab five bytes, insert EEX 9 and BG. In all these cases, the idea is the same. The processor only requires three bytes for the byte grabber. If you open 7 bytes with an insertion and fill four of them (for example by inserting 1E 9) and press BG, the byte grabber will drop into the three remaining nulls. But since the TEXT 7 instruction is 8 bytes long, it must get its last 5 character bytes from the existing program.

Be very careful when grabbing more than one byte. You might accidentally grab part of the .END.. If you do this, don't backarrow! Immediately BST and BG again to release the .END. from the previous byte-grabber text line.

You might be under the impression that packing removes any and all nulls from a program. Not so. Occasionally a null carries essential information and cannot be deleted.

The first such case occurs when the null is located between successive numeric entry instructions. Let's continue where we left off with the FØ label, which when we left it looked like this:

LBL "BD□123456EFGHIJ" .

SST once to the - (subtract) instruction just ahead of the * instruction which corresponds to the character B. Key in the two successive numeric entry instructions 1E3 and 56. Switch into ALPHA mode and back to terminate the 1E3 instruction before starting on the 56. Now go back to the FØ label and you'll see

LBL "‾▦▦▦‾▦▦‾‾‾‾‾‾‾B" .

The first three starburst characters comprise the 1E3 instruction, while the next pair of starbursts is the 56 digit entry. Now PACK to see the result

LBL "▦▦▦‾▦▦BD□123456" .

All the nulls disappeared except the one between the two numeric entry instructions. That null is needed to prevent the two instructions from merging into a single program line. This is why a null between successive numeric entry instructions is nonpackable. The need for nulls to separate numeric entry instructions from each other explains the nulls we saw before packing in this example. The HP-41 operating system insists on adding a null in front of every numeric entry instruction at the time it is keyed in. This null will be removed by packing unless the previous instruction is also a numeric entry. The operating system also insists, for similar reasons, that there be at least one null separating the numeric entry instruction from the following instruction as the numeric entry is being keyed in. In the preceding example, seven bytes were opened up when the 6 of the 56 numeric entry was keyed in. If no bytes had been opened, there would have been no space isolating the 56 from the following program instruction. If that following instruction had been a numeric entry, the 56 would have merged into it to create a single (incorrect) numeric entry instruction. Thus at least one null separator byte was required. Since the HP-41 opens 7 bytes at a time, seven nulls were created.

Any null byte that is part of a multi-byte instruction is nonpackable. For instance the instruction ST+ ØØ appears in an FØ label as █¯ . The second byte is a null. This byte cannot be removed by packing, since it is part of an instruction and thus carries essential information, in this case the register number. Given the complex rules for removing nulls, it's no wonder that the PACK instruction can take a long time to execute.

One additional obscure point involving nulls deserves to be covered. Normally when you key in an instruction, it is inserted after the current instruction, overwriting any existing nulls and opening seven new nulls if space is needed. However if the current instruction is an END (or the .END.), the new instruction is inserted precisely where the END was, with the END being shifted down 7 bytes. This occurs even if there were sufficient nulls preceding the END.

To illustrate this behavior at ENDs, start with the sequence: FØ label, -, *, END. Go to the FØ label, PACK, and you'll see LBL^{T} B█¯█ followed by more characters. The second, third, and fourth characters visible are the END. Now delete the * instruction. If you inserted a new * instruction here it would exactly take the place of the old one. If however you SST to the END and then insert a new * instruction, the result is

LBL^{T} ¯B¯¯¯¯¯¯█ẋ█ plus four more characters.

The * instruction was inserted where the END used to be, while the END was shifted down 7 bytes. Six additional nulls were created where none were really needed. Therefore it is good programming practice not to make insertions into a program with the END in the display. Instead BST before making the insertion to take advantage of any nulls preceding the END. Of course PACK will eliminate the nulls anyway, but this technique may help you avoid having to resize to key in a program that barely fits in memory.

You'll note that in the last example the END changed its appearance when it moved. This is because part of the first two bytes of an END or a global alpha label is used to store a relative address to the preceding element in Catalog 1. Thus if Catalog 1 contains LBL "ABC", END, .END., then the .END. contains a pointer to the END, the END contains a pointer to LBL "ABC", and LBL "ABC" contains a blank relative address field, indicating the top of Catalog 1. The calculator uses this linked list, climbing the chain of labels and ENDs from the .END. up each time a global label search is undertaken. The linked list is also used for backstepping. When BST is pressed the calculator finds the nearest preceding global label or END and counts down from there to find the correct instruction. This is necessary because line number information is not stored in program memory. Without starting from a known position like a Catalog 1 label or END, the calculator cannot know whether a given byte constitutes an instruction or a suffix for a preceding instruction. The BST operation is implemented the only way it can be, by counting downward from a known position. This explains why BST can take so long near the end of a long program that has a lone global label at line Ø1.

Relative address information is also contained within local (non-text) GTO and XEQ instructions, as was mentioned in Chapter 3. The first execution of one of these instructions requires a time-consuming search for the corresponding LBL. But when this search is completed the relative address is filled in, allowing much faster branching on subsequent executions. With the FØ label it is possible to observe GTO and XEQ instructions before and after the relative address information is filled in. The structure of this relative address information is explained in detail on page 21 of the August 1979 PPC Calculator Journal.

Problems

5.1 Predict the result of the following steps, including the number and location of invisible nulls. Use the FØ label to verify your prediction.

a) Key in the instructions +, 3, -, 4, 5, and *. Insert Σ+ and Σ- after the +. Insert RCL Ø5 after the 4.

b) Key in the instructions +, -, XEQ ØØ, GTO 99, *, and /. Delete the GTO 99 and key in ST+ 75.

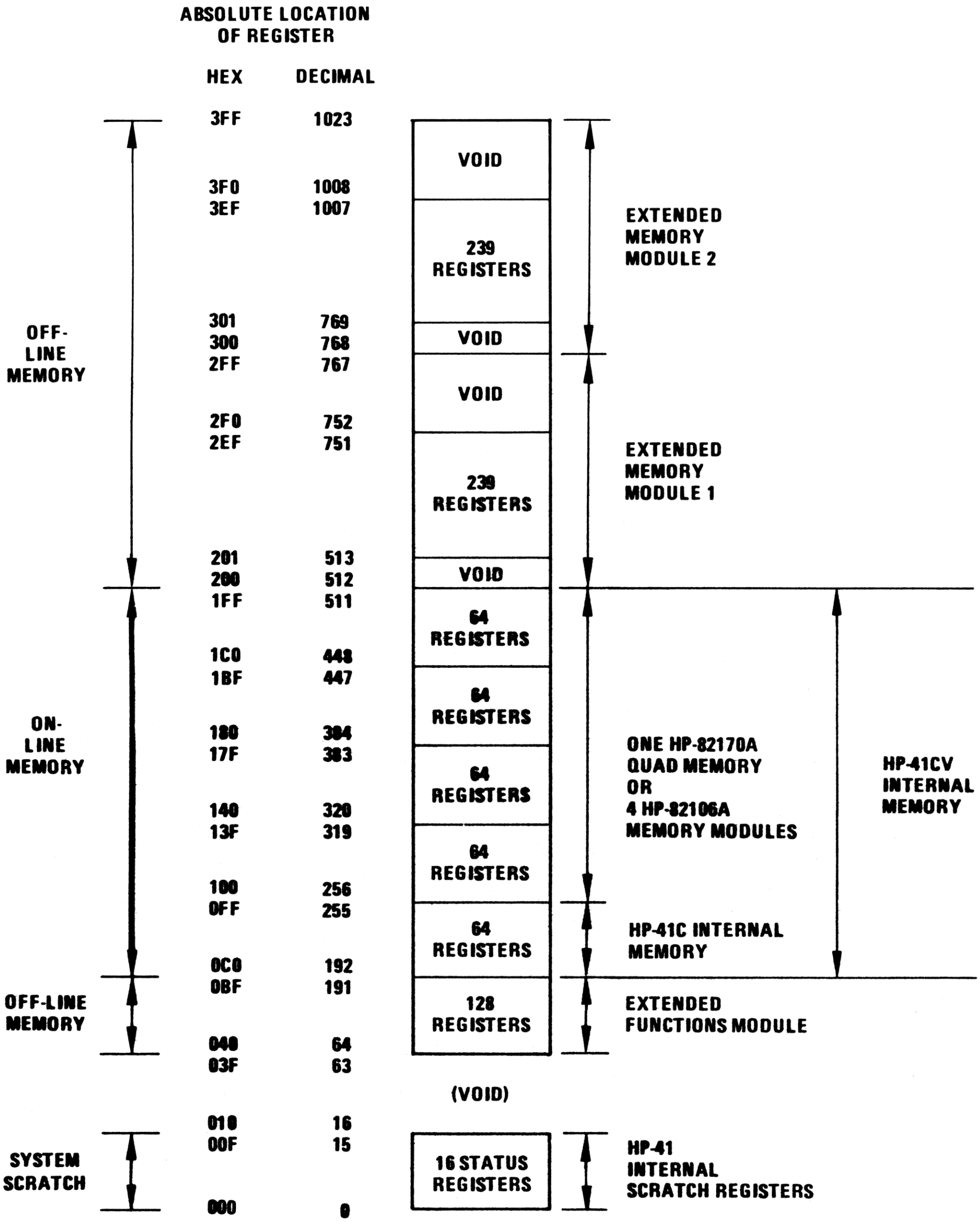


Figure 6.1 Overall Structure of HP-41 Memory

CHAPTER SIX

HP-41 MEMORY STRUCTURE AND STATUS REGISTER APPLICATIONS

This chapter will complete your knowledge of the basics of the workings of the HP-41. Some of the details given here may not be of immediate use, but they are presented to provide a reference. They also provide a point of departure for those of you who want to write your own "bit-fiddling" synthetic programs. Even if you plan only to use the simpler techniques of synthetic programming, and use "canned" synthetic programs from the PPC ROM or the HP User's Library for the fancy stuff, this information will help you get a general idea of how such "bit-fiddling" synthetic programs work.

6A. Memory Structure

Figure 6.1 on the facing page illustrates the organization of program, data, system scratch, and extended memory on the HP-41. The extended memory, including that portion contained in the extended functions module, is called off-line because programs cannot be executed directly from extended memory. They must first be brought into the main (on-line) memory.

Details of the contents and structure of extended memory can be found on page 18 of the March 1982 PPC Calculator Journal. Another article on page 26 of the April 1982 PPC CJ shows how synthetic techniques can permit execution of programs directly from extended memory.

The functional organization of main memory is shown in Figure 6.2 on the next page. The data registers extend upward from a partition (more about this when we discuss status

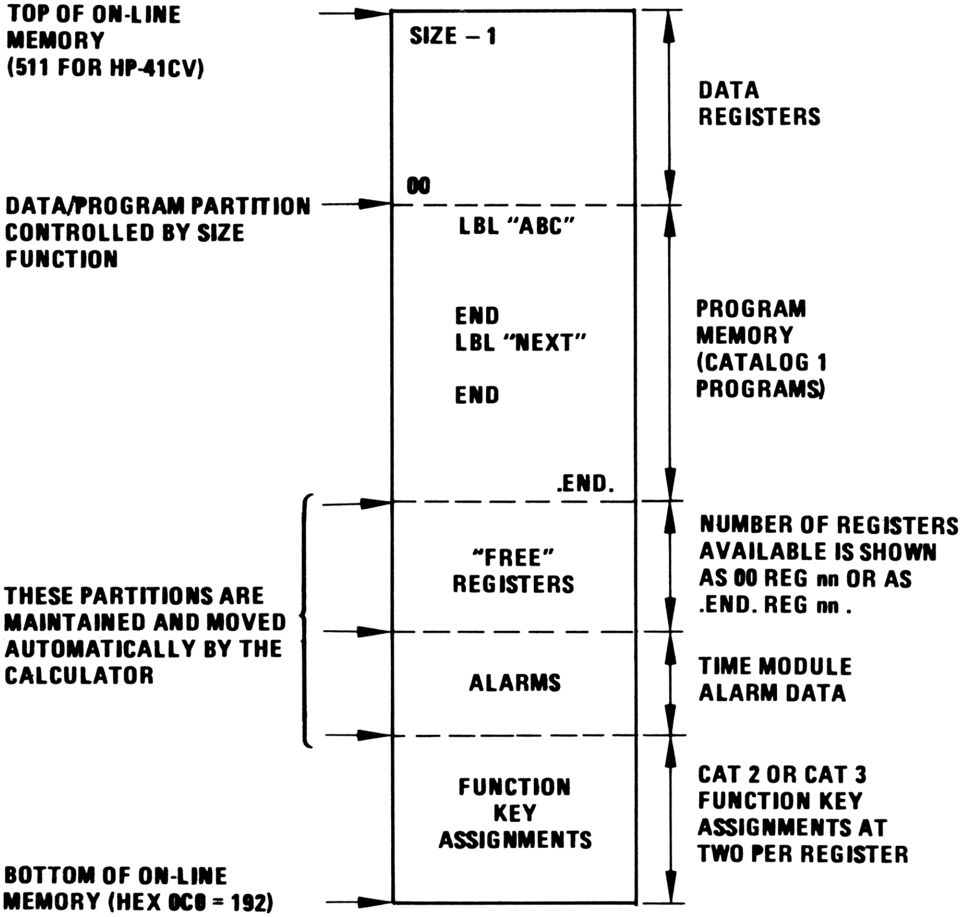


Figure 6.2 On-Line Memory Usage

register c) to the top of main memory. User programs extend downward from the same partition to the .END., which is moved automatically by the calculator as required. Below the .END. are the "free" registers -- those available for additional programs, timer alarms, or key assignments. They can also be converted to data registers by increasing the SIZE, which pushes down all data and programs into the free register block. Decreasing the SIZE pushes the program and data upwards in memory, adding to the number of free registers and causing some of the higher numbered data registers to be lost off the top of memory. The number of free registers present at any time can be checked by executing GTO .ØØØ in PRGM mode or else RTN in RUN mode then switch to PRGM mode. In either case the display will show ØØ REG nn, where nn is the number of free registers.

Below the free registers are the alarms and key assignments. Key assignments of Catalog 2 (peripheral) or Catalog 3 (built-in) functions occupy registers starting at decimal location 192 and proceeding upward. Each register that contains key assignments begins with a hex FØ marker byte. The other six bytes of the key assignment register contain a pair of function key assignments, each of which requires three bytes. Of these three bytes, the first two define the function. These are the two bytes that you provide decimal values for when using MK. The third byte defines which key the function is assigned to. The specifics of what byte is used to define a given key can be found in William C. Wickes's classic article on page 28 (second column) of the November 1979 PPC Calculator Journal. Page 28Ø of the PPC ROM User's Manual has a clear summary as well.

