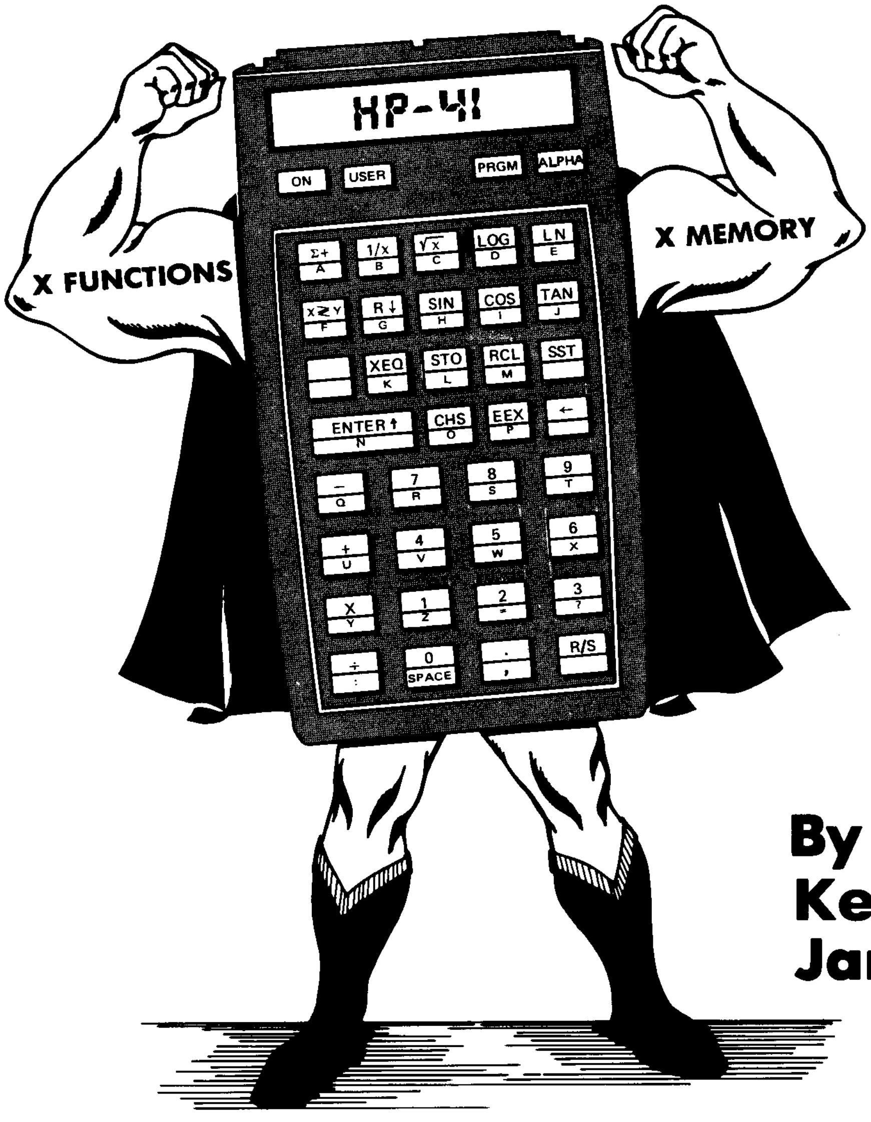
HP-41 EXTENDED FUNCTIONS MADE EASY



By Keith Jarett

For the HP-41C, HP-41CV, and the New HP-41CX

HP-41
EXTENDED FUNCTIONS MADE EASY

By Keith Jarett

© 1983, SYNTHETIX
P.O. Box 113
Manhattan Beach
CA 90266 USA

Library of Congress Card Catalog Number: 83-050952
ISBN: 0-9612174-1-3

PREFACE

DEDICATION

This book is dedicated to my wife, Catherine Van De Rostyne, who could not have been more helpful and supportive.

ACKNOWLEDGEMENT

The most important contributions to this book were made by Clifford Stern, who was the technical consultant throughout its development. Clifford is one of a handful of "grand masters" of HP-41 programming. He is probably more familiar with the subtleties of the HP-41 operating system than any other individual. Clifford wrote most of the very advanced utility programs in Chapter 10, checked the book for technical errors, and provided valuable suggestions for its improvement.

Other contributors to this book were Erik Christensen, who wrote the text editor program and documentation that appear in Chapter 8, Tapani Tarvainen and Gerard Westen, who wrote the amazing key assignment program in Chapter 10, and Alan McCornack, who wrote the mailing list program in Chapter 7 and provided many helpful suggestions during editing.

ABOUT THE AUTHOR

Keith Jarett has been addicted to Hewlett-Packard calculators since he bought an HP-45 in 1973 and wrote manual keystroke programs for it. In 1980 and 1981, he coordinated the development of 67 synthetic routines for the PPC ROM (see Appendix C), a custom program module by and for HP-41 users.

He is currently a Systems Engineer in Hughes Aircraft's Space and Communications Group. He graduated from Culver Military Academy in 1972, received a B.S. degree in Electrical Engineering from Cornell University in 1975, an M.S. in E.E. from Stanford in 1976, and a Ph.D. in E.E. from Stanford in 1979.

For price information on this book, write to: SYNTHETIX, P.O. Box 113, Manhattan Beach, CA 90266. Enclosed an addressed return envelope for faster reply. Dealer and distributor inquiries are welcome.

TABLE OF CONTENTS

INTRODUCTION

The Extended Functions/Memory module, built into the new HP-41CX and available separately for the HP-41C and CV, provides many new functions for your HP-41[1]. It also provides 127 additional registers of memory. Up to two Extended Memory modules can be added, each of which contains 238 registers. Thus, depending on how many Extended Memory modules you plug in[2] to accompany your Extended Functions/Memory module, you will have 127 to 6Ø3 registers of extended memory available.

Extended memory is an example of "off-line" storage. Programs in extended memory are not directly usable; they must first be brought into main, or "on-line", memory. Once in main memory, programs can be modified or executed as needed. Data can also be stored in extended memory.

In this respect, the operation of extended memory is similar to that of the HP821Ø4A Card Reader. The card reader also has the capability to save programs and data outside main memory. The most important differences between extended memory and the card reader are:

Card reader	Extended memory
Unlimited capacity	Limited capacity (127 to 6Ø3 regs.)
Manual operation	Keyboard or program control
Long-term storage	Short- to medium-term storage (susceptible to MEMORY LOST)
Write-all, status, data, program cards	Program, data, text file types

The extended memory equivalent of a magnetic card set is called a <u>file</u>. Just as there are different formats for magnetic cards (program cards, data cards, etc.), there are different types of extended memory files. The three file types are program, data, and text, also called ASCII[3].

A program file, as the name implies, holds a program. A data file holds one or more registers of data. A text, or ASCII file holds a collection of character strings. These file types will be introduced and explained in the first three chapters where examples of their use will be given.

---

[1]The Extended Functions/Memory module provides 47 functions. The HP-41CX includes these 47 functions plus 14 more, for a total of 61 extended functions.

[2]Do not plug in two Extended Memory modules one above the other. The Extended Functions/Memory module can be plugged in anywhere. Consult Section 1 of your Extended Functions/Memory module Owner's Manual for details of which module configurations are allowed. If you have only one Extended Memory module, you should plug it into port 1 or port 3 (top left or bottom left as viewed from the top end of the calculator).

[3]American Standard Code for Information Interchange -- a system for expressing character values in 7 binary digits. Each character uses one byte (1/7 of a register) in the HP-41.

"BUGS"

A "bug" is defined as any behavior of a function that is either undesirable and unexpected or that is different from what is described in the appropriate Owner's Manual. Because of the incredible complexity of the internal programming in the HP-41's Extended Functions ROM (Read-Only-Memory), a few bugs managed to survive the normal "debugging" procedure at Hewlett-Packard. Under certain somewhat unusual circumstances, some of the extended functions can have unpleasant results, sometimes even MEMORY LOST.

If you have either an Extended Functions/Memory module or a card reader manufactured before September 1983, you should read this section before proceeding. Otherwise your HP-41 system is essentially free of bugs, and you may skip this discussion for now.

This book gives full details and techniques to help you prevent problems with any extended functions bugs that you may have. In some cases, you can even repair the damage caused by a bug after it "bites". The purpose of this brief summary is to give you enough information so that you can avoid trouble until later in this book, where prevention and repair techniques are explained in detail.

You should not fault HP for the existence of bugs. They spend much time testing their work before releasing it in a product, but no set of tests can cover all operating conditions. At some point testing has to end and production begin. If you want perfection, you will have to wait a long time.

Hewlett-Packard produces calculators and modules with fewer problems than any other manufacturer. What few bugs remain are simply the price for a product that is ahead of its competition in performance and value.

There are three bugs in the early versions of the HP-41 Extended Functions/Memory module. Techniques presented in this book will eliminate all problems with these bugs. Usually, preventative steps are described, but a repair procedure is also possible for the most frequently occurring bug.

The first extended functions bug is that the SAVEP (save program) and PCLPS (programmable clear programs) functions must not be executed if the calculator is positioned in an application module program, outside main memory. This is covered in detail on pages 12 and 13. If you use the "XF" program (Section 1ØB) to execute extended functions from the keyboard, this problem will be automatically avoided.

The second bug is that the PURFL (purge file) function sets up a dangerous situation in which a wrong move can lose access to the entire contents of extended memory. Fortunately, synthetic programming techniques (Section 1ØE) can be used to repair the damage. Other simple measures can help to prevent the problem from occurring in the first place. For more information on PURFL, refer to pages 19 through 21. In case you have never heard of synthetic programming, Section 1ØA has a short description of what synthetic programming is and how you can learn more about it.

The third bug is that the card reader's VER and 7CLREG functions can alter the contents of extended memory. Until you read the full details in Appendix A, do not use VER when an Extended Memory module is present in port 2 or port 4 or in a combined module. The Extended Functions/Memory module can be plugged in anywhere without risk. A short synthetic program in Section 1ØD will allow you to use VER without harming extended memory.

The first two bugs listed above appear only in Revision 1B of the Extended Functions module. You can find out which revision you have by executing Catalog 2 (press shift CATALOG

2). Somewhere in the list of peripheral functions will appear the message

```
          -EXT FCN 1B
          -EXT FCN 1C
     or   -EXT FCN 2C   (HP-41CX only)
```

If the message goes by too fast, you can press R/S to interrupt the listing, then use BST to get back to it. Revisions 1C and up are free of the SAVEP/PCLPS and PURFL bugs.

The third bug is actually due to the operation of the card reader. It appears in all but the most recent card readers. Check the revision number of your card reader by running Catalog 2. If you see

```
          CARD READER
          CARD RDR 1D
          CARD RDR 1E
     or   CARD RDR 1F
```

then your card reader has the VER bug. If you have a revision 1G or higher, your card reader is free of this bug.

Note that the HP-41CX has no bugs in its extended functions, but if you use it with an older card reader, you will still have to avoid using the VER function until you read Section 1ØD.

Now let's find out what extended memory is all about.

CHAPTER ONE
SAVING PROGRAMS IN EXTENDED MEMORY

1A. Creating a Program File

The most commonly used extended memory operations are **SAVEP (save program)** and **GETP (get program).** These operations save a program in extended memory and retrieve it from extended memory. To illustrate these functions, turn on your HP-41, press GTO.. and key in the following program:

```
                                                             54 RCL 02
01♦LBL "JNX"  12 *         22 RCL 03   33 X≠0?    44 STO 01   55 ST/ 01
 02 STO 03    13 STO 07    23 /        34 /       45 RCL 05   56 ST/ 04
 03 ABS       14 1/X       24 RCL 04   35 ST+ 02  46 STO 06   57 ST/ 05
 04 5         15 STO 02    25 STO 05   36 RCL 00  47 RCL 07   58 ST/ 06
 05 +         16 STO 04    26 *        37 2                   59 RCL 05
 06 X<>Y      17 STO 05    27 +        38 ST- 07  48♦LBL 01   60 RCL 04
 07 STO 00                 28 STO 04   39 *       49 X≠0?     61 RCL 06
 08 X<Y?      18♦LBL 00    29 RCL 07   40 RCL 07  50 GTO 00   62 RCL 01
 09 X<>Y      19 RCL 05    30 4        41 X≠Y?    51 RCL 04   63 END
 10 INT       20 CHS       31 /        42 GTO 01  52 ST- 02
 11 4         21 RCL 07    32 FRC      43 RCL 04  53 RCL 01        80 BYTES
```

If you see the message PACKING followed by TRY AGAIN, you will need to reduce the SIZE to make more registers available for this program. An alternative to reducing the number of data registers allocated is to use the CLP function to clear one or more programs (choose ones that are expendable or that you have saved on cards or tape) to make space available.

When you are done keying in this program, GTO.. to pack and attach an END to it. This is important to minimize the space required to store the program in extended memory. Another thing you should do before saving the program is to execute each GTO (lines 42 and 5Ø) at least once. This is explained more fully on page 189.

To execute these lines, switch to RUN (non-PRGM) mode and press GTO .Ø42. Next press SST and hold the key down until the program line appears in the display, then release the key. When the display clears, you know that line 42 has been executed. Next press

GTO.050

SST (hold until line 50 appears, then release).

The program is now ready to be saved in extended memory. The procedure for saving a program will be described in section C of this chapter.

Description of the "JNX" program

The "JNX" program computes Bessel functions of the first kind of integer order, $J_n(x)$, with eight-digit accuracy. This program may or may not be useful to you, but it is a good example of the power of the HP-41. The Bessel function program "JNX" will be used in Chapter 6, so you may wish to save it on a magnetic card or the cassette drive before proceeding. To test that your version of "JNX" is operating correctly, try

2 ENTER↑ 1.2 XEQ "JNX"

to calculate $J_2(1.2)$. The result should be $1.593490184 \times 10^{-1}$.

You may now skip to the beginning of the next section (page 10) unless you are particularly interested in Bessel functions. The following detailed discussion describes exactly how the "JNX" program works.

Line-by-line analysis of "JNX"

The algorithm used by "JNX" is based on the recurrence relation

$$J_{i-1}(x) = (2m/x)J_i(x) - J_{i+1}(x) \quad .$$

The process starts with initial estimates

$$J_m(x) = J_{m+1}(x) = 1/2m, \text{ where } m = 2*\mathrm{INT}(\max(n,x+5)).$$

The recurrence relation is evaluated for decreasing values of n, until n=0 is reached. During this process, the sum

$$J_0(x)+2J_2(x)+2J_4(x)+2J_6(x)+\ldots$$

is evaluated. Since this sum theoretically should equal 1, it is used to normalize the previously computed value of $J_n(x)$.

Now to the specifics of this program. The data register usage of "JNX" is

register	contents	
ØØ	n	
Ø1	$J_n(x)$	
Ø2	normalization sum	
Ø3	x	
Ø4	$J_i(x)$	(starts at 1/2m)
Ø5	$J_{i+1}(x)$	(starts at 1/2m)
Ø6	$J_{n+1}(x)$	
Ø7	2i	(starts at 2m)

Lines Ø1-17 of "JNX" initialize the data registers. The LBL ØØ loop computes $J_{i-1}(x)$ from the previous estimates $J_i(x)$ and $J_{i+1}(x)$. This new estimate replaces the old $J_i(x)$ (line 28), while $J_i(x)$ replaces the old $J_{i+1}(x)$ (lines 24 and 25).

Lines 3Ø-35 add $2J_{i-1}(x)$ to the normalization sum only if i is odd (that is, if the fractional part of 2i/4 is Ø.5).

Lines 37-38 decrement i (register Ø7) for the next time through the loop. Then, if i=n, lines 43-46 save the current values of $J_i(x)$ and $J_{i+1}(x)$ for later use. Otherwise these lines are skipped over. Unless i=Ø (lines 49-5Ø) the LBL ØØ loop is repeated, counting down one more step toward $J_\emptyset(x)$.

When i=Ø is reached, registers Ø4 and Ø5 contain estimates of $J_\emptyset(x)$ and $J_1(x)$. Lines 51-52 adjust the sum for the extra $J_\emptyset(x)$ term added at line 35. Then the normalization factor is applied to all four Bessel function estimates. When the program halts, the following results are in the stack:

register	contents
T	$J_1(x)$
Z	$J_\emptyset(x)$
Y	$J_{n+1}(x)$
X	$J_n(x)$

1B. The EMDIR function

Before saving the "JNX" program in extended memory, it is advisable to check the status of extended memory. Press

```
          XEQ ALPHA E M D I R ALPHA  .
```

EMDIR is the Extended Memory DIRectory function. If you have an HP-41CX, a shortcut is available. You can press

```
          shift CATALOG 4
```

to request the extended memory directory.

If you haven't used extended memory yet, you will see the message DIR EMPTY. If you have saved programs or data in extended memory, note that each entry in the directory describes a "file", which is a set of extended memory registers allocated to storing a program or a block of data. The file description consists of three items: the file name, the file type, and the file size.

The file name is a string of up to 7 characters. The file type is designated by a single letter: P for program, D for data, and A for ASCII (text). These are covered in Chapters 1, 2, and 3, respectively. The file size is the number of extended memory registers allocated. Actually, two more registers per file are used for the file header. One header register holds the file name, while the other holds file type, length, and pointer codes. (Full details are given in section 1ØC.) Thus if you create a 12-register file, whether it be a program file or any other type of file, the number of free registers in extended memory will drop by 14.

You can check the number of extended memory registers available by letting EMDIR run to completion. The number will then appear in the X-register (raising the stack). This number is always two less than the number of unused registers left in extended memory, because the calculator automatically subtracts the two registers that will be needed by the next file created. Thus if EMDIR yields a register count of 53, there are actually 55 unused registers, but the largest file you can create is a 53-register file.

The EMDIR function is somewhat analogous to the CATALOG 1 function for main memory. Because of its usefulness, you should consider assigning EMDIR to a key. Press

shift ASN ALPHA E M D I R ALPHA,

followed by the key of your choice. With an HP-41CX, the extended memory directory can be interrupted, single-stepped, and back-stepped in the same way as Catalog 1. With an HP-41C or CV you cannot interrupt the extended memory directory. Instead, you can "freeze" the display by pressing and holding any key except R/S or ON. Release the key to resume the listing. Press R/S to interrupt (terminate) the listing.

On the HP-41CX, there are two other functions that are related to EMDIR. The **EMROOM (extended memory room)** function returns the number of extended memory registers available for data, just as does EMDIR when run to completion. The difference is that no directory is displayed. This makes EMROOM more suitable than EMDIR for use in a program that creates extended memory files. You can test the number of registers available before trying to create a file, perhaps reducing the requested file size to match the EMROOM.

Also on the HP-41CX is the function **EMDIRX (extended memory directory -- file X)**. When you put a number **n** in X and execute EMDIRX, the name of the nth file in extended memory will be returned in the ALPHA register and the file type will be returned to X as a two character string (PR, DA, or AS for program, data, or ASCII files).

1C. Using SAVEP and GETP

To save the "JNX" program in extended memory, simply press

ALPHA J N X ALPHA

followed by

XEQ ALPHA S A V E P ALPHA .

The display will blank for a few moments, except for the annunciators, while the operation is performed.

This procedure for using **SAVEP (save program)** is similar to the procedure that you will use for many other extended memory operations. First you load the name of the program or file into the ALPHA register, then you execute the function.

To check the results of your SAVEP operation, execute EMDIR (or Catalog 4 for the HP-41CX). You should see:

```
        JNX     PØ12.
```

If it went by too fast, you can try EMDIR again. On the HP-41C or CV, holding down any key freezes the display until you release the key. On the HP-41CX you an press R/S to halt the directory and SST to step through it.

Another way to check the existence or size of an extended memory file is to use the **FLSIZE (file size)** function. Just put the file name in the ALPHA register and execute FLSIZE. For the above example, you would press

```
        ALPHA J N X ALPHA
        XEQ ALPHA F L S I Z E ALPHA    .
```

The result should be the number 12 in the X register. This is the size of the "JNX" file, exclusive of the 2 header registers that are needed by any extended memory file.

When you use **FLSIZE**, the result is the size of the named file if it exists, or the error message FL NOT FOUND if the named file does not exist. FLSIZE works the same for all three types of files: program, data, and ASCII. If you execute FLSIZE with the ALPHA register empty, the size of the "working" file will be returned. The "working", or "current", file is the last file that you referred to by name in an extended memory function like SAVEP, or the file at which an EMDIR display was terminated. More details on "working" files will be given on page 2Ø.

WARNING: (Revision 1B only) When you use SAVEP, make sure that the calculator is positioned in main memory, not in an application module's read-only memory (ROM). Be especially wary when an application module (Math Pac, Standard Pac, etc.) is plugged in. Even the printer (or the HP-IL module with the printer switch on) contains three programs, PRAXIS, PRPLOT and PRPLOTP, in read-only memory. The "XF" program presented in Chapter 10 guarantees that you will be in main memory when you execute SAVEP. This warning does not apply to the HP-41CX, nor does it apply to revisions 1C and up of the Extended Functions/Memory module.

To find out whether or not you are in an application module program, start in RUN (non-PRGM) mode, press

shift RTN

and switch into PRGM mode. This moves you to line 00 of the current program. If you see just

00

then you are in an application ROM program. You can press

shift CATALOG 1 or GTO ..

to get back to main program memory. If you know the name of a specific program in main memory, you can press

GTO ALPHA (program name) ALPHA .

This will also return you to main memory. Once in main memory if you press

shift RTN PRGM

to move to line 00 of the current program, you should see

00 REG nnn ,

where nnn is the current number of free registers in main program memory.

If you accidentally do a SAVEP while you are outside main memory, do a PURFL(purge file - page 19) immediately. The program file that SAVEP creates under these conditions is likely to be quite large and is certain to be unusable. One of these files can even lock up the calculator's keyboard if you try to bring it into main memory.

Incidentally, if you want to transfer an application module program to extended memory, you will have to do a COPY first, to bring the program into main program memory.

This same warning against being outside main memory applies even more strongly to PCLPS (programmable clear programs

-- to be discussed on page 18). In the case of PCLPS, the penalty for a mis-step is the dreaded MEMORY LOST. Once again, this warning does not apply to the HP-41CX or to revisions 1C and up of the Extended Functions/Memory module.

Now let's retrieve the "JNX" program from extended memory. Make sure that the ALPHA register still contains "JNX", then GTO.. and press XEQ ALPHA G E T P ALPHA. Assuming that you had sufficient program space available, you should now have a second copy of "JNX" in main memory. Execute Catalog 1 and you should see LBL^{T}JNX, END, LBL^{T}JNX, and .END. REG xx as the last items listed.

This illustrates the fact that **GETP (get program)** retrieves the designated program and places it between the last END and the .END., even if this means that a program must be overwritten. In particular, if you had not performed the GTO.. to attach an END to the "JNX" program before saving it, the GETP operation would have overwritten the old copy of "JNX" with the new one, leaving only one copy instead of two. You may wish to practice more with SAVEP and GETP, using your own library of programs.

When they are assigned to keys, the functions SAVEP and GETP provide single-keystroke equivalents of recording and reading magnetic program cards. A typical application is moving a program down to the bottom of Catalog 1 for faster response when editing or PACKing. [When you insert an instruction or PACK a program that is located near the top of Catalog 1, all lower programs in Catalog 1 must be shifted. This slows the calculator's response noticeably.] Use SAVEP to save the program, CLP to clear it from main memory, and GETP to bring it back at the bottom of Catalog 1. Techniques like this are useful, but the full power of SAVEP and GETP is harnessed by using these instructions within your programs.

Before exploring this subject, you need to know a little more about SAVEP and GETP.

1D. Advanced features of SAVEP

The above examples might have led you to believe that a program can only be saved in an extended memory file having the same name as the program. With a little more effort, it is possible to save a program under any file name you like. Instead of just putting the program name (actually the name of any Catalog 1 ALPHA label in the program) in the ALPHA register, you can follow the program name with the desired file name, using a comma as a separator. If you want to save the current program, it is OK to omit the program name and simply load ALPHA with a comma, followed by the file name.

[The **current program** is the one that appears when you switch into PRGM mode. This is usually the program you executed most recently, unless you have pressed GTO.. or CATALOG 1, both of which can move you to a different portion of program memory.]

The allowable ALPHA contents for SAVEP are as follows:

ALPHA contents	result
"program name"	The program containing an ALPHA label of this name is saved in extended memory in a file of the same name.
"program name,file name"	The named program is saved in a file of the designated name. CAUTION: Do not leave a space after the comma, unless you want the space to become part of the file name.
",file name"	The current program is saved in a file of this name.

Note: Commas are not allowed in file names, since the comma is interpreted as a name separator. File names cannot exceed seven characters (any excess characters are ignored).

1E. Advanced features of GETP, including GETSUB and PCLPS

Unlike SAVEP, the GETP function operates differently when it is executed as part of a program rather than from the keyboard. Either way, GETP brings the program file named in the ALPHA register into program memory, putting it just after the last END in Catalog 1 and before the .END. . As in the case of reading a program from cards, barcode, or cassette, the program's ALPHA label key assignments will only take effect if the GETP is performed in USER mode. (Any existing conflicting key assignments are overwritten.)

When called from the keyboard, GETP sets the calculator to the first line of the retrieved program. This makes it convenient either to switch into PRGM mode and review the program, or to press R/S and run the program.

When it is encountered in a running program, GETP reads in the named program file and continues to run the original program. An exception is made for the case in which the original program was the last program in main memory. [The last program in main memory is the one that has the .END. as its last line.] In this case, when the program file is read in, the original program is overwritten. Clearly the original program cannot continue to run. Instead, execution resumes at the first line of the new program.

When you write a program that uses GETP, you must carefully plan the GETP operations so that no important programs are accidentally overwritten. Often it is helpful to place a note in the program's operating instructions, requiring that the user either clear the last program in memory or to GTO.. before running the program. It is good operating practice not to GTO.. until you make sure either that the last program area in main memory is blank (no lines other than the .END.) or that it contains at least one ALPHA label. This precaution will help prevent the annoying proliferation of excess ENDs in Catalog 1.