Timer alarms reside immediately above the key assignment registers. Each alarm requires one register for the alarm time, plus additional spaces if there is a message and/or a repeat interval associated with the alarm. One "header" register at the bottom of the alarm registers, just above the

	BYTE NUMBER WITHIN REGISTER: 6 | 5 | 4 | 3 | 2 | 1 | 0	REGISTER NUMBER
a	BIT MAP FOR SHIFTED ASSIGNED KEYS | SCRATCH | LINE NUMBER	15
b	USER FLAGS: 0 TO 29 | SYSTEM FLAGS: 30 TO 55	14
c	Σ REG POINTER | NOT USED | COLD START CONSTANT | CURTAIN POINTER | .END. POINTER	13
d	THIRD RTN | SECOND RTN POINTER | FIRST RTN POINTER | PROGRAM POINTER	12
e	SIXTH RTN POINTER | FIFTH RTN POINTER | FOURTH RTN POINTER | THIRD RTN	11
⊦	BIT MAP FOR UNSHIFTED ASSIGNED KEYS | SCRATCH	10
Q	TEMPORARY SCRATCH FOR ALPHA LBL, GTO, XEQ, OR WHEN KEYING IN DIGIT ENTRY INSTRUCTIONS	9
P	DISPLAY FORMAT | CAT LN NUMBER | ALPHA REGISTER (26) (25) 24 23 22	8
O	ALPHA REGISTER 21 20 19 18 17 16 15	7
N	ALPHA REGISTER 14 13 12 11 10 9 8	6
M	ALPHA REGISTER 7 6 5 4 3 2 1	5
L	LAST X REGISTER	4
X	STACK REGISTER X	3
Y	STACK REGISTER Y	2
Z	STACK REGISTER Z	1
T	STACK REGISTER T	0
	SIGN | ◄— MANTISSA (10 DIGITS) —► | SIGN | EXPONENT	

Figure 6.3 The Status Registers

uppermost key assignment register, is required to define the total number of alarm registers in use. Another register delimits the top of the alarms.

This completes the description of HP-41 memory structure, except for one very important area -- the status, or system scratch, registers. The name "status registers" is due to the fact that the contents of these 16 registers is recorded on track 1 of a status card by the card reader's WSTS function.

The 16 system scratch registers reside at the very bottom of the HP-41 address space, at locations Ø through 15 (decimal). The register names are T, Z, Y, X, L, M, N, O, P, Q, ⊢, a, b, c, d, and e, respectively. You are already familiar with most of these registers; the first five are described in your Owner's Manual, while several of the others were introduced in Chapter 2. Figure 6.3 is a brief summary of the processor's usage of these registers.

The stack registers, T, Z, Y, X, and L are available to the user through normal means. In addition to the ENTER↑, RDN, R↑, and LASTX instructions that have been incorporated in many HP calculators, the HP-41 allows direct access to all the stack registers through instructions like RCL Z or X<> L. With synthetic programming, the use of STO, RCL, and X<> can be extended to the other status registers as well.

Registers M, N, O, and P contain the 24-character ALPHA register. The ALPHA register contents are always right-justified in the status registers. The rightmost byte, byte Ø, of the M register contains the rightmost character. Byte 1 contains the second-to-last character, and so on. If the ALPHA register contains 7 or fewer characters, only the M register is used. As more characters are appended, the leading characters are bumped right-to-left then upward into registers N, O, and P. When the 24th position is filled (in register P), a warning tone sounds. Appending more characters will then push the leftmost characters into the scratch portion of register P. However if you remain in ALPHA mode, or at least have a non-numeric display, the four characters in positions 25 to 28 (the leftmost 4 bytes of P) will remain in place for extraction by synthetic methods such as RCL P. The Morse code program in Appendix B uses this 28-character capability.

The leftmost two bytes of P are used by the processor under some conditions. The first byte is an encoded representation of the numeric display status (FIX, SCI, ENG, Flag 28, Flag 29, and the number of digits). This byte is set up by the processor whenever a numeric display is needed or when a digit entry instruction is executed. The second byte of P is used for digit entry, whether it be manual or in a running program.

Executing the CATalog function also alters the first and second bytes of P. The first byte contains the catalog number (1, 2, or 3), while the second byte contains the line number within the catalog.

Details of the bit usage in the first two bytes of the P register can be found on page 13 of the July 1981 PPC Calculator Journal.

The Q register is used whenever an ALPHA label name is spelled out. This happens when the label instruction is keyed in or when the corresponding GTO or XEQ is keyed in or executed. The label name is placed, in byte-reversed order, in Q.

The Q register is also used during digit entry, whether manual or in a running program. The number is composed in Q before being transferred to the X register.

Details of Q register usage can be found on page 78 of the August 1981 PPC Calculator Journal. Be aware that the Q register is also used by the printer if one is connected.

The ⊦ register contains a bit map for the unshifted assigned keys in its first four bytes and half of the fifth byte. This is part of a clever technique that the HP-41 operating system uses to speed execution of functions from the keyboard. When an unshifted key is pressed in USER mode, the processor checks the corresponding bit of the ⊦ register. If the bit is clear, the processor knows that the key has not been assigned, and one of two actions is taken.

If the key in question is not in the top row or in the unshifted second row (ALPHA keys A-J and a-e), the default function (that is, the one that is printed on the key) is executed. If the key is in the top row or unshifted second row, a search of the current program for the corresponding local label (A through J or a through e) is initiated. If the label is found, program execution begins at that point. If the entire program is searched without finding the label, the processor (finally!) executes the default function.

If the bit in the ⊦ register is set the processor knows that the key has been assigned. It then searches for the key assignment information first in the key assignment registers. If no function assignment is found, the processor checks the key assignment byte (the fourth byte) in each global label in Catalog 1, from the .END. up to the curtain. If no global label assignment is found (this is not a normal case), then a function like CAT, ABS, or 1/x is executed.

Thanks in part to the key assignment bit map, the first step in the above USER mode execution sequence occurs quite rapidly. However the local label search can be very time consuming if the current program is more than 100 lines or so. This is why it is a good idea to assign X<>Y and RDN to their default keys. In USER mode the seemingly redundant function assignment takes precedence over the local label search, eliminating the delay associated with that search.

The rightmost two and a half bytes of the ⊦ register contain the hexadecimal code for the last function executed from the keyboard. The printer may make use of this area as well.

Registers a and b contain the program pointer and the stack of return pointers. Each pointer occupies two bytes, expressible in four hexadecimal digits. Bytes 1 and Ø of register b contain the current program pointer. When an XEQ instruction is encountered, this pointer is pushed onto the return stack -- that is, into bytes 3 and 2 of register b. If another XEQ is encountered before the RTN from the first one, the program pointer and the first return are pushed leftward two more bytes. The return stack in registers a and b can accommodate up to six pending return addresses in this way.

When a RTN instruction is encountered, the first return address in bytes 3 and 2 of register b is checked. If its value is zero, the current program pointer is retained and control returns to the keyboard. Otherwise the return stack is shifted leftward two bytes, with the former first return address being moved into the program pointer slot. Execution continues from that location in program memory, one step past the XEQ instruction that caused the return address to be pushed onto the return stack.

Now for a little technical detail on program pointers. The four hexadecimal digits of the program pointer are interpreted one way for RAM (read/write Random Access Memory) and another way for ROM pointers (those from a plug-in Read Only Memory). For RAM the first four bits denote the byte number within the register, while the other 12 bits denote the register's absolute address from the bottom of memory. The format is

Øbbb,ØØØr,rrrr,rrrr ,

where bbb denotes the byte number (expressible in three bits since the maximum value is 6 = Ø11Ø base 2) and where r,rrrr,rrrr denotes the register number (expressible in 9 bits since the maximum value is 511 = ØØØ1,1111,1111 base 2). For example Ø1Ø1,ØØØ1,1Ø1Ø,111Ø = hex 51AE denotes byte 5 of register 1AE (= 43Ø decimal). Byte numbers range from 6 to Ø as the program pointer moves downward through one register of a program. Thus 61AE is above 41AE in a program, and 41AE is above 61AD.

RAM return address pointers are the same as ordinary RAM pointers, except that the three bits that designate the byte number within the register are shifted to the right. These bits, normally the second, third, and fourth from the left of the 16-bit pointer, are shifted three positions over, to the fifth, sixth, and seventh bit positions. The RAM return pointer format is

ØØØØ,bbbr,rrrr,rrrr .

ROM pointers consist of a port address in the first four bits plus a 12-bit byte number within that port:

pppp,bbbb,bbbb,bbbb .

The port address part of a ROM pointer is not the same as the physical port number. The correspondence is:

port address	physical port or device
Ø	internal ROM Ø
1	internal ROM 1
2	internal ROM 2
3	not used
4	Service ROM module
5	Time module
6	Printer
7	Tape Drive (IL monitor)
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

Each port address can accomodate a 4 Kilobyte ROM (4Ø96 = hex FFF +1 bytes). The 12-bit byte number starts at zero and increases toward FFF as sequential ROM program instructions are executed.

Another important detail: When you RCL b in RUN mode at a specific line of program memory, the pointer value is usually one byte above the location where the instruction resides. Thus if a RCL M instruction is located in bytes 6 and 5 of register 1AE, and you RCL b at this line of program memory, the resulting pointer value will be Ø1AF hex, one byte above the actual location of the RCL M instruction. Where nulls are present, the pointer will be farther above the instruction. In fact it will be one byte above the group of nulls preceding the instruction.

Status register c contains essential pointer information needed to define the configuration of memory usage. Referring to Figure 6.3, we'll proceed right to left through the c register.

The last (rightmost) three hexadecimal digits of register c contain a pointer to the register containing the .END., which marks the bottom of user program memory. The .END. is always positioned in the rightmost three bytes of the register, with nulls preceding it as needed to occupy the space between the last instruction and the .END.

The next three hex digits of c contain a pointer to data register ØØ. This pointer, often called the "curtain", effects the separation of program and data memory. Any time the SIZE is changed, this pointer is adjusted and the contents of memory are shifted. Several short synthetic programs have been written to move the curtain, transforming program steps to data or vice versa. In Section 6C you will encounter one such program, together with an introduction to curtain moving. Within **LB** and **MK** are instruction sequences that temporarily place the curtain at Ø1Ø hex = 16 decimal. This allows program memory or the key assignment registers to be accessed by STO IND and RCL IND instructions. RCL will, of course, normalize the register contents. The previous contents of register c are held in the stack or in other status registers for replacement before the program halts. LB and MK illustrate the power of curtain control.

The next three hex digits of c contain the "cold start constant". These three digits are 1, 6, and 9 in every HP-41 manufactured so far. If the processor ever finds that these digits have been altered, it clears all of memory, giving the MEMORY LOST message in the display. The rationale behind this action is that since the processor never alters these digits, any alteration must be due to power failure. (No provisions were made for errant synthetic programmers.) Presumably other parts of memory would also have been altered, so clearing the memory is required to prevent an unsuspecting user from getting erroneous results. The main thing to remember about the cold start constant is not to store anything in c unless these three hex digits are 169, under penalty of MEMORY LOST. Incidentally, if the register immediately below the curtain pointer is nonexistent, you'll also get MEMORY LOST. So watch what you store in c.

The fourth and fifth hex digits from the left are apparently not used by the operating system or the printer.

The leftmost three hex digits of c constitute a pointer to the lowest register of the summation register block. For example if the curtain is at hex 1EB (SIZE Ø2Ø with full memory) and a ΣREG Ø1 command is executed, the ΣREG pointer will be set to hex 1EC which is 1EB + 1.

The d register contains all 56 flags. Byte 6, the leftmost byte, contains flags Ø through 7, while byte Ø contains flags 48 to 55. The flag register is used as the cornerstone of synthetic programming. Until the advent of the extended functions module, most bit manipulation could be done only by dropping one or more bytes of data into the flag register. Once in the flag register, the first thirty bits of the data can be directly modified as flags ØØ through 29. A prime example of this technique is the "RAMBYT" program of Chapter 4. You'll find pairs of X<> d instructions, separated by several lines of bit-fiddling flag operations, in many of the synthetic routines in the PPC ROM.

The e register contains a bit map for shifted assigned keys. This bit map is precisely analogous to the one for unshifted keys in the ⊢ register. It also occupies the leftmost four and a half bytes of the register.

The next two hex digits, half of byte number 2 and half of byte 1, are used as scratch by the processor.

The last three hex digits of the e register constitute the program line number. Since the line number is not stored with the instructions in program memory, and since instructions vary in length from 1 to several bytes, the processor must calculate the line number. This calculation is time consuming and must be redone every time you execute the Catalog function, SST a GTO or XEQ instruction in RUN mode, or otherwise jump to a location with an unknown line number. Because the calculation is time consuming, it is not performed in a running program. This speeds program execution, but it also causes a noticeable delay when you try to switch to PRGM mode after running a program. The processor will not show you the program instruction until it has computed the line number that goes with it. How does the processor know that the line number needs to be recomputed? It's simple. Before the processor starts running a program (SST execution does not count as "running a program" in this context), it sets the line number to hex FFF = decimal 4095. The line number remains FFF as the program is executed. When you try to SST or to switch to PRGM mode, the processor sees that the line number is FFF and automatically recomputes the correct line number for the current program pointer by counting down from the preceding END.

The mysterious line 4094 you saw in Chapter 1 when you created the byte grabber was due to the fact that when you pressed backarrow in ALPHA mode, the calculator decremented the line number by 1 without realizing that the FFF line number was invalid. The RCL Ø1 that you saw was a phantom instruction that appears when the program pointer register (status register b) contains zero.

6B. Status Register Application 1 -- Suspend Key Assignments

As part of its compatibility with HP-67 operation, the HP-41 has 15 keys (top two rows unshifted plus top row shifted) which, when pressed in USER mode, will find and execute the corresponding local label (A-J and a-e). But this feature conflicts with any global label or function key assignments to these keys, since the HP-41 gives precedence to function and global label assignments. How many times have you wanted to use the automatic assignment of local labels A-J and a-e, but found a function or global label key assignment in your way? You press LOG to execute LBL D, but instead you get another function that you have assigned to that key. Wouldn't it be nice if there were a way to temporarily eliminate the conflicting key assignment, then bring it back later?

Synthetic programming techniques permit this to be done, and the PPC ROM contains two routines that do it. You use **SK** to suspend the function and global label key assignments, and **RK** to reactivate them.

To use **SK** , simply key in a register number k, and XEQ "SK". The key assignment bit maps from status registers ⊢ and e are stored in data registers k and k+1, while the bit map areas in the ⊢ and e registers are cleared. Because the bit maps are clear, the calculator thinks that there are no key assignments present. Therefore you can press the LOG key in USER mode to execute LBL D. Any function or global label key assignments that are present are held in suspended animation.

When you want to reactivate the global label and function key assignments, just key in the same data register number k, and XEQ "RK". The contents of data registers k and k+l are recalled and put into status registers ⊢ and e. Since the calculator now has the proper bit maps, the key assignments operate normally again.