A program can use GETP to load the subroutines it needs from extended memory, each time overwriting the previously used subroutine. This technique, called overlaying, is necessary for very large programs when all the subroutines will not fit into main memory at the same time.

The precautions needed when GETP is used may tempt you to use GETSUB instead. The **GETSUB (get subroutine)** function is almost the same as GTO.. followed by GETP. The difference is that a new END is created even if the last program in main memory was blank. If you make a habit of using GETSUB, you will soon find that your Catalog 1 is full of extra END's. These END's will have to be deleted using the following procedure:

If you have an extra END with no ALPHA label preceding it in Catalog 1, the only way to get rid of it is to run Catalog 1 and interrupt it at the extra END. Then you can either switch into PRGM mode and backarrow the END or you can press

XEQ ALPHA C L P ALPHA ALPHA ALPHA.

(When no program name is supplied, the CLP function clears the current program.) If several ENDs are back-to-back, stop Catalog 1 at the first one, switch into PRGM mode, and press backarrow and SST alternately until a non-END line appears, indicating that the entire group of ENDs has been deleted.

Whether GETSUB is executed from a program or from the keyboard, the result is the same. The .END. is converted to an END, the program file named in the ALPHA register is read in, and a new .END. is added. Execution is not transferred to the new program.

The only valid use of GETSUB is to retrieve a program from extended memory when the last program in Catalog 1 has the .END. as its last line, and you do not want that program to be overwritten. This can occur when several program files need to be read in from extended memory. The first file can be read in using GETP, while GETSUB can be used for the subsequent files. This procedure avoids the creation of extra END's. However, if the last program in Catalog 1 has a nonpermanent END, or if you do not want to save that program, then use GETP rather than GETSUB.

Incidentally, if you are familiar with the card reader functions, GETP is precisely analogous to reading in a program card set, and GETSUB is almost analogous to executing the RSUB card reader function. The difference is that RSUB only converts the .END. to an END if you are in the last program in main memory (the one that has the .END. as its last line).

When you are done using all the programs that were read in, you can use the **PCLPS (programmable clear programs)** function to delete them. Just load the ALPHA register with the name of the first program that you read in from extended memory (the one that you read in with GETP, rather than GETSUB). Then PCLPS will clear that program <u>and all programs following it in Catalog 1</u>. If the ALPHA register is clear, the current program and all following programs will be cleared. This will occur exactly the same whether PCLPS is called from the keyboard or encountered in a running program. Execution of the running program will continue after the affected programs have been cleared.

WARNING: (Revision 1B only) If the calculator is positioned outside main program memory (in an application module program) and the ALPHA register is <u>not</u> clear, executing PCLPS will give **MEMORY LOST**, after a delay of several seconds for dramatic effect. Refer to the SAVEP warning (page 12) for more details. The "XF" program in Section 10B has the incidental benefit of preventing this problem from occurring.

Astute readers will notice that PCLPS has an exception much like the one for GETSUB. PCLPS clears the named program and all the programs following it. If the PCLPS function is executed in a running program that is one of the programs being cleared, execution will terminate at the .END. . However, if the PCLPS instruction was part of a subroutine, the

.END. will be executed and interpreted as a RTN. Control will return to the calling program <u>which may no longer exist</u> if you did not plan things correctly. If the calling program does not exist (due to the action of PCLPS), you will find yourself outside the program area of the HP-41, in the I/O buffer and key assignment registers which lie beyond the .END. . This will occur regardless of which revision of extended functions you have. Although synthetic programmers will relish the possibilities, this situation should be avoided. If you are unlucky enough to have this problem occur, just interrupt the program (if it doesn't stop itself with an error of some sort) then press

shift CATALOG 1 or GTO ..

to reposition the calculator to main program memory.

<u>1F. Clearing a file from extended memory</u>

Just as the CLP (clear program) instruction clears the named program from main memory, the **PURFL (purge file)** function removes the named file from extended memory. The named file can be a program, data, or ASCII file. After the file is removed, extended memory is automatically packed to free the space formerly used by the file. This operation is somewhat similar to the packing performed by CLP.

WARNING: (Revision 1B only) The PURFL function has one very dangerous feature. After PURFL is executed, there is no "working" file. If a working file is not quickly re-established, the entire contents of extended memory can be rendered inaccessible. Any instruction that operates on the working file will <u>destroy the extended memory directory</u> if a working file does not exist at the time the instruction is executed. Then special techniques (Section 1ØE) are necessary to restore the directory.

Note: If your Extended Functions module is a revision 1B, this "bug" can be useful. The sequence PURFL, RCLPT is a quick and easy way to clear the extended memory directory without disturbing main memory. This sequence should never be used in a program, though.

The **"working" file** , which is called the **current file** in the HP-41CX Owner's Manual, is the last file used or created, unless an EMDIR instruction (or CATALOG 4 on the HP-41CX) was executed since then. When you run the extended memory directory on an HP-41C or CV, the last file displayed becomes the working file. This is true regardless of whether the directory ran to completion or was interrupted. On an HP-41CX, the working file changes only if you do not let the directory run to completion.

The "working" file in extended memory is analogous to the "working", or current, program in main memory. The current program in main memory is the program which appears when you switch into PRGM mode. All program-related operations which do not specify an ALPHA label name operate on the current program. Instructions like GTO .ØØ1, LIST 999, DEL ØØ5, and CLP ALPHA ALPHA all operate on the current program. In main memory, the current program is selected one of two ways. It is normally the program last accessed by a GTO "label" or XEQ "label" instruction. However, if you subsequently execute Catalog 1, the program at which Catalog 1 stops becomes the current program. Thus by carefully choosing a point at which to halt Catalog 1, you can select a current program without having to spell out a GTO "label" instruction (if indeed the program you want contains an ALPHA label).

Just like Catalog 1, EMDIR can be prematurely halted (by pressing R/S) in order to select a working file. The other ways to establish a working file are to create a new file or to execute any instruction that refers to an existing file by name.

Now suppose that you have just executed PURFL, but you have not yet established a new working file. If you now execute a function that operates on the working file, for example FLSIZE with the ALPHA register empty, you will get the message FL NOT FOUND. This is certainly reasonable, since there was no working file to operate on. However, with a revision 1B Extended Function/Memory module, if you then execute an EMDIR, you will see the DIR EMPTY message. Your entire extended memory directory is gone!

There are several ways to alleviate the problem with PURFL if you have a revision 1B Extended Functions/Memory module. Your first line of defense is to make a habit of executing EMDIR or otherwise establishing a new working file immediately after using PURFL. This includes any uses of PURFL in your programs. Your second defense is to ask yourself "Have I defined the correct working file?" before any instruction that operates on the working file. Where an instruction that can operate on either the named file or the working file is to be used in a program, you can precede it by the steps ALENG (alpha length -- see page 63) and 1/X to make sure that the ALPHA register is not empty. Lastly, a short program called "PFF" (Purge File Fix) in Section 1ØE permits the damage to be repaired after the fact. This program contains some synthetic instructions which cannot be keyed in by normal means, but barcode for the program is provided in Appendix D. Once again, this problem with PURFL does not occur with the HP-41CX or with revisions 1C and up of the Extended Functions/ Memory module.

One more detail about SAVEP deserves to be mentioned. If you save a new version of a program that is already saved in extended memory, the old file will automatically be purged and the new version will be added <u>at the end of the extended memory directory</u>. Knowing this ahead of time may save you a few moments of panic when you run the directory. You can retain the old program file if you choose to use a different name for the new program file (see page 14).

CHAPTER TWO
SAVING DATA IN EXTENDED MEMORY

2A. File structure

Saving data in extended memory requires a few more steps than saving a program. Rather than simply saving the data, we must first create an empty data "file" in extended memory in which to save the data. Figure 2.1 shows the structure of such a file. This structure is the same for all three types of files (program, data, and text), except that that the information within the file is organized differently. For program and text files, each register equals 7 bytes.

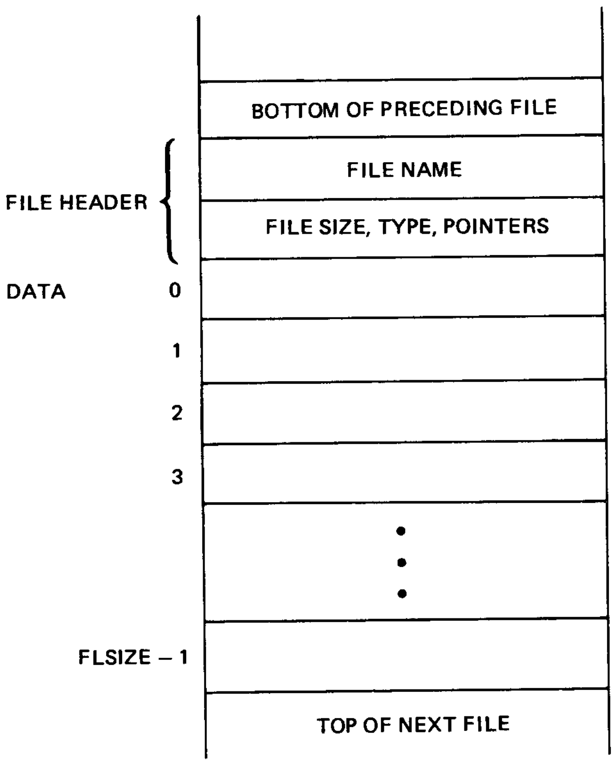


Figure 2.1. Register Usage for an Extended Memory File
Header information is available through FLSIZE, EMDIR, etc.

To make the examples in this chapter easier to follow, use the following short routine to pre-load the data registers with values that are the same as the register numbers. Data register ØØ gets the value Ø, register Ø1 gets 1, and so on, until a NONEXISTENT register is encountered.

```
Ø1 LBL "PRELOAD"
Ø2 1              First load the stack with 1's
Ø3 ENTER↑         for repeated addition.
Ø4 ENTER↑
Ø5 ENTER↑
Ø6 CLX            Start with X=Ø.
Ø7 LBL Ø1
Ø8 STO IND X      Store X in register number X.
Ø9 +              Add 1.
1Ø GTO Ø1         Go back to line Ø7.
11 END
```

2B. The SAVERX function and the "working" file

Suppose you want to save the contents of data registers 2 through 9. If you were using magnetic cards, you would key in

    2.ØØ9 XEQ "WDTAX".

To save these registers in extended memory you must first create a data file of at least 8 registers. There is no instruction analogous to SAVEP that creates the file and transfers the data in one operation.

Let's name the data file "ABC". Press

    ALPHA A B C ALPHA

to name the file, then

    8

to designate the file size, followed by

    XEQ ALPHA C R F L D ALPHA (create file -- data)

to create an empty 8-register file. The **CRFLD (create file -- data)** instruction expects the file name in the ALPHA register and the file size (the number of data registers in the file) in X. CRFLD clears the registers in a file as it is created.

Execute EMDIR and you should see

```
        ABC       D008
```

as the last entry in the directory. To save the contents of registers 2 through 9 in the "ABC" data file, press

```
        2.009    XEQ "SAVERX"
                     (press XEQ ALPHA S A V E R X ALPHA).
```

The **SAVERX (save registers designated by X)** function accepts a number in the X-register of the form **bbb.eee** . A block of data registers beginning with register number **bbb** and ending with register number **eee** is transferred to the "working" data file in extended memory. The "working" file is the last file used or created, unless an EMDIR or PURFL was executed. When you run the extended memory directory, the working file becomes the file at which you stopped the directory. On the HP-41CX, if you let the directory run to completion, the working file is left unchanged. On the HP-41C or CV, letting the directory run to completion selects the last file in the directory as the working file.

2C. The GETRX function and the register pointer

If you have been thinking ahead, you might suspect that the sequence 12.019, XEQ "GETRX" would retrieve the 8 numbers and place them in data registers 12 through 19. As logical as this may seem, it is not the case. If you try this sequence, you will get the error message END OF FL. Some explanation is in order.

A data file in extended memory can be very large. A single such data file can be used to save several blocks of data. Furthermore, you need not retrieve the entire file at once. Small blocks of data or even single registers can be retrieved from a data file. The price for this flexibility is that a pointer is required to specify where in the data file you wish to store or retrieve data. Without a pointer it would be impossible to guarantee that GETRX would retrieve the right block of data.

But if a pointer is necessary, why didn't we have to set it up before doing the SAVERX operation? Normally it is necessary to set up the pointer, but in that case, the "ABC" file had just been created. A newly created file has a pointer that is automatically initialized to zero, meaning that any SAVE or GET operations are performed beginning at register Ø, the first register of the file. [Registers in an extended memory data file are numbered starting with Ø, just as the data registers in main memory are numbered.] Therefore the sequence 2.ØØ9, SAVERX stored the contents of data registers 2 through 9 into the first 8 registers (and only 8 registers) of the "ABC" data file. This sequence had the additional unobserved effect of advancing the pointer from Ø (the first register) to 8 (one past the last register in the file). Any further operations such as GETRX will give the END OF FL error message until the pointer is re-set.

To set the pointer, we use the SEEKPTA function. The **SEEKPTA (seek pointer for the file named in ALPHA)** function simply sets the pointer to the value specified in X. The file name should also be specified in the ALPHA register. If the ALPHA register is clear, SEEKPTA will operate on the working file. Each data file has its own pointer, stored in one of the file's two header registers. A SEEKPTA on one file will not affect the pointer for another file.

For example, suppose we want to retrieve the former contents of data registers 4 through 7 from the data file "ABC" and place these four values in data registers 11 through 14. Figure 2.2 illustrates this operation. Note in Figure 2.2 that the former contents of data register 4 reside in the third register of the file "ABC". Therefore we press

ALPHA A B C ALPHA 2 XEQ ALPHA S E E K P T A ALPHA

to set the pointer value to 2, the third register of the file. Once this is done we simply press

11.Ø14 XEQ ALPHA G E T R X ALPHA

to retrieve the data. Use RCL to check that registers 11, 12, 13, and 14 contain the same values as registers 4, 5, 6 and 7.

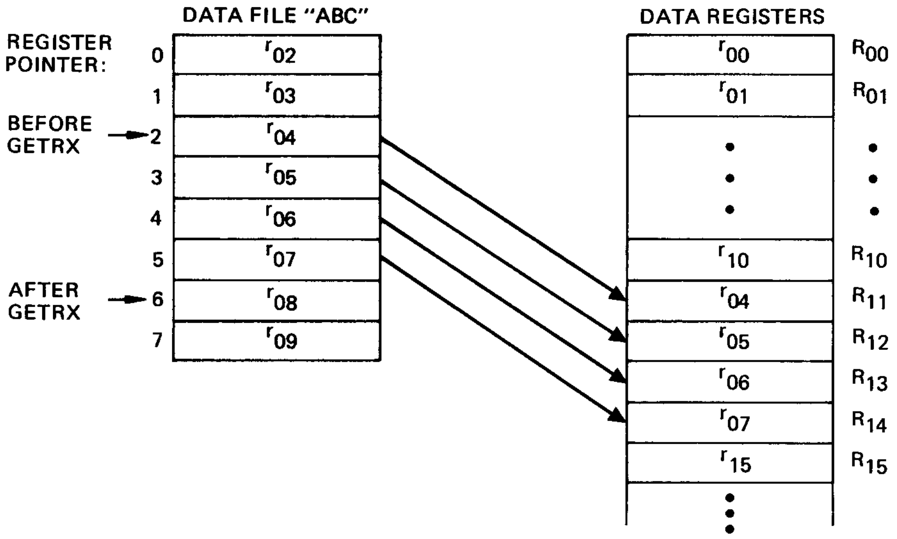


Figure 2.2. Effect of the Sequence: "ABC" 2 SEEKPTA 11.014 GETRX.

The **GETRX (get registers designated by X)** function accepts a number in the X-register of the form bbb.eee, designating the data register block in which the retrieved data is to be placed. GETRX retrieves the designated number of data registers from the working extended memory file, starting at the current register of the file. GETRX also advances the register pointer to the first register beyond the block that was retrieved from extended memory.

SAVERX, like GETRX, advances the register pointer to the first register beyond the block that was written into extended memory. In fact, this automatic advancing of the register pointer is common to all data file SAVE and GET functions.

The **RCLPTA (recall pointer for file named in ALPHA)** function provides an easy way to check the current value of any file's pointer, in case you do not remember it. Just put the file name in ALPHA and execute RCLPTA, and the pointer value will be recalled to X. If ALPHA is clear, RCLPTA will operate on the working file. As for any RCL operation, the stack will be lifted unless the RCL was immediately preceded by an ENTER↑, CLX, or similar stack lift-disabling operation.

As an example of RCLPTA, let's now check the pointer. Since you just recalled registers 4 through 6 of the "ABC" file, if you execute

```
        RCLPTA
```

the result should be 7.

Tip: RCLPTA is a convenient way to select a file to be the working file without altering the contents of the file.

2D. The SEEKPT and RCLPT functions

The **SEEKPT (seek pointer)** function operates identically to SEEKPTA, except that the operation of setting the pointer is performed on the "working" file, rather than the file named in ALPHA. If you are sure which file is the "working" file, SEEKPT saves a few steps. Otherwise use SEEKPTA.

The same advice applies to using SEEKPT in a program. If a preceding step of the program established the correct "working" file, it is OK to use SEEKPT. If not, use SEEKPTA.

The **RCLPT (recall pointer)** function is a version of RCLPTA that operates on the working file. Use it instead of RCLPTA when you are sure which file is the working file.

2E. More data file functions: SAVEX, GETX, SAVER, GETR, CLFL

The **SAVEX (save X register)** function transfers the contents of the X register to the working file, which must be a data file, in extended memory. The current pointer value designates which register of the data file is used. After the value is saved, the pointer value is increased by 1. For example, to store the value 15 in the third register (register number 2) of the data file, press

```
2  XEQ "SEEKPT"
15  XEQ "SAVEX"   .
```

The pointer value is now 2+1 = 3, so that a second SAVEX instruction would store X in register 3 of the file. This automatic incrementing of the register pointer with SAVEX is extremely useful. You can write a program that computes one result each time through a loop, with a single SAVEX instruction to store the result:

```
"filename"
CLFL  or  CRFLD   (CLFL will be covered on page 31)
LBL Ø1
(insert steps here to compute result)
SAVEX
GTO Ø1      .
```

There is no need to mess around with register pointers or ISG counters for storage. If you need a counter for the computation, you may be able to use RCLPT as a built-in counter. If you like, you can even let the END OF FL error condition terminate the computations. This makes for a very simple program structure.

The **GETX (get current register, transfer to X)** function is the inverse operation of SAVEX. The current register of the working file is retrieved and brought into X. The register pointer is increased by 1, and the stack is lifted to accommodate the retrieved data, just as for RCL.

For example, to retrieve the number 15 from register 2 of the "ABC" data file (which should still be the working file if you have been following along with the examples), press

```
2
SEEKPT
GETX     .
```

The result should be 15 in the X register. The register pointer is changed from 2 to 3, as executing RCLPT will reveal. Just as for SAVERX, this automatic pointer incrementing makes it convenient to use GETX inside a loop.

The **SAVER (save registers)** function transfers all the data registers to the file named in ALPHA, or to the working file if ALPHA is empty. Unlike SAVEX and SAVERX, SAVER totally ignores the register pointer. Data register ØØ goes into register Ø of the file, data register Ø1 goes into file register 1, and so on. Unfortunately, although SAVER does not use the pointer, it does change it! The pointer is left just beyond the last register written into the file.

If the extended memory file is not large enough to hold all the data registers, SAVER displays an END OF FL error meesage and refuses to transfer even a single register. This feature, which cannot be overridden by flag 25, limits the usefulness of SAVER.

Unless the current SIZE precisely matches the amount of data that you have to save (and does not exceed the FLSIZE of the selected data file), you should consider using SAVERX rather than SAVER to avoid wasting space in extended memory. Of course, you can always reduce the SIZE to match the number of data registers you want to save. For example, to save data register ØØ through 23, use the sequence

```
24
PSIZE
"file name"
SAVER        .
```

The PSIZE (programmable SIZE -- see page 75) function reduces the SIZE to 24, throwing out the data beyond register 23. The rest of the data is then saved by SAVER. However, this technique really isn't much easier than

```
"file name"
Ø
SEEKPTA
.Ø23
SAVERX
```
but it may be preferred for some applications.

The **GETR (get registers)** function is far more useful than SAVER. GETR retrieves data beginning with register Ø of the named file (the working file if ALPHA is empty), and places it in data registers ØØ and up. Unlike SAVER, GETR will <u>not</u> give an END OF FL error message if the file is smaller than the current SIZE. GETR will stop either when the file runs out of registers <u>or</u> when the current SIZE is used up. This means that recalling an entire data file is as simple as

```
"file name"
GETR    ,
```
perhaps followed by a REGMOVE or a REGSWAP instruction (see pages 76 to 79) to move the data to a different block of registers if you didn't want it to start at register ØØ.

Like SAVER, GETR ignores the pointer but sets it to the END OF FL position, 1 register beyond the last register retrieved.

The **CLFL (clear file)** function clears the contents of a data file; that is, it sets all the registers to zero. The register pointer is also set to zero so that you can immediately begin using SAVE instructions to store data in the file. Just put the file name in ALPHA and execute CLFL. A typical application for CLFL is initialization before re-using an existing data file. Since data files are cleared when they are created, you do not need to use CLFL on a newly-created data file.

If a file name is not present in ALPHA when you execute CLFL, a NAME ERR message will appear. CLFL will not operate on the working file. However, like the other file-handling functions, CLFL does make the named file the working file. If you attempt to use CLFL to clear a program file, a FL TYPE ERR will result. Program files can only be replaced with a new program (using SAVEP) or purged entirely.

The **PURFL (purge file)** function eliminates the named file from extended memory, freeing its registers for other uses. Like CLFL, it must have a valid file name in ALPHA. See page 19 for an important **WARNING** about PURFL.

On the HP-41CX, the **RESZFL (resize file)** function changes the size of an existing data or text file. RESZFL operates only on the working file. Use RCLPTA with the desired file name in ALPHA, or use any other means to select the desired file as the working file. Then put the new FLSIZE in X and execute RESZFL. If you decrease the file size, the highest numbered registers will be eliminated. If there is nonzero data in these registers, the calculator will give a FL SIZE ERR message. You can override this protective feature by specifying the negative of the desired FLSIZE in X.

As you have seen, extended memory is much more flexible in storing data than the card reader. Extended memory allows easy access to individual registers of data, and to sub-blocks of registers within a block of saved data. This provides a convenient method to analyze large data bases without tying up all your data registers. You can pull out the numbers as needed, in blocks or one by one.

The full power of extended memory data files will be illustrated in Chapter 6 with the application programs "SOLVE", "DERIV", and "INTEG". These programs use extended memory to guard their data while a user-supplied program is called to evaluate a function f(x).

CHAPTER THREE
TEXT FILES IN EXTENDED MEMORY

3A. What is a text file?

Twelve of the 47 functions provided in the Extended Functions/Memory module and 14 of the 61 HP-41CX extended functions deal exclusively with text files. This chapter explains how ASCII files are used and how they provide powerful new string handling capability. If none of your applications involve long ALPHA strings, you may wish to skip this chapter for now.

Prior to the advent of extended memory, dealing with character strings on the HP-41 was cumbersome. Strings had to be broken up into segments of 6 characters or less, because the ASTO operation cannot fit more characters into a register.

Extended memory offers a new way to deal with strings that does not require that a string be broken into register-sized pieces. Instead, the strings are stored unbroken in an extended memory text, or ASCII, file. [The terms "text file" and "ASCII file" are used interchangeably in this book, as they are in HP's documentation.] Each string, or record, can be up to 254 characters long. The number of different strings that you can have in a single text file is limited only by the size of extended memory. As for a data file, you must specify the number of registers to be allocated when you create an ASCII file. This number must be at least:

$$N_{registers} = \text{INT}[(N_{records}+N_{characters}+7)/7],$$

where $N_{records}$ is the maximum number of records you will need, and $N_{characters}$ is the maximum number of characters that will be stored. The +7 accounts for one end-of-file byte (see section 10C) and rounding up. For example if you wish to store 2Ø names of at most 25 characters each, you will need

$$N_{registers} = \text{INT}[(2Ø+2Ø*25+7)/7]=\text{INT}(75.29)= 75 \text{ regs.}$$

It is a good idea to use a somewhat larger number than needed when creating an ASCII file in case your storage needs grow. The Extended Functions Module Owner's Manual suggests adding 2Ø% to your best estimate of the number of characters to be stored, and dividing the result by 7. If you have an HP-41CX, you need not be as cautious, because the RESZFL (resize file) function makes it easy to increase the file size later. Two programs presented in section G of this chapter give a similar file resizing capability to the HP-41C and CV.

Just as extended memory data files have a pointer to the current register, text files have a pointer to the current record. In addition, text files have a pointer to the current character position within the record. The record pointer **rrr** and the character pointer **ccc** are combined into a single decimal number **rrr.ccc** for all pointer operations like SEEKPT (setting the pointer values) and RCLPT (reading the current pointer values).

To illustrate these points, let's try an example. We will need a 25-register text file called "NAMES". Make sure that there are at least 25 registers available for data in extended memory. To do this, execute EMDIR and let the directory run to completion. On the HP-41CX you can use the EMROOM function or CATALOG 4 instead of EMDIR for this purpose. The number in X is then the number of registers available for data in extended memory. Then, to create the "NAMES" file, press

25 ALPHA N A M E S ALPHA XEQ ALPHA C R F L A S ALPHA.

The usage of the **CRFLAS (create file -- ASCII)** function is very similar to the usage of CRFLD (create file -- data). You put the file name in ALPHA, the number of registers in X, and execute "CRFLAS".