There is another way to reactivate your function key assignments. You need only read in a program card on the card reader. It doesn't matter whether you read the card in USER mode or not, but it must be a program card. This technique is valuable if you accidentally disturb data registers k and k+l that hold the key assignment bit maps after you execute "SK".

Let's analyze the workings of PPC ROM routines **SK** and **RK** (suspend and reactivate key assignments). If you don't have a PPC ROM, key in **SK** and **RK** using LB:

```
01 LBL "SK"              "LB" inputs:
02 SIGN
03 CLX
04 X<> ⊢                 206, 122
05 XEQ 14
06 ISG L
07 TEXT 0                240
08 .
09 X<> e                 206, 127
10 LBL 14
11 "*"
12 X<> M                 206, 117
13 STO N                 145, 118
14 ASTO IND L
15 RDN
16 RTN

17 LBL "RK"
18 SIGN
19 ARCL IND L
20 hex F2 7F 00          242, 127, 0
21 ISG L
```

```
22 TEXT Ø                240
23 ARCL IND L
24 hex F3 7F ØF FF       243, 127, 15, 255
25 X<> N                 2Ø6, 118
26 STO ⊢                 145, 122
27 X<> M                 2Ø6, 117
28 STO e                 145, 127
29 RDN
3Ø CLA
31 END
```

The accompanying "Stack and ALPHA Register Analysis Form" is an indispensible tool for step-by-step tracing of synthetic programs. You'll understand its value after you've used it to trace **SK** and **RK** .

When you execute **SK** , the register number k is first stored in LASTX by the SIGN function. Then an X<> ⊢ instruction is used to extract the contents of the ⊢ register and simultaneously clear it. The LBL 14 subroutine uses the ASTO function to store a six-character string in register k. This six-character string consists of an asterisk character followed by the first five bytes of the former ⊢ register contents. The asterisk is needed as a place holder in case the leftmost byte of the ⊢ register is zero. The three-step sequence "*", X<> M, STO N, sets up the ALPHA register contents for the ASTO operation, as you can see on the ALPHA register analysis form. Take the time to understand this three-step sequence if you want to write your own synthetic programs.

The rest of the **SK** routine performs a similar operation, extracting the contents of register e and clearing it, and storing a similar six-character string in data register k+1.

When you execute **RK** the data register number k is first stored in LASTX by the SIGN function. Then the six-character string is ARCL'ed from register k and shifted left one byte

Stack and Alpha Analysis Form

LINE	INSTRUCTION	L	X	Y	Z	T	P	O	N	M
53	LBL"SK"		x	y	z	t				
54	SIGN	x	1							
55	CLX		0							
56	X<>⊦		H							
57	XEQ 14									
62	LBL 14									
63	"*"						Cleared	Cleared	Cleared	*
64	X<> M		------*							1 2 3 4 5 6 7
65	STO N								*	
66	ASTO IND L									
67	RDN		y	z	t	------*				
68	RTN									
58	ISG L	x+1								
59	"" (NOP)									
60	.		0	y	z	t				
61	X<> e		e							
62	LBL 14									
63	"*"						Cleared	Cleared	Cleared	*
64	X<> M		------*							8 9 10 11 12 13 14
65	STO N								*	
66	ASTO IND L									
67	RDN		y	z	t	------*				
68	RTN									
84	LBL"RK"		x	y	z	t				
85	SIGN	x	1							
86	ARCL IND L									* 1 2 3 4 5
87	"⊦‾"									* 1 2 3 4 5 ‾
88	ISG L	x+1								
89	"" (NOP)									
90	ARCL IND L								* 1 2 3 4 5	‾ * 8 9 10 11 12
91	"⊦Φ"							*	1 2 3 4 5 ‾ *	8 9 10 11 12 0F FF
92	X<> N		⊦'						[illegible]	
93	STO ⊦									
94	X<> M		e'							1 2 3 4 5 ‾ *
95	STO e									
96	RDN		y	z	t	e'				
97	CLA						Cleared	Cleared	Cleared	
98	RTN									

STACK AND ALPHA ANALYSIS FORM

LINE	INSTRUCTION	L	X	Y	Z	T	P	O	N	M

by appending a null, though an asterisk would do just as well. Register k+1 is then ARCL'ed, shifting the previous string another six characters to the left. Two more bytes, hex ØF and FF, are appended, causing a further two-byte shift to the left. The ALPHA analysis form reveals all this action in detail.

At this point the N register contains the required 7 bytes for ⊦, while the M register contains the correct bytes for e. The last several lines of **RK** extract the contents of N and M, store them in ⊦ and e, and clean up ALPHA and the stack. Note that the last two bytes of e are ØF FF, requiring the calculator to compute a correct line number. Earlier versions of **RK** stored ØØ ØØ in the rightmost bytes of e, causing the line number to be incorrect if the program was single-stepped or run in TRACE mode.

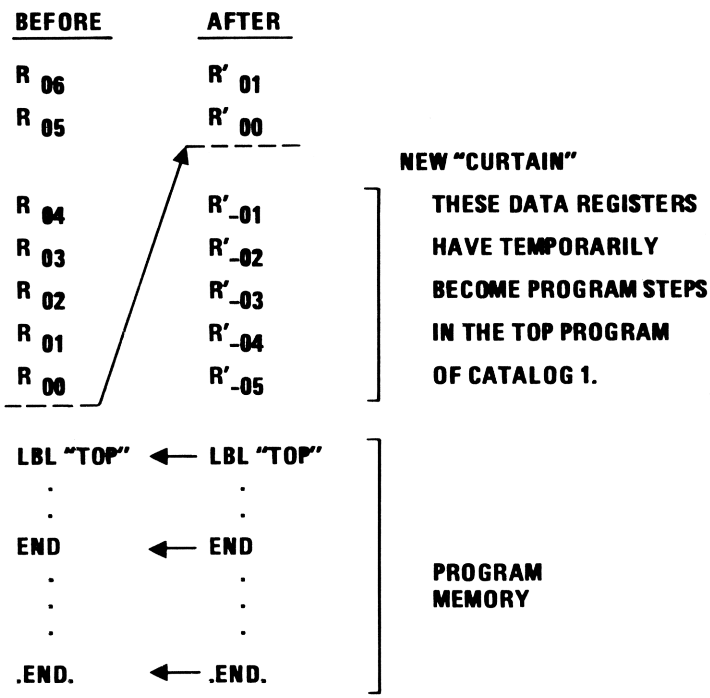
6C. Status Register Application 2 -- Register Renumbering

Suppose you have a program which calls a user-supplied program as a subroutine. A typical example would be a root finder program which finds a value of x such that f(x) = Ø. In this case f(x) is calculated by a user-supplied subroutine. The user supplies the name of the f(x) program, the root finder stores the name in a data register and calls it as needed with an XEQ IND nn instruction.

In writing such a root finder program, you have a difficult decision to make. The root finder will need to use some numbered data registers to hold its data, and it is essential that these registers not be disturbed by the user's f(x) program. No matter which registers you choose, there is always the possibility of a register usage conflict between the root finder and the f(x) program. You might try using data registers 5Ø and up for the root finder, figuring that most reasonable f(x) programs wouldn't be using those registers. But even if this would work, it is wasteful. In most cases the user's f(x) program won't use anywhere near 5Ø registers.

Synthetic programming provides a way out of this predicament. A short synthetic routine can reposition the curtain that separates data registers from program memory, effectively renumbering the data registers.

For example, suppose the root finder program uses the five data registers ØØ through Ø4. Just before calling the f(x) program, the root finder calls the synthetic routine "CU" (curtain up) to raise the curtain five registers. The figure below shows the effect of raising the curtain five registers. Although the contents of the registers haven't changed, a RCL ØØ will now extract the contents of what used to be called data register Ø5.



Similarly a RCL Ø1 instruction will produce the contents of what used to be register Ø6. The important registers that the root finder needs to protect from the user's f(x) program are now inaccessible by STO and RCL instructions. The contents of what used to be called data registers ØØ through Ø4 are now regarded as part of program memory by the calculator. In fact if you were to go to the top program of Catalog 1, you'd find this data at the top of the program. Of course it would appear in the form of program instructions rather than as numbers.

The important point is that after raising the curtain by five registers, the root finder program can call the f(x) program without fear that its essential data will be disturbed. The f(x) program will have free use of what it thinks are data registers ØØ and up.

When the f(x) program returns control to the root finder program, the first thing the root finder does is to lower the curtain back to the original location. This restores the original data register numbering and makes the root finder's data accessible again as data registers ØØ through Ø4.

The accompanying program listings for the curtain-raising routine "CU" and a typical root finder program "SOLVE" illustrate the principles we've been discussing. This version of "CU" was written by Tapani Tarvainen, and represents a major breakthrough from previous versions.

LB inputs for "CU":

Line Ø3 144, 125 Line Ø4 145, 117
Line Ø5 245, 127, Ø, Ø, Ø, 33
Line Ø8 2Ø6, 117 Line Ø9 2Ø6, 126 Line 1Ø 145, 119
Line 13 17Ø, 245 Line 15 24Ø Line 21 168, 245
Line 22 151, 117 Line 27 2Ø6, 119 Line 28 2Ø6, 126
Line 29 145, 117
Line 3Ø 244, 127, Ø, Ø, Ø
Line 31 2Ø6, 118 Line 32 2Ø6, 125

```
01♦LBL "SOLVE"
 02 "FNAME?"
 03 AON
 04 STOP
 05 ASTO 00
 06 AOFF
 07 "XGUESS1?"
 08 PROMPT
 09 STO 03
 10 "XGUESS2?"
 11 PROMPT
 12 -
 13 STO 02
 14 RCL 00
 15 LASTX
 16 XEQ 14
 17 STO 01

18♦LBL 10
 19 RCL 00
 20 RCL 03
 21 XEQ 14
 22 ENTER↑
 23 ENTER↑
 24 X<> 01
 25 -
 26 /
 27 RCL 02
 28 *
 29 CHS
 30 STO 02
 31 RCL 03
 32 +
 33 STO 03
 34 RCL 01
 35 ABS
 36 E-6
 37 X<=Y?
 38 GTO 10
 39 RCL 03
 40 BEEP
 41 RTN

42♦LBL 14
 43 4
 44 XEQ "CR"
 45 XEQ IND Y
 46 4
 47 CHS
 48 XEQ "CR"
 49 END
LBL'SOLVE
END
      97 BYTES
```

```
01♦LBL "CU"
 02 INT
 03 RCL c
 04 STO [
 05 "├♦♦♦!"
 06 RDN
 07 11
 08 X<> [
 09 X<> d
 10 STO ]
 11 RDN

12♦LBL 03
 13 FS?C IND [
 14 ISG X
 15 ""
 16 2
 17 /
 18 ENTER↑
 19 FRC
 20 X≠0?
 21 SF IND [
 22 DSE [
 23 ABS
 24 -
 25 X≠0?
 26 GTO 03
 27 X<> ]
 28 X<> d
 29 STO [
 30 "├♦♦♦"
 31 X<> \
 32 X<> c
 33 RDN
 34 CLA
 35 END
LBL'CU
END
      67 BYTES
```

Barcode for "SOLVE" and "CU" can be found in Appendix E.

The SOLVE routine starts by asking for the name of the user-supplied f(x) program and for two initial guesses at the root, that is, the value of x such that f(x) = Ø. SOLVE then proceeds to apply Newton's method to find the actual root of f(x) = Ø. To do this it will need to evaluate f(x) at several points. Each evaluation of f(x) is accomplished through the LBL 14 subroutine, which raises the curtain 4 registers, calls f(x), then lowers the curtain 4 registers to its original location.

The "CU" routine raises the curtain by the number of registers specified in X. If this number is negative the curtain is lowered. Two stack registers are preserved, so that the original contents of Y and Z (before executing "CU") end up in X and Y. This feature is used in the "SOLVE" program to preserve the function name and the trial value of x in the stack. Then an XEQ IND Y instruction is sufficient to call the f(x) function with the correct input.

To try out the SOLVE/CU combination, try this example. GTO.. and key in:

```
01 LBL"TEST"
02 1/X
03 LASTX
04 -
05 1
06 +
```

This short program calculates f(x)=(1/x)-x+1. Comparing problem 2.4, you can confirm that the solution to f(x)=0 is x=1+1/x, which is the Golden Ratio.

XEQ"SOLVE" now and supply the requested information:

Prompt	Response
FNAME?	TEST (R/S)
XGUESS1?	1 (R/S)
XGUESS2?	2 (R/S)

After about 40 seconds you'll hear a BEEP and see the result 1.618033989. This example does not really make use of the full capabilities of the SOLVE/CU combination, but you can be assured that SOLVE and CU will work just as well with any user-supplied f(x) program, regardless of any apparent register usage conflicts. Of course the usual limitations of root finding by Newton's method still apply. Certain ill-behaved functions can cause problems, as can bad initial guesses. But in most real-world cases, it works quickly and well.

Constraints on the use of "CU"

1.) While the curtain is in a raised position, data registers temporarily become program steps at the top of the first program in program memory. Some of these temporary program steps may be labels. Therefore do not branch backwards to a local label in the first program block when the curtain is up.

2.) Don't PACK program memory while the curtain is raised. It is more than likely that the protected data registers will contain null bytes which will be removed by packing. You can partially protect yourself from data alteration by PACKing before raising the curtain. This way the processor thinks your top program is already packed. Also make sure that several free registers (below the .END.) are present before using "CU". Then if you insert a program instruction, make a key assignment, or set an alarm you won't inadvertently cause a PACK to occur.

3.) Always restore the curtain to its original position. This is a matter of good programming practice. If you accidentally leave the curtain up you'll have to go into the first program in memory, delete the extraneous instructions at the top (thereby clearing your protected data), and PACK to bring the program up to the new curtain.

4.) Don't put the curtain immediately above a void, or nonexistent, location. For example a curtain location of 16 (decimal) is OK since register 15 (status register e) exists. But if you put the curtain at 17 you'll get MEMORY LOST, since register 16 does not exist. MEMORY LOST can be avoided if you bring the curtain back to an allowable location before halting ("MK" and "LB" do this), but you'd better know exactly what you're doing.

With the "CU" program, not only can one program renumber the registers before calling another program, but this second program can do a second renumbering before calling a third program. The process can be continued indefinitely, creating a multi-level data "stack". The critical sequence of steps to be embedded in any program to allow it to guard data registers 0 through k-1 from a subroutine is:

```
k
XEQ "CU"
XEQ subroutine
-k
XEQ "CU"
```

Register renumbering through curtain control adds greatly to program flexibility. For example a program that uses data registers 10 through 19 can be run with a SIZE of only 10. You need only lower the curtain 10 registers before executing the program, transforming registers 00 through 09 into registers 10 through 19. Don't forget to put the curtain back where it was immediately after running the program -- an inadvertent RCL 00 could wipe out part of your programs.