After you have created the text file called "NAMES", the next step is to use the **APPREC (append record)** instruction to load some data into it. Suppose you want to store the three names:

record number	name
Ø	RICHARD NELSON
1	ROGER HILL
2	JOHN MCGECHIE .

The name-storing process is simple. Just load a name into the ALPHA register and XEQ "APPREC". The APPREC function adds a new record to the working text file by appending the entire contents of the ALPHA register to the file. The pointer is advanced to one character beyond the last character appended. The following sequence of operations loads the three names:

```
"RICHARD NELSON" XEQ "APPREC"
"ROGER HILL" XEQ "APPREC"
"JOHN MCGECHIE" XEQ "APPREC"  .
```

Each quote mark (") indicates that the ALPHA key must be pressed.

Loading ALPHA data into a text file is much easier than storing it in data registers. The APPREC function handles up to 24 characters, rather than the 6 that ASTO can handle. Just to store the name "RICHARD NELSON" in data registers requires 5 instructions: ASTO Ø1, ASHF, ASTO Ø2, ASHF, ASTO Ø3. This is clearly very cumbersome compared to a single APPREC instruction. It becomes even more cumbersome if the string is reconstructed or needs to remain unchanged in ALPHA (add CLA, ARCL Ø1, ARCL Ø2, ARCL Ø3).

Ease of loading is by no means the only advantage of using extended memory text files to hold ALPHA data. The real power of text files lies in their data access, insertion, and deletion capabilities.

3B. Accessing data in text files

There are two functions that recall data from a text file. These are **ARCLREC (alpha recall record)** and GETREC (get record). As its name implies, ARCLREC recalls characters from the working text file, starting at the current pointer location, until the ALPHA register is full or the end of the record is reached. The character pointer is advanced to one position beyond the last character recalled. The ARCLREC function sets or clears flag 17 (the "record incomplete" flag) to indicate whether or not more data remains in the record. ARCLREC works like ARCL, in that it appends the recalled characters to any existing characters in the ALPHA register.

The **GETREC (get record)** function is precisely equivalent to the sequence CLA, ARCLREC.

As an example, suppose you want to review the data in the "NAMES" file, which should still be the working file since you just created it. The sequence

```
Ø  SEEKPT
GETREC  AVIEW
```

shows you the contents of the first record, "RICHARD NELSON". If you then try the sequence

```
ARCLREC  AVIEW
```

the result will be "RICHARD NELSONROGER HILL". Whoops! We forgot to do a CLA before the ARCLREC. Most of the time, you will find that GETREC is handier to use than ARCLREC, because GETREC automatically clears the ALPHA register before recalling the record. The ARCLREC function will be useful for those special instances in which the contents of a record are to be attached to a message.

In the preceding example, the ARCLREC function was able to fit the entire record being recalled into the ALPHA register, so flag 17 was cleared. If the record had not fit into the ALPHA register, ARCLREC would have set flag 17 to indicate that more data remained in the record. When you write a program that prints strings from text files, you will use sequences that test flag 17. For example:

```
   (record number)
   SEEKPT         Set pointer to the beginning of the record
   LBL Ø1
   GETREC         Recall 24 characters of the record
   ACA (or OUTA)  Send chars. to printer, but do not print yet
   FS? 17         If record is incomplete, get 24 more chars.
   GTO Ø1
   PRBUF          Otherwise print the accumulated string.
```

Some HP-IL peripherals automatically make use of the status of flag 17 after ARCLREC or GETREC. If flag 17 is set, the normal carriage return/line feed is suppressed so that the rest of the record may be included on the same line.

3C. Insertion of data into text files

Suppose you want to change the first name record from "RICHARD NELSON" to "RICHARD NELSON, FOUNDER OF PPC". The first thing you must do is to set the record pointer to zero, which is the first record of the file. If you have done nothing to designate a different working file, the "NAMES" file is still the working file. The sequence

```
Ø  SEEKPT
```

will therefore set the pointers to character Ø (the first character) of record zero (the first record).

The **APPCHR (Append characters)** instruction appends the contents of the ALPHA register to end of the current record, ignoring the character pointer. The pointer is advanced to the end of the current record, one position beyond the character appended. Unlike APPREC, APPCHR does not create a new record. To make the desired change to record Ø, press

```
", FOUNDER OF PPC"  XEQ "APPCHR"  .
```

You can use the sequence

```
Ø  SEEKPT
GETREC  AVIEW
GETREC  AVIEW
```

to check your results. This is much easier than using ARCL, APPEND, and ASTO to modify a string stored in data registers.

The next example of insertion uses the **INSCHR (insert character)** instruction. The goal is to change the first record from "RICHARD NELSON, FOUNDER OF PPC" to "RICHARD J. NELSON, FOUNDER OF PPC". This requires inserting the characters "J. " ahead of the "N" in "NELSON".

Before the INSCHR instruction can be successfully used, you must tell the HP-41 exactly where to insert the characters. In this example, that means that the record pointer must be Ø (the first record) and the character pointer must be 8, corresponding to the 9th character, "N". INSCHR always inserts the contents of ALPHA ahead of the current pointer location and advances the character pointer by the number of characters inserted. As with the other data insertion functions, the pointer ends up one position past the last character inserted. The sequence

```
.ØØ8   SEEKPT
"J. "   INSCHR   (don't forget the space)
```

performs the insertion of the middle initial "J." . Use

```
Ø   SEEKPT
GETREC  AVIEW
GETREC  AVIEW
```

to check your results. The sequence that would be required to do this insertion using ARCL and ASTO instructions defies description!

The final example of text file insertion is the addition of a new record in the middle of an existing file. The function **INSREC (insert record)** is provided for this purpose. Analogously to INSCHR, INSREC inserts a new record ahead of the current record pointer, loading it with the contents of ALPHA. INSREC also advances the pointer to the end of the new record, one position beyond the last character.

As an example of the INSREC function, try inserting the name "CLIFFORD STERN" between "ROGER HILL" and "JOHN MCGECHIE". Since the insertion is to be made ahead of the third record (record number 2), the sequence is:

```
        2  SEEKPT
        "CLIFFORD STERN"  XEQ "INSREC"  .
```

The POSFL (position in file) instruction, described on page 41, makes it easy to insert characters or records in the right place relative to any selected string of characters in a file. First you use POSFL to find the string before which the insertion is to be made, then you use INSCHR or INSREC as needed.

3D. Deletion of data from text files

Continuing the previous example, we have:

record number	name
Ø	RICHARD J. NELSON, FOUNDER OF PPC
1	ROGER HILL
2	CLIFFORD STERN
3	JOHN MCGECHIE

Suppose you want to delete the last record of the file, record number 3. The function needed for this operation is DELREC.

The **DELREC (delete record)** function deletes the current record (as designated by the record pointer) from the working text file. DELREC does not change the record pointer, but it does zero the character pointer. To delete record number 3, the sequence is

```
        3  SEEKPT
        DELREC    .
```

To check the result, use GETREC (the record pointer is still 3). You should get an END OF FL error message, indicating that record 3 no longer exists. If you had deleted record number 1, records 2 and 3 would have moved up to become the new records 1 and 2, respectively. Incidentally, both DELREC and INSREC deal with only a single record. If you need to insert or delete several records at one point in a file, you may need a short looping sequence containing DELREC or INSREC.

Now suppose you want to delete the string ", FOUNDER OF PPC" from record Ø. The **DELCHR (delete character)** function deletes characters starting from the current pointer position. The number of characters to be deleted is specified by the integer part of the number in the X register. If this number is larger than the number of characters from the current pointer position to the end of the record, the deletion is only performed up to the end of the record. The record and character pointers and X register are left unchanged by DELREC. For this example, the sequence

```
Ø.Ø17   SEEKPT
16  DELCHR
```

performs the deletion of ", FOUNDER OF PPC". The comma was the 18th character (character number 17) of record Ø. The number of characters to be deleted was 16. Actually, since you were deleting all the remaining characters of record Ø, you did not have to count the number of characters to be deleted. The number 99 would have served as well as 16; you only needed a number at least as big as the number of characters remaining in record Ø.

To clear the entire contents of a text file without deleting the file itself, use the **CLFL (clear file)** instruction, with the file name in the ALPHA register. CLFL needs a file name, and will not operate on the working file. The named file becomes the working file, and the number of records is set to zero. This is useful to initialize an existing text file for re-use as if it were a new file. The CLFL instruction is the same one that clears data files.

To delete the text file itself and free its extended memory registers for other uses, put the file name in ALPHA (this is <u>not</u> optional) and execute **PURFL (purge file).** For more details on PURFL, including an important **WARNING,** see page 19.

3E. Miscellaneous text file operations

POSFL, SAVEAS, GETAS

The **POSFL (position in file)** function searches the working text file, starting at the current pointer position, for a string that exactly matches the contents of the ALPHA register. This string is not allowed to span more than one record; it must be fully contained in a single record. If the search is successful, the pointer is moved to the first character of the string and the new pointer value is placed in the X register. If the search is not successful, no error message is displayed, but the number -1 is placed in X. Thus, if you are using the POSFL function in a program, a simple X<Ø? instruction will tell you if the string was not found.

POSFL can work in combination with DELCHR or DELREC to delete strings or records, or in combination with INSCHR or INSREC to insert new strings or records.

The stack usage of POSFL is quite unusual. On the HP-41CX, the stack is raised and LASTX is not disturbed. This is just as it would be if RCLPT were executed at the point where the match was found. On the HP-41C or CV, POSFL works this way only if the string is found. If you are using an HP-41C or CV and the string is not found, POSFL overwrites the X register with the number -1 and places the former contents of X in LASTX.

Let's try an example. Suppose you want to locate the last name "HILL" in the "NAMES" file. This is easy to do. Just press

```
Ø SEEKPT
ALPHA space H I L L ALPHA
POSFL
```

The result should be 1.ØØ5, indicating that the space character before "HILL" is character number 5 of record 1. The space character was used to ensure that "HILL" was not found as a first name or as a string embedded within another name.

The **SAVEAS (save ASCII file)** and **GETAS (get ASCII file)** functions are usable only if you have an HP-IL mass storage device, such as the HP 82161A Digital Cassette Drive. These functions are described in the Extended Functions/Memory Module Owner's Manual.

If you plan to make heavy use of ASCII files, an HP-IL mass memory device will be very useful. Through SAVEAS and GETAS, it provides a convenient way to permanently save your ASCII files. If you need to merge the contents of two ASCII files, which SAVEAS and GETAS are not meant to do, you can use the programs presented in Section 3G.

3F. Viewing the contents of an ASCII file

The program "VAS" (view ASCII file) presented here will display the entire contents of a text file, one record at a time. It uses a few of the extended functions that are explained in Chapter 4, so you will have to read that chapter before you try to understand how "VAS" works.

To view a text file, put the file name in the ALPHA register and execute "VAS". If a printer is attached, turned on, and enabled (flag 21 set), the ASCII file contents will be printed out. Otherwise they will be displayed. The LBL 10 subroutine performs this "print or display" operation. It is an excellent application for the RCLFLAG and STOFLAG extended functions (see page 66).

Here is a typical printout from "VAS":

```
RECORD 0:
THIS EXAMPLE ILLUSTRATES
 THE PRINTOUT/DISPLAY PR
ODUCED BY VAS.
RECORD 1:
SHORT RECORDS USE 1 LINE
RECORD 2:
LONGER RECORDS SPILL OVE
R INTO TWO OR MORE LINES
RECORD 3:
END OF FL
```

If you want to list only a part of the file, put a record counter in X in the ISG format (**bbb.eee**, where bbb is the starting record and eee is the last record to be viewed). Put the file name in ALPHA and execute "PVAS" (partial view ASCII).

One error trap is included in "VAS" and "PVAS". If you get the DATA ERROR message at line Ø5, you should load a file name into ALPHA and press BST and R/S. This error trap is intended to prevent you from losing your extended memory directory if you have a revision 1B Extended Functions/Memory module and you just used PURFL. With ALPHA clear, the SEEKPTA instruction would operate on the working file, causing disaster (see page 19) if there were no working file.

"VAS"/"PVAS" program listing

```
01♦LBL "VAS"
 02 .9

03♦LBL "PVAS"
 04 ALENG
 05 1/X
 06 RDN
 07 INT
 08 SEEKPTA

09♦LBL 01
10 "RECORD "
 11 LASTX
 12 INT
 13 RCLFLAG
 14 CF 29
 15 FIX 0
 16 ARCL Y
 17 STOFLAG
 18 "⊢:"
 19 XEQ 10

20♦LBL 02
 21 GETREC
 22 XEQ 10
 23 FS? 17
 24 GTO 02
 25 ISG L
 26 GTO 01
 27 RTN

28♦LBL 10
 29 SF 25
 30 PRA
 31 RCLFLAG
 32 FS?C 21
 33 FC?C 25
 34 AVIEW
 35 STOFLAG
 36 RDN
 37 FS?C 25
 38 FC? 21
 39 PSE
 40 END

         89 BYTES
```

Line-by-line analysis of "VAS"/"PVAS"

Line Ø2 provides a default record counter of Ø.9ØØ for "VAS", so that all records will be displayed. Lines Ø4 and Ø5 constitute the error trap that detects an empty ALPHA register (length of the string in ALPHA = Ø). You may delete these two lines if you have an HP-41CX or a revision 1C and up Extended Functions/Memory module. Lines Ø7-Ø8 set the pointer to the beginning of the first record to be viewed. The LBL Ø1 sequence forms the message "RECORD n:" in the ALPHA register. Then XEQ 1Ø (line 19) displays or prints the string.

The LBL Ø2 sequence uses GETREC to recall 24 characters. Then an XEQ 1Ø prints or displays the string. If flag 17 is set, indicating an incomplete record, another GETREC is done. Otherwise the record counter is incremented (line 25). The GTO Ø1 instruction causes the same process to be performed to print the next record. When the counter reaches its limit, the GTO Ø1 is skipped and the RTN is executed instead. When "VAS" is used, termination will be caused by an END OF FL error stop at line 21. This is normal.

3G. Saving text files on magnetic cards

If you have a card reader, the program "WAS" (write ASCII file) presented here can be used to write a text file into the data registers, from which a WDTAX (write data registers designated by X) instruction transfers the information to magnetic cards. The "RAS" (read ASCII file) program performs the reverse operation. These programs have only one constraint: the text file should not contain any null characters (decimal code Ø -- see page 62). This is not a serious constraint since null characters are not ordinarily used.

To use "WAS", simply put the file name in ALPHA and XEQ "WAS". The file name must be provided to avoid an error stop at line Ø3. This error trap is intended to prevent the FLSIZE instruction from wiping out your extended memory directory if you just used PURFL. If you get the DATA ERROR message, you should load a file name into ALPHA and press BST and R/S. The "WAS" program will make sure that the SIZE is sufficient to hold all the data, using the PSIZE (programmable SIZE) extended function to increase the SIZE if necessary. The PSIZE function will be explained on page 75. If you get a NO ROOM error stop at line 2Ø, you will have to either delete some programs to make more space or use the "PWAS" (partial write ASCII) program described below. The number in X at this error stop indicates the required SIZE for this "WAS" operation.

When the card reader prompt RDY Ø1 OF nn appears, you may either insert the card to be recorded or you may press R/S twice to avoid recording a card. When "WAS" is finished, a number of the form Ø.nnn is in the X register. This number indicates that a representation of the text file data resides in data registers ØØ through nnn. The total number of data registers used is nnn+1, while the number of tracks used to record the data is 1+INT(nnn/16).

To use "RAS", put the file name in ALPHA (not optional) and XEQ "RAS". Supply data cards at the prompt or press R/S twice if the file representation already resides in the data registers. The "RAS" program automatically knows where the data ends. You don't need to specify a number of registers.

The "WAS" and "RAS" programs are very helpful in dealing with a problem commonly encountered with ASCII files. Suppose you have created an ASCII file of 5Ø registers and have started to load your data into it. All of a sudden, you get the END OF FL error message. The file is full! It would appear that you have to purge this file, create a new, larger one, and start re-entering data from the beginning. But wait! You can use "WAS" to write the file contents into data registers before you purge the file. Then, after you create the larger file, you can use "RAS" to re-load the data into the new file. This saves a lot of work.

On the HP-41CX, the **RESZFL (resize file)** function can be used for this purpose instead of "WAS" and "RAS". First select the file you want to resize as the working file. You can do this by interrupting the extended memory directory or by naming the file and executing FLSIZE or RCLPTA. Then put the desired file size in X and execute RESZFL. RESZFL allows you to increase or decrease the size of the working file, as long as no records would be lost by the file size reduction. **Caution:** Even if you put a different file name in ALPHA, RESZFL will still resize the working file. You should run Catalog 4 (EMDIR) after using RESZFL to check the result.

If you want to record only part of a text file on cards, put a record counter of the form bbb.eee in X, the file name in ALPHA, and execute "PWAS" (partial write ASCII). Records starting with bbb and ending with eee will be copied into the data registers, and onto magnetic cards if you so choose. This is helpful when insufficient SIZE is available for "WAS", or when you are merging parts of two different text files.

If you want to replace part of the data in a text file with data from cards, you can use the "PRAS" (partial read ASCII) program. Put a record control number in X, the file name in ALPHA, and execute "PRAS". The records from bbb to eee will be deleted and replaced by the data from cards or from the data registers. If you want to append records to the end of a file, use a record control number xx.9, where xx is greater than the number of records in the file , and 900-xx is greater than the number of records to be added. To find out the number of records in the file, see problem 3.1 on page 53.

Line-by-line analysis of "WAS"/"PWAS"/"RAS"/"PRAS"

Lines 02 and 03 make sure that the ALPHA register is not empty, so that the FLSIZE function at line 05 will not operate on the "working" file (which might not exist). See page 63 for an explanation of the ALENG function. If you want to be able to use "WAS" and "RAS" with the "working" file, you may delete lines 02, 03, 95, and 96. If you have a revision 1B Extended Functions/Memory module, see page 19 for an explanation of the risk you take by deleting these error traps.

Lines 05 through 18 calculate the number of data registers needed to store the representation of the text file. This representation is best shown by an example. Suppose you have the text file:

record number	name
0	RICHARD J. NELSON
1	ROGER HILL
2	CLIFFORD STERN

"WAS"/"PWAS"/"RAS"/"PRAS" program listing

```
01♦LBL "WAS"
 02 ALENG
 03 1/X
 04 SIZE?
 05 FLSIZE
 06 7
 07 *
 08 APPREC
 09 DELREC
 10 RCLPT
 11 5
 12 *
 13 4
 14 +
 15 +
 16 6
 17 /
 18 INT
 19 X>Y?
 20 PSIZE
 21 0.9
 22 ENTER↑
 23 GTO 00

24♦LBL "PWAS"
 25 ALENG
 26 1/X
 27 X<>Y
 28 SIZE?
 29 2
 30 -
 31 1 E3
 32 /
 33 X<>Y

 34♦LBL 00
 35 ENTER↑
 36 INT
 37 SEEKPTA
 38 SF 25
 39 DSE L

 40♦LBL 01
 41 GETREC
 42 FC? 17
 43 ISG L
 44 FC? 25
 45 GTO 03

 46♦LBL 02
 47 R↑
 48 R↑
49 ASTO IND X
 50 ISG X
 51 X<0?
 52 GTO 04
 53 ALENG
 54 11
 55 ASHF
 56 X<=Y?
 57 GTO 02
 58 RDN
 59 5
 60 X<Y?
 61 FS? 17
 62 GTO 01
 63 GTO 02

 64♦LBL 03
 65 LASTX
 66 R↑
 67 CLA
68 ASTO IND X
 69 INT
 70 1 E3
 71 /
 72 SF 25
 73 WDTAX
 74 CF 25
 75 RTN

 76♦LBL 04
 77 6
 78 R↑

 79♦LBL 05
 80 DSE Z

 81♦LBL 05
 82 CLA
83 ARCL IND Z
 84 RDN
 85 ALENG
 86 X=Y?
 87 GTO 05
 88 DSE L

 89♦LBL 04
 90 ISG Z
 91 ACOS
 92 GTO 03

 93♦LBL "RAS"
 94 CF 05
 95 ALENG
 96 1/X
 97 CLFL
 98 .9
 99 SIGN
100 GTO 08

101♦LBL "PRAS"
102 CF 05
103 ALENG
104 1/X
105 RDN
106 ENTER↑
107 INT
108 SF 25
109 SEEKPTA
110 FC? 25
111 GTO 07

112♦LBL 06
113 DELREC
114 FC? 25
115 GTO 07
116 ISG Y
117 GTO 06
118 CF 25

119♦LBL 07
120 APPREC
121 DELREC
122 RCLPT
123 X<>Y
124 SF 25
125 SEEKPT
126 CF 25
127 X<Y?
128 SF 05

129♦LBL 08
130 SF 25
131 RDTA
132 CF 25
133 CLX
134 SF 06
135 LASTX
136 6

137♦LBL 09
138 CLA
139 ARCL IND Z
140 ALENG
141 X=0?
142 FC? 06
143 X<0?
144 RTN
145 FC? 06
146 APPCHR
147 FC?C 06
148 CLA
149 FS? 05
150 INSREC
151 FC? 05
152 APPREC
153 X≠Y?
154 SF 06
155 X≠Y?
156 FC? 05
157 GTO 10
158 RDN
159 RCLPT
160 INT
161 ISG X

162♦LBL 09
163 SEEKPT

164♦LBL 10
165 RDN
166 ISG Z
167 ACOS
168 FS? 06
169 ISG Y
170 GTO 09
171 END

    291 BYTES
```

The representation of this file generated by "WAS" would be:

data register	contents
ØØ	"RICHAR"
Ø1	"D J. N"
Ø2	"ELSON"
Ø3	"ROGER "
Ø4	"HILL"
Ø5	"CLIFFO"
Ø6	"RD STE"
Ø7	"RN"
Ø8	"" (empty string)

The end of each record is marked by a string of less than 6 characters. This end-of-record marker will be an empty string if the number of characters in the record is a multiple of 6. An empty string at the beginning of a new record (register Ø8 in this example) signifies the end of the file.

This particular representation of the text file represents a good compromise between speed and register utilization. Further packing of the data is practical only by using synthetic programming techniques (see Section 1ØI).

The number of data registers needed to represent a text file depends on the file size N (in registers) and on the number of records R in the file. The program needs to compute an upper bound on the number of data registers needed, so that it can adjust the SIZE to be large enough. The worst possible case is when the first R-1 records are 6 characters each, meaning that they each take 2 data registers, and the last record uses all the remaining space in the file. In this case, a lengthy computation shows that the total number of data registers needed to store the text file data cannot exceed:

$$\begin{aligned} D &= 2(R-1)+1+\text{INT}(\ (7(N-R)+10)/6\ ) \\ &= \text{INT}\ (\ (7N+5R+4)/6\ )\ . \end{aligned}$$

In "WAS", line Ø5 computes the file size N, while lines Ø8-1Ø compute the number R of records in the file. This latter computation requires that there be up to 8 spare character positions in the text file, so that a record consisting of the file name can be temporarily appended without running into the END OF FL.

Lines 19 and 2Ø compare the maximum number of data registers required with the current SIZE, and resize if necessary. All of these registers will probably not be used, but this is part of the price for user convenience. Line 21 sets the default record counter (Ø.9) so that all records will be written.

The LBL ØØ sequence sets the pointer to the beginning of the first record designated. Flag 25 is set so that the GETREC on line 41 will not halt the program.

The LBL Ø1 loop fetches a record from the file. If the END OF FL is encountered, the GETREC clears flag 25 and causes the GTO Ø3 branch to be taken. Otherwise the LBL Ø2 loop is used to store the ALPHA register contents in 6-character pieces.

First the leftmost 6 characters are stored in register ØØ. Line 43 increments the record counter in LASTX once for each new record retrieved. Lines 53-57 return to LBL Ø2 to store another 6 characters if 12 or more characters were present in ALPHA before the ASHF at line 55 removed the 6 that were just stored. If 12 or more characters were not present, then at most one more ASTO will be needed before the next GETREC. If 5 or fewer characters were present, the record is complete and has already been stored. In this case, line 6Ø causes the GTO Ø1 to be executed.

If more than 5 characters were present, another ASTO operation will usually be needed. The only exception occurs when flag 17 is set, indicating that GETREC retrieved 24 characters but did not reach the end of the record. In this case, the last 6 of these 24 characters have just been ASTO'd and we do not want to store a blank end-of-record marker. So the flag 17 test on line 61 sends us back for more characters from the current record. If flag 17 is not set, the record is complete and we do need an end-of-record marker (which will contain Ø to 5 characters). The GTO Ø2 instruction at line 62 takes care of this case.

LBL Ø3 marks the beginning of the termination procedure that occurs after the END OF FL is reached by GETREC or after the ISG L on line 43 reaches a skip condition. The end-of-file marker (a blank string) is stored in the current data register, so that "WAS" will know where the "RAS" data ends. Then the number Ø.nnn is constructed for the use of WDTAX (write data registers designated by X).

The "RAS" program begins by checking that the ALPHA register is not empty. The named text file is cleared, which automatically sets its pointer to record Ø. The default record counter of Ø.9 is placed in LASTX by the SIGN instruction. "PRAS" starts similarly, but the file is not cleared. If the SEEKPTA on line 1Ø9 fails, the program assumes that the new records are to be appended, and no records need to be deleted. Otherwise the LBL Ø6 loop deletes the number of records requested by the record counter that was originally placed in X. Flag 25 is tested in case you specified too many records and END OF FL is encountered. Flag Ø5 is set at line 128 if the first designated record is within the file, rather than at or beyond the end of the file. This means that INSREC will be used later instead of APPREC.

At LBL Ø8, the cards are read in (if desired), and flag Ø6 is set, indicating that the next register to be read begins a new record. The value Ø in Z (line 133) is the initial register pointer, the value in Y (line 135) is the ISG record counter, and the 6 in X (line 136) is to be used for ALENG comparisons. When the length of a data register string is less than 6, the end of a record has been reached.

The LBL 09 loop first gets a string of 0 to 6 characters from the current data register. If the length is 0 and flag 06 is set, indicating that this register is supposed to begin a new record, then the RTN at line 144 terminates "RAS". Otherwise, if flag 06 is clear, the APPCHR function at line 146 adds the ALPHA contents to the current record. If flag 06 is set, APPREC or INSREC is executed, depending on flag 05, to use the ALPHA contents to start a new record. If the string length was not exactly 6 characters, then flag 06 is set to indicate that the next string recalled will start a new record. Lines 155-164 advance the pointer to the next record if flags 05 and 06 are set, so that the next INSREC will put the next record in the right place. The register counter in Z is then incremented so that the next register can be recalled.

3H. Additional text file functions on the HP-41CX

The HP-41CX includes two additional functions dedicated to text files. The first of these is **ASROOM (ASCII file room)**. ASROOM returns the number of bytes available in the named file, or the working file if ALPHA is clear. If you have a file to which you will not be adding information frequently, you can use the following sequence to minimize its usage of extended memory registers:

```
(file name)
FLSIZE       Gives the number of registers allocated
ASROOM       Gives the number of bytes free
7
/
INT          Number of registers free
-            Number of registers in use
RESZFL       Resize to minimum.
```

If you have an HP-41C or CV, you can use "WAS" and "RAS" to reduce the file size to the minimum, but you will have to use the following short routine to duplicate the ASROOM function in the preceding sequence:

```
Ø1 LBL "ASROOM"
Ø2 ALENG      These lines are an error trap for the
Ø3 1/X        PURFL bug.  You may remove them if
Ø4 RDN        you have Revision 1C or higher.
Ø5 FLSIZE     Number of registers in named file.
Ø6 7
Ø7 *
Ø8 Ø
Ø9 SEEKPTA    Go to beginning of file.
1Ø +          Total number of bytes in file.
11 SF 25      Prevent error stop at line 14.
12 LBL Ø1
13 CLA
14 GETREC
15 ALENG      Subtract the number of characters
16 -          in this record.
17 FC? 17     Subtract one byte for each record,
18 DSE X      one byte at the end of file.
19 FS? 25
2Ø GTO Ø1     Repeat if END OF FL is not reached.
21 END
```

This routine gives the true ASROOM as long as there are no null bytes in the text file.

The second HP-41CX text file function is **ED (edit).** The ED function is described fully in the HP-41CX Owner's Manual, and the description is too lengthy to repeat here. When you execute ED, the keyboard is redefined to allow easy motion through the file as well as insertion and deletion of data.

If you have an HP-41C or CV, Chapter 9 presents a text editor program called "TE" that, while slower than ED, contains all its features plus a few more. You will find "TE" or ED quite helpful for creating and modifying text files.

PROBLEMS (Solutions follow Chapter 1Ø)

3.1. Write a short sequence of instructions to determine how many records there are in a text file (assume that the file is the working file).

3.2. Write a short program to print an entire text file, one record to a line (unless the record overflows the print line). Assume that the file name is in the ALPHA register at the start of the program.

CHAPTER FOUR
MORE EXTENDED FUNCTIONS

Not all the functions built into the Extended Functions module or built into the HP-41CX extended functions are directly concerned with using extended memory. Sixteen of the 47 functions (25 of 61 for the HP-41CX) provide operating system enhancements that aid immeasurably in dealing with ALPHA strings, flags, blocks of data, and key assignments. One function, GETKEY, has the potential to allow complete customization of the keyboard under control of a program. This is demonstrated in the application programs in Chapters 7, 8, and 9.

4A. Stack Usage and Input Flexibility

There is one important difference between extended functions and normal built-in (Catalog 3) functions that is not mentioned in the Owner's Manual. Most of the extended functions that use an input from the X register just leave the input in X when they are done (X<>F, POSA, POSFL, and GETKEYX are the only exceptions). Except for POSA, POSFL, and GETKEYX they do not even copy X into LASTX. In this respect, extended functions are much more similar to indirect functions like ARCL IND X than they are to direct functions like 1/X.

This difference in stack usage is easy to deal with in your programs once you are aware of it. At worst you will need an extra roll-down instruction here and there to get rid of a used function input. At best you will be able to make use of the fact that LASTX is not disturbed by keeping a loop counter there.

Those extended functions that bring a result into X work just like RCL. The stack is raised unless a CLX, ENTER↑, or other stack lift disabling function was just executed. There are two exceptions to this rule. The first exception is POSA, which overwrites X and saves the previous value of X in LASTX. The second exception, which only applies to the HP-41C or CV, is POSFL. On the HP-41C or CV, when POSFL does not find the string, X is overwritten and the previous value of X is saved in LASTX. If the string is found, or if you are using an HP-41CX, the stack is raised and LASTX is undisturbed.

Another feature common to the extended functions is that they ignore any digits in X beyond those normally required. Often this means that the fractional part of X is ignored. For example, if you want to use STOFLAG to restore the status of flags 36-39, the number in X can be 36.39xxxxxx, where the digits xxxxxx can be nonzero. A case in which this characteristic of the extended functions can be helpful can be found on page 31. There, the sequence

```
"file name"
Ø
SEEKPTA
.Ø23
SAVERX
```

was mentioned as a way to save data registers Ø to 23 in a data file. Because data file pointers are always integers, the SEEKPTA instruction would have ignored any fractional part of the number in X. Thus you could have used the sequence

```
"file name"
.Ø23
SEEKPTA
SAVERX   ,
```

which is one step and one byte shorter. An additional unexpected benefit is that the SEEKPTA function is actually faster with .Ø23 in X than it is with Ø in X. It is not often that situations like this arise, but if you keep in mind this input flexibility of the extended functions, you will be able to write more efficient programs.

Another input flexibility feature of the extended functions is that negative numbers are usually treated as if they were positive numbers. The exceptions are AROT, to be discussed in section B of this chapter, and the HP-41CX functions RESZFL (resize file, pages 32 and 45) and GETKEYX (page 9Ø). These last two functions use the sign of X as a flag to override an error trap and to select a different mode of operation, respectively. One possible benefit of this sign-ignoring feature is that negative pointer values are treated as if they were positive. You can thus simulate a "decrement and skip if less than zero" instruction by using a negative integer with an ISG instruction. Incrementing a negative number decrements its absolute value.

4B. ALPHA manipulation

A "bare" HP-41C or CV has very limited alphabetic capability. With just a 24-character ALPHA register and a primitive set of alpha operations (append, ASTO, ARCL, ASHF, etc.), its alpha capabilities are well-suited to message displays but inadequate for much more.

Extended memory adds the ability to store text files (collections of ALPHA strings) and adds instructions for selectively changing, recalling, or finding a string. Moreover, there are six ALPHA-related functions in the set of extended functions which operate directly on the contents of the ALPHA register rather than on strings within an text file. These six functions, ALENG, ANUM, AROT, ATOX, POSA, and XTOA, add significant capability, but still do not permit extensive ALPHA processing.

If you remember that the HP-41 is not intended to be capable of word processing, you will realize that its ALPHA capabilities, especially with the addition of extended functions, are more than adequate for its 12-character display.

The **AROT (ALPHA rotate)** function rotates the contents of the ALPHA register leftward by the number of character positions specified in X. A negative number in X produces a rightward rotation. The absolute value of X must be less than 256, or a DATA ERROR message will result.

The primary use of AROT is to bring a selected character to one end of the string in ALPHA. For example, a selected character brought to the left end of ALPHA can be decoded by the ATOX function (see below). A single character that has just been appended to the right end of ALPHA can be moved from its initial position at the end of the string to a position at the front of the string by the sequence 1, CHS, AROT.

The AROT function does not drop the stack or disturb LASTX. The number of positions rotated remains in X, so you will usually have to follow AROT with a RDN instruction.

The function **XTOA (X to ALPHA)** appends a single character to the rightmost part of ALPHA. This character is designated by a decimal number from Ø to 255 in X. This decimal number is called the ASCII (American Standard Code for Information Interchange) equivalent of the character. The correspondence of display and printer representations to the decimal ASCII code is shown in the table on pages 6Ø and 61.

If ALPHA already contains 23 or 24 characters, TONE 7 is sounded. This warning tone is sounded even if XTOA is used in a program, unless flag 26 is clear. If X contains alpha data, XTOA will act like ARCL X, appending the characters to the right end of ALPHA. It is better to use ARCL X in this case because its meaning is more clear in a program listing. The stack is not dropped by XTOA, nor is LASTX updated.

The XTOA function can be used to construct ALPHA strings containing non-keyable characters such as parentheses and ampersand. For example, the following sequence creates the string "X(1)= " in the ALPHA register:

```
"X"
4Ø
XTOA
"├1"
```

```
        1          (note that these two steps make use of the
        +           fact that XTOA does not drop the stack)
        XTOA
        "⊢= "    .
```

An ARCL instruction and an AVIEW can then be used to append a number and display the message. XTOA can also be used to form strings containing lower case or special printer characters, although the printer's ACCHR function does the same thing. Except for a-e, these characters will appear as starbursts in the display. This is due to the limitations of a 14-segment display.

If the contents of the string are known ahead of time, synthetic programming techniques allow you to put a text instruction in your program that contains any such special characters. This is much more efficient than using XTOA. See page 29 of "HP-41 Synthetic Programming Made Easy". XTOA is best suited to appending one or two characters to ALPHA, where the actual character to be appended depends on the result of a computation in the program. This technique is used in the base conversion portion of the "HP-16" program in Chapter 9.

The **ATOX (ALPHA to X)** function is almost the inverse of XTOA. XTOA converts a decimal number to a character that is appended at the right end of the ALPHA register. In contrast, ATOX converts the character at the <u>left</u> end of the ALPHA register to the corresponding decimal code. The stack is raised, but LASTX is not affected.

In addition to its primary use for decoding a character from ALPHA, ATOX is often used simply to delete a character from the ALPHA register. An AROT operation can be used to move any desired character to the front of the string in ALPHA, where ATOX can remove and decode it.

(continued on page 63)

decimal code	display char	printer char	decimal code	display char	printer char
0	(null) ‾	♦	32	(space)	(space)
1	x̄	x	33	!	!
2	▓	x̄	34	"	"
3	▓	←	35	#	#
4	α	α	36	$	$
5	β	β	37	%	%
6	Γ	Γ	38	&	&
7	▓	↓	39	'	'
8	▓	Δ	40	(	(
9	▓	σ	41	)	)
10	▓	♦	42	*	*
11	▓	λ	43	+	+
12	µ	µ	44	,	,
13	∡	∡	45	-	-
14	▓	τ	46	.	.
15	▓	Φ	47	/	/
16	▓	Θ	48	0	0
17	▓	Ω	49	1	1
18	▓	δ	50	2	2
19	▓	Å	51	3	3
20	▓	å	52	4	4
21	▓	Ä	53	5	5
22	▓	ä	54	6	6
23	▓	Ö	55	7	7
24	▓	ö	56	8	8
25	▓	Ü	57	9	9
26	▓	ü	58	:	:
27	▓	Æ	59	;	;
28	▓	æ	60	<	<
29	≠	≠	61	=	=
30	▓	£	62	>	>
31	▓	▒	63	?	?

decimal code	display char	printer char	decimal code	display char	printer char
64	@	@	96	┬	┬
65	A	A	97	a	a
66	B	B	98	b	b
67	C	C	99	c	c
68	D	D	1ØØ	d	d
69	E	E	1Ø1	e	e
7Ø	F	F	1Ø2	█	f
71	G	G	1Ø3	█	g
72	H	H	1Ø4	█	h
73	I	I	1Ø5	█	i
74	J	J	1Ø6	█	j
75	K	K	1Ø7	█	k
76	L	L	1Ø8	█	l
77	M	M	1Ø9	█	m
78	N	N	11Ø	█	n
79	O	O	111	█	o
8Ø	P	P	112	█	p
81	Q	Q	113	█	q
82	R	R	114	█	r
83	S	S	115	█	s
84	T	T	116	█	t
85	U	U	117	█	u
86	V	V	118	█	v
87	W	W	119	█	w
88	X	X	12Ø	█	x
89	Y	Y	121	█	y
9Ø	Z	Z	122	█	z
91	[	[	123	█	π
92	\	\	124	█	|
93	]	]	125	█	→
94	↗	↑	126	Σ	Σ
95	_	_	127	⊦	⊦

Notes to ASCII character table

If you are using an HP-IL printer, decimal codes 9, 10, and 27 have a different meaning. Code 9 generates a "line-feed" character, code 10 generates a "carriage return", and code 27 generates an "Escape" character. The "Escape" character signifies that the following characters constitute a special control message to the printer. This message is not printed. Escape mode is exited automatically when enough control characters have been received to complete a valid command sequence.

The decimal codes 128-255 give starbursts (all segments lit) in the display. The printer characters for codes 128-255 are the same as those for 0-127, respectively, except for the three special HP-IL printer codes.

Decimal code 0 gives a "null" character, which is not related to the NULL message when a key is held down too long. Unless you do a lot of synthetic programming, you will probably never use a null character. Read Appendix C of the Extended Functions/Memory module Owner's Handbook for a complete summary of the strange behavior they can exhibit.

Character number 255 has a few strange properties as well. When it is displayed as part of the ALPHA register, it appears as a starburst. However, when you ASTO a string that contains this character and then display the ASTO'd string, what you see in the display will be misleading. The decimal 255 character and any characters that follow it will be invisible. If you have a revision 1B Extended Functions/Memory module, you must observe another caution involving decimal 255 characters: do not store more than 6 consecutive decimal 255 characters in a text file. You risk losing that file and all subsequent files the next time you purge a file closer to the beginning of extended memory. This is because the HP-41 uses a register of 7 of these characters to mark the last occupied register of extended memory. This caution does not apply to the HP-41CX or to extended functions revisions 1C and up.

(continued from page 59)

The **ALENG (ALPHA length)** function computes the number of characters in the ALPHA register, from Ø to 24. This number is placed in the X register, while the former contents of X, Y, and Z, are raised to Y, Z, and T. LASTX is unchanged.

For example, suppose you want to check whether the ALPHA register is empty and branch to LBL 99 if it is. The sequence

```
ALENG
X=Ø?
GTO 99
```

will accomplish this. If you just want to generate an error message if ALPHA is empty, you can use the sequence

```
ALENG
1/X         (gives DATA ERROR if X=Ø).
```

Another use for ALENG is to determine how many ASTO and ASHF operations are needed to store a long ALPHA string. Since each ASTO stores 6 characters (and ASHF then removes these 6 characters), we can divide the initial length by 6 and round up to the next highest integer to determine how many ASTO's will be needed. Another approach is to check the length after each ASHF and continue as long as the ALPHA register is not empty.

An advanced application of ALENG is to aid in rotating strings containing null characters. If a null character is rotated to the front (leftmost part) of a string, it will disappear. The only way you can tell that this happened is to check the ALENG before and after the rotation to see whether it decreased. Unless you do a lot of synthetic programming (see section 1ØA), you probably will not use ALENG this way. But now, when you see ALENG preceding and following AROT in a synthetic program, you will know why.

The function **POSA (position in ALPHA)** accepts a decimal character code in X. It then searches the string in ALPHA, from left to right, for the first occurrence of the specified character. A position code is returned to the X-register, overwriting the character code. The character code is saved in LASTX. POSA, POSFL (page 41), and GETKEYX (page 9Ø) are the only extended functions that alter LASTX.

The position code returned by POSA is an integer from Ø to 23. A value of Ø indicates a match at the first (leftmost) character of ALPHA, while a value of 23 indicates a match at the 24th character. If no match is found, the value -1 is placed in X. These rules for the position code may seem strange, but they are designed with a specific application in mind. If you want to locate a particular character and bring in to the front of ALPHA, you can use the very simple sequence

```
(character code)
POSA
X<Ø?       (If the character is not found,
SF 99       then display "NONEXISTENT")
AROT    .
```

The located character can then be removed by an ATOX. If you have separator characters in the ALPHA register, POSA, AROT, and ATOX working together can find the separators, remove them, and prepare the ALPHA register for each separate string to be processed.

As an example of POSA, suppose you have someone's name in the ALPHA register in the standard form "firstname lastname", and you want to change it to the form "lastname, firstname". The following sequence should do the trick:

```
"ROGER HILL"    (for example)
"├, "          Place a comma and space after last name
32             Decimal code for the space character
POSA           Locate the space after "ROGER"
AROT           Bring it to the front of the string
ATOX           Delete the space.
```

The **POSA** function has a second mode of operation that allows the ALPHA register to be searched for a string of 1 to

6 characters. Instead of putting a decimal character code in X, you can ASTO a string there. For example, try the following:

```
"FGH"              (press ALPHA F G H ALPHA)
ASTO X             (press ALPHA shift STO . 6 ALPHA)
"WXYXABCDEFGHIJ"
POSA               (XEQ ALPHA P O S A ALPHA)
```

The result should be 9, indicating that the string "FGH" begins at the 10th character of ALPHA. Note that the string "FGH" is still available in LASTX if you need it.

In this second mode, POSA is very similar to the POSFL function (page 41). Because it is not limited to 6-character substrings or 24-character strings, the POSFL function is far more useful than POSA for substring searches. However, for single character searches, either by decimal code or single-character substring, the POSA function is often simpler to use than POSFL.

The **ANUM (ALPHA number)** function is a near-inverse to ARCL. The primary use of the ARCL (ALPHA recall) function is to append a number to the ALPHA register. The ANUM function extracts a number from the ALPHA register. For example, if the ALPHA register contains the string "A=452", executing ANUM will put the result 452 into the X register. The stack is raised and LASTX is not affected.

ANUM searches the ALPHA register from left to right, returning the first legitimate number found. This is affected by commas and periods in ALPHA, and the status of flags 28 and 29. When the ALPHA register contains one or more periods or commas, things start getting complicated. First, the period and comma are interpreted according to the status of flags 28 and 29. If flags 28 and 29 are set, a period is interpreted as a decimal point and a comma as a digit separator. If flag 28 is clear and 29 is set, a comma is interpreted as a decimal point and a period as separator. This is the standard European notation. If flag 29 is clear, digit separators (comma if flag 28 set, period if flag 28 clear) are treated as alpha characters. Therefore if the number "12,ØØ3.Ø5" is in ALPHA and you execute ANUM with flag 28 set but flag 29 clear, the result will be 12. Because flag 29 was clear, the comma was regarded as a character, splitting the number into two parts. For most applications, you should try to avoid this problem by making sure that flag 29 is set before you use ANUM. One more caution: if you use ANUM with a nonstandard number format in ALPHA, the results may not be what you intended. For example "-34-" XEQ "ANUM" yields the positive result 34. The second negative sign cancelled the effect of the first. Also, two or more numbers separated only by + or - symbols will be interpreted as a single number.

PROBLEMS

4.1. Write a 4-step sequence to append a character to the left end of the ALPHA register.

4.2. Write a short sequence to ASTO the ALPHA register contents without wasting any registers on empty strings.

4.3. Write sequences to delete n characters from ALPHA:
a) from the left, and b) from the right.

4.4. Modify the above "lastname, firstname" rotation sequence to handle a possible middle initial.

4C. Flag Manipulations

The extended functions provide three new functions that are very helpful in controlling the status of flags. Most important of these are RCLFLAG and STOFLAG.

Consider the following situation. You are writing a program that needs to round or display a result in a certain format, for example FIX 2. You would like the program to be able to restore the original display format before returning control to the user. Before the advent of extended functions, this seemingly simple task was very difficult to do. Elaborate flag testing was needed at the start of the program to determine the original display setting. Then, after the display setting was changed, additional complicated operations were needed to restore the original setting.

The availability of the RCLFLAG (recall flags) and STOFLAG (restore flags) functions eliminates all this difficulty. You simply use RCLFLAG to recall the flag setting before changing the display, then use STOFLAG to restore the original flag status. A typical instruction sequence might look like this:

```
RCLFLAG      Places a flag-equivalent string in X
"AMT= $"
FIX 2
ARCL Ø1
AVIEW
STOFLAG      Uses the string to restore flags.
```

The LBL 92 subroutine of Chapter 7's "NAP" program is an example of this technique. Now for some details about the RCLFLAG and STOFLAG functions.

The **RCLFLAG (recall flags)** function recalls to the X-register an unintelligible alpha string that represents the current status of flags Ø to 43. Like a standard RCL instruction, RCLFLAG raises the contents of X, Y, and Z into Y, Z, and T, unless it is immediately preceded by an ENTER↑, CLX, or other stack lift disabling operation. LASTX is not changed.

The alpha string formed by RCLFLAG can be stored in a data register or kept in the stack. The only use of this string is to later restore some or all of the original flag status by using STOFLAG.

The **STOFLAG (restore flags)** function has two modes of operation. The one illustrated in the above example is the simpler mode. Simply put the RCLFLAG alpha data into X, and execute STOFLAG. The original status of flags Ø to 43 (as of the time RCLFLAG was executed) is restored.

STOFLAG's second mode of operation permits selective restoration of the previous flag settings. To use this mode, put the RCLFLAG alpha string in the Y register, and a number of the form **bb.ee** (not bb.eee) in **X**. Then, when you execute STOFLAG, the block of flags from flag number bb to flag number ee (including bb and ee) will be restored to their original status. To restore a single flag, put the flag number **bb** in X.

This second mode of operation allows you, for example, to restore just the display setting, just the general-purpose flags, or just the triginometric mode. To restore only the display setting, you would use a sequence like:

```
RCLFLAG       Save the original flag settings
STO Ø5        in data register Ø5
.
.             (display-altering program steps)
.
RCL Ø5        Bring back the RCLFLAG string
36.41         Flags 36-41 control the display setting
STOFLAG       Restores flags 36-41 only.
```

Using RCLFLAG and STOFLAG, it is possible to have several sets of flag settings appropriate to different sections of a program. Each flag setting can be stored in a separate data register in the form of a RCLFLAG alpha string. Each section of the program can then simply use RCLFLAG to establish its flag settings, rather than having to deal with the flags individually. This should noticeably speed program execution.

Another application of RCLFLAG/STOFLAG is the following short routine that will print the contents of the ALPHA register if the printer is turned on and enabled (flag 21 set), or AVIEW and PSE otherwise. This is superior to a simple AVIEW because:

1) It does not halt if flag 21 is set but the printer is turned off, and
2) It does not force you to wait for a slowly scrolling display if the printer is in use.

This sequence was used in the "VAS" (view ASCII file) program of section 3F. Here it is as a program, "PVA" (print or VIEW ALPHA):

```
01 LBL "PVA"
02 SF 25
03 PRA          Attempt to print ALPHA
04 RCLFLAG
05 FS?C 21      Clear flag 21 for later AVIEW
06 FC? 25       If print was not successful,
07 AVIEW        or if flag 21 was clear, then AVIEW.
08 STOFLAG      Restore flags
09 RDN
10 FS?C 25      If print was not successful,
11 FC? 21       or if print was disabled,
12 PSE          then PSE after the AVIEW.
13 END
```

Yet another RCLFLAG/STOFLAG application allows you to obtain FIX/ENG display format by setting flags 40 and 41. This display mode looks like a normal FIX format until the number in X becomes large or small enough that an exponent is needed. Then the ENG mode takes over. Just put a number from 0 to 9 in X, and execute "FEX" to set FIX/ENG mode with the specified number of digits displayed to the right of the most significant digit.

```
01 LBL "FEX"      FIX/ENG INDirect X
02 ENG 0          Set flag 41
03 RCLFLAG
04 FIX IND Y      Set flag 40 and select the
05 X<>Y             correct number of digits
06 RDN
07 41
08 STOFLAG        Set flag 41 (others unchanged)
09 R↑
10 R↑             Put the stack back in order.
11 END
```

The third flag-related function is **X<>F (X exchange flags).** This function treats general-purpose flags 00 through 07 as a "mini-register", and performs an exchange with that register. This "mini-register" can only hold integer numbers from 0 to 255 inclusive. Therefore any fractional part of X is discarded before the exchange is performed, and the sign of X is ignored. In effect, the X<>F function incorporates the sequence ABS, INT as its first two steps, except that LASTX is not altered. In fact, the sequence X<>F, X<>F can be used to perform ABS, INT on a number up to 255 without altering the stack or LASTX. No DATA ERROR message is given by X<>F unless INT(ABS(X)) is larger than 255.

The power of X<>F is that, like STOFLAG and RCLFLAG, it gives you the ability to maintain several sets of general-purpose flags in data registers.

After an X<>F is performed, the settings of flags 00 through 07 express, in binary form, the former value of X. If you are mathematically inclined, the formula is

$$\text{former X} = \sum_{i=0}^{7} f_i * 2^i \quad ,$$

where f_i = 1 if flag i is set, 0 if flag i is clear.

In this binary representation, flag Ø has the value 1, flag 1 has the value 2, flag 2 has the value 4, and so on. This equivalence can be represented in tabular form. The example shown below gives the binary representation of the decimal number 133.

flag number	value if set	set?	current value
ØØ	1	Y	1
Ø1	2	N	Ø
Ø2	4	Y	4
Ø3	8	N	Ø
Ø4	16	N	Ø
Ø5	32	N	Ø
Ø6	64	N	Ø
Ø7	128	Y	128
		Total:	133

As a simple example of the usefulness of X<>F, suppose you have a program that starts by clearing flags ØØ through Ø3 and setting flags Ø4 and Ø5. Rather than use the sequence

```
CF ØØ
CF Ø1
CF Ø2
CF Ø3
SF Ø4
SF Ø5        (12 bytes)
```

one can use the sequence

```
48           (flag Ø4 = 16, flag Ø5 = 32)
X<>F         (4 bytes).
```

If you were not familiar with the binary equivalence, you could have verified that 48 was the correct number as follows:

```
Ø          This clears flags ØØ through Ø7,
X<>F       a very useful technique.
SF Ø4
SF Ø5
X<>F
```

The result of this sequence is 48. This shows that the number 48, when followed by X<>F, will set flags Ø4 and Ø5, while clearing the others.