Tapani Tarvainen's "CU" program is functionally equivalent to Bill Wickes's **CU** (curtain up) program that is in the PPC ROM, so they may be used interchangeably. If speed is important you should be aware that Tapani's "CU" is significantly faster than **CU** . Also available are the even-faster PPC ROM curtain control routines **HD** , **UD** , and **ΣC** . These three routines have additional restrictions on their use which you should understand before you use them. For background information on curtain moving in general and on the routines named here, see the PPC Calculator Journal: May 1980 page 23, June 1980 page 45, July 1980 page 2, and March 1981 page 2. The programs "MS" and "RS" discussed in the PPC CJ articles are earlier versions of **HD** and **UD** . The PPC ROM User's Manual contains helpful information in the writeups for **CU**, **HD**, **UD**, and **ΣC**. Appendix M of the ROM Manual contains even more background material on curtain moving.

How the "CU" routine works

First the contents of status register c are placed in the rightmost part of the ALPHA register. Then line 05 appends four bytes. At this point status register M, which consists of the the last seven characters of ALPHA, contains the last three bytes of c, followed by three null bytes and a hexadecimal 21 byte. The curtain pointer resides in the first byte and a half of M.

Next M is extracted and swapped with the flags. The curtain pointer now resides in flags 0 through 11. Actually flags 0 and 1 are guaranteed to be clear, since the curtain is always less than or equal to 512 = 0010,0000,0000 base 2. The original flags are saved in status register O for later restoration, while the number 11 is stored in M for later use as a loop index.

The mysterious hex 21 byte sets flags 50 and 55. Flag 50 prevents any message in the display from moving (see Example 6 under **IF** in the PPC ROM User's Manual). Flag 55 must be set to allow "CU" to be interrupted or single-stepped with a printer attached. If flag 55 were clear, flags 55 and 21 would both be set on interruption, possibly altering the portion of the flag register that corresponds to the .END. pointer.

The LBL 03 loop performs binary addition in the flag register using Tapani's unique, elegant algorithm. The binary number in flags 0 through 11 is converted to decimal and added to the decimal increment (the number of registers by which the curtain is to be raised). Then the resulting decimal sum is converted back to binary and placed in flags 0 through 11.

The feature that makes Tapani's program unique is that this binary to decimal to binary conversion is completed at each bit before the next bit is considered. Each time through the LBL Ø3 loop one bit of the current curtain pointer is replaced by the correct bit for the new curtain pointer. Consider the way this process works for the least significant bit, the first time through the LBL Ø3 loop.

When LBL Ø3 is encountered for the first time, X contains the curtain increment you asked for. Lines 13 and 14 clear flag 11, the "ones" bit of the curtain pointer, and add 1 to X if flag 11 was set. This effectively converts the flag 11 bit to decimal, adding it to X. The flag 11 bit of the new curtain pointer will be set if and only if the number in X is now odd. If you don't see why this is so, consider that the new curtain pointer is the sum of the number in X plus the binary number residing in flags Ø through 11. Since flag 11 is clear, the binary number is divisible by 2. Thus the sum is odd, and flag 11 is to be set, if and only if X is odd.

Lines 15 through 24 perform several operations that are equivalent in effect to setting flag 11 and subtracting 1 from X if X is odd, otherwise leaving flag 11 clear, then dividing X by two. This division has an integer as the result because the previous step ensured that X would be even. The flag index is decremented from 11 to 1Ø for the next pass through the loop. Flag 11 attains the proper state for the new curtain pointer: set if and only if X was odd. Lines 25 and 26 cause the addition to proceed to the next most significant bit if the increment has not been reduced to zero yet.

The second time through the loop the binary number is only 11 bits long (flags Ø through 1Ø). We had to divide X by 2 so that it would be a decimal increment consistent with the new "ones" bit at flag 1Ø. The number in X does not merely represent the originally requested curtain increment. It now contains a component corresponding to a "carry", if there was any, from the previous bit.

This time through the loop flag 10 is cleared and transferred to X, then flag 10 is set if and only if X is odd. Once again, X is made even and divided by 2 for the next pass. This procedure continues until X is reduced to zero, as it must eventually be because of the repeated division by 2.

Notice that nowhere in the routine do we require knowledge of whether X is positive or negative. "CU" works the same in either case. When a flag is cleared X is incremented. When a flag is set X is decremented. Each time through the loop X is divided by 2, until eventually X becomes zero.

Lines 27 through 29 extract the contents of the flag register and place them in status register M, restoring the original flags and placing the modified last three bytes of c adjacent to the first four bytes of c which still occupy the rightmost 4 bytes of N. The ALPHA register is shifted left three bytes by an append instruction. All seven bytes of the new c register are now in status register N. They are extracted and stored in c. The X<> c instruction is used in case you want to restore the old curtain later with a simple STO c. Of course to do that you'll have to find the old c register contents in the stack, if it's still there.

The last few lines clear the ALPHA register for neatness and straighten out the stack. The former Y and Z end up in X and Y; Z contains the previous c register contents, and T contains zero.

Follow through this analysis a few times until you understand it. It may help to load the stack with 4 ENTER↑ 3 ENTER↑ 2 ENTER↑ 1 and GTO "CU". Make sure the SIZE is at least 001. Then you can SST through the routine and see what's going on for this simple case of raising the curtain 1 register.

Don't be concerned if much or even most of this Chapter is difficult to fathom at first reading. After all, that's why I saved it for last. Consider that the byte grabber and the "bootstrap" method of assigning it to a key were both discovered two years after synthetic programming began. There is undoubtedly much more yet to be discovered about your HP-41. Perhaps you will be the one to do it.

SOLUTIONS TO PROBLEMS

CHAPTER 2

2.1 Here's one version of "CQ":

```
Ø1 LBL"CQ"              LB / MK inputs:
Ø2 RAD
Ø3 CLX
Ø4 TONE 8
Ø5 TONE P               159, 12Ø
Ø6 TONE 8
Ø7 TONE P               159, 12Ø
Ø8 SIN
Ø9 TONE 8
1Ø TONE 8
11 TONE P               159, 12Ø
12 TONE 8
13 END
```

2.2 Key in

```
Ø1 ENTER↑
Ø2 1E1
```

GTO .ØØ1, key in RDN, BG, and backarrow twice. You now have E1 on line Ø2. Next key in STO 28, PACK, BST, BG, and backarrow. The PACKing placed the 28 suffix byte adjacent to the E1 instruction, purging the intervening nulls. When the STO prefix is grabbed, the 28 suffix becomes a NEG digit entry byte and is incorporated in the adjacent E1 instruction.

LB inputs for -E1 are 28, 27, 17.

2.3

```
Ø1 LBL"VX"              LB / MK inputs:
Ø2 "  "    (2 spaces)
Ø3 RCL d                144, 126
Ø4 SCI 9
Ø5 ARCL Y  (not X since the stack was raised by RCL d)
```

```
Ø6 STO d                    145, 126
Ø7 RDN
Ø8 AVIEW
Ø9 END
```

In cases like this you should get in the habit of doing the AVIEW after the STO d rather than before. This prevents altering system flags. In this particular case the display will revert to normal (the AVIEWed number will disappear) at completion of the program if the AVIEW is done first, since STO d clears flag 5Ø, the message flag.

2.4 Here's one solution to the Golden Ratio problem.

```
Ø1 LBL"GR"                  LB / MK inputs:
Ø2 FIX 9
Ø3 E                        27 or 27, Ø
Ø4 RCL b                    144, 124
Ø5 X<>Y
Ø6 1/X
Ø7 E                        27 or 27, Ø
Ø8 +
Ø9 X<>Y
1Ø VIEW Y
11 STO b                    145, 124
```

It converges to a 1Ø-digit solution in 8 seconds.

2.5 a)

```
Ø1 LBL"PX"                  LB  inputs:
Ø2 FIX Ø
Ø3 CF 29
Ø4 "X("                     242, 88, 4Ø
Ø5 ARCL ØØ
Ø6 "⊦)=?"                   244, 127, 41, 61, 63
Ø7 PROMPT
```

To generate the synthetic lines using the byte grabber, key in

```
Ø1 ENTER↑
Ø2 "XX"
Ø3 "⊦X=?"   .
```

GTO .ØØ2, BG, GTO .ØØ5, backarrow, RCL Ø9, GTO .ØØ2, BG, DEL ØØ2, GTO .ØØ1, BG, GTO .ØØ4, backarrow, RCL Ø8, GTO .ØØ1, BG, DEL ØØ2, backarrow, and key in the nonsynthetic lines.

b) To preserve the display mode, insert RCL d and STO d as shown:

```
Ø1 LBL"PX"                LB / MK  inputs:
Ø2 RCL d                  144, 124
Ø3 CF 29
Ø4 FIX Ø
Ø5 "X("
Ø6 ARCL ØØ
Ø7 "⊦)=?"
Ø8 STO d                  145, 124
Ø9 RDN
1Ø PROMPT
```

It is possible to save one byte by replacing lines Ø2 - Ø3 of this program by

```
Ø2 .     (decimal point)
Ø3 X<> d                  2Ø6, 126
```

This stores zero in the flag register, clearing all 56 flags. The we need only to FIX Ø to get the desired status of flags 29 and 36-41. The old flag register contents are in X just as before, ready for the subsequent STO d that restores the previous flag settings. To make the X<>d instruction using the byte grabber, start with STO IND 78 followed by AVIEW. Grab the STO byte and backarrow. The IND 78 becomes X<> and the AVIEW becomes the d suffix.

```
2.6     Ø1 LBL"OX"                 LB inputs:
        Ø2 RCL d                   144, 126
        Ø3 FIX 2
        Ø4 "OUT="
        Ø5 ARCL Y
        Ø6 STO d                   145, 126
        Ø7 RDN
        Ø8 "⊦μV"                   243, 127, 12, 86
```

Line Ø8 can be constructed using the byte grabber as follows. Key in

```
        Ø1 ENTER↑
        Ø2 "⊦XV"
```

GTO .ØØ1, BG, GTO .ØØ4, backarrow, LBL 11, GTO .ØØ1, BG, DEL ØØ2, backarrow.

```
2.7     LBL"CMOD"                  LB / MK inputs:
        Ø2 X<>Y
        Ø3 STO M                   145, 117
        Ø4 X<>Y
        Ø5 MOD
        Ø6 ST- M                   147, 117
        Ø7 LASTX
        Ø8 ST/ M                   149, 117
        Ø9 CLX
        1Ø X<>M                    2Ø6, 117
```

Lines Ø1-Ø4 save y in M and x in L. Then y mod x is subtacted from M. Lines Ø7-1Ø divide M by X, bring M back to X, and clear M.

CHAPTER 3

3.1 GTO.. and key in LBL"++", at least 45 +'s, and XEQ"LB". Switch out of PRGM mode, R/S, and respond to the prompts as follows:

prompt	response
1?	27 R/S
2?	145 R/S
3?	119 R/S
4?	146 R/S
5?	119 R/S
6?	206 R/S
7?	119 R/S
1?	145 R/S
2?	117 R/S
3?	150 R/S
4?	117 R/S
5?	240 R/S
6?	153 R/S
7?	245 R/S
1?	152 R/S
2?	119 R/S
3?	172 R/S
4?	245 R/S
5?	159 R/S
6?	106 R/S
7?	244 R/S
1?	1 R/S
2?	4 R/S
3?	5 R/S
4?	6 R/S
5?	242 R/S
6?	127 R/S
7?	96 R/S
1?	154 R/S
2?	118 R/S
3?	152 R/S
4?	118 R/S
5?	R/S

When the program stops you can press SST to get back to

LBL"++" and see your new synthetic instructions.

3.2 Here's a simple nonsynthetic program to compute the LB inputs from XROM numbers. This program takes advantage of the fact that 64*(i mod 4) is the same as 256*FRC(i/4). At the right we note how the stack register contents change through the program. Where there is no entry, the contents of that register are unchanged from the previous step.

LBL"XRLB"	L	X	Y	Z	T
X<>Y		i	j	z	t
4		4	i	j	z
/	4	i/4	j	z	z
INT	i/4	INT(i/4)			
X<>Y		j	INT(i/4)		
LASTX		i/4	j	INT(i/4)	z
FRC	i/4	FRC(i/4)			
256		256	FRC(i/4)	j	INT(i/4)
*	256	64(i mod 4)	j	INT(i/4)	INT(i/4)
+	64(i mod 4)	byte 2	INT(i/4)		
X<>Y		INT(i/4)	byte 2		
160		160	j	byte 2	
+	160	byte 1	byte 2	INT(i/4)	INT(i/4)
END					

To use XRLB, key in i ENTER↑ j and XEQ"XRLB". The output in X is byte 1 in decimal. Byte 2 is in the Y register.

Here's a synthetic version of "XRLB" that does not disturb stack registers Z and T. At the right are noted the important stack and status register contents as they change through the program.

LBL"XRLB"	N	M	L	X	Y	Z	T
STO M		j		j	i	z	t
RDN				i	z	t	j
4				4	i	z	t
/			4	i/4	z	t	t
STO N	i/4						
FRC			i/4	FRC(i/4)			
256				256	FRC(i/4)	z	t
*			256	64(i mod 4)	z	t	t
RCL M				j	64(i mod 4)	z	t
+			j	byte 2	z	t	t
160				160	byte 2	z	t
ST+ N	160+i/4						
X<> N	160			160+i/4			
INT			160+i/4	byte 1	byte 2	z	t
CLA	0	0					
END				byte 1	byte 2	z	t

3.3 Use at least 17 +'s and execute LB. The 7 inputs are 207, 120, 159, 37, 208, 0, 120.

3.4 Use at least 31 +'s and load decimal values 192, 0, 255, 0, 82, 80, 78, 32, 67, 65, 76, 67, 85, 76, 65, 84, 79, 82. PACK to incorporate this new global label into Catalog 1. Since this label is longer than 6 characters it cannot be the object of a GTO IND or XEQ IND instruction.

3.5 The proper LB inputs are 144, 124, 206, 117, 206, 118, 145, 117, 206, 117, 206, 125, 145, 125, 242, 127, 0, 206, 125, 144, 117, 145, 125.

CHAPTER 4

4.2 The decimal byte equivalents required are 244, 127, 0, 0, 2, 27, 20, 206, 125, 145, 125, 242, 127, 0, 206, 125, 145,

125. GTO.. and key in LBL "LB". Then in RUN mode do CLA, 125, XTOA, 145, XTOA, 125, XTOA, 2Ø6, XTOA, Ø, XTOA, 127, XTOA, 242, XTOA. GTO "LB", RCL M, STO Q, enter PRGM mode, Q-LOAD, BG, and backarrow twice.
Switch back to RUN mode and do CLA, 125, XTOA, 145, XTOA, 125, XTOA, 2Ø6, XTOA, 2Ø, XTOA, 27, XTOA. GTO "LB", RCL M, STO Q, and enter PRGM mode. No PACKing is required here since the 242 byte is not part of a preceding instruction. Thus no direct attachment to the new bytes is required.
Still in PRGM mode at LBL "LB", Q-LOAD, BG, and backarrow twice.
Continue with CLA, 2, XTOA, Ø, XTOA, Ø, XTOA, 127, XTOA, 244, XTOA. GTO "LB", RCL M, STO Q, enter PRGM mode, Q-LOAD, BG, and backarrow twice. The fact that we did not include the decimal 2 byte in the second group of bytes saved us from the need to PACK before loading the third group. Moreover, this procedure was essential anyhow since the one weakness of Q-loading is its inability to load trailing null bytes. We could not have loaded the sequence hex F4 7F ØØ ØØ successfully by itself.

4.3 a) XROM 61,25
    b) XROM 57,26
    c) XROM 27,54

CHAPTER 5

5.1 The byte sequences in hexadecimal are as follows:

a) 4Ø, 47, 48, ØØ, ØØ, ØØ, ØØ, ØØ, ØØ, 13, 41, ØØ, 14, 25, 15, 42. There was room for the Σ+ (hex 47), but the Σ- opened seven bytes. The RCL Ø5 fit in the null that was already present between the 4 and 5 digit entry instructions.

b) 4Ø, 41, EØ, ØØ, ØØ, 92, 4B, ØØ, 42, 43. The ST+ 75 takes two of the 3 bytes formerly used by GTO 99.

APPENDIX A

INSTRUCTION TIMING

In reading Chapter 2, you might have wondered how anyone could determine that the synthetic digit entry instruction E executes faster than 1, or that the decimal point executes faster than the digit zero. In HP-67 days, these results were obtained by keying in a sequence of 100 or more identical instructions, measuring the time needed to execute the entire sequence, then dividing by the number of instructions in the sequence. Needless to say, this procedure was both laborious and time-consuming.

Synthetic programming permits automation of the procedure of entering hundreds of copies of a particular instruction (or even copies of a short sequence of instructions). The proper byte sequences are created and stored, in 7-byte groups, in contiguous registers. The bytes can then be executed as program instructions by placing the proper code in the program pointer register.

As a measure of the capability of the HP-41 system, the HP 82182A time module allows even the timing of the sequence of synthetically stored instructions to be automated. Clifford Stern has written a synthetic program which uses the time module to time an arbitrary group of one to seven bytes. The program creates and stores as many replicas of the byte group as it can within the unused portion of program memory. It then executes the full sequence of byte groups, measures the elapsed time, divides by the number of identical groups, and displays the resulting time per group.

Table A.1 gives typical results for instruction execution time. Emphasis has been placed on instructions for which alternatives are available. If you need a LOG function, it doesn't realy matter how long it takes since you don't have any faster way to calculate the logarithm. But to increment a register, you may be interested to know that the sequence E,

+, at 78.7 msec, is slightly slower than the sequence ISG X, TEXT 0, at about 74 msec. If you need the speed you may be willing to use the extra byte of program memory to get it. Other conclusions from the timing chart are:

- R↑ R↑ is faster than RDN RDN ;
- X<> is faster than RCL but slower than STO ;
- Status register operations are always faster than the corresponding numbered register operations ;
- compiled GTO's are very fast, with XEQ being a bit slower ;
- digit entry is very slow. This is due to the fact that status registers P and Q must be loaded before the X register ;
- For faster numeric entry use E instead of 1, and the decimal point instead of zero. Note that CLX, SIGN is a much faster way to get 1.
- For faster entry of negative numbers, use a positive number entry followed by a separate CHS instruction, rather than a single instruction containing the negative number. Press ALPHA ALPHA to terminate the positive number entry, then press CHS to get the separate CHS instruction. CHS is much faster than NEG (negation within a number entry instruction).

These results from the timing program are another example of how knowledge of synthetic programming can improve your general programming technique.

If you have a PPC ROM, an extended functions module, and a time module, you can use Clifford Stern's program to do some instruction timing of your own. Here are the instructions:

1) Make sure that there is an END above this program in the Catalog 1 list. This is necessary to allow the GTO instructions to work properly with the program/data "curtain" positioned at hex 010. For further explanation, see "CU" constraint 1 in Section 6C.

Table A.1 Typical execution times (in milliseconds)

Stack operations

ENTER↑	11.7
X<>Y	1Ø.3
RDN	16.9
R↑	12.Ø
CLX	9.8
LASTX	13.Ø
CLST	1Ø.5
SIGN	13.3
CHS	12.5
CLA	9.5
RCL status	2Ø.3
STO status	16.8
X<> status	19.7

Misc instructions

LBL ØØ-14	1Ø.6
two-byte LBL	13.1
CLD	2Ø.6
TEXT Ø	12.3
AON, AOFF	19.Ø
ADV (no printer)	9.2
BEEP (flag 26 set)	1Ø42.4
(flag 26 clear)	14.9
DEG	19.8
RAD	19.9
GRAD	2Ø.5
PSE	1333.2
NULL	5.7

Storage register operations

STO ØØ-15	19.3
STO 16-99	2Ø.6
STO status	16.8
STO IND ØØ-99	32.3
STO IND status	32.1
RCL ØØ-15	22.8
RCL 16-99	24.1
RCL status	2Ø.3
RCL IND ØØ-99	35.7
RCL IND status	35.6
X<> ØØ-99	23.4
X<> status	19.7
X<> IND ØØ-99	35.1
X<> IND status	35.Ø

ST+ ØØ-99	38.9	
ST+ status	35.3	
ST- ØØ-99	4Ø.8	
ST- status	37.3	
ST* ØØ-99	46.8	
ST* status	43.Ø	
ST/ ØØ-99	49,5	
ST/ status	45.8	
ISG X , TEXT Ø (skip)	73.2	(x = 1)
(non-skip)	74.4	(x = -1)
DSE X , TEXT Ø (skip)	72.9	(x = 1)
(non-skip)	74.Ø	(x = 2)

Digit Entry

Ø	69.7
1 through 9	59.8
.	61.8
E	53.6
- (NEG, negates the mantissa or exponent. By itself, it places a zero in X.)	6Ø.9

Miscellaneous multi-byte instructions

GTO ØØ-14 , compiled	17.3
GTO(three byte),compiled	24.5
XEQ, compiled	35.2
global LBL, 1 character	45.4
2 character	49.3
3 character	51.9

2) Clear flag Ø2 and set SIZE at least ØØ4. Clear all timer alarms (you can use the "SA" program from Section 4E). Make any key assignments you want now. <u>Do not</u> make any key assignments (except global labels) after you've started step 3 and before you've finished step 9.

3) Enter the number of registers to be used for storing the byte sequence. The number of registers should be selected to provide an exact multiple of the number of bytes per group of instructions, except that 1- and 7-byte groups are always OK. For example if the group is 3 bytes long, the number of registers should be a multiple of 3. If it

is not a multiple of the number of bytes per group, you'll eventually get DATA ERROR at line 114. If you pick a multiple of 60 registers, you can't go wrong. XEQ "IN" to initialize to this number of registers. The timing program will adjust the SIZE if needed to provide the requested number of free registers below the .END. . If the existing combination of SIZE and free registers is not sufficient to allow the requested number of free registers to be provided for timing, a DATA ERROR message will appear at line 49. If this happens, clear a program or reduce the number of free registers requested and repeat from the beginning of step 3.

4) The "IN" procedure automatically falls into LBL "S", the instruction storage routine. The "S" routine will prompt you for a group of one to seven bytes. Key in a decimal number between 0 and 255 for each byte, and press R/S without an input to indicate the end of a byte group. The group of bytes will then be duplicated and stored throughout the initialized block of registers below the .END. and above the key assignments.

5) With flag 01 clear the "S" routine halts at LBL "T", the timing routine. At this point the stack is clear. You are free to load the stack as needed for your instruction sequence. Press R/S or XEQ "T" to start the timing. The result, expressed in milliseconds per group of bytes, is returned in the X register when the timing routine halts. If you happen to have an error condition that causes a halt in the stored instruction sequence, you must press GTO "S" and XEQ 10. You can then store a new sequence of instructions as in step 4, or simply enter a valid argument and XEQ "T".

6) To repeat the timing for another initial condition, reload the stack and XEQ "T" again (do <u>not</u> simply press R/S -- see step 9). If you want to set up the alpha register as well as stack contents, just set flags 1 and 2 before executing "T". The timing routine will stop for

you to load the alpha register (as well as the stack, if you like). Note that "T" can be called as a subroutine for automated timing of the same function with a variety of stack inputs.

7) To switch to timing a different group of instructions, XEQ "S" again. You have the option of setting flag 1 first if you wish the timing to proceed automatically with a clear stack. Set flags 1 and 2 if you need to load the alpha register for timing.
8) To select a different number of registers for instruction storage, enter the number and XEQ "IN" again.
9) To clear out the free register block at the end of the timing session, press RTN and R/S, or just R/S after using the "T" routine.
10) Three additional convenience routines are provided in this program. They are each non-prompting versions of the instruction storage routine "S".

XEQ "1" with a decimal input (0 to 255) to store a sequence of one-byte instructions.

XEQ "2" with a decimal input to store the repeating sequence: one-byte instruction, LASTX. This sequence is helpful when timing unary operations like SIN or LN.

XEQ "3" with a decimal input to store the repeating sequence: one-byte instruction, X<> L. This is useful for timing binary operations like + or MOD. Just initialize by filling the stack with "Y" arguments, then putting the "X" argument in X and executing "T".

When you use "2" or "3" you'll have to separately time LASTX or X<> L and subtract to get the net execution time for the particular function you're timing.

When you time numeric entry instructions, you must separate them so they don't run together into a single huge instruction. Use a null or LASTX, and subtract the time for the separator.

Barcode for the complete instruction timer program is included in Appendix E.

```
01 XROM "RF"
02 AVIEW
03 XROM "LF"
04 XROM "OM"
05 X<>Y
06 ISG X
07 XROM "BC"
08 GTO 13

09+LBL "3"
10 "t"
11 3
12 GTO 01

13+LBL "2"
14 "v"
15 2
16 GTO 01

17+LBL "1"
18 CLA
19  E

20+LBL 01
21 STO a
22 ASTO X
23 CLA
24 AVIEW
25 CF 29
26 FIX 0
27 X<>Y
28 XTOA
29 ARCL Y
30 GTO 16

31+LBL "IN"
32 STO 03
33 XROM "F?"
34 INT
35 ENTER↑
36 XROM "E?"
37 X<>Y
38 -
39 STO 01
40 SIZE?
41 ENTER↑
42 R↑
43 +
44 RCL 03
45 -
46 7
47 -
48 X<0?
49 SQRT
50 4
51 +
52 X<Y?
53 PSIZE
54 XROM "OM"
55 R↑
56  E
57 +
58 XROM "CX"
59 X<> c
60 RCL 03
61  E
62 +
63 X<>Y
64 X<> c
65 "              ne"
66 RCL [
67 STO 00
68 "          x̄
            "
69 ASTO IND Z
70 RDN
71 X<> c
72 X<> 01
73 +
74 2561
75 +
76 7
77 *
78 XROM "DP"
79 ASTO 02
80 BEEP

81+LBL 10
82 STOPSW
83 CLX
84 SETSW

85+LBL "S"
86 CF 29
87 FIX 0
88 CLA
89 CLX

90+LBL 11
91 XTOA
92 ISG a
93 -
94 X<> [
95 "DEC. "
96 ARCL a
97 "├?"
98 AVIEW
99 STO [
100 STOP
101 FS?C 22
102 GTO 11
103 CLA
104 AVIEW
105 STO [
106 DSE a

107+LBL 16
108 RCL 03
109 7
110 *
111 RCL a
112 /
113 STO 00
114 OCT
115 GTO IND a

116+LBL 07
117 X<> [
118 X<> ]
119 STO a
120 GTO 12

121+LBL 04
122 FIX 1

123+LBL 05
124 SF 29

125+LBL 06
126+LBL 03
127+LBL 02
128+LBL 01
129 ASTO X
130 17
131 RCL a
132 /
133 INT
134 RCL b
135 ARCL Z
136 DSE Y
137 STO b
138 "├*"
139 FC? 29
140 "├**"
141 RCL a
142  E5
143 /

144+LBL 12
145 RCL 03
146 +
147 ABS
148 RCL 01
149 X<> c
150 RCL ]
151 GTO 09

152+LBL 00
153  E
154 ST- L
155 ARCL X
156 LASTX
157 R↑
158 RCL ]

159+LBL 09
160 STO IND Z
161 DSE Z
162 GTO 09
163 DSE a
164 GTO 00

165+LBL 13
166 CLD
167 X<>Y
168 STO c
169 CLST
170 FC? 02
171 FC? 01
172 TONE 8
173 FC? 01
174 RTN

175+LBL "T"
176 ARCL 02
177 XROM "XE"
178 SETSW
179 X<>Y
180 36 E5
181 *
182 RCL 00
183 /
184 FIX 9
185 TONE 8
186 END
LBL'3
LBL'2
LBL'1
LBL'IN
LBL'S
LBL'T
END          329 BYTES
```

The complete instruction timer program listing is shown on the previous page. A few of the synthetic lines have ambiguous representations in the printout. These are listed here together with their decimal equivalents for LB:

Line	hex	decimal
10	F2 CE 74	242 206 116
14	F1 76	241 118
65	F7 A6 99 A6 93 6D 1C 85	247 166 153 166 147 109 28 133
68	F5 AC 02 84 A6 94	245 172 2 132 166 148

Lines 65 and 68 contain printer control characters. The hex A6 character causes 6 spaces to be skipped; hex AC causes 12 spaces to be skipped.

Summary of Error Traps:

Line 49 DATA ERROR means available memory is insufficient to produce the requested number of storage registers.

Line 114 DATA ERROR means that the number of bytes per group does not evenly divide the number of registers allocated ("IN") for storage of the full instruction sequence.

Line 115 NONEXISTENT means that you tried to time an 8-byte group. This program will handle 1- to 7-byte groups.

Timer program data register usage:

R_{00} = scratch (number of instruction groups)
R_{01} = curtain lowering code (temporarily placed in c)
R_{02} = return pointer for the stored byte sequence
R_{03} = number of storage registers
If any of R_{01} through R_{03} are altered, you must re-initialize (enter the number of registers and XEQ"IN").

APPENDIX B

MORSE CODE AND STO b

The idea of using the HP-41 to produce machine-perfect Morse code was introduced by Richard Nelson (the founder of PPC and editor of the PPC Calculator Journal) on page 50 of the February 1980 PPC CJ. His program employed the synthetic TONE P, but at that time synthetic programming was in its infancy, so the execution logic was confined to standard techniques. As a result, transmission speed was only about 6 words per minute. However a General class amateur radio license requires you to be able to receive 13 words per minute. Conventional methods are clearly inadequate to produce code at this speed.