If it were important in the above example to preserve the status of flags Ø6 and Ø7, you could have used this sequence:

```
Ø
X<>F        Recalls the flag status
64
/           Flags Ø6 and Ø7 are now in the
INT         one's and two's digits
LASTX
*
48          These two steps add in the number
+           to set flags Ø4 and Ø5.
X<>F
```

Further analysis of this sequence is left as an exercise. When you understand it, you will be able to fully utilize X<>F. But do not be misled into using X<>F everywhere. For instance, in the example just shown, a simple set of six instructions to clear flags ØØ-Ø3 and set flags Ø4-Ø5 saves two bytes over the X<>F method!

PROBLEMS

4.5. Write a sequence of instructions that evaluates the function

$$f(x) = \frac{SIN(PI*x)}{PI*x}$$

The SIN function must be evaluated in RADian mode, but the original trig mode is to be restored.

4.6. Write a sequence to activate FIX/ENG mode without changing the currently selected number of digits (flags 36-39).

<u>Synthetic Programming applications of RCLFLAG and STOFLAG</u>

When a printer is attached, program execution is slowed. The amount of slowing can be reduced if you synthetically clear flag 55. Flag 55 will only remain clear as long as the program continues to run. Once it stops, flags 55 and 21 will both be set. The following short sequence, developed by Steve Wandzura, clears flag 55 without disturbing any other important flags:

```
RCLFLAG
SIGN        stores flags in LASTX, sets X=Ø.
STO d       clears all flags.
X<> L       brings flags back to X.
STOFLAG     restores flags (up to 43).
RDN         restores the stack (except T).
```

The exact format of the ALPHA string generated by RCLFLAG is, in hexadecimal,

1F Ff ff ff ff ff ff,

where the f's denote flag information, corresponding to flags Ø to 43, left to right. The flags are shifted one-and-a-half bytes to the right from their normal position in the flag register. The extra half-byte shift can be useful in advanced synthetic programming applications.

4D. SIZE-related functions

Two of the extended functions allow you to check and adjust the SIZE *under program control*. This is a powerful new capability that, before the introduction of the extended functions module, was available only through synthetic programming techniques.

The **SIZE? (SIZE finder)** function finds the number of data registers currently allocated and places that number in the X register. So, for example, if you set a SIZE of 020, and then you execute SIZE?, the result will be the number 20 in X. The stack is lifted just as for a RCL operation.

The SIZE? function is the classic example of an essential operating system function that the designers left out of the original HP-41. If you have used an HP-41 without extended functions, you know this already. How many times have you wanted to check the current SIZE before starting a program or manual data entry? The usual procedure was to try several RCL operations in an attempt to get an approximate idea of what the first NONEXISTENT register is. This procedure could be automated by programs like this simple but slow one:

```
01 LBL "SZFIND"
02 CLX              These 2 lines set X=0, set flag 25
03 SF 25            to avoid stopping at line 05.
04 LBL 01
05 RCL IND X        Attempt to recall a register.
06 FC? 25           If the register was NONEXISTENT,
07 RTN              the value in X is the SIZE.
08 RDN
09 1
10 +                Add 1 to the register number.
11 GTO 01           then try the next one.
12 END
```

The SIZE? function is incomparably faster than this approach, and much more practical too. You might use it often enough to warrant assigning it to a key, but even if you do not, it is quickly accessible by the key sequence

```
          XEQ ALPHA S I Z E ? ALPHA   .
```

The **PSIZE (programmable SIZE)** function does the same thing as SIZE, except that it does not give the familiar three-underscore prompt. Instead, the SIZE is adjusted to equal the value in X. PSIZE can be used in a running program, even in a sixth-level subroutine, without any adverse effect on the program's operation. This means that you can write programs and subroutines that automatically increase or decrease the SIZE as necessary.

The following short sequence of instructions checks whether the current SIZE is sufficient for a specific purpose, and uses PSIZE to increase the SIZE if necessary.

```
(required SIZE)
SIZE?
X<>Y
X>Y?
PSIZE
```

Another variation provides an audible warning of the impending PSIZE operation, in case the user of the program wants to press R/S to prevent resizing:

```
(required SIZE)
SIZE?
X>Y?
GTO Ø1
TONE 9
X<>Y
PSE
PSIZE
LBL Ø1
```

4E. Block operations

The extended functions REGMOVE and REGSWAP allow you to copy, exchange, or rotate blocks of data registers. The HP-41CX adds the function CLRGX, which clears a block of data registers. If you have an HP-41C or CV, a short "block clear" program does the same job.

The **REGMOVE (register move)** function accepts an input of the form **sss.dddnnn** in the X register. Executing REGMOVE copies a source block of nnn data registers beginning at register sss to a destination block of nnn data registers beginning at register ddd. If nnn is zero, one register is copied. REGMOVE does not alter the stack or LASTX. As an example, the sequence

```
        6.001003   REGMOVE
```

copies a block of 3 registers. The source block is registers 06, 07, and 08, while the destination block is composed of registers 01, 02, and 03.

To make the examples easier to follow, first set the SIZE to 020 and run the "PRELOAD" program from page 24. This will "tag" all your data registers. When you press

```
        XEQ "PRELOAD"
```

the value 0 is stored in register 00, 1 in register 01, and so on. The value in each register matches its number.

As a simple example of REGMOVE, press

```
        3.007006   XEQ ALPHA R E G M O V E ALPHA.
```

This will cause registers 03-08 (a 6-register block) to be copied into registers 07-12, as shown on the next page.

register:	Ø3	Ø4	Ø5	Ø6	Ø7	Ø8	Ø9	1Ø	11	12
start:	3	4	5	6	7	8	9	1Ø	11	12
										8
									7	
								6		
							5			
						4				
					3					
result:	3	4	5	6	3	4	5	6	7	8

The intermediate steps shown in this diagram are invisible to you. They are included so that you can visualize how the copying process is implemented. Where there is no entry, the register contents are not changed at that step.

The **REGSWAP (register swap)** function exchanges the contents of two blocks of data registers. Like REGMOVE, it accepts a number of the form **sss.dddnnn** in X, where sss denotes the beginning of the source block, ddd denotes the beginning of the destination block, and nnn denotes the number of registers in each block. If nnn is zero, the HP-41 assumes that you want nnn=1 and it swaps only registers sss and ddd. The stack and LASTX are unchanged.

The internal programming of REGSWAP interchanges one pair of registers at a time. If sss<ddd, the highest numbered register is swapped first and the lowest numbered register is swapped last. If sss>ddd, the lowest numbered register is moved first and the highest numbered register is moved last. This internal order of operations is the same for REGSWAP as it is for REGMOVE. Normally you would not need to know in what order these operations are performed. However, if the source and destination blocks overlap, the order of operations affects the result. As an example of REGSWAP, try this:

```
        XEQ "INIT"
        3.ØØ7ØØ6   XEQ "REGSWAP"    .
```

The following diagram shows how the register exchange is performed.

register:	Ø3	Ø4	Ø5	Ø6	Ø7	Ø8	Ø9	1Ø	11	12
start:	3	4	5	6	7	8	9	1Ø	11	12
						12				8
					11				7	
				1Ø				6		
			9				5			
		12				4				
	11				3					
result:	11	12	9	1Ø	3	4	5	6	7	8

As you can see, REGSWAP can really scramble the registers when there is a significant overlap of the two blocks. This feature can be turned to an advantage, however, in constructing a "block rotate" function. Consider the following example. Press

```
        XEQ "PRELOAD"      (initializes the registers)
        4.ØØ3ØØ9   XEQ "REGSWAP"    .
```

The internal steps in the register swap are as shown on the next page. Because 4 is greater than 3, the swap proceeds from low to high numbered registers.

register:	Ø3	Ø4	Ø5	Ø6	Ø7	Ø8	Ø9	1Ø	11	12
start:	3	4	5	6	7	8	9	1Ø	11	12
	4	3								
		5	3							
			6	3						
				7	3					
					8	3				
						9	3			
							1Ø	3		
								11	3	
									12	3
result:	4	5	6	7	8	9	1Ø	11	12	3

The result is that the block of 1Ø registers from Ø3 to 12 is rotated one register downward. If you had pressed

```
3.ØØ4ØØ9  XEQ "REGSWAP"
```

the 1Ø-register block would have been rotated upward one register. This result can be easily generalized.

To rotate a block of nnn registers beginning at register sss, use the REGSWAP input

```
sss.(sss+1)(nnn-1)  to rotate upward 1 register, or
(sss+1).sss(nnn-1)  to rotate downward 1 register.
```

If you want to rotate a block of **n** registers by **r** registers upward or downward, you may be able to accomplish the desired result with a single REGSWAP instruction. If the number **r** divides **n** evenly (without a remainder), use

```
sss.(sss+r)(nnn-r)  to rotate upward r registers, or
(sss+r).sss(nnn-r)  to rotate downward r registers.
```

The **CLRGX (clear registers designated by X)** function on the HP-41CX accepts an input of the form **bbb.eeeii** in the X register, where **bbb** is the first register to be cleared, **ii** is the increment between registers to be cleared, and registers beyond **eee** are not to be disturbed.

If ii is not supplied (ii=Ø), then a default value of ii=1 is assumed, so that registers from bbb up to and including eee are cleared. If you are familiar with the ISG (increment and skip if greater than) instruction on the HP-41, these rules will not be new to you. For example, to clear registers Ø4 through Ø8, you would press

```
4.ØØ8  XEQ ALPHA C L R G X ALPHA  .
```

To clear registers Ø1, Ø3, Ø5, Ø7, and Ø9, you would press

```
1.ØØ9Ø2  XEG ALPHA C L R G X ALPHA  .
```

Actually it is a very rare application in which you need to use a nonzero value of ii. One example would be when you have stored a matrix, one entry per register, in a block of data registers. Nonzero values of ii allow you to selectively clear one column or the diagonal elements.

CLRGX leaves its input in X and does not disturb LASTX.

If you have an HP-41C or CV, it is easy to write an instruction sequence to clear a block of registers. The following very simple program will clear a block of data registers beginning at register bbb and ending with register eee. Just put the number **bbb.eee** in X and execute "BC" (block clear).

```
Ø1 LBL"BC"
Ø2 SIGN          Stores bbb.eee in LASTX
Ø3 CLX           The value Ø is to be stored.
Ø4 LBL Ø3
Ø5 STO IND L     Clear the register.
Ø6 ISG L         Increment the counter.
Ø7 GTO Ø3
Ø8 END
```

If you plan to clear large blocks of data registers, you can use the faster program "BCΣ" (block clear using summation registers) that uses the CLΣ function to clear 6 registers at a time. To clear 100 registers, "BC" uses 13 seconds, while "BCΣ" takes less than 4 seconds. To use "BCΣ", just put the bbb.eee control number in X and execute "BCΣ".

BCΣ program listing

```
 01♦LBL "BCΣ"      08♦LBL 01      14 DSE X       19♦LBL 03
  02 6 E-5          09 CLΣ                       20 STO IND Y
  03 +            10 ΣREG IND X   15♦LBL 02       21 ISG Y
04 ΣREG IND X      11 ISG X      16 LASTX        22 GTO 03
  05 ISG X          12 GTO 01     17 -            23 END
  06 X<0?                         18 0
  07 GTO 02         13♦LBL 02                          44 BYTES
```

PROBLEM
4.7. Write a short program to rotate a block of nnn registers starting at register sss upward by rrr registers (downward if rrr is negative). At the start of the program, assume that sss is in X, nnn is in Y, and rrr is in Z. Use only the stack and LASTX. (It is not as easy as it looks.)

4F. Key assignment control

Two extended functions, CLKEYS and PASN, enhance your ability to control USER mode key assignments. Another extended function, GETKEY (plus GETKEYX on the HP-41CX), allows your program to "read" the keyboard, providing the ultimate in redefinition of the keyboard.

First a few words about key assignments. The ability to assign a function to a single key is one of the features that distinguishes the better programmable calculators. The HP-65 and HP-67 programmable calculators had a top row of keys labeled **A** through **E** (the shifted top row was labeled **a** through **e**). At the touch of one of these keys, you could execute a section of the calculator's program that began with the corresponding label (A-E or a-e).

The HP-41 is a major advance over its predecessors in key assignment capability. The HP-41's USER mode allows virtually every key to be redefined with an assignment of a global label or a function. (Global labels are those labels that appear in Catalog 1, while functions appear in Catalog 2 or Catalog 3.) Also, to maintain compatibility with the HP-67 and HP-97, the top row of keys can access local labels A-E (unshifted) and a-e (shifted) in the current program. In addition, the unshifted second row can access local labels F-J. This will work as long as no global label or function is assigned to the key in question. This automatic label search feature is described under "Local labels" in the HP-41 Owner's Handbook. If a key in the top row, shifted or unshifted, or in the second row, unshifted only, is pressed in USER mode, and if no global label or function is assigned to that key, a search is begun. If the corresponding local label (**A** through **J** or **a** through **e**) is found in the current program, the calculator starts executing the program at that point. Hold the key down to preview its function.

Incidentally, because this local label search can take a relatively long time, it is often useful to assign the **X<>Y** and RDN functions to their own keys. This assignment has higher priority than a local label, so no label search is performed. The response time to these keys in USER mode is noticeably improved.

The **PASN (programmable ASN)** function works almost like the ASN (assign) function does from the keyboard. Recall that when you use ASN, you have to enter ALPHA mode and spell out a function name. It's the same with PASN, except that you spell the function name out in the ALPHA register <u>before</u> you execute

PASN. With ASN, you designate the key to which the function is to be assigned by actually pressing the key after spelling out the function name. If you hold that key down for a moment, a keycode appears in the display. This keycode is a two-digit number. The first digit is the row number of the key (1 through 8), while the second digit indicates the column (1 through 5). It is this row/column keycode that you have to put in the X register before executing PASN.

The ASN function can be used to manually clear a key of its assignment. You just press ALPHA ALPHA for the function name. When no function is named, the HP-41 assumes that you want the key to be free of any assignment. Once again, PASN works similarly. Just make sure the ALPHA register is empty, put the row/column keycode in X, and execute PASN.

Summarizing: to use PASN, load the ALPHA register with the name of the function to be assigned, put the row/column keycode in X, and execute PASN. The specified function will be assigned to the designated key. If the ALPHA register is empty, the designated key will be cleared of its assignment. These instructions apply identically whether PASN is an instruction in a program or whether it is executed from the keyboard.

The PASN function lets you write programs that make key assignments. For example, suppose you had a program to update text files. It could prove quite helpful to have operations like INSREC (insert record) and DELREC (delete record) assigned to keys, even though these functions may not be useful enough to keep assigned to keys all the time. The answer is to use PASN at the beginning of your program to assign these functions to convenient keys. At the end of the program you can use PASN again, with the ALPHA register empty, to clear the assignments.

The short routine listed at the top of the next page was written by Alan McCornack. It clears the top row (unshifted only) of any function or global label key assignments. This technique can be easily extended to meet your needs for selected key assignment clearing.

```
Ø1 LBL "CT"      (clear top row)
Ø2 CLA
Ø3 11.Ø15        ISG counter for keycodes 11 to 15
Ø4 LBL Ø5
Ø5 PASN          Clear key x.
Ø6 ISG X
Ø7 GTO Ø5
Ø8 END
```

The **CLKEYS (clear keys)** function clears all USER mode key assignments of global labels or functions. Note that when you use CLKEYS to delete global label and function key assignments, local label pseudo-assignments (keys A-J and a-e) will no longer be masked by the presence of any higher-priority global label and function assignments.

CLKEYS is a drastic solution to problems of conflicting key assignments. In most cases you are better off using PASN to clear or reassign individual keys. Section 1ØG has an even better method.

Here is a typical application of PASN and CLKEYS. Suppose you have two different sets of key assignments that you like to use with your HP-41, depending on what you are using it for. You can write two programs, one to set up each set of assignments. Each program would have this general form:

```
LBL"KB1"         (keyboard 1)
CLKEYS           Eliminate previous assignments
"function 1"
(keycode 1)
PASN
```

```
"function 2"
(keycode 2)
PASN
"function 3"
(keyode 3)
PASN
.
.
.
END
```

Since PASN, like ASN, will overwrite any assignment already made to the designated key, you may find that CLKEYS is unnecessary here, especially if several assignments for the different keyboards are the same functions, or use the same keys. If you want to selectively clear keys of assignments, include a sequence like

```
CLA
(keycode 1)
PASN
(keycode 2)
PASN
etc.
```

in the "KB1" program.

The third extended function that is related to key assignments is GETKEY. This is a very special function that is perhaps more powerful than any other extended function, as you will see in Chapters 8 and 9.

The **GETKEY (get keycode)** function is an entirely new type of function for the HP-41. When you execute GETKEY as part of a program, the calculator pauses for up to 1Ø seconds waiting for you to press a key. If a key is pressed, the row/column keycode for that key is placed in the X register. If no key is pressed within 1Ø seconds, the number Ø is placed in X. In either case, the stack is raised and LASTX is not disturbed. If you want the program to keep waiting until a key is pressed, a simple loop will do the job:

```
LBL ØØ
GETKEY
X=Ø?
GTO ØØ
```

As long as no key is pressed, this program segment will keep looping. To avoid raising the stack each time the GETKEY is unsuccessful, you can use a RDN instruction between LBL ØØ and GETKEY.

Unlike PASN, the GETKEY function has keycodes for the 4 mode switches. For GETKEY, these keys are assigned a row number of Ø. The ON key has a keycode of 1 and the ALPHA key has a keycode of 4. Remember that these keycodes only appear as results from GETKEY; they will not work with PASN.

When you use GETKEY, avoid sequences like this:

```
LBL ØØ
GETKEY
GTO ØØ
```

The omission of the X=Ø? test causes an "infinite loop". But you can't stop this one by just pressing R/S. After all, R/S is just key 84. Even pressing the ON switch won't stop it. One way to stop it is to take out the batteries. A better way is to press and hold the R/S key, press the ON key, release the R/S key, and release the ON key. The best bet is to make sure that you have a normal way out of any GETKEY loop.

With GETKEY, a program can simulate the local label assignment feature on all 35 unshifted keys. What is more, the program can do this without worry of conflict with other key assignments and without the necessity of setting USER mode. The interpretation of each key can even change within a program.

The most commonly used type of sequence with GETKEY is:

```
LBL ØØ
RDN
GETKEY
X=Ø?            If no key was pressed,
GTO ØØ          then try again.
XEQ IND X       Execute a subroutine corresponding
.               to the key that was pressed.
.               This portion of the program displays
.               results or does other operations
.               that are the same for all keys.
RTN or GTO ØØ
LBL 11          This section is executed if the key
.               at row 1, column 1 was pressed.
.
RTN
LBL 12          This section is executed if the key
.               at row 1, column 2 was pressed.
.
RTN
.               Use a LBL for each key that you
.               want your program to respond to.
.
END
```

This sequence waits until a key is pressed, then executes whatever program steps follow the corresponding numeric local label. For example, if you were to press the backarrow key, the HP-41 would look for LBL 44 (row 4, column 4) and start executing that portion of the program as a subroutine. If you want more keys to have functions, just add the corresponding numeric labels to the program, followed by sequences that do whatever you want the key to do.

USER mode key assignments do not conflict with GETKEY, because the GETKEY function temporarily pre-empts them, just as it pre-empts the on/off and mode selection keys. **GETKEY can function as another level of custom key assignments.**

Here is a simple example of how GETKEY can be used. This sequence prompts for a YES/NO response. Either R/S or "Y" (the multiplication key) is accepted as a YES response; any other key is assumed to be a NO response.

```
"message"
SF 25
LBL ØØ
AVIEW
GETKEY
RDN          Put keycode in stack register T
GTO IND T
.            (NO response drops into here)
.
RTN
LBL 71       (YES response goes here)
LBL 84       (R/S response goes here)
CF 25
.
.
RTN
```

The GTO IND T instruction branches back to LBL ØØ if no key was pressed, or to LBL 71 or 84 if the "Y" or R/S key was pressed. Otherwise there is no LBL corresponding to the keycode. This causes a NONEXISTENT error which clears flag 25. Execution then drops into the NO response sequence.

This GETKEY technique implicitly requires, as do most uses of GETKEY, that there be no extraneous local LBL's that have keycode-like numbers. This means that within this program, the following LBL's are not allowed, except where such a LBL is needed as the object of the GTO IND instruction:

Ø1	Ø2		Ø3	Ø4	(the rocker switches)
11	12	13	14	15	(row 1)
21	22	23	24	25	(row 2)
31	32	33	34	35	(row 3)
41		42	43	44	(row 4)
51		52	53	54	(row 5)
61		62	63	64	(row 6)
71		72	73	74	(row 7)
81		82	83	84	(row 8)

Of course, you can violate this constraint in your programs if you do not mind invalid results when an illegal key is pressed in response to a GETKEY prompt.

This kind of YES/NO response testing technique is used in the mailing list program in Chapter 7. Two more response options are added, but the principle is the same. A much more elaborate example of GETKEY is given in Chapter 9, where a program is presented that simulates the single-key base conversion functions of the HP-16 calculator.

4G. Added functions on the HP-41CX

The HP-41CX includes 14 more extended functions than the Extended Functions/Memory module for the HP-41C or CV. Six of these functions have already been described: EMROOM (page 11), EMDIRX (page 11), RESZFL (pages 32 and 45), ED (page 52), ASROOM (page 51), and CLRGX (page 8Ø). The remaining eight functions, GETKEYX, ΣREG?, X<NN?, X<=NN?, X=NN?, X≠NN?, X>=NN?, and X>NN? will be described in this section.

The **GETKEYX (get keycode, wait X seconds)** function is an extended version of the GETKEY. A number in X up to ±99.9 specifies the number of seconds that the calculator will wait for a key to be pressed when you execute GETKEYX. If X is less than 0.1 the calculator will do its best, but you may get a wait that is slightly longer than you requested. GETKEYX returns a keycode to the Y-register (not the X-register) and a character code to the X-register, as explained below. The interval that was specified in X is saved in LASTX, while the former contents of stack registers Y and Z are raised to Z and T, respectively.

If you press an unshifted key within the specified time, the keycode is placed in the Y-register. If you press the shift key, the key code 31 (row 3, column 1) is <u>not</u> returned as it would be if you had used GETKEY. Instead, the calculator restarts the specified interval and waits for another key to be pressed. When you press the second key, the negative of its keycode is placed in the Y register. This feature lets you "GET" shifted keys as well as unshifted keys.

If the specified time interval expires before a key is pressed, the value zero is placed in Y to indicate that no key was pressed.

The value returned to the X register depends on the ALPHA mode status (flag 48) and on which key is pressed. If ALPHA mode is on (flag 48 set) and you press a key (or shift plus another key) which corresponds to a character, the ASCII equivalent of the character is returned to X. This makes it simple to create a copy of the selected character in the ALPHA register--just use a single XTOA instruction.

If ALPHA mode is off (flag 48 clear) when you use GETKEYX, ASCII codes are returned only for the digit keys, the radix (decimal point) key, and the CHS key. Once again, this code enables you to create a copy of the selected key in the ALPHA register simply by using XTOA. If the key pressed does not correspond to an ALPHA character (ALPHA mode on), the value zero is returned to the X register.

If you specify a negative time interval in the X register for GETKEYX, the calculator will not wait until the key is released, as it normally would. Instead, execution will resume as soon as electrical contact is made. Thus a sequence like this:

```
LBL Ø1
.
.
.
-.1
GETKEYX
RDN
X≠Ø?
GTO 01
```

will continue to loop as long as any key is held down. It is not very likely that your applications will lead you to use this feature of GETKEYX, but if you must use it, there is a possible problem you should be aware of. Although the GETKEYX instruction will indeed read the proper key code, releasing the key causes the normal function to be executed. For all but the ON and R/S keys, nothing will happen because the program is running. In contrast, releasing the ON key will shut off the calculator, and releasing the R/S key will halt the program. Thus when you execute GETKEYX with a negative number in X, avoid pressing the R/S or ON keys unless you are done using the program.

Here are two sample applications for GETKEYX. The first, "VREG" (view register), views a selected register for as long as the corresponding key is held down.

"VREG" program listing

```
01♦LBL "VREG"     09 CHS          19 -              28 GETKEYX
 02 CF 21         10 GETKEYX      20 X<0?           29 RDN
                  11 SIGN         21 CLX            30 X≠0?
 03♦LBL 01        12 X<>Y        22 VIEW IND X      31 GTO 03
 04 "REG?"        13 X=0?         23 .1             32 GTO 01
 05 AON           14 GTO 02       24 CHS            33 END
 06 AVIEW         15 X=Y?         25 SIGN
                  16 RTN
 07♦LBL 02        17 LASTX        26♦LBL 03        57 BYTES
 08 10            18 64           27 X<> L
```

When you run "VREG", the prompt "REG?" appears, requesting a register selection. The "A" key selects register Ø1, the "B" key selects Ø2, and the "Z" key selects 26. Press the ON key twice to quit the program.

Here is a brief explanation. The first GETKEYX loop detects when a key has been pressed. If the keycode is Ø1 (the ON key), a RTN stops the program. (Releasing the ON key turns off the calculator, and pressing it the second time turns the calculator back on.)

If the keycode is not Ø1, the ASCII code is retrieved from LASTX and adjusted by subtracting 64. This converts "A" to Ø1 and "Z" to 26. The next two lines replace negative values by zero. Then the selected register is VIEWed. The second GETKEYX loop simply continues to loop as long as a key is held down. When the key is released, the zero keycode causes a branch back to LBL Ø1 at the top of the program, where the REG? prompt is regenerated.