Clifford Stern has written a Morse code program that brings the full power of synthetic programming to bear on the problem. To understand the technique used, first consider the following execution loop which appeared in an earlier version of this program:

```
LBL 01
RCL IND L
XEQ IND X
ISG L
GTO 01
```

The individual characters of the message have been stored in a series of data registers, and the LASTX register contains a counter for those registers. The RCL IND L instruction puts a single character in the X register, then XEQ IND X calls a short tone routine corresponding to the character in X. For example if X contains the letter "C", then the following sequence is executed:

```
LBL "C"
TONE 8
TONE P
TONE 8
TONE P
TONE 8
RTN
```

The simplicity of this procedure is due to the use of synthetic single-character global labels. These are used for three of the punctuation marks and the letters A through J. The non-synthetic labels for those letters are local, not global, and cannot be the object of indirect addressing.

However, speed is still a problem with this approach. Because XEQ IND X has to search Catalog 1 to find the proper tone sequence, it requires a relatively long time to execute. In fact, 16 milliseconds per label is spent climbing up the global label chain from the .END. in the search for a specified global label. This causes a noticeable delay for labels placed high in the catalog.

The major breakthrough for this Morse code program is replacing XEQ IND X with a STO b instruction so as to jump directly to each tone sequence. Not only does this provide a dramatic breakthrough in speed, but it is a striking example of how synthetic programming makes possible that which cannot be done by normal means, no matter how elaborate. In effect, synthetic techniques are used to compile indirect branching addresses.

Some details have to be considered when applying this procedure. First, there must be a method to determine the correct address to branch to. This is accomplished here by inserting a RCL b instruction before each set of tones; for example:

```
LBL "C"
RCL b
TONE 8     (STO b will cause execution to pick up here)
TONE P
TONE 8
TONE P
RTN
```

The sequences are called with flag 26 clear during the setup process. The RCL b results are incorporated into codes which are stored in a series of data registers. The other detail to be taken care of is the inclusion of return addresses in the code so that the RTN at the end of each tone sequence brings execution back to the ISG L instruction.

For the ultimate in speed, the GTO Ø1 instruction is replaced by a RTN. A second return address is included with the one just discussed to make this work. This second return address is set up to transfer execution directly to the RCL IND L instruction, eliminating the need for LBL Ø1. Furthermore, RTN is 15% faster than a compiled two-byte GTO.

The primary pointer and two return pointers account for six bytes of each STO b code. The leading byte is taken from row 1 of the QRC to avoid normalization problems when recalling the stored codes from data registers. (The fact that the first byte is from row 1 guarantees that the code will be treated as legitimate alpha data.) Because the leftmost byte is nonzero, a STOP instruction, rather than a RTN, is required to halt execution.

In the system used here, both of the return pointers are constructed by normal subroutine calls. This technique is much simpler than synthesizing the pointers because it does not require calculation of the program's location in memory or merging return addresses onto a program pointer. The first return pointer is constructed by the XEQ IND T instruction at line 58, while the second pointer is constructed by XEQ Ø5 at line 45. Thus the RCL b instruction preceding each set of tones provides the complete code for storage, since the two returns are pending at that time.

The result is a Morse code program that produces code at 16 words per minute -- a substantial improvement over conventional methods. Also, the true capacity of the ALPHA register is highlighted, as 28 characters may be entered at a time during the setup phase. This capability is made possible by the fact that the calculator remains in ALPHA mode during data entry (see the information on status register P in Section 6A). Ambitious synthetic programmers should also consult the P register summary on page 13 of the July 1981 PPC CJ for full details of how the digit entries on lines 42 and 52 are used to modify the P register.

Here are the instructions for using Clifford's Morse code program "MC":

1) Execute a SIZE of at least one greater than the number of characters in the message.
2) XEQ "MC". Enter the message in groups of 1 to 28 characters. The tone prompt that signals the end of the standard ALPHA register indicates here that 4 more characters can still be entered. Press R/S to process each group. If you get NONEXISTENT, increase the SIZE and start over.
3) Push R/S without making an entry to transmit the message. Press R/S or XEQ 10 to repeat the message.
4) To get slower code output, insert any instructions which do not affect LASTX between lines 45 and 46 and XEQ "MC" again. This change increases the character spacing.

If you have an optical wand, use the barcode in Appendix E to load the Morse code program. If you do not have a wand, there are a few things you can do to speed up keying in the program.

The following synthetic key assignments will facilitate keying in "MC" from the listing: 159, 120 (TONE P); 159, 8 (TONE 8); and 205, 0 (the global label counterpart of the Q-loader). This last assignment was discovered by Tom Cadwallader, and can be used to produce the required synthetic labels. For example to create LBL "A", key in XEQ A or LBL A. This loads the character "A" into the Q register. Delete that instruction (if you were in PRGM mode when you keyed it in), and press the assigned key in PRGM mode to create LBL "A". This procedure was discovered by Valentin Albillo, another synthetic programming pioneer, and can be used to key in the program's global labels for A-J.

A different process must be used to produce labels for the colon, period, and comma. One method is to enter the punctuation mark into the ALPHA register, ASTO X, and press GTO IND X (all in RUN mode). This loads the punctuation mark into Q. After NONEXISTENT appears, switch to PRGM mode and press the assigned key to obtain the corresponding global label.

As an alternative, the byte grabber can be used to synthesize any of these labels:

01 ENTER↑	LB inputs:
02 STO IND 66	192,
03 SIN	0, (any value is OK)
04 "Z:"	242, 0, character byte.

Pressing the byte grabber at line 01 removes the STO byte and creates LBL ":" . PACKing is essential to incorporate these synthetic labels into the global chain, regardless of the means by which they are created.

```
01♦LBL "MC"
02 SF 26
03 ",x"
04 X<> [
05 X<> d
06 RCL b
07 FC?C 26
08 GTO 01
09 CLA
10 ASTO Z
11 X<> [
12 SIGN
13 ASTO X
14 "≠"
15 ARCL X
16 ASTO b

17♦LBL 01
18 SF 26
19 "CHARACTERS?"
20 PROMPT
21 FC?C 23
22 GTO 06
23 VIEW Z
24 CF 26
25 CLX
26 ENTER↑
27 X<> ↑
28 X=Y?
29 GTO 02
30 SF 05
31 X<> ]
32 X<> \
33 X<> [
34 X<>Y

35♦LBL 02
36 "├♦"
37 X<> ↑
38 X=0?
39 GTO 02
40 STO ↑
41 RDN
42 0
43 FC?C 29
44 GTO 05
45 XEQ 05
46 RCL IND L
47 STO b

48♦LBL 03
49 STO IND L
50 RDN

51♦LBL 04
52 .
53 "├♦"

54♦LBL 05
55 X<> ↑
56 RDN
57 SF 25
58 XEQ IND T
59 ISG L
60 RTN
61 FS?C 25
62 GTO 03
63 FS? 26
64 GTO 07
65 DSE L
66 FC?C 05
67 GTO 01
68 STO ]
69 GTO 04

70♦LBL 06
71 LASTX
72  E3
73 +
74 LASTX
75 /
76 STO 00
77 SIGN
78 R↑
79 STO d
80 RCL 01
81 STO b

82♦LBL 07
83 RCL 00
84 SIGN
85 STOP

86♦LBL 10
87 RCL 01
88 STO b

89♦LBL ":"
90 RCL b
91 TONE 8
92 TONE 8
93 TONE 8
94 TONE ↑
95 TONE ↑
96 TONE ↑
97 RTN

98♦LBL "-"
99 RCL b
100 TONE 8
101 TONE ↑
102 TONE ↑
103 TONE ↑
104 TONE 8
105 RTN

106♦LBL "/"
107 RCL b
108 TONE 8
109 TONE ↑
110 TONE ↑
111 TONE 8
112 TONE ↑
113 RTN

114♦LBL "?"
115 RCL b
116 TONE ↑
117 TONE ↑
118 TONE 8
119 TONE 8
120 TONE ↑
121 TONE ↑
122 RTN

123♦LBL "."
124 RCL b
125 TONE ↑
126 TONE 8
127 TONE ↑
128 TONE 8
129 TONE ↑
130 TONE 8
131 RTN

132♦LBL ","
133 RCL b
134 TONE 8
135 TONE 8
136 TONE ↑
137 TONE ↑
138 TONE 8
139 TONE 8
140 RTN

141♦LBL "0"
142 RCL b
143 TONE 8
144 TONE 8
145 TONE 8
146 TONE 8
147 TONE 8
148 RTN

149♦LBL "9"
150 RCL b
151 TONE 8
152 TONE 8
153 TONE 8
154 TONE 8
155 TONE ↑
156 RTN

157♦LBL "8"
158 RCL b
159 TONE 8
160 TONE 8
161 TONE 8
162 TONE ↑
163 TONE ↑
164 RTN

165♦LBL "7"
166 RCL b
167 TONE 8
168 TONE 8
169 TONE ↑
170 TONE ↑
171 TONE ↑
172 RTN

173♦LBL "6"
174 RCL b
175 TONE 8
176 TONE ↑
177 TONE ↑
178 TONE ↑
179 TONE ↑
180 RTN

181♦LBL "5"
182 RCL b
183 TONE ↑
184 TONE ↑
185 TONE ↑
186 TONE ↑
187 TONE ↑
188 RTN

189♦LBL "4"
190 RCL b
191 TONE ↑
192 TONE ↑
193 TONE ↑
194 TONE ↑
195 TONE 8
196 RTN

197♦LBL "3"
198 RCL b
199 TONE ↑
200 TONE ↑
201 TONE ↑
202 TONE 8
203 TONE 8
```

```
204 RTN

205♦LBL "2"
206 RCL b
207 TONE ↑
208 TONE ↑
209 TONE 8
210 TONE 8
211 TONE 8
212 RTN

213♦LBL "1"
214 RCL b
215 TONE ↑
216 TONE 8
217 TONE 8
218 TONE 8
219 TONE 8
220 RTN

221♦LBL "Z"
222 RCL b
223 TONE 8
224 TONE 8
225 TONE ↑
226 TONE ↑
227 RTN

228♦LBL "Q"
229 RCL b
230 TONE 8
231 TONE 8
232 TONE ↑
233 TONE 8
234 RTN

235♦LBL "J"
236 RCL b
237 TONE ↑
238 TONE 8
239 TONE 8
240 TONE 8
241 RTN

242♦LBL "X"
243 RCL b
244 TONE 8
245 TONE ↑
246 TONE ↑
247 TONE 8
248 RTN

249♦LBL "K"
250 RCL b
251 TONE 8
252 TONE ↑
253 TONE 8
254 RTN

255♦LBL "V"
256 RCL b
257 TONE ↑
258 TONE ↑
259 TONE ↑
260 TONE 8
261 RTN

262♦LBL "B"
263 RCL b
264 TONE 8
265 TONE ↑
266 TONE ↑
267 TONE ↑
268 RTN

269♦LBL "G"
270 RCL b
271 TONE 8
272 TONE 8
273 TONE ↑
274 RTN

275♦LBL "W"
276 RCL b
277 TONE ↑
278 TONE 8
279 TONE 8
280 RTN

281♦LBL "Y"
282 RCL b
283 TONE 8
284 TONE ↑
285 TONE 8
286 TONE 8
287 RTN

288♦LBL "P"
289 RCL b
290 TONE ↑
291 TONE 8
292 TONE 8
293 TONE ↑
294 RTN

295♦LBL " "
296 RCL b
297 FC? 26
298 RTN
299 LASTX
300 LN
301 RTN

302♦LBL "M"
303 RCL b
304 TONE 8
305 TONE 8
306 RTN

307♦LBL "U"
308 RCL b
309 TONE ↑
310 TONE ↑
311 TONE 8
312 RTN

313♦LBL "F"
314 RCL b
315 TONE ↑
316 TONE ↑
317 TONE 8
318 TONE ↑
319 RTN

320♦LBL "C"
321 RCL b
322 TONE 8
323 TONE ↑
324 TONE 8
325 TONE ↑
326 RTN

327♦LBL "L"
328 RCL b
329 TONE ↑
330 TONE 8
331 TONE ↑
332 TONE ↑
333 RTN

334♦LBL "D"
335 RCL b
336 TONE 8
337 TONE ↑
338 TONE ↑
339 RTN

340♦LBL "H"
341 RCL b
342 TONE ↑
343 TONE ↑
344 TONE ↑
345 TONE ↑
346 RTN

347♦LBL "S"
348 RCL b
349 TONE ↑
350 TONE ↑
351 TONE ↑
352 RTN

353♦LBL "I"
354 RCL b
355 TONE ↑
356 TONE ↑
357 RTN

358♦LBL "R"
359 RCL b
360 TONE ↑
361 TONE 8
362 TONE ↑
363 RTN

364♦LBL "N"
365 RCL b
366 TONE 8
367 TONE ↑
368 RTN

369♦LBL "O"
370 RCL b
371 TONE 8
372 TONE 8
373 TONE 8
374 RTN

375♦LBL "A"
376 RCL b
377 TONE ↑
378 TONE 8
379 RTN

380♦LBL "T"
381 RCL b
382 TONE 8
383 RTN

384♦LBL "E"
385 RCL b
386 TONE ↑
387 END

LBL'MC
LBL':
LBL'-
LBL'/
LBL'?
LBL'.
LBL',
LBL'0
LBL'9
LBL'8
LBL'7
LBL'6
LBL'5
LBL'4
LBL'3
LBL'2
LBL'1
LBL'Z
LBL'Q
LBL'J
LBL'X
LBL'K
LBL'V
LBL'B
LBL'G
LBL'W
LBL'Y
LBL'P
LBL'
LBL'M
LBL'U
LBL'F
LBL'C
LBL'L
LBL'D
LBL'H
LBL'S
LBL'I
LBL'R
LBL'N
LBL'O
LBL'A
LBL'T
LBL'E
END
845 BYTES
```

Three of the text instructions in the Morse code program appear in an ambiguous form in the printed listing. These are:

line	hex	decimal
Ø3	F4 2C Ø1 8Ø 81	244 44 1 128 129
36	F2 7F ØØ	242 127 Ø
53	F2 7F ØØ	242 127 Ø

APPENDIX C

SYNTHETIC PROGRAMMING REFERENCES

Here is a list of sources for information on HP-41 synthetic programming:

1. PPC Calculator Journal, published by Personal Programming Center, a non-profit, public benefit California corporation dedicated to personal computing. The issues from July 1979 (Volume 6, Number 4) to the present contain a wealth of information on the HP-41 in general, and on synthetic programming in particular. The PPC CJ is still the most up-to-date and comprehensive source for synthetic programs, techniques,and discoveries.

To obtain a PPC membership application and a price list for back list for back issues of PPC CJ, send a 9" by 12" self-addressed stamped envelope with 3 ounces of postage to:

PPC
2545 W. Camden Place
Santa Ana, CA, 92704

To speed the processing, mark the lower left corner of your outer envelope with "New member info plus HP-41 back issues." You don't need to enclose a letter; it will only slow things down.

2. PPC Technical Notes, published by the Melbourne, Australia chapter of PPC. PPC TN is a smaller-scale publication than PPC CJ, but it specializes in synthetic programming. Issue number 9 contains the best summary of HP-41 microcode currently available. The current subscription price is 20 Australian dollars per year to US and Europe. Mail Australian currency, a check payable through an Australian bank, or an Australian currency money order to:

R.M. Eades
P.O. Box 15
Hampton, Victoria, 3188
AUSTRALIA

Since the subscription rate may have changed by the time you read this, be prepared to send an additional payment.

3. PPC-UK Journal, published by the United Kingdom chapter of PPC. PPC-UK J is a relatively new publication, but so far it has placed considerable emphasis on tutorials and other helpful information for beginners. For more information and a membership application, send a self-addressed stamped envelope to:

David M. Burch
Astage
Rectory Lane
Windlesham, Surrey
GU2Ø 6BW
ENGLAND

Overseas inquiries should include an addressed envelope with an international postal reply coupon or two magnetic cards in lieu of postage.

4. The Hewlett-Packard Users' Library catalog contains a few synthetic programs. The Users' Library did not accept synthetic programs until January 1982, so the current catalog may not reflect the extent of synthetic programs in the Library.

The current membership fee for the Users' Library is $2Ø.ØØ in the US or Canada, and $3Ø.ØØ elsewhere. Mail your payment in the form of a check payable through a US bank to:

HP Users Library
1ØØØ N.E. Circle Boulevard
Corvallis, Oregon 9733Ø

5. HP Key Notes, published 3 times a year by Hewlett-Packard. A limited number of synthetic programs have appeared in Key Notes since the January 1982 initiation of synthetic programming to the Users' Library. Key Notes is a bargain at the current rate of $5 per year for US and Canadian residents. Residents of other countries can obtain Key Notes with a Users' Library membership. Send a check drawn on a US bank to:

HP Key Notes
1000 N.E. Circle Boulevard
Corvallis, Oregon 97330

For 1982, a Users' Library membership carries with it a free Key Notes subscription. This offer may or may not be continued through 1983.

6. Synthetic Programming on the HP-41C, a book by Bill Wickes, published by Larken Publications. This book was the first compilation of synthetic programming information and techniques. Because it was written in 1980, Wickes' book does not contain any examples using the byte grabber or Extended Functions module or Time Module functions. Nevertheless it remains a excellent reference book. Wickes's approach is substantially different than that of **HP-41 Synthetic Programming Made Easy**. Each subject is covered in full depth before the next subject is begun.

If you want to learn more about synthetic programming, I strongly recommend that you read "Synthetic Programming on the HP-41C". The knowledge you've gained from reading **HP-41 Synthetic Programming Made Easy** will enable you to get through Bill Wickes's book more quickly and with better understanding of the details. Wickes's book contains several interesting synthetic programs together with line-by-line analysis that will help complete your mastery of synthetic programming.

"Synthetic Programming on the HP-41C" is available at many calculator dealers and college bookstores. Alternatively, you may mail your order to:

Larken Publications
Dept. SPME
4517 NW Queens Ave.
Corvallis, Oregon, 97330
U.S.A.

The current price is $11 postpaid, by surface mail. For airmail, add: for USA, Mexico, Canada $1, for Europe and South America $2, for elsewhere $3. Payment should be in the form of a check payable through a US bank.

7. The PPC ROM User's Manual, which accompanies the PPC ROM. The PPC ROM is a custom ROM module for the HP-41 designed by PPC members and manufactured by Hewlett-Packard. The PPC ROM contains over 60 synthetic programs, each of which is analyzed line-by-line in the User's Manual.

By the time you read this, the PPC ROM may be available at calculator dealers. You may also order the PPC ROM from Personal Programming Center. For price and ordering information mail a self-addressed stamped envelope to :

PPC
2545 W Camden Place
Santa Ana, CA 92704

Mark the lower left corner of your outer envelope "PPC ROM ordering info". A substantial discount is available to PPC members. This discount could almost pay for your first year's membership.

8. Calculator Tips and Routines (Especially for the HP-41C/41CV), edited by John Dearing, published by Corvallis Software Inc. This book contains listings for many of the PPC ROM routines, some of which are synthetic. A great number of nonsynthetic programming tricks are also described.

"Calculator Tips and Routines" is available from dealers or directly from :

Corvallis Software, Inc.
Dept. SPME
P.O. Box 1412
Corvallis, Oregon 97339-1412
U.S.A.

The current price is $15 within the USA and Canada, $20 elsewhere, airmail postpaid. Payment should be in the form of a check in US dollars, payable through a US bank.

9. The HP-41 SYNTHETIC Quick Reference Guide, a pocket-sized (3-1/2 inch by 6 inch) compilation of synthetic programming information. Slightly wider than than the plastic Quick Reference Card for Synthetic Programming (so that the card will fit inside), the booklet contains XROM listings, a memory map, a byte table, tone tables, function timings, and some more exotic goodies. This is a reference book and not a "how to" book. However reference to the PPC Calculator Journal and other sources are included where further explanation is required. The HP-41 SYNTHETIC Quick Reference Guide is available from:

J.J. Smith
Dept. SPME
19451 Mesa Drive
Villa Park, CA 92667
USA

The price is $5.00 plus postage of $1.00 (US or Canada) or $2.00 (elsewhere). Instead of postage you may include a self-addressed stamped envelope with sufficient postage for two ounces.

10. **The HP-41C Quick Reference Card for Synthetic Programming.** Extra copies of this 2-7/8 inch by 6 inch plastic card are available from some dealers and college bookstores. Check the dealer from whom you bought this book.

Alternatively you may mail your order to:

Synthetix
Dept. SPME
1540 Mathews Ave.
Manhattan Beach CA 90266 USA

The price is $3 per card plus $1.50 per order shipping charge. US orders can enclose a self-addressed stamped envelope in lieu of the shipping charge. Payment should be in the form of a check payable through a US bank. If this is a problem, US currency is equally acceptable.

An earlier, more compact, black-and-white version of the QRC is also available while supplies last. It is 2-5/8 inch by 4-1/2 inch, so like the QRC it fits in the HP-41 carrying case alongside the calculator. Called the "HP-41C Combined Hex/Decimal Byte Table", it contains essentially the same basic byte table as the QRC. The only noticeable differences are the lack of a flag listing, multi-byte structure summary, and color tinting. The price is lower than the QRC at $2 for one card plus either $1 shipping or a self-addressed stamped envelope. Additional cards on the same order are $1 each to USA, Canada, and Mexico, $1.20 each to other countries. Checks (payable through a US bank) should be made payable and mailed to: Keith Jarett, Dept. SPME, 1540 Mathews Ave., Manhattan Beach, CA 90266 USA.

APPENDIX D

THE QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING ("QRC")

The QRC is a 2-7/8 inch by 6 inch plastic card that contains a wealth of information that is essential for synthetic programming. Each copy of **HP-41 Synthetic Programming Made Easy** comes with a QRC on the back cover.

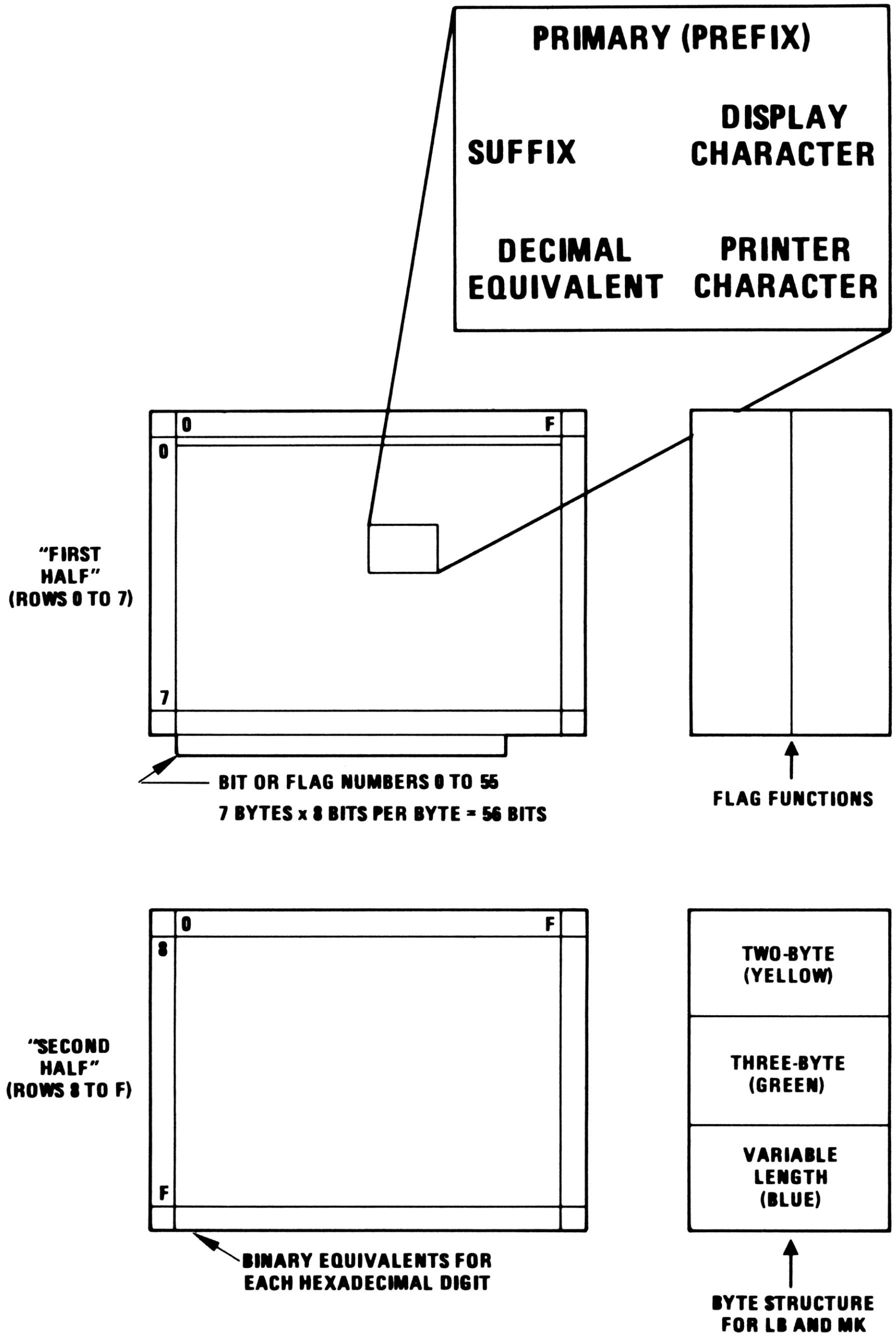
The leftmost two-thirds of the QRC is occupied by a byte table. Each box in the byte table illustrates the several possible interpretations of a byte. Refer to the "Legend for the QRC" on the next page. These equivalences are introduced and explained in Chapters 1 and 2.

Display characters are not shown for the second half of the byte table (rows 8 through F), since they are all starbursts (all 14 segments lit). This allows the full indirect suffix equivalents to be shown on the second line of each box. Printer characters shown are those that result from PRA when the byte in question resides in the ALPHA register. At the bottom of each half of the byte table are binary equivalents for the hexadecimal digits Ø through F.

To the right of the first half of the byte table is a summary listing of the functions of all 56 HP-41 flags. Next to the second half of the byte table is a quick reference summary of LB inputs (decimal byte equivalents) for each type of instruction. Chapter 3 covers this subject.

Obscure aspects of the QRC: Characters from rows 8 through F disappear in printed program **listings** (not PRA output), except that characters that are shaded will cause additional strange behavior (see Section 2E). Row Ø shows the required MK inputs, Ø through 15, for non-programmable functions in small letters. See Section 4A for details. Row 1 includes the W^{T} function which has no effect except to lock up the keyboard until the batteries are removed. The SPARE bytes will form two-byte No Operation instructions.

If this summary of the QRC seems confusing, you probably haven't read Chapters 1 and 2. Go back and read them!



Legend for the QRC

FLAGS (Register d)

00-10 general purpose
11 auto execute
12 doublewide
13 lower case
14 overwrite
15-16 IL printer
 0 0 MAN
 0 1 NORM
 1 0 TRACE
 1 1 TR/STACK
17 record incomplete
18 ⎤general use
19 ⎥cleared at
20 ⎦turn-on
21 prtr enable
22 num. entry
23 alpha entry
24 range ignore
25 error ignore
26 audio enable
27 USER mode
28 dec./comma
29 digit grouping
30 CAT
31 timer DMY/MDY
32 manual IL I/O
33 IL absolute manual
34 not used
35 not used
36-39 number of digits
40-41 display
 0 0 SCI
 0 1 ENG
 1 0 FIX
 1 1 FIX/ENG
42-43 trig mode
 0 0 DEG
 0 1 RAD
 1 0 GRAD
 1 1 RAD
44 cont. ON
45 system data entry
46 partial key sequence
47 SHIFT
48 ALPHA
49 low BAT
50 message
51 SST
52 PGRM
53 I/O
54 PSE
55 printer existence

Structure of multi-byte instructions

Two-byte instructions
 STO 16=145,16 DSE IND 55 =151,183
 LBL e =207,127 FS?C IND Y =170,242
 RCL b =144,124 TONE 89 =159,89
 X<>M =206,117 ST+ IND N =146,246
 LBL Q =207,121 VIEW H(109)=152,109
Two-byte special cases
 GTO IND=174,reg. XEQ IND=174,128+r
 GTO IND 09=174,9 XEQ IND X =174,243
 XROM i,j =160+i/4,64(i mod 4)+j
  WSTS =XROM 30,10 =167,138
 short form GTO =177+label,0
  GTO 12 =189,0
Three-byte instructions
 long form GTO =208,0,label
  GTO 32 =208,0,32
 XEQ =224,0,label
  XEQ D =224,0,105
 END =192,0,9+ sum of status indicators
  32(.END.), 4(rePACK), 2(decompile)
Variable length instructions
 TEXT =240+n, n character bytes
  Append symbol counts as first char.
  ⊤& =241,38 ⊤⊦)? =243,127,41,63
 GTO ⊤ =29,240+n, n character bytes
  GTO ⊤XYZ =29,243,88,89,90
 XEQ ⊤ =30,240+n, n character bytes
  XEQ ⊤A =30,241,65 (synthetic)
 LBL ⊤ =192,0,241+n, (key), n chars.
  LBL ⊤: =192,0,242,0,58 (synthetic)

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 α 4 α	LBL 04 05 β 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 æ	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		

HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING

	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 ȧ	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 œ	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 IND M [245 u	TEXT 6 IND N \ 246 v	TEXT 7 IND O] 247 w	TEXT 8 IND P ↑ 248 x	TEXT 9 IND Q _ 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	

For price information and a list of dealers in your area, send a self-addressed stamped envelope to: SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, CA 90266, USA

APPENDIX E

BARCODE FOR PROGRAMS

Barcode is provided here for all of the utility programs in this book, so that you may conveniently enter these programs into your HP-41 using the 82153A Optical Wand. If you have a wand or if you can borrow one, this will save you some time.