Another, more straightforward application of GETKEYX is to permit selection of a key for an assignment needed by a program. For example, suppose you have a base conversion program that makes assignments of the functions MOD, INT, and FRC to USER mode keys. In section 4F you learned how the PASN function can be used to make key assignments under program control. The GETKEYX function lets the user of the program select keys for these assignments "in real time", as the program is running, without having to go to the trouble of figuring out the keycode. A typical program might use a structure like this:

```
"MOD"       The LBL 99 subroutine prompts
XEQ 99      the user to press a key, uses
"INT"       GETKEYX to get the key code, then
XEQ 99      uses PASN to make the requested
"FRC"       assignment.
XEQ 99
.
.
.
LBL 99
"├ KEY? "     Note the spaces before and after KEY?.
AVIEW         Display message requesting a key.
1Ø
LBL ØØ
GETKEYX     Get the keycode in Y.
X<>L
X>Y?        If keycode <1Ø, try again.
GTO ØØ
RDN         These five steps remove the 6 characters
6           " KEY? " from the ALPHA register.
CHS
AROT
ASHF
RDN         Keycode is now in X.
PASN        Make the requested key assignment.
RTN
```

The **ΣREG? (summation register finder)** function on the HP-41CX gives the currently selected location of the summation register block, a block of 6 registers used by the calculator for the Σ+, Σ-, MEAN, and SDEV statistical operations. The Catalog 3 function ΣREG selects a starting register for this 6 register block. When you execute ΣREG?, the number returned is the same as the last location selected by ΣREG, or 11 if you have not executed ΣREG since the calculator was last cleared. The stack is raised and LASTX is undisturbed.

The functions of the 6 registers of the summation register block are as follows:

$R_{\Sigma REG?}$	Σx
$R_{\Sigma REG?+1}$	Σx^2
$R_{\Sigma REG?+2}$	Σy
$R_{\Sigma REG?+3}$	Σy^2
$R_{\Sigma REG?+4}$	Σxy
$R_{\Sigma REG?+5}$	n

The primary application of the ΣREG? function is to recall data from the statistical registers regardless of where those registers are located. For example, the sequence

```
2
ΣREG?
+               ΣREG? in L
RDN             ΣREG? + 2 in T
RCL IND T       Σy
RCL IND L       Σx
```

simulates the HP-67/97 function RCLΣ , bringing the sum of y values into the Y register and the sum of x values into the X register. You can modify this sequence to recall any of the six registers in the ΣREG block for your calculations.

The remaining six functions on the HP-41CX, **X<NN?**, **X<=NN?**, **X=NN?**, **X≠NN?**, **X>=NN?**, and **X>NN?**, allow you to compare the contents of X with any other register. The location of the other register is designated in Y. If you are familiar with indirect functions, these functions are effectively "X compare indirect Y" functions. To use one of these six functions, for example X>=NN?, just put a register number in Y (from Ø up to SIZE?-1) and press

```
     XEQ ALPHA X shift J = N N ? ALPHA
```

[Instead of a register number, Y can contain alpha data designating a stack register: "Z", "T", or "L". "X" and "Y" will work, but they are not useful.] The result will be displayed: YES if the contents of X are greater than or equal to the contents of the register specified in Y, NO otherwise.

If you use one of these instructions in a program, the YES or NO display will not appear. Instead, the instruction that follows will be executed only if the result is YES, otherwise it will be skipped. This operation conforms to the standard "do if true" rule for all test instructions.

One important feature distinguishes these six comparison functions from their Catalog 3 counterparts. These indirect comparison functions allow you to compare alpha data as well as numeric values. Strings are compared on the basis of ASCII codes. The effect is the same as if you used ATOX to compare the strings character by character from left to right, stopping at the first position that revealed a difference between the strings. The ASCII code ordering of alpha strings is similar to normal lexicographic ordering except that:

1) numeric and punctuation characters are less than alphabetic characters, and
2) lower case characters are greater than uppercase characters.

For more details on ASCII code ordering, see the ASCII equivalence table on pages 6Ø and 61.

The "ALSORT" (alphabetic sort) program listed on the next page will sort a block of registers from register **bbb** to register **eee**, inclusive, in increasing order. It uses a simple bubble-sort algorithm. Just put the number **bbb.eee** in X and execute "ALSORT".

"ALSORT" program listing

```
01♦LBL "ALSORT"
 02 ENTER↑
 03 ISG Y
 04 X<0?
 05 RTN
 06 INT
 07 1 E3
 08 /

 09♦LBL 01
 10 CF 06
 11 RCL X
 12 RCL Z
 13 INT
 14 +
 15 DSE X
 16 X<0?
 17 SF 06
 18 RCL IND L

 19♦LBL 02
 20 X>=NN?
 21 GTO 03
 22 X<> IND Y
 23 STO IND L
 24 FS?C 06
 25 GTO 03
 26 RDN
 27 ABS
 28 RCL IND X
 29 DSE Y
 30 FS? 53
 31 SF 06
 32 GTO 02

 33♦LBL 03
 34 R↑
 35 R↑
 36 ISG Y
 37 GTO 01
 38 END

 71 BYTES
```

Because the X>=NN? function is used for the comparisons, the "ALSORT" program will sort either numeric or alpha data (or both). The bubble sort algorithm is quite simple. In BASIC it might look something like this:

```
For i = bbb + 1 to eee
     For j = i-1 to bbb by -1
          If $R_{j+1} > R_j$, go to new i
          Else interchange $(R_{j+1}, R_j)$
          Next j
new i: Next i
```

In the "ALSORT" program, LBL Ø1 starts the **i** loop, which uses an ISG counter of the form **i.eee**. LBL Ø2 starts the **j** loop which uses a DSE counter **j.(bbb-1)**. If you want to trace the stack usage of "ALSORT", it may be helpful to know that at LBL Ø1, the important stack contents are

X=Ø.(bbb-1) and Y=i.eee .

At LBL Ø2, the stack contains:

L=j+1, X=R_{j+1}, Y=j.(bbb-1), Z=Ø.(bbb-1), and T=i.eee.

CHAPTER FIVE
A PROGRAM BYTE COUNTER

The HP-41 Owner's Manual mentions that the byte is the basic unit of program memory, and that each instruction in a program occupies one or more bytes. In fact, the Owner's Manual gives a tabular summary of the byte count for each different type of instruction.

If you have an HP-41CX and you want to know how many bytes one of your programs occupies, you can just execute CATALOG 1 and press R/S to halt it at the END of the selected program. The number at the right side of the display indicates the number of bytes of main memory that the program currently occupies. If the last line of the program is .END., you will need to press GTO.. to give the program its own END. No byte count is supplied with the .END. .

If you have an HP-41C or CV and you want to know how many bytes one of your programs occupies, you could refer to the tabular summary in your Owner's Manual and count the bytes by hand. Dividing by seven and rounding up gives the number of registers required to hold the program. Naturally this manual counting procedure seems like a waste of time when you have a powerful tool like the HP-41 at your disposal.

With the Extended Functions/Memory module, you can automate the byte counting procedure. The short utility routine "CBX" (Count Bytes using XMemory) presented here does the whole job. First "CBX" saves your program in extended memory, creating a new program file (unless that program was already saved in extended memory). Then "CBX" performs a RCLPT instruction which, for a program file, returns the program's byte count to the X register. Finally, "CBX" clears the temporary program file it created. If your program was already saved in extended memory, the file is not cleared.

After "CBX" gets the byte count, it computes the number of program registers required. This number would be equal to the FLSIZE, except that there is one extra byte in the file for the program's checksum (see page 181). So sometimes the number of program registers needed is one less than the FLSIZE.

<u>Instructions</u> <u>for</u> <u>"CBX"</u>

1. Make sure the program you want to count has a non-permanent END (not the .END.) as its last line, and that the program is packed. These are the same things you should do before saving a program in extended memory, in order to minimize the space used.
2. Load the ALPHA register with the name of the program for which you want a byte count. This name must not conflict with the name of an existing data or ASCII file.
3. Execute "CBX" (Press XEQ ALPHA C B X ALPHA).
4. If the program was already saved in extended memory, the result will appear very quickly. The byte count will be in X, and the number of program registers will be in Y. To see the number of program registers, press X<>Y or RDN (roll down).
5. If the program was not already saved in extended memory, "CBX" will take a few seconds longer to get the result. First an extended memory directory will appear. About half a second after the directory is finished, the byte count appears in X, with the register count in Y. To speed things up, you may interrupt the directory display and restart the "CBX" program by pressing R/S twice.

<u>"CBX"</u> <u>Example</u> <u>1:</u>

Count the number of bytes in "CBX" itself.

Solution:

Load the ALPHA register with the program name "CBX". Then XEQ "CBX". The result should be a count of 52 bytes in

X, and a count of 8 registers in Y. If "CBX" was not packed or if it did not have a nonpermanent END attached to it, your count may be slightly larger.

"CBX" program listing

```
01♦LBL "CBX"     07 SAVEP      13 CLD        18 +
 02 SF 25        08 RCLPT      14 RDN        19 7
 03 RCLPTA       09 "**CBX"                  20 /
 04 FS?C 25      10 PURFL      15♦LBL 01     21 INT
 05 GTO 01       11 ENTER↑     16 RCL X      22 X<>Y
 06 "⊢,**CBX"    12 EMDIR      17 6          23 END      52 BYTES
```

Line-by-line analysis of "CBX"

At the start of "CBX", the ALPHA register should contain the name of the program for which the byte count is desired.

Line 03 will return the byte count if the program is already saved in extended memory. If the program was not already in extended memory, the RCLPTA instruction will cause flag 25 to be cleared. Line 04 clears flag 25 and branches to LBL 01, the final computation sequence, if the RCLPTA was successful. Otherwise the named program is saved in a temporary extended memory file called "**CBX". Line 08, RCLPT, gives the byte count from this temporary file. The temporary file is then purged.

The EMDIR instruction is included to re-establish a working file after the PURFL instruction. If a working file is not defined and you have a revision 1B Extended Functions/ Memory module, the extended memory directory is in danger of being cleared. See page 19 for details. If your revision is 1C or higher (including the HP-41CX), you can safely delete lines 11 through 14.

The ENTER↑ and RDN instructions ensure that whether or not the directory is interrupted, the X register will contain the byte count. A completed EMDIR instruction raises the stack, giving the number of free registers in extended memory. An interrupted EMDIR instruction does not raise the stack.

Either way, the RDN instruction will leave the byte count in the X register, since the byte count was in X and Y before the EMDIR instruction. The CLD instruction is included so that the last directory entry does not remain in the display after "CBX" finishes.

The LBL Ø1 sequence starts with the byte count in X and computes the number of program registers required:

$$N_{reg} = \mathrm{INT}[(N_{bytes}+6)/7] \quad .$$

This formula accomplishes division by 7 and rounding up to the next highest integer. The "CBX" program finishes with N_{reg} in Y and N_{bytes} in X.

"CBX" Example 2:

Count the bytes in the "JNX" program from Section 1B. This example assumes that you have a copy of "JNX" in either main memory or extended memory.

Solution:

Load the ALPHA register with "JNX" and press XEQ "CBX". The result should be 8Ø bytes (12 registers).

CHAPTER SIX
DATA FILE APPLICATIONS

6A. A Universal Root Finder

One frequent application of programmable calculators is solving equations of the form f(x)=Ø ; that is, finding the value of x that makes this equation true for a user-supplied function f. For example, suppose the cost of producing n items using Machine 1 is SQRT(n), while the cost of producing n items on Machine 2 is 1Ø+LN(n+1). Because the LN function is "flatter" than the SQRT function, Machine 2 will be more economical for very large values of n. But at what value of n does Machine 2 become more economical? To find the crossover point, we need to solve the equation SQRT(n) = 1Ø+LN(n+1) for n. This equation can be rewritten in the form f(x)=Ø, where f(x) = 1Ø+LN(x+1)-SQRT(x). This example will be solved later in this section.

Minimization and maximization problems can be solved in the form f(x)=Ø by using the appropriate first derivative function for f. If the function being maximized or minimized has a relatively simple form, it is fastest to use calculus to find the correct first derivative. However, if finding the derivative analytically is not practical, the program "DERIV" in the next section can compute it numerically.

Any program that solves f(x)=Ø will need to call the f(x) program several times. The root-finder program will also need a few data registers for its own use. These registers must be ones that are not disturbed by the evaluation of f(x), so that the necessary information from previous evaluations of f(x) can be retained.

If the f(x) program uses data registers, the possibility of a register usage conflict cannot be overlooked. No matter which data registers the root-finder program uses, there will be some possible f(x) program that uses the same data registers.

One "solution" to this problem is to check the root-finder and f(x) programs for conflicting register usage, and re-write one of the programs to eliminate the conflict.

The Extended Functions/Memory module solves register usage conflicts once and for all. It allows you to write a universal root-finder program that will work with any f(x) program. (Of course the f(x) program must have a global label of 6 characters or less, so that it can be reached through an XEQ IND instruction.) Rather than leaving its essential data in the numbered registers, where it would be susceptible to alteration by the f(x) program, this root-finder saves its data in an extended memory file before calling f(x). After f(x) returns a value, the root-finder program can recall its essential data, untouched, from extended memory. This is a classic example of the power of extended memory.

The program listing below includes "SOLVE" plus two other routines that will be covered in the next two sections. These three routines are combined into one program because they use some of the same instruction sequences, and because they will often be used together. Key in the program exactly as shown, so that you may try the examples that follow.

Extended Memory requirements:

program	free registers needed to run
"SOLVE"	4
"DERIV"	7
"INTEG"	2Ø

"SOLVE"/"DERIV"/"INTEG" program listing

```
01♦LBL "SOLVE"
02 ASTO 00
03 STO 01
04 1
05 %
06 +
07 STO 02
08 "**SOLVE"
09 4
10 XEQ 05
11 2 E-3
12 SAVERX
13 RCL 01
14 XEQ IND 00
15 "**SOLVE"
16 SAVEX
17 GETR

18♦LBL 01
19 CLX
20 SEEKPTA
21 3 E-3
22 SAVERX
23 RCL 02
24 FS? 10
25 VIEW X
26 XEQ IND 00
27 "**SOLVE"
28 GETR
29 ENTER↑
30 ENTER↑
31 X<> 03
32 -
33 X≠0?
34 /
35 RCL 02
36 ENTER↑
37 X<> 01
38 -
39 *
40 ST- 02
41 RCL 01
42 RND
43 RCL 02
44 RND
45 X≠Y?
46 GTO 01
47 LASTX
48 GTO 21

49♦LBL "DERIV"
50 ASTO 03
51 STO 04
52 RDN
53 STO 05
54 3 E-3
55 STO 06
56 "**DERIV"
57 7
58 XEQ 05

59♦LBL 02
60 CLX
61 SEEKPTA
62 6 E-3
63 SAVERX
64 RCL 06
65 INT
66 RCL 05
67 *
68 RCL 04
69 +
70 XEQ IND 03
71 "**DERIV"
72 GETR
73 STO IND 06
74 ISG 06
75 GTO 02
76 RCL 03
77 RCL 02
78 RCL 01
79 ENTER↑
80 +
81 -
82 9
83 *
84 -
85 +
86 RCL 00
87 11
88 *
89 -
90 6
91 /
92 RCL 05
93 /

94♦LBL 21
95 ENTER↑
96 PURFL
97 EMDIR
98 CLD
99 RDN
100 RTN

101♦LBL "INTEG"
102 ASTO 00
103 STO 01
104 X<>Y
105 -
106 4
107 /
108 STO 02
109 ST+ X
110 ST- 01
111 CLX
112 STO 03
113 STO 06
114 STO 07
115 SF 20
116 20
117 "**INTEG"
118 XEQ 05

119♦LBL 22
120 2
121 RCL 03
122 CHS
123 Y↑X
124 STO 05
125 ST+ 05
126 1
127 -

128♦LBL 03
129 STO 04
130 "**INTEG"
131 CLX
132 SEEKPTA
133 .019
134 SAVERX
135 FS? 49
136 OFF
137 3
138 RCL 04
139 X↑2
140 -
141 RCL 04
142 *
143 RCL 02
144 *
145 RCL 01
146 +
147 XEQ IND 00
148 "**INTEG"
149 GETR
150 1
151 RCL 04
152 X↑2
153 -
154 *
155 ST+ 06
156 1
157 RCL 04
158 RCL 05
159 +
160 X<Y?
161 GTO 03
162 RCL 03
163 STO 04
164 RDN
165 7
166 STO 05
167 SIGN
168 ST+ 03
169 -
170 RCL 02
171 *
172 RCL 06
173 *
174 3
175 *

176♦LBL 04
177 ENTER↑
178 X<> IND 05
179 ST- Y
180 RND
181 X<> Z
182 4
183 *
184 STO Z
185 DSE X
186 /
187 RCL IND 05
188 +
189 ISG 05
190 LN
191 DSE 04
192 GTO 04
193 STO IND 05
194 FS? 10
195 VIEW X
196 RND
197 R↑
198 FC?C 20
199 X≠Y?
200 GTO 22
201 LASTX
202 GTO 21

203♦LBL 05
204 SF 25
205 CRFLD
206 FS?C 25
207 RTN
208 SF 25
209 PURFL
210 FC?C 25
211 GTO 06
212 SF 25
213 CRFLD
214 FS?C 25
215 RTN

216♦LBL 06
217 "NO ROOM - EM"
218 PROMPT
219 END

405 BYTES
```

"SOLVE" Example 1:

Continuing the example given at the beginning of this section, we want to find the value of n such that SQRT(n) = 10+LN(n+1). Below this value, Machine 1 will be more economical, while above this value, Machine 2 will be cheaper to use.

The first step is to write a program to compute f(x). In this case x is the number of units to be produced, and f is the cost difference between Machine 2 and Machine 1. The following program computes the cost difference:

```
01 LBL"CDIFF"        Start with n in the X-register.
02 SQRT               SQRT(n)
03 LASTX
04 1
05 +                 n+1
06 LN                LN(n+1)
07 X<>Y
08 -                  LN(n+1)-SQRT(n)
09 10
10 +                  10+LN(n+1)-SQRT(n)
11 END
```

Now that you have the "CDIFF" program and the "SOLVE" program ready, obtaining the solution is simple:

1. Make sure the SIZE is at least 004.
2. Put the function name in the ALPHA register. (In this case press ALPHA C D I F F ALPHA.)
3. Key in an initial guess for the root finder. (In this example you can use 100. Since there is only one root, any positive value should work.)
4. Select a display mode according to the accuracy you desire. For example, if you want four significant digits, set SCI 3. If you need an accuracy of .0001 (which gives a different number of significant digits depending on the value of the root), set FIX 4. The root finder quits when two successive approximations are equal, with-

in the specified display accuracy. Do not use FIX 9, ENG 9, or SCI 9, because roundoff errors can hurt the accuracy of the formula used when you ask for too many digits. For this example, FIX 2 is sufficient.

5. Set flag 10 if you want to view the successive approximations to the root; clear flag 10 if you want the "flying goose" display.
6. XEQ "SOLVE" to start the root finder.
7. The root finder finishes with an extended memory directory. You may interrupt this directory to see the answer, but it is not necessary to do so. The EMDIR instruction was put at the end of "SOLVE" to compensate for the PURFL bug. If your extended functions are revision 1C or higher (including the CX), you may safely delete lines 95, 97, 98, and 99. This will eliminate the extended memory directory at the end of "SOLVE".

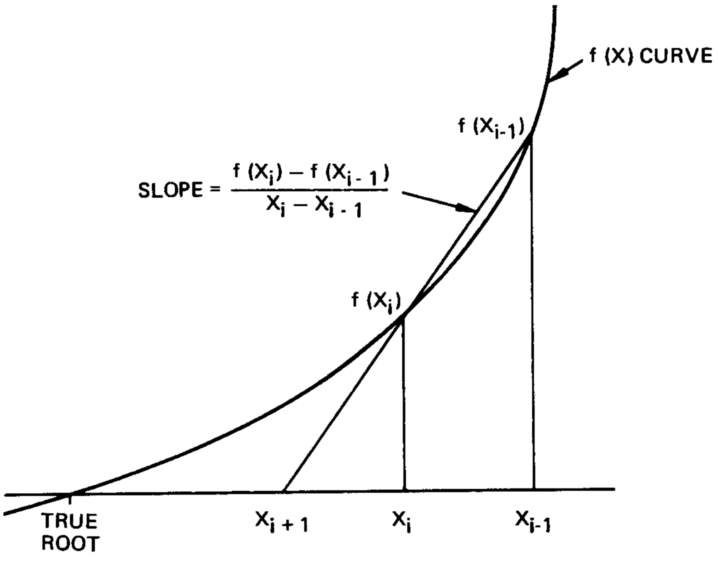
For the "CDIFF" example, the following series of approximations will be displayed if flag 10 is set:

```
101.00
215.31
236.07
239.66
239.75
```

Thus for 239 units or less, Machine 1 is better, while Machine 2 will be better for 240 units or more. You may wish to explore the effects of different initial guesses on the root finding process. You will notice that, in all cases, once the root finder gets near the correct solution, convergence is very rapid.

The Root-Finding Algorithm

The "SOLVE" program uses a simple secant algorithm to produce successive approximations x_i to the true root of f(x).



The value of f(x) at the current approximation x_i and at the previous approximation x_{i-1} are used to compute the next approximation,

$$x_{i+1} = x_i + \frac{(x_i - x_{i-1}) * f(x_i)}{(f(x_i) - f(x_{i-1}))} \quad .$$

There are some ill-behaved functions that will give this algorithm trouble, but in cases of practical interest you are not likely to encounter such functions. Therefore the complexity required to deal with such functions has not been included in "SOLVE".

You should also be aware that as x_i and x_{i-1} get very close to each other (within $10^{-9}x_i$), the calculator cannot accurately compute the difference. Multiplication by the number $x_i - x_{i-1}$ can then cause a substantial error in the calculated value of x_{i+1}. This is why you should not request 1Ø-digit accuracy from "SOLVE".

Line-by-line analysis of "SOLVE"

The first 7 lines of "SOLVE" store the function name and initial guess, and compute the second guess as 1.Ø1 times the first guess. You can change this to permit user input of the second guess, if you like. The register usage of "SOLVE" is:

Register	Contents
ØØ	function name
Ø1	previous approximation x_{i-1}
Ø2	current approximation x_i
Ø3	$f(x_{i-1})$

The LBL Ø5 subroutine sets up a 4-register data file called "**SOLVE". This requires more than a simple CRFLD (create file -- data) instruction, because the program should be able to automatically handle the case in which a file named "**SOLVE" already exists in Extended Memory. This case can result when the "SOLVE" program is terminated abnormally, before the PURFL instruction on line 2Ø9 can be executed.

Lines 11 and 12 save the contents of registers ØØ through Ø2 in Extended Memory in preparation for calling the f(x) program (which may alter the contents of these registers). Then f(x) is evaluated at the initial guess $x_Ø$. Lines 15 - 17 save the value of $f(x_Ø)$ in the fourth register of the "**SOLVE" data file and use GETR to bring all the data back. At this point all four registers are initialized, and the iterative procedure can begin.

Lines 19 - 22 save the contents of registers ØØ through Ø3 in preparation for executing f(x). This is not necessary the first time through the LBL Ø1 loop, but it will be necessary in the subsequent iterations. If flag 1Ø is set, x_i is VIEWed before f(x) is called. After the evaluation of f(x), GETR brings back the contents of registers ØØ through Ø3.

Lines 29 through 4Ø update the contents of these registers as shown:

Register	Old contents	New contents
ØØ	function name	function name
Ø1	x_{i-1}	x_i
Ø2	x_i	$x_{i+1} = x_i + \dfrac{(x_i-x_{i-1})*f(x_i)}{(f(x_i)-f(x_{i-1}))}$
Ø3	$f(x_{i-1})$	$f(x_i)$

Next the contents of registers Ø1 and Ø2 are extracted, rounded, and compared. If the rounded versions are equal, execution halts with the LBL 21 "cleanup" routine. Otherwise the LBL Ø1 loop is repeated.

The LBL 21 sequence first purges the "**SOLVE" file from extended memory. This sets up a dangerous situation if you have a revision 1B Extended Functions/Memory module, due to what is known as the PURFL bug. After PURFL is executed, there is no "working" file. This might not seem like a problem, but if you accidently execute an instruction like SEEKPT or SAVERX that operates on the working file, disaster will strike: Your entire extended memory directory will vanish! Section 1ØE gives more details and an outline of how this DIR EMPTY condition can be fixed using synthetic programming techniques. This problem does <u>not</u> occur with revisions 1C and higher, including the HP-41CX.

To avoid catastrophe, the LBL 21 sequence re-establishes a working file the only way it can without manual intervention, with an EMDIR instruction. However, executing EMDIR in a program has two undesirable side effects. First, displaying the directory takes valuable time. Second, when the directory is complete, the number of free registers in extended memory is placed in X. Since we want the "SOLVE" result to be in X, the EMDIR instruction is preceded by ENTER↑ and followed by RDN. This way, even if you choose to interrupt the directory and restart the program (as you might if "SOLVE" were being used as a subroutine of another program), the X register will still contain the correct result.

If you have revision 1C or higher extended functions, including the CX, you should delete lines 95, 97, 98, and 99. This will save a significant amount of execution time.

These details on the LBL 21 sequence may not be of immediate interest to you, but they are provided for your reference. If you write a program that uses a temporary data file, and you want it to be usable with revision 1B extended functions, you will have to deal with the same situation.

<u>"SOLVE" Example 2:</u>

Find the second zero of $J_3(x)$, the Bessel function of the first kind, order three. This is the second non-zero value of x for which $J_3(x)=0$. Of course you will need a copy of the Bessel function program "JNX" from page 5. Since you want to compute $J_3(x)$, you could construct a "shell" program:

```
        01 LBL "J3X"
        02 3
        03 X<>Y
        04 XEQ "JNX"
        05 END
```

This program simply takes the value of x that it is given, and calls "JNX" with Y=3. This simple (even trivial) technique is often used with programs like "SOLVE". If you have a function that needs more than a single input, you must either modify that program or create a "shell" routine to use with "SOLVE". The name of the shell routine should be no longer than <u>six</u> characters, so that the XEQ IND instruction in "SOLVE" will work properly. In fact, it is good HP-41 programming practice to avoid using 7-letter alpha labels wherever your programs might need to be called indirectly as subroutines.