Always protect the surface of the barcode with a clear plastic sheet. It may also be helpful to place a ~~clean white~~ solid black sheet of paper behind the barcode to improve the contrast.

This barcode was tested in a trial printing and found to be readable. If your barcode is not readable, try inking in any incomplete bars, scanning the rows faster with the aid of a straightedge, or holding the wand at a different angle. If all else fails, try another wand.

If you have a card reader, you should record these programs in case your dog finds this book. Other methods of storing the programs include mass storage (IL tape drive) or extended memory. Extended memory should not be considered as a permanent storage, however, since it is susceptible to MEMORY LOST.

DECIMAL TO CHARACTER PROGRAM REGISTERS NEEDED: 8

ROW 1 (1 : 6)

ROW 2 (6 : 12)

ROW 3 (13 : 18)

ROW 4 (18 : 25)

ROW 5 (25 : 25)

LOAD BYTES

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 31

ROW 1 (1 : 6)

ROW 2 (6 : 15)

ROW 3 (15 : 22)

ROW 4 (22 : 25)

ROW 5 (26 : 31)

ROW 6 (32 : 38)

ROW 7 (38 : 44)

ROW 8 (45 : 53)

ROW 9 (53 : 61)

ROW 10 (61 : 68)

ROW 11 (69 : 76)

ROW 12 (76 : 82)

ROW 13 (83 : 88)

ROW 14 (88 : 95)

ROW 15 (96 : 103)

ROW 16 (103 : 110)

ROW 17 (110 : 112)

LOAD BYTES
(EXTENDED FUNCTIONS VERSION)
PROGRAM REGISTERS NEEDED: 23

ROW 1 (1 : 6)

ROW 2 (6 : 14)

ROW 3 (15 : 21)

ROW 4 (22 : 25)

ROW 5 (25 : 32)

ROW 6 (33 : 42)

ROW 7 (42 : 51)

ROW 8 (52 : 57)

ROW 9 (57 : 65)

ROW 10 (65 : 73)

ROW 11 (73 : 80)

ROW 12 (80 : 87)

ROW 13 (87 : 88)

MAKE KEY ASSIGNMENTS

PROGRAM REGISTERS NEEDED: 45

ROW 1 (1 : 5)

ROW 2 (6 : 12)

ROW 3 (12 : 19)

ROW 4 (19 : 22)

ROW 5 (23 : 28)

ROW 6 (29 : 31)

ROW 7 (31 : 35)

ROW 8 (35 : 40)

ROW 9 (40 : 48)

ROW 10 (48 : 57)

ROW 11 (58 : 67)

ROW 12 (68 : 77)

ROW 13 (78 : 85)

ROW 14 (85 : 93)

ROW 15 (93 : 99)

ROW 16 (99 : 103)

ROW 17 (103 : 108)

ROW 18 (109 : 116)

ROW 19 (116 : 121)

ROW 20 (121 : 128)

ROW 21 (129 : 136)

ROW 22 (137 : 143)

ROW 23 (143 : 149)

ROW 24 (149 : 154)

ROW 25 (154 : 154)

MAKE KEY ASSIGNMENTS
(EXTENDED FUNCTIONS VERSION)
PROGRAM REGISTERS NEEDED: 18

ROW 1 (1 : 3)

ROW 2 (3 : 8)

ROW 3 (8 : 14)

ROW 4 (14 : 18)

ROW 5 (18 : 26)

ROW 6 (27 : 31)

ROW 7 (31 : 37)

ROW 8 (38 : 41)

ROW 9 (41 : 48)

ROW 10 (49 : 51)

RAM BYTE COUNTER

PROGRAM REGISTERS NEEDED: 23

ROW 1 (1 : 3)

ROW 2 (4 : 8)

ROW 3 (8 : 14)

ROW 4 (15 : 23)

ROW 5 (23 : 27)

ROW 6 (28 : 31)

ROW 7 (31 : 38)

ROW 8 (39 : 43)

ROW 9 (44 : 50)

ROW 10 (50 : 56)

ROW 11 (57 : 63)

ROW 12 (63 : 69)

ROW 13 (70 : 70)

SAVE ALARMS/ RECALL ALARMS

PROGRAM REGISTERS NEEDED: 25

ROW 1 (1 : 6)

ROW 2 (7 : 14)

ROW 3 (15 : 23)

ROW 4 (23 : 30)

ROW 5 (30 : 36)

ROW 6 (36 : 41)

ROW 7 (41 : 49)

ROW 8 (50 : 57)

ROW 9 (57 : 63)

ROW 10 (64 : 72)

ROW 11 (73 : 81)

ROW 12 (82 : 87)

ROW 13 (88 : 96)

ROW 14 (97 : 99)

EXTENDED FUNCTIONS / TIME MODULE

PROGRAM REGISTERS NEEDED: 9

PAGE 1 OF 1

ROW 1 (1 : 5)

ROW 2 (6 : 12)

ROW 3 (12 : 16)

ROW 4 (16 : 23)

ROW 5 (24 : 26)

SUSPEND KEY ASSIGNMENTS/ REACTIVATE KEY ASSIGNMENTS

PROGRAM REGISTERS NEEDED: 10

PAGE 1 OF 1

ROW 1 (1 : 5)

ROW 2 (6 : 13)

ROW 3 (14 : 19)

ROW 4 (20 : 25)

ROW 5 (25 : 31)

SOLVE f(x) = 0 for x

PROGRAM REGISTERS NEEDED: 14

ROW 1 (1 : 2)

ROW 2 (2 : 7)

ROW 3 (7 : 10)

ROW 4 (10 : 19)

ROW 5 (20 : 29)

ROW 6 (30 : 39)

ROW 7 (40 : 48)

ROW 8 (48 : 49)

CURTAIN UP

PROGRAM REGISTERS NEEDED: 10

ROW 1 (1 : 5)

ROW 2 (5 : 10)

ROW 3 (11 : 21)

ROW 4 (21 : 29)

ROW 5 (29 : 35)

ROW 6 (35 : 35)

INSTRUCTION TIMER

PROGRAM REGISTERS NEEDED: 47

ROW 1 (1 : 8)

ROW 2 (8 : 13)

ROW 3 (13 : 17)

ROW 4 (17 : 26)

ROW 5 (26 : 31)

ROW 6 (31 : 40)

ROW 7 (40 : 52)

ROW 8 (53 : 61)

ROW 9 (62 : 66)

ROW 10 (66 : 71)

ROW 11 (72 : 79)

ROW 12 (79 : 85)

ROW 13 (86 : 94)

ROW 14 (94 : 98)

ROW 15 (99 : 106)

ROW 16 (107 : 116)

ROW 17 (117 : 124)

ROW 18 (124 : 133)

ROW 19 (134 : 139)

ROW 20 (140 : 147)

ROW 21 (148 : 155)

ROW 22 (156 : 163)

ROW 23 (164 : 172)

ROW 24 (172 : 177)

ROW 25 (178 : 185)

ROW 26 (185 : 186)

MORSE CODE

PROGRAM REGISTERS NEEDED: 121

ROW 1 (1 : 3)

ROW 2 (4 : 10)

ROW 3 (11 : 18)

ROW 4 (18 : 19)

ROW 5 (20 : 27)

ROW 6 (28 : 35)

ROW 7 (36 : 43)

ROW 8 (43 : 49)

ROW 9 (50 : 58)

ROW 10 (58 : 65)

ROW 11 (65 : 72)

ROW 12 (73 : 83)

ROW 13 (84 : 90)

ROW 14 (91 : 97)

ROW 15 (98 : 102)

ROW 16 (103 : 108)

ROW 17 (108 : 114)

ROW 18 (114 : 120)

ROW 19 (120 : 125)

ROW 20 (126 : 132)

ROW 21 (132 : 137)

ROW 22 (138 : 143)

ROW 23 (143 : 149)

ROW 24 (149 : 155)

ROW 25 (155 : 160)

ROW 26 (161 : 166)

ROW 27 (166 : 173)

ROW 28 (173 : 178)

ROW 29 (178 : 183)

ROW 30 (184 : 189)

ROW 31 (189 : 195)

ROW 32 (196 : 201)

ROW 33 (201 : 206)

ROW 34 (207 : 213)

ROW 35 (213 : 218)

ROW 36 (219 : 224)

ROW 37 (224 : 229)

ROW 38 (230 : 235)

ROW 39 (235 : 242)

ROW 40 (242 : 247)

ROW 41 (247 : 252)

ROW 42 (253 : 258)

ROW 43 (258 : 263)

ROW 44 (264 : 269)

ROW 45 (269 : 275)

ROW 46 (275 : 281)

ROW 47 (281 : 286)

ROW 48 (287 : 292)

ROW 49 (292 : 297)

ROW 50 (298 : 304)

ROW 51 (305 : 310)

ROW 52 (310 : 315)

ROW 53 (316 : 321)

ROW 54 (321 : 327)

ROW 55 (327 : 333)

ROW 56 (334 : 338)

ROW 57 (339 : 344)

ROW 58 (344 : 349)

ROW 59 (350 : 355)

ROW 60 (355 : 360)

ROW 61 (361 : 366)

ROW 62 (366 : 371)

ROW 63 (372 : 377)

ROW 64 (377 : 382)

ROW 65 (383 : 387)

INDEX

Errata and Selected Useful Facts

Printer slows execution

Having a printer attached to your HP-41 will slow execution of your programs, regardless of whether flag 21 is set or the printer is turned on. Even instructions that are not intended to involve the printer are slowed.

This speed penalty can be reduced by synthetically clearing flag 55, the printer existence flag. Any of the following sequences of instructions will accomplish this:

with "bare" HP-41:	with XFUNCTIONS module*:	with PPC ROM:
SF Ø7**	RCLFLAG	55
RCL d	SIGN	FS? 55
CLA	STO d	RDN
STO M	X<> L	FC? 55
ASTO M	STOFLAG	XROM IF
├⁻	RDN	
X<> M		
STO d		
RDN		

*this routine was written by Steve Wandzura

**any flag from ØØ to Ø7 can be used.

As long as your program continues to run without encountering a printer function, flag 55 will remain clear and execution will be speeded. If flag 21 is clear, encountering a printer function will not set flag 55 either. The function will be ignored just as it would normally.

If flag 21 is set, the behavior depends on the type of printer present. With an 82143A printer, all printer functions are disabled until the program halts, at which time flags 21 and 55 are immediately set (even if 21 was clear). With an HP-IL printer, the set status of flag 21 will cause the printer function to be executed and flag 55 to be set. Simply halting execution will not set flag 55 as for the 82143A printer, but executing a flag test, VIEW, or related instruction from the keyboard will set flag 55.

Avoid decompiling

Suppose you record a program on magnetic cards after executing it once to compile all the GTO's and XEQ's. (Refer to page 60 for a definition and explanation of compiling.) When you read the cards back in, the GTO's and XEQ's will still be compiled, so that no searches for the LBL's are required. However the branching information contained in the GTO's and XEQ's will be lost the next time you GTO.. or PACK. A simple synthetic technique invented by Clifford Stern allows you to pack without losing this information:

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, BST, BG, 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.

This method applies identically to programs read in from tape, extended memory, or any other source.

ROM/RAM distinction with STO b

Most RAM program pointers would constitute equally valid ROM program pointers (see pages 114 and 115). The HP-41 therefore must remember internally with some sort of flag whether the current location is in ROM or RAM. This flag cannot be changed by STO b.

Thus STO b can only be used to jump from one ROM location to another or one RAM location to another. A common mistake is to press a STO b assigned key while the program pointer is in ROM, expecting to jump to a particular location in RAM. This will not work. Instead you should execute Catalog 1 (it is OK to R/S immediately) to get back to RAM before pressing STO b.

Q-register shortcuts

When you spell out an ALPHA label name from the keyboard (while keying in a LBL, a GTO, or an XEQ), the name will be loaded into the Q register. This fact is helpful when using eGOBEEP 77 to execute PRP (see page 76). For example, to print a program that contains LBL"ABC", you can press GTO ALPHA A B C ALPHA, eGOBEEP 77. You can get even fancier by pressing eGOBEEP ALPHA A B C ALPHA, eGOBEEP 77. This latter example makes use of the obscure fact, discovered by Robert Edelen, that eGOBEEP"name" has the same result as LBL"name".

Another useful shortcut, discovered by Clifford Stern, is to clear the Q register by pressing XEQ ALPHA backarrow. You can then obtain a TEXT Ø instruction by pressing Q-LOAD (MK inputs 27, Ø) and backarrow. Refer to page 7Ø. If you press eGOBEEP 77 after clearing Q, you will cause the current program to be printed, just as if you had pressed PRP ALPHA ALPHA.

Subroutine use of "RA"

If "RA" (recall alarms, see page 89) must be called as a subroutine, replace line 38 (the OFF instruction) by ALMNOW and RTN. The ALMNOW instruction will reset the Time Module's countdown to the pending alarm.

"EFT" use of PCLPS

The useful PCLPS function can be executed by means of the "EFT" routine (page 94) as long as "EFT" itself is not cleared in the process. PCLPS provides the fastest method of clearing main memory programs.

“SOUP UP” YOUR HP-41 — It’s Easy and Fun!

Synthetic programming encompasses the creation and use of synthetic instructions — those instructions that cannot be keyed up by normal means. Applications of synthetic instructions included expanded key assignment capability (assign SF 14 or GTO IND X to a key), 21 additional display characters, and renumbering of data registers under program control.

If you have heard about synthetic programming and want to know more, or if you have found other sources of information on synthetic programming confusing or difficult to read, try this book. **HP-41 SYNTHETIC PROGRAMMING MADE EASY** uses all the latest synthetic programs and techniques, and gives many cross-references to other sources, all of which will be much more readable after you have been through this book. Barcode for all programs is included for those readers who have access to an optical wand. Also included is the handy plastic **QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING**, a $3.00 value.

If you like your HP-41, you’ll like **HP-41 SYNTHETIC PROGRAMMING MADE EASY.** Thousands of HP-41 owners have learned synthetic programming. Shouldn’t you?