Now let's get on with the example. First we need to know something about the behavior of $J_3(x)$. Assign "J3X" to a key (shift ASN ALPHA J shift 3 X ALPHA key) and try a few values of x to get a general idea of what $J_3(x)$ looks like. But beware that the "JNX" program gives DATA ERROR when x=Ø.

x	$J_3(x)$
Ø.Ø1	2.1E-8
1	Ø.Ø2
2	Ø.13
3	Ø.31
4	Ø.43
5	Ø.36
6	Ø.11
7	-Ø.17
8	-Ø.29
9	-Ø.18
1Ø	Ø.Ø6

From these points, it is apparent that the second zero of $J_3(x)$ is located between x=9 and x=1Ø.

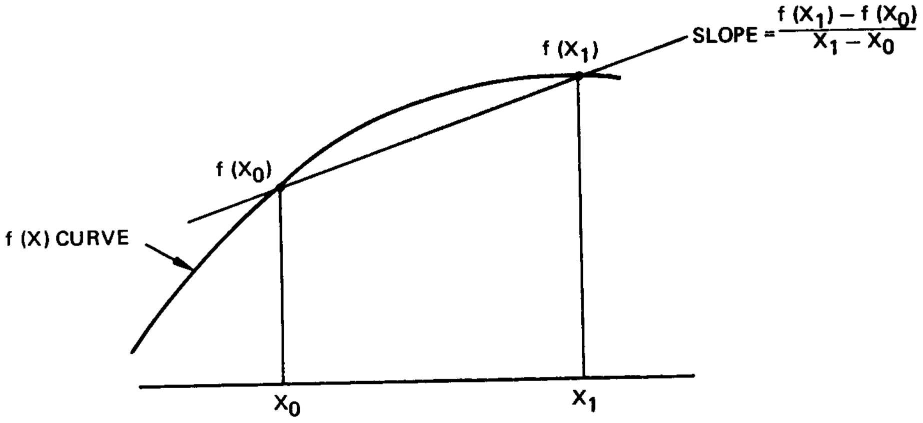
To find the value more exactly, set a SIZE of ØØ8 or more, select FIX 8 display mode, load the ALPHA register with "J3X", key in an initial guess of 9.5, SF 1Ø, and press XEQ "SOLVE". You will see the following series of approximations:

```
9.595ØØØØØ
9.76155455
9.761Ø2548
9.761Ø2313
```

For further information on root-finding, including a discussion of more sophisticated algorithms, consult any good book on Numerical Analysis or read the writeup of the "SV" program in the PPC ROM User's Manual (see Appendix C for a description of the PPC ROM).

6B. Numeric Differentiation

Many types of problems, particularly those involving maximization and minimization, require numeric evaluation of the derivative of a function. The preferred technique is first to use the rules of calculus to construct an equation for the derivative, then to write a program to evaluate the equation. If the function does not have a simple closed-form expression, numeric methods can be used. The simplest such method is to evaluate the function at two closely-spaced points, and to compute a slope based on these two values.



A much more accurate estimate of the first derivative is given by the expression

$$f'(x) = [2f(x+3h)-9f(x+2h)+18f(x+h)-11f(x)]/6h \quad .$$

This estimate is <u>exact</u> for any polynomial function of degree three or less. Otherwise the error term is of the order of h^3. However, it does not automatically follow that you should use the smallest possible value for h. The problem is that roundoff error can and will destroy the accuracy that would otherwise be obtained by decreasing the value of h. A typical example illustrating this roundoff error effect will be given in Example 2 of this section.

Because of subtraction roundoff error, the most accuracy that can be expected in the derivative estimate is 6 digits. Any further accuracy is purely coincidental.

The "DERIV" program that is part of the "SOLVE"/"DERIV"/"INTEG" package evaluates the above equation for f'(x) by calling the user-supplied f(x) program four times. Like "SOLVE", the "DERIV" program protects its data from the f(x) program by creating a temporary extended memory data file. "DERIV" is therefore compatible with <u>any</u> user-supplied f(x) program.

<u>Instructions for "DERIV"</u>

Using "DERIV" is similar to using "SOLVE". The SIZE should be Ø07 or more. The name of the user-supplied function should be in the ALPHA register. The Y register should contain the step size **h**, which can be positive or negative. This allows derivatives to be evaluated where the function is discontinuous on one side. Use h>Ø to evaluate the derivative to the right, or h<Ø to evaluate the derivative to the left. The X register should contain the value of x at which f'(x) is to be estimated.

Executing "DERIV" then produces the derivative estimate. The accuracy of the estimate depends on the step size (see Example 2), but in no case can more than 6-digit accuracy be expected. The display setting does not affect the accuracy, since no rounding is performed.

<u>"DERIV" Example 1:</u>

Verify the following derivative properties of Bessel functions:

$$J_0'(x) = -J_1(x), \text{ and}$$
$$J_1'(x) = J_0(x)-J_1(x)/x \quad .$$

First you will need to construct "shell" functions for $J_0(x)$ and $J_1(x)$:

```
01 LBL "J0X"          01 LBL "J1X"
02 0                  02 1
03 X<>Y               03 X<>Y
04 XEQ "JNX"          04 XEQ "JNX"
05 END                05 END
```

To estimate the derivative of $J_0(x)$ at x=1, press .01, ENTER↑, 1, ALPHA J shift 0 X ALPHA, XEQ ALPHA D E R I V ALPHA. The step size of .01 gives approximately 6-digit accuracy. Step sizes of 0.1 or .001 give 4-digit accuracy, which is quite reasonable too.

Compare your results to the following table:

	Derivative estimates		True derivative	
x	$J_0'(x)$	$J_1'(x)$	$-J_1(x)$	$J_0(x)-J_1(x)/x$
1	-0.440050350	0.325147317	-0.440050586	0.325147101
2	-0.576724900	-0.064471700	-0.576724808	-0.064471625
3	-0.339059100	-0.373071483	-0.339058958	-0.373071608
4	0.066043167	-0.380638968	0.066043328	-0.380638978
5	0.327579167	-0.112081133	0.327579138	-0.112080944

These results were obtained by using the program "COMPARE", which automates the entire procedure. When used with a printer, a program like "COMPARE" can save many minutes of manual keypunching and writing down results. Instead, you can just turn on the printer, start the "COMPARE" program, and come back 15 minutes later to check the results.

"COMPARE" program listing

```
01♦LBL "COMPARE"
 02 1.005
 03 STO 09

 04♦LBL 01
 05 RCL 09
 06 INT
 07 STO 10
 08 "X="
 09 ARCL X
 10 AVIEW
 11 "J0X"
 12 .01
 13 X<>Y
 14 XEQ "DERIV"
 15 "J0E="
 16 ARCL X
 17 AVIEW
 18 .01
 19 RCL 10
 20 "J1X"
 21 XEQ "DERIV"
 22 "J1E="
 23 ARCL X
 24 AVIEW
 25 RCL 10
 26 XEQ "J1X"
 27 CHS
 28 "J0T="
 29 ARCL X
 30 AVIEW
 31 RCL 10
 32 /
 33 X<> 10
 34 XEQ "J0X"
 35 RCL 10
 36 +
 37 "J1T="
 38 ARCL X
 39 AVIEW
 40 ADV
 41 ISG 09
 42 GTO 01
 43 END

   115 BYTES
```

Line-by-line analysis of "DERIV"

The data register usage of "DERIV" is

register	contents
ØØ	f(x)
Ø1	f(x+h)
Ø2	f(x+2h)
Ø3	function name -> f(x+3h)
Ø4	x
Ø5	h
Ø6	i, a loop counter (originally Ø.ØØ3).

When "DERIV" starts, the stack and ALPHA contents are

register	contents
Y	h, the step size
X	x, the point at which to estimate f'(x)
ALPHA	function name.

Lines 49-55 store these inputs in the appropriate registers. Lines 56-58 set up a 7-register data file called "**DERIV" in extended memory. Lines 59-75 constitute the loop which is executed four times to evaluate f(x), f(x+h), f(x+2h), and f(x+3h). Register Ø6 contains the ISG counter for this loop. The counter is also used as a pointer indicating where the result is to be stored (line 73).

Within the loop, lines 59-63 save the contents of registers 00 through 06 in extended memory. Lines 64-70 calculate f(x+ih). Actually the INT function must be used to chop off the .003 from the loop counter i. Lines 71-73 recall registers 00-06 from extended memory and store the result f(x+ih) in data register i. Lines 74-75 cause the loop to be repeated for the next value of i, until the function f has been evaluated at all four points.

The last step in computing the derivative estimate is to use the four results in registers 00-03 to form the result

$$f'(x) = [2f(x+3h)-9f(x+2h)+18f(x+h)-11f(x)]/6h.$$

The factorization that is used in the computation is

$$f'(x) = \{f(x+3h)+f(x+3h)-9[f(x+2h)-2f(x+h)]-11f(x)\}/6h.$$

The "DERIV" program ends with the same EMDIR sequence as "SOLVE". The same option to interrupt the extended memory directory applies. As with "SOLVE", you can delete the EMDIR instruction if you have revision 1C or higher.

<u>"DERIV" Example 2:</u>

Use "DERIV" to compute the derivative of the function

$$f(x) = x^4+10x^3+100x^2+1000x+10000$$

at x=1. Show how the estimate's accuracy varies with step size.

From differential calculus, the derivative of f(x) is

$$f'(x) = 4x^3+30x^2+200x+1000$$
$$= 1234 \text{ at } x=1.$$

Check your results against the following table:

Step size	Derivative estimate	
1.0	1240.000000	
.1	1234.006000	
.01	1234.000000	(this is the optimum step size)
.001	1234.016667	
.0001	1233.833333	
.00001	1233.333333	

If you find it difficult to construct a program to evaluate f(x), study the sequence below. It uses the factorization

$$f(x) = (((x+10)x+100)x+1000)x+10000.$$

This factorization technique can be applied to any polynomial function.

```
01 LBL "FX"
02 ENTER↑
03 ENTER↑
04 ENTER↑
05 10
06 +
07 *
08 1E2        (press 1 EEX 2)
09 +
10 *
11 1E3
12 +
13 *
14 1E4
15 +
16 END
```

Finding the derivative at x=1 is simple. Just press ALPHA F X ALPHA, key in the step size, ENTER↑, the number 1, and XEQ "DERIV". Your results should agree with the table above. You may even wish to automate the procedure by writing a short program like this:

```
01 LBL "STEP"
02 "FX"
03 1
04 XEQ "DERIV"
05 END
```

Note that the ENTER↑ is not included because lines 01 and 02 enable the stack lift. Consult your Owner's Manual for details on stack lift.

"DERIV" Accuracy

As the preceding example showed, the accuracy of the derivative estimate gets better as the step size decreases, but then gets worse if the step size is made too small. It is possible to write a program that calls "DERIV" repeatedly, decreasing the step size each time. The series of derivative estimates D_i should have the following properties:

1) D_i should be monotonic, and
2) $| D_i - D_{i-1} |$ should be monotonic and decreasing.

When the step size gets so small that one of these conditions is violated, the previous derivative estimate D_{i-1} is the best available estimate. This approach is used in the PPC ROM routine "FD", where a factor of Ø.7 is used to decrease the step size in each iteration. Check page 146 of the PPC ROM User's Manual for more details. The main disadvantage of this approach is that it greatly slows the evaluation of the derivative. In most cases, including maximization and minimization, the additional accuracy is not needed. Moreover, if "DERIV" is to be called by "SOLVE", each derivative evaluation should be as fast as possible. Your application may even warrant using the very simple estimate

$$f(x) = [f(x+h)-f(x)]/h \quad ,$$

which is about twice as fast as "DERIV".

"DERIV" Theory

The expression $f'(x) = [2f(x+3h)-9f(x+2h)+18f(x+h)-f(x)]/6h$ is one of a class of derivative estimates. These estimates can be derived through a Taylor series expansion of f(x). For example, a four-point second derivative estimate can be derived as follows:

$$\begin{aligned}
f''(x) &= a_0f(x)+a_1f(x+h)+a_2f(x+2h)+a_3f(x+3h) \\
&= a_0f(x) \quad +a_1f(x) \qquad +a_2f(x) \qquad +a_3f(x) \\
&\qquad +a_1hf'(x) \qquad +2a_2hf'(x) \qquad +3a_3hf'(x) \\
&\qquad +a_1\frac{h^2}{2}f''(x) \qquad +4a_2\frac{h^2}{2}f''(x) \qquad +9a_3\frac{h^2}{2}f''(x) \\
&\qquad +a_1\frac{h^3}{6}f^{(3)}(x)+8a_2\frac{h^3}{6}f^{(3)}(x) \quad +27a_3\frac{h^3}{6}f^{(3)}(x)
\end{aligned}$$

Fourth and higher derivatives of f(x) are omitted from the Taylor series expansion because a four-point estimate does not allow derivatives beyond the third to be considered. If the above equation is to be true for all values of h, the following equations must be true of the coefficients a_0, a_1, a_2, and a_3:

$$\begin{aligned}
a_0 + a_1 + a_2 + a_3 &= 0 \\
a_1 + 2a_2 + 3a_3 &= 0 \\
a_1 + 4a_2 + 9a_3 &= 2/h^2 \\
a_1 + 8a_2 + 27a_3 &= 0
\end{aligned}$$

These four equations are sufficient to define the coefficients. This also shows why four coefficients are not sufficient to consider fourth and higher derivatives. When this system of equations is solved, the result is the four-point estimate:

$$f''(x) = [2f(x)-5f(x+h)+4f(x+2h)-f(x+3h)]/h^2 \quad .$$

You may wish to write your own "DERIV2" program analogous to "DERIV" and based on this formula.

Another interesting exercise is to derive the equation for a four-point estimate of f'(x) and verify that it is the same one used by "DERIV".

<u>"DERIV" Example 3:</u>

Find the maximum value of $J_1(x)-J_\emptyset(x)$, which occurs just past the first peak of $J_1(x)$. Although this function is analytically differentiable, this example is meant to illustrate how to use the "SOLVE"/"DERIV" combination. The idea is to "SOLVE" for the value of x that makes the derivative of $J_1(x)-J_\emptyset(x)$ equal to zero.

The following "shell" routines are needed for the solution:

```
Ø1 LBL "J1-JØ"
Ø2 Ø               This program makes use of the fact
Ø3 X<>Y            that J1(x) ends up in the Y
Ø4 XEQ "JNX"       register after JØ(x) is
Ø5 -               calculated by "JNX".
Ø6 END
```

```
Ø1 LBL "DJ1-JØ"
Ø2 "J1-JØ"         This program uses "DERIV" to compute
Ø3 .Ø1             the first derivative of J1(x)-JØ(x).
Ø4 X<>Y
Ø5 XEQ "DERIV"
Ø6 END
```

After keying in these shell routines, press "DJ1-JØ", 2, FIX 4, XEQ "SOLVE". This will compute the location of the peak of $J_1(x)-J_Ø(x)$. To find the value of $J_1(x)-J_Ø(x)$ at the peak, leave the location in the X register and XEQ "J1-JØ". Your result should be:

location of peak	peak value of $J_1(x)-J_Ø(x)$
2.9386	Ø.6ØØ2

High accuracy in determining the location of the peak is not necessary to find the value at the peak. The flatness of the function in the vicinity of the peak is very forgiving of errors in x.

6C. A Universal Integration program

Integration, like root-finding, is a frequent application of programmable calculators. The integration program presented here, "INTEG", starts with a user-supplied function g and user-supplied values **a** and **b**. Then "INTEG" calculates

$$\int_a^b g(z)\, dz \quad .$$

Together with the root-finder program "SOLVE", "INTEG" allows equations of the form

$$\int_a^b g(x,z)\, dz = c$$

to be solved for x. The process of integration usually requires many more evaluations of the user-supplied function g(z) than would differentiation or root-finding. This makes the "INTEG" program much slower to give an answer than either "SOLVE" or "DERIV". The particular algorithm used in "INTEG" is the same one used in the PPC ROM program "IG", and is very similar to the algorithm used by the HP-34C's "integrate" function.

Instructions for "INTEG"

To calculate the integral of the function g(z) from z=a to z=b, put the name of the program that calculates g(z) in the ALPHA register, key in **a** ENTER↑ **b**, then XEQ "INTEG". A SIZE of 020 or more is required. The display setting determines the accuracy of the result and the amount of time that the calculation will take, just as with "SOLVE". The calculation is an iterative procedure that halts when two successive estimates are equal, when rounded to the current display setting. If you set flag 10, the successive estimates will be VIEWed (and printed if a printer is attached). The time needed to compute each new estimate is approximately the same as the total time already used. That is to say, the total elapsed time doubles at each step. So be sure not to specify more accuracy than you really need.

"INTEG" Example 1:

Use "INTEG" to calculate the integral

$$\int_{-1/2}^{1/2} 3(1-z^2)^{-1/2}\,dz$$

to 6 significant digits.

The first step is to write a short program to calculate the integrand:

```
01 LBL"I1"
02 X↑2
03 1
04 X<>Y
05 -
06 SQRT
07 1/X
08 3
09 *
10 END
```

Next set the display mode to SCI 5. Since "INTEG" repeatedly refines its estimate until two successive estimates are equal when rounded, SCI 5 mode will yield an accuracy of approximately 6 digits. Next, place the function name "I1" in the ALPHA register. Set flag 10 so that you will see the sequence of estimates. Then key in the limits of integration .5 CHS

ENTER↑ .5 and XEQ "INTEG". You should see the following estimates:

```
3.00000        00
3.14601        00
3.14214        00
3.14158        00
3.14159        00
3.14159        00
```

If you get impatient with the progress, you can press R/S to interrupt "INTEG", change the display mode, and press R/S to restart. The correct answer for this integral is PI.

"INTEG" is compatible with functions that are not defined at the limits of integration, because it never tries to evaluate the integrand at the limits. The next example illustrates this.

<u>"INTEG" Example 2:</u>

Evaluate

$$\int_0^1 z^{-1/2}\,dz$$

to 4 significant digits.

Solution:

```
01 LBL"I2"
02 SQRT
03 1/X
04 END
```

Press SCI 3, "I2", Ø ENTER↑ 1, XEQ "INTEG". The results should be:

```
          1.414
          1.71Ø
          1.865
          1.934
          1.967
          1.984
          1.992
          1.996
          1.998
          1.999
          1.999     (The true value of the integral is 2.)
```

This example illustrates how slow convergence can be when the integrand increases without bound at one or both limits of integration. The maximum number of iterations that "INTEG" allows is 13, encompassing $2^{13}-1$ evaluations of the integrand. This maximum number was chosen because it will take over 8 hours to complete, even with the simplest integrand. If you are very patient and want to try more iterations, simply change lines 116 and 133 to reflect the increased usage of data registers. Of course, you need not make any changes to "INTEG" if the integrand function does not alter data registers 2Ø and up.

"INTEG" Theory

The algorithm used by "INTEG" is the same one used by the PPC ROM program "IG", and it is very similar to the algorithm used by the HP-34C. The essence of the algorithm is repeated interval-halving. First the integrand g(z) is evaluated at the midpoint and an estimate of the integral is produced. Then g(z) is evaluated at two more points between the midpoint and the limits of integration to produce a 3-point histogram estimate of the integral. In the third iteration, 4 more points are added straddling the previous 3 points. With each step, the number of points is approximately doubled.

There are two improvements that are applied to this integration procedure. Two successive histogram estimates are used to construct a Simpson's rule estimate, without any more evaluations of g(z). Two successive Simpson's rule estimates are used to construct a Newton-Cotes estimate. This refinement of estimates is continued for a total of k-1 refinements at the kth iteration.

The second improvement to the integration procedure is the use of non-uniform sampling points to cover the interval of integration more quickly. The non-uniform sampling is implemented by performing the change of variables

$$I = \int_a^b g(z)\ dz$$

$$= 3(b-a)/4 * \int_{-1}^{1} f\{u(3-u^2)(b-a)/4+(a+b)/2\}*(1-u^2)\ du$$

and using a uniform sampling procedure for the new integrand as a function of u. Compare the uniform interval-halving for the u_i's to the non-uniform spacing of the z_i's for (a,b)=(-1,1):

u_i	z_i
Ø	Ø
$\pm.5$	$\pm.6875$
$\pm.25$, $\pm.75$	$\pm.3672$, $\pm.9141$
$\pm.125$, $\pm.375$, $\pm.625$, $\pm.875$	$\pm.1865$, $\pm.5361$, $\pm.8154$, $\pm.9775$

Although the z_i's approach the limits of integration much more quickly than the u_i's, the penalty is that the simple histogram estimate becomes more difficult to calculate. Each value $g(z_i)$ must be weighted by the width of the sub-interval centered at z_i.

The full explanation of how "INTEG" works would take several more pages. If you want to find out more, consult the following references:

1. PPC ROM User's Manual, pages 222-224 under the writeup for the "IG" program.
2. P.J. Davis and P. Rabinowitz, "Methods of Numerical Integration", Section 6.3, Academic Press, New York, 1975.
3. W.M. Kahan, "Handheld Calculator Evaluates Integrals" [HP-34C], Hewlett-Packard Journal, August 1980.
4. B. Carnahan, H.A. Luther, and J.O. Wilkes, "Applied Numerical Methods", Section 2.7, J. Wiley, New York, 1969. The notation in this reference is
   $T_{i,j} = M(i+j-1,j-1)$.

<u>Formulas used by "INTEG"</u>

Iteration number $k = 0, 1, 2, \ldots$

For each value of k, the following are calculated:

First sample point $u_0 = -1+2^{-k}$

ith sample point $u_i = u_{i-1} + 2^{1-k}$

kth histogram estimate $M(k,0)$

$$S_k = S_{k-1} + \int_{0}^{2^k-1} (1-u_i^2)*f\{u_i(3-u_i^2)(b-a)/4 + (a+b)/2\}$$

$$M(k,0) = [3(b-a)/4]*2^{-k}*S_k$$

Refinement of estimates uses this formula for j=1, 2, ... k,

$$M(k,j) = M(k,j-1) + [M(k,j-1)-M(k-1,j-1)]/[4^j-1]$$

The series of estimates M(k,k) converges to the true value of the integral I. Actually, the estimates M(k,Ø) also converge to I, but the convergence of M(k,k) is faster. The amount of computation needed to construct M(k,k) is quite small compared to the amount of computation needed to construct M(Ø,Ø), M(1,Ø), ..., M(k,Ø), which are needed to get M(k,k).

Line-by-line analysis of "INTEG"

The "INTEG" program is a modified and re-optimized version of the PPC ROM program "IG", which was written and revised by PPC members Read Predmore and John Kennedy, respectively. Bytes have been saved in a few places, one fewer data register is used, and, most importantly, the capabilities of Extended Memory make "INTEG" compatible with any user-supplied integrand function.

The data register usage of "INTEG" is

register	contents	contents for LBL Ø6 loop
ØØ	function name	
Ø1	(a+b)/2	
Ø2	(b-a)/4	
Ø3	k	
Ø4	u_i	DSE loop count k-j
Ø5	$u_{i+1}-u_i = 2^{1-k}$	M(k,j) register number
Ø6	S_{k-1}	
Ø7	M(Ø,Ø)	
Ø8	M(1,Ø) -> M(1,1)	
Ø9	M(2,Ø) -> M(2,1) -> M(2,2)	
1Ø	M(3,Ø) -> M(3,1) -> M(3,2) -> M(3,3)	
etc.		

When "INTEG" starts, the stack and ALPHA contents are

register	contents
Y	a, the lower limit of integration
X	b, the upper limit of integration
ALPHA	function name.

Lines 1Ø1-115 use these inputs to initialize the data registers. Flag 2Ø is set so that the termination test will be bypassed for k=Ø (line 198). Next, a 2Ø-register data file called "**INTEG" is created in extended memory. The LBL Ø5 subroutine automatically handles the case in which a file named "**INTEG" already exists in extended memory.

The LBL 22 sequence computes $u_Ø$ and the increment $u_{i+1}-u_i = 2^{1-k}$. The LBL Ø3 loop is where most of the time is spent. First the registers are saved in preparation for calculating $g(z_i)$. Lines 137-146 calculate $z_i = u_i(3-u_i^2)(b-a)/4 + (a+b)/2$. Next $g(z_i)$ is evaluated and the register contents are restored. The value of $g(z_i)$ is weighted by $(1-u_i^2)$ before being added into the current sum S_k. Lines 155-16Ø calculate $u_{i+1} = u_i + 2^{1-k}$ and exit the LBL Ø3 loop when the result becomes greater than 1, and thus outside the limits of integration.

Lines 135 and 136 are provided to prevent memory loss due to low battery voltage. Some integral evaluations can run a very long time, possibly even exceeding the life of a fully-charged battery pack. If a weak battery condition halts "INTEG" at line 136, you can change to a fresh battery pack and press R/S to continue. If you don't have a spare battery pack, it should be safe to leave the batteries out for several hours. Most new HP-41's will retain their memory contents for much more than 24 hours after the batteries are removed. Even the oldest HP-41C's appear to be good for at least 8 hours. But don't turn on the calculator while the battery pack is removed! That will almost certainly result in MEMORY LOST.

At the completion of the LBL Ø3 loop, X is $1+2^{-k}$ and Y is 1. Lines 162-175 set up the data registers for the LBL Ø4 loop, which applies the M(k,j) formula to the previously calculated values M(k-1,Ø), M(k-1,1), ..., M(k-1,k-1), which are stored in registers Ø7 and up. First, k is stored in register Ø4 as a DSE counter, whose value is k-j. The first time through the loop, j is zero. The number 7 is stored in register Ø5 as a pointer to the register containing M(k-1,Ø). This pointer will be incremented each time through the LBL Ø4 loop, just as the counter in register Ø4 will be decremented. Lines 167 and 168 increment k for the next time through the LBL 22 loop. Line 169 obtains 2^{-k} by subtracting 1 from the value that was in X at the completion of the LBL Ø3 loop. Lines 17Ø-175 compute $M(k,Ø) = 3*2^{-k}*S_k*(b-a)/4$. Note that at this point Y still contains 1. The Y register will be used in the LBL Ø4 loop to hold the value 4^j, which starts at 1 for j=Ø the first time through the loop.

At the top of the LBL Ø4 loop, X contains M(k,j) and Y contains 4^j. After lines 177-186, X contains [M(k,j)-M(k-1,j)] /[4^j-1], Y contains $4*4^j = 4^{j+1}$, and Z contains a rounded version of M(k-1,j). Next M(k,j) is added to X, producing the value M(k,j+1). Lines 189-192 increment the register counter and decrement register 4. This sets up conditions for the next time through the LBL Ø4 loop with the next value of j. After k times through the loop, the GTO Ø4 instruction is skipped with the value M(k,k) in X and the rounded version of M(k-1,k-1) in Z and T. M(k,k) is stored in the proper register, then it is rounded and compared to M(k-1,k-1). If the two rounded versions are equal, M(k,k) is extracted from LASTX and returned as the result. Otherwise a branch to LBL 22 begins the calculation of M(k+1,Ø) and eventually M(k+1,k+1).

Like "SOLVE" and "DERIV", "INTEG" ends with an EMDIR instruction, so that there will be a working file when the program halts. This is only needed for revision 1B extended functions.

"INTEG" Example 3:

Verify that

$$\int_2^3 J_1(x)\,dx = J_0(2) - J_0(3) \quad .$$

Solution:

Set FIX 4 and key in the "shell" function

```
01 LBL"J1X"
02 1
03 X<>Y
04 XEQ"JNX"
05 END
```

Press 2 ENTER↑ 3, "J1X", SF 10, and XEQ "INTEG". The result is:

```
0.4971
0.4831
0.4839
0.4839
```

To compare this to the true value of the integral, press

0 ENTER↑ 2 XEQ "JNX"

to compute $J_0(2)=0.2239$. Next press

0 ENTER↑ 3 XEQ "JNX"

to compute $J_0(3)=-0.2601$. The difference is 0.4840, which agrees with the computed value of the integral.

CHAPTER SEVEN
A MAILING LIST PROGRAM

The "NAP" (Name/Address/Phone) program presented in this chapter illustrates how text files can be used to save blocks of ALPHA data in an organized manner. The program was written by Alan McCornack. It is included here primarily as a learning tool, but you may find it suitable for general use. Its relatively straightforward approach can be a model to help you develop a program to manage your own data base. Alan's use of GETKEY in the program's Edit section is also instructive, showing how the use of GETKEY can be confined to a portion of a program if desired.

The "NAP" program assumes that the name, address, and phone number data for each entry in the list has the following format:

item	maximum length
Name	24 chars.
Address	
line 1	24 chars.
line 2	24 chars.
line 3	24 chars.
Phone number	22 chars.
Miscellaneous	
information	22 chars.

The 24-character length limit is imposed so that each item may be fully listed on a single printed line. The last two items are limited to 22 characters so that they may appear on a single line after being indented two spaces. Any characters may be used, including special characters accumulated in the ALPHA register using XTOA.

The "NAP" program forms a block of 6 records for each entry. The record contents are as shown, where **n** is the entry number:

record number	contents
6n	name
6n+1	address line 1
6n+2	address line 2
6n+3	address line 3
6n+4	phone number
6n+5	miscellaneous data

Entries are numbered in sequence, starting with zero. Thus, if you have 1Ø entries, the text file will contain 6Ø records. The entries will be numbered Ø to 9.

Instructions for using the "NAP" program

1. Before you use "NAP", you must create a text file named "ML" (mailing list). Pick a file size, key that number into X, put "ML" in the ALPHA register, and execute CRFLAS. The file size should be large enough to handle your projected mailing list needs. Each entry will use one register for each 7 characters of text plus another register for the 6 record separator bytes. A typical entry uses 8 to 12 registers. The maximum number of entries that can be accomodated is therefore about 5Ø to 75 with a full 6Ø3-register complement of extended memory. If this limit is too low, you may be able to do a little better by abbreviating some of the information to cut down on the number of characters used.
2. Once the "ML" ASCII file is in place, the "NAP" program provides several options. These options are available on the five top row keys in USER mode. If you already have some functions or global labels assigned to the top row, you can use the "SK" (suspend key assignments) program from Section 1ØG to temporarily de-activate them. Other-

wise you will have to clear the assignments from these keys by pressing

ASN ALPHA ALPHA (key)

or by using the "CT" program from page 84, then manually clearing the X↑2 key. The top row assignments must be cleared because global label or function key assignments are always given priority over local labels where there is a confict.

3. Switch into USER mode. Make sure that the SIZE is at least ØØ1, because the "NAP" program uses register ØØ. Set flag 26 manually or by turning the calculator off, then on. Clear flag 21 unless you have a printer attached and you want printed output.
4. Press GTO "NAP". You are now in the "NAP" program. If you have removed or suspended any top row key assignments, the top row keys will have the following functions:

		List all		
Add entry	**Find string**	**List entry**	**Delete entry**	**Edit**

The "List all" function is obtained by pressing

shift $\sqrt{x}$ (the "c" key).

Each of these functions will now be described in detail. If you are starting with an empty "ML" text file, you will want to use "Add entry" first, so press the "A" key (row 1, column 1). Actually, you can start with the "Add entry" function by using XEQ "NAP" instead of GTO "NAP".

(A) Add entry

The program will prompt for a NAME to fill the first line of the six-line entry. The program will stop in ALPHA mode, so that you can key in a name of up to 24 characters. You do not need to count the characters, because the calculator will give a warning tone as the 24th character is entered. After you have keyed in the name, press R/S to restart the program. The next prompt will ask for "ADD. L1?", line 1 of the address. Key it in and press R/S. Proceed this way, entering line 2 and line 3 of the address. If you want to leave a line blank, you can just press R/S without an entry in response to the prompt. When the prompt "PHONE?" appears, key in the phone number. Make sure that you use no more than 22 characters if you expect neat printer output. To check the number of characters, add spaces until the warning tone sounds. This indicates that 24 characters are present. Then press backarrow to get rid of the extra spaces you added. Press R/S when the entry is ready to be processed. The next prompt, "MISC.?" will be for the miscellaneous data field. This field should also be limited to 22 characters if a printer will be used. When you press R/S, the program will store this data and quit.

After you finish making a full name/address/phone entry, you can press R/S or the "A" key to add another entry.

Caution: Never quit this part of the program without adding all 6 lines of the entry. If you make a mistake, simply continue making entries until all 6 are done, then correct the mistake using the edit function (key "E") described later. If you want to delete an incomplete entry, you can use the delete function (key "D") described on the next page, but <u>only if</u> no further entries have yet been added to the file.

(B) Find string

Press "B", the 1/x key, to execute this routine. The program will prompt you for a string to find. The program stops in ALPHA mode so you can key in the string. The string can be up to 24 characters long. The program will find the string, wherever it appears in the file, as long as the string is fully contained in a single line.

If the string is not found, a value of -1 is returned in X. If the string is found, the message "ENTRY NO. n" tells you which entry contains the string. All 6 lines of the entry will then be shown.

Press R/S to halt the program if this is the entry you were looking for. Otherwise the search will automatically be continued. This second search (and all subsequent searches) will use only the first 6 characters of the string you entered. As long as you do not interrupt the searching by pressing R/S, the search will continue to display entries for which a match is found. When the END OF FL is reached and no more matches are found, the value -1 is returned to X and the program halts.

The string to be found need not be in the NAME line. It will be found on any line anywhere in the "ML" file. For example, suppose you wanted to record birthdays in the MISC line. You could use the notation "bmm/dd/yy". The lowercase **b** is a "tag" that indicates birthday data. To list all entries that have June birthdays, you could then search for the string "b6". The lowercase characters **a** through **e** make especially handy "tags" because they are easy to key in and not often used otherwise.

(C) List entry

Press "C", the $\sqrt{x}$ key, to execute this routine. The program asks for the number of the entry to be listed. Just key in the number and press R/S. Remember that the first entry is number zero, not 1. The 6 lines of the entry will be displayed in sequence, and printed if the printer is attached and flag 21 is set.

(c) List all

Press "c", shift $\sqrt{x}$, to execute this routine. No input is needed. The entire name and address list will be displayed in sequence. If the printer is attached and flag 21 is set, the list will be printed out. The phone number and miscellaneous lines will be indented by two spaces in the printed listing. Successive entries will be separated by two spaces.

At the end of the listing, the number of registers used and the total number of registers in the file will be displayed (and printed if the printer is attached and enabled).

(D) Delete entry

Press "D", the LOG key, to execute this routine. The program asks for the number of the entry to be deleted. Be sure you know the right number for the entry you want to delete. If you are in doubt, press the "C" key to check the entry. When you are sure which entry you want to delete, key in the number and press R/S. Again, remember that the first entry is number zero.

(E) Edit

Press "E", the LN key, to execute this routine. The program will ask you for the number of the entry you want to edit. Key in this number and press R/S. The program will proceed to show you each line of the entry (and print it if the printer is attached and enabled). After a pause, the prompt "OK?" will appear. If the line is OK and does not need to be changed, press "Y" or R/S. If you want to see the line again, press the ALPHA key. To quit the EDIT mode, press the backarrow key or the ON key. Press "N" (the ENTER key) if you want to change the line. If you do not press any key, the line will be displayed again and the "OK?" prompt will be repeated, just as if you had pressed the ALPHA key. If you press any key other than those mentioned above, the result will be the same as if you pressed "N".

If you press "Y" (the multiplication key) or R/S to indicate the line is OK, the next line will be displayed.

If you press "N" (the ENTER key), or any key other than

ALPHA, "Y", R/S, ON, or backarrow, the prompt "LINE?" will appear. Key in a string to replace the line and press R/S. If you want to clear the line, you can just press R/S. If you made a mistake and you do not want to change the line, press ALPHA to exit ALPHA mode, then either XEQ 84 or the "E" key to restart the editing process. The "E" key will start over at the beginning of the entry; XEQ 84 continues with the next line of the current entry.

Each line of the entry is displayed in sequence, with the "OK?" prompt. Each time you can press "Y" or "N" to indicate whether it is OK to proceed to the next line. When you have finished editing all 6 lines of the entry, the program will automatically continue on to the next entry, if there is one.

When you want to quit, simply press the backarrow key in response to the "OK?" prompt.

Line-by-line analysis of "NAP"

The "NAP" program contains several subroutines that are used by more than one local label. This technique is known as modular programming, and greatly reduces the number of bytes used in a complex task by dividing the job into sections.

Label 90 selects an initial pointer of zero (beginning of the "ML" file). LBL 91 selects a pointer to the beginning of a selected entry. Line 52 initializes the character count in the X register. The integer part of this character count will be displayed later as the number of registers occupied.

Label 92 appends the integer part of X to the ALPHA register for display purposes. The RCLFLAG and STOFLAG functions ensure that the display setting is restored. Label 98 displays the contents of the ALPHA register without printing it. Label 99 prints the contents of the ALPHA register, or displays ALPHA and pauses if the printer is off, not present, or disabled. This routine differs from the "PVA" routine presented in section 4C in that it preserves the status of flag 25, which is used in "NAP" to detect the END OF FL.

"NAP" program listing

```
01♦LBL "NAP"
 02♦LBL A
 03 XEQ 90
 04 SF 25
 05 5
 06 CHS

 07♦LBL 05
 08 6
 09 +
 10 SEEKPT
 11 FS? 25
 12 GTO 05
 13 LASTX
 14 /
 15 "NAME "
 16 XEQ 92
 17 SIGN
 18 X<>Y
 19 AON
 20 XEQ 94
 21 XEQ 93
 22 XEQ 93
 23 XEQ 93
 24 "PHONE"
 25 XEQ 94
 26 "MISC. "
 27 XEQ 94
 28 AOFF
29 "RGS. = "
 30 7
 31 /
 32 XEQ 92

 33♦LBL 98
 34 RCLFLAG
 35 CF 21
 36 AVIEW
 37 STOFLAG
 38 RDN
 39 RTN
 40 GTO A

 41♦LBL 90
 42 0
 43 GTO 06

 44♦LBL 91
45 "ENTRY NO. ?"
 46 PROMPT
 47 6
 48 *

 49♦LBL 06
 50 "ML"
 51 SEEKPTA
 52 7
 53 RTN

 54♦LBL 92
 55 INT
 56 RCLFLAG
 57 FIX 0
 58 CF 29
 59 ARCL Y
 60 STOFLAG
 61 RDN
 62 RTN

 63♦LBL 93
 64 "ADD. L"
 65 X<>Y
 66 XEQ 92
 67 X<>Y
 68 ISG Y

 69♦LBL 94
 70 "⊦?"
 71 XEQ 98
 72 " "
 73 STOP
 74 APPREC

 75♦LBL 07
 76 1
 77 +
 78 RCLPT
 79 FRC
 80 1 E3
 81 *
 82 +
 83 RTN

 84♦LBL B
 85 XEQ 90
 86 "FIND?"
 87 AON
 88 STOP
 89 AOFF
 90 ASTO 00

 91♦LBL 08
 92 POSFL
 93 X<0?
 94 RTN
 95 6
 96 /
 97 LASTX
 98 X<>Y
99 "ENTRY NO. "
 100 XEQ 92
 101 XEQ 99
 102 *
 103 SEEKPT
 104 RDN
 105 XEQ 95
 106 ARCL 00
 107 GTO 08

 108♦LBL 96
 109 SF 25
 110 ARCLREC
 111 XEQ 99
 112 CLA
 113 FS? 25
 114 GTO 07
 115 RTN

 116♦LBL C
 117 XEQ 91

 118♦LBL 95
 119 ADV
 120 CLA
 121 XEQ 96
 122 XEQ 96
 123 XEQ 96
 124 XEQ 96
 125 "  "
 126 XEQ 96
 127 "  "
 128 XEQ 96
 129 ADV
 130 CLD
 131 RTN

 132♦LBL D
 133 XEQ 91

 134♦LBL 09
 135 DELREC
 136 DSE L
 137 GTO 09
 138 RTN

 139♦LBL c
 140 XEQ 90
 141 CLA

 142♦LBL 10
 143 SF 25
 144 X<>Y
 145 SEEKPT
 146 6
 147 +
 148 X<>Y
 149 FS? 25
 150 XEQ 95
 151 FS? 25
 152 GTO 10
 153 7
 154 /
 155 FLSIZE
 156 X<>Y
 157 XEQ 92
 158 "⊦ : "
 159 X<>Y
 160 XEQ 92
 161 "⊦ RGS."

 162♦LBL 99
 163 RCLFLAG
 164 SF 25
 165 PRA
 166 RCLFLAG
 167 FS?C 21
 168 FC? 25
 169 AVIEW
 170 STOFLAG
 171 RDN
 172 FS? 21
 173 FC? 25
 174 PSE
 175 STOFLAG
 176 RDN
 177 RTN

 178♦LBL E
 179 XEQ 91

 180♦LBL 71
 181♦LBL 84
 182 SF 25
 183 GETREC
 184 FC? 25
 185 GTO 01
 186 XEQ 99
 187 "OK?"
 188 XEQ 98
 189 GETKEY
 190 CLD
 191 RDN
192 GTO IND T
 193 "LINE?"
 194 XEQ 98
 195 " "
 196 AON
 197 STOP
 198 AOFF
 199 DELREC
 200 INSREC
 201 GTO 84

 202♦LBL 00
 203♦LBL 04
 204 RCLPT
 205 INT
 206 SEEKPT
 207 GTO 84

 208♦LBL 44
 209♦LBL 01
 210 CLST
 211 END

    447 BYTES
```

Label 96 outputs a record (line) from the "ML" file. The ARCLREC function is used instead of GETREC so that the PHONE and MISC lines can be indented. Label 99 is called for the actual output. If the END OF FL was not encountered, the GTO Ø7 instruction causes the LBL Ø7 character count routine to be executed. This sequence computes the number of characters in the current record (RCLPT, FRC, 1E3, *), and adds that number plus 1 (for the record length byte) to the character counter in X. Details of ASCII file register usage can be found in Section 1ØC.

Label 95 is used by labels B, c, and C to output six consecutive records of the file. It advances the paper, clears ALPHA, then outputs the first four lines using label 96. The next two lines are preceded by 2 spaces that are loaded into the ALPHA register before label 96 is called.

Label A adds one entry to the "ML" text file. Lines Ø4-12 find the end of the file and set the pointer there. Flag 25 is used both to suppress the END OF FL error message and to test when the END OF FL is reached. Lines 15-28 prompt for the 6 lines of the entry and append the 6 records to the "ML" file. Label 93 is a byte-saving device that generates the three prompts "ADD. L1?", "ADD. L2?", and "ADD. L3?". Lines 17-18 set up a counter in Y with an initial value of 1. The ISG Y at line 68 then increments this counter each time label 93 is called. Label 94 merely appends a question mark, calls label 98 to display the prompt without printing it, loads the ALPHA register with a single space, and halts for input. If the user just presses R/S, the single space will be used for the record. Otherwise whatever ALPHA string was keyed in will be appended to the "ML" file at line 74. The LBL Ø7 sequence updates the character counter in X as described above.

When LBL A concludes, the number of registers used is displayed, rounded to the next higher integer.

Label B prompts for a string to find, storing the left-most 6 characters in register ØØ for subsequent searches.

Label Ø8 performs the search (line 92), computes and displays the entry number (lines 95-1Ø1), sets the pointer to the beginning of the entry (1Ø2-1Ø3), and calls label 95 to display the entry. The search is then resumed, using the leftmost 6 characters of the target string. When the END OF FL is reached, the POSFL instruction returns a value of -1 to X, and the test at line 93 halts the program.

Label C calls label 91 to set the pointer to a selected entry, then drops into label 95 to display the entry.

Label c first calls label 9Ø to initialize the character counter. Then it sets the pointer to the beginning of the file (line 145). Lines 146-148 prepare the pointer value that will be needed the next time through the loop. If the END OF FL was not reached, label 95 displays the 6 lines of the entry, and the GTO 1Ø proceeds to the next entry. When the END OF FL is reached (flag 25 clear), the character count in X is divided by 7, giving the number of registers occupied. Lines 155-161 construct and display or print a message that compares this number with the FLSIZE.

Label D calls label 91 to set the pointer to the beginning of an entry, then it deletes 6 records.

Label E calls label 91 to select an entry. The first record of the entry is then printed or displayed (line 186). Then the prompt "OK?" is displayed and GETKEY is executed. After the key is pressed or the time expires with no key having been pressed, lines 19Ø-192 effect a GTO IND X, branching to the label designated by the keycode. Labels 71 ("Y") and 84 (R/S) cause the next line to be displayed. Labels ØØ (no key) and Ø4 (ALPHA) set the pointer back to the beginning of the current record, and cause the current line to be redisplayed. Labels 44 (backarrow) and Ø1 (ON) cause the program to halt. If any other key is pressed, execution will continue with line 193, because flag 25 was set. This sequence prompts for a new line, then uses that string to replace the current record (lines 199-2ØØ).

CHAPTER EIGHT
TEXT EDITING ON THE HP-41

This chapter introduces a Text Editor program, called "TE", that allows any text file to be reviewed and edited with a minimum of keystrokes. Like the "HP-16" program of Chapter 9, it uses the GETKEY function to achieve a remarkable degree of user convenience and "friendliness". If you have an HP-41CX, its built-in ED function does essentially the same job as this Text Editor program. The major advantage of ED is that it responds much more quickly than "TE". Advantages of "TE" are its search and special (non-keyable) character entry features. Even if you have an HP-41CX, do not overlook the capabilities that "TE" provides.

This powerful program and its associated documentation were written by Erik Christensen, and are reproduced here by permission.

"TE" (text editor) is a text editor program for use with a HP-41C or CV that has an Extended Functions/Memory module plugged in. The program will also work with an HP-41CX. Additional Extended Memory modules are optional (up to two can be used). The HP-41 used must have at least 115 free program registers, and one free data register.

The "TE" program provides a quick way to view, add, delete, and change text in an extended memory text (ASCII) file. An edit mode is included that totally redefines the keyboard for file processing. In this edit mode, you press a certain key that corresponds to the operation you wish to perform on the file. Then you are prompted accordingly, and the editing continues. You view the file through a twelve character "window" that you can move throughout the file using different one-key operations. You are notified if an error condition occurs, but execution is not interrupted. The following pages will describe the different functions of the keys (A) - (H). The last one described will be the (E) key, for it represents the largest part of the program, including the edit mode.

As with "NAP" in Chapter 7, you need to execute "SK" or clear any key assignments from keys (A) - (H) before you can effectively use "TE". Section 10G has a full explanation. Once these keys are clear of assignments, execute "TE" to start the text editor. Keys (A) - (H) will then be redefined as follows:

row 1:	**Add file**	**chr free?**	**clear fl**	**delete fl**	**edit fl**
row 2:	**file dir**	**goto file**	**HELP**		

Here is a brief explanation of each of these functions:

(A) Add a file to memory
This routine sets up the memory allocation for a new text file in extended memory. The program prompts for the number of lines the text file is to have ("LINES?"). Key in a number, and then press R/S. The program then prompts for the total number of characters that will be allocated for the file ("CHAR"). Key in a number, and press R/S. The routine then prompts for the name that the new file shall be called ("NAME?"). Type in a file name that is from 1 to 7 letters long, and press R/S. If the name keyed in has already been used as a name for another file, then you will be reprompted for another name. If there is currently not enough free memory available to create the file, you will be prompted again for the number of lines and characters to be allocated. Upon completion, the display will show "OK" and the program will stop. You may now use any of the other local labels.

(B) Count the number of free characters in a file
This routine checks how many characters may be added to the specified file. The routine prompts for the name of the file to be analyzed ("FILE NAME"). Key in a name from 1 to 7 characters long that corresponds to an already created file, and press R/S. If you specify a name not yet used as a name for a file, or the name of a program or data file, you will be reprompted. When a file has been picked, the program will proceed to count the free characters, and then will stop with "CHAR LEFT=n" where n equals the number of free characters. Press R/S and you will see "OK" in the display. You can now use any of the other routines.

(C) Clear out the contents of a file
This routine erases all the text stored in a file, but leaves the file intact, including the name, memory allocation, and directory placement. The routine prompts for the name of the file to be cleared out ("FILE NAME?"). Key in the name of the file to be cleared (1 to 7 characters), and press R/S. If you specify a name that has not already been used in memory, or the name of a data or program file, you will be reprompted for the name of the file. The routine ends with "OK" in the display. You can now use any of the other routines.

(D) Delete a file from memory
This routine purges a text file from extended memory. This routine would be the one to use to make room for new files, by deleting old ones. The routine prompts for the name of the file to be deleted ("FILE NAME?"). Enter the name of the file to be cleared (1 to 7 characters long), and press R/S. If you specify the name of a file not already used in memory, or the name of a data or program file, you will be reprompted for the name. The routine ends with "OK" in the display. You may now use any of the other routines.

(F) View the file directory

This routine will show the names, types, and memory allocations for each file in extended memory, including data and program files. The order of viewing is the order in which the files were created, so the first one shown is the first one created. The display for each file is: **FFFFFFF TMMM** , where FFFFFFF is the 1 to 7 character name of the file, T denotes the type of file, and MMM stands for the number of 7 character registers that are allocated to the file. T can be A,D, or P, where A=ASCII text file, D=data file, and P=program file. When all the files in extended memory have been listed, the program will halt with "FREE=n" in the display, where n equals the number of free characters that can be allocated to files, in extended memory. Press R/S once more, and "OK" will be displayed. You may now use any of the other routines.

(G) Go to a file

This routine is used to position the editor to any text file in memory. Once positioned to a text file, you may perform any operation on the current file, by just hitting R/S after a "FILE NAME?" prompt. This works for routines (B), and (E). To position the editor to a specific file, answer the prompt in this routine ("FILE NAME?") with the name of the file desired, and press R/S. If the file specified is not found, or is the name of a program or data file, you will be reprompted for the name. Upon completion, "OK" will be seen in the display. You may now use any of the other routines.

(H) Help associating a key

This routine is a convenience routine to determine the function of certain local label key (A)-(G). It will display a mnemonic that corresponds to the key pressed. It will prompt with "PROBLEM KEY?". Respond by pressing the key that you are uncertain about. The display will show a 3 character mnemonic. The codes for keys (A)-(G) are as follows: (A)=ADD, (B)=FRE, (C)=CLR, (D)=DEL, (E)=ED ,(F)=DIR, (G)=GTO. The mnemonic will be shown for about one second, and then the routine will go back and ask for another key. It will continue to do so until you press another key, or leave the HP-41 unattended during the prompt. When you press any other key than (A)-(G), the routine will stop prompting and display "OK", meaning that it is possible to use any of the other routines.

(E) Edit a file

When you press the "E" key, you enter the **edit mode.** The keyboard is completely redefined for file processing. All input will be done while the program is running, except alpha character entry, for which the program will stop, and for which R/S is necessary to continue.

To select the edit mode, press the (E) key. You will see the message "FILE NAME?" in the display. Key in the name of the file to be edited (1-7 characters) and press R/S. If you specify the name of a program or data file, you will be re-prompted for another name. If a text file has already been selected using local label (G), then that file will be used if you just press R/S without a name input. After the name has been specified, you will see a portion of text in the file through a 12 character window (which is movable using various commands). The window will initially be positioned to line Ø, character Ø. You will also see the flag Ø annunciator. This means that the calculator is ready for your selection of any editing function. When flag Ø is set, you may press the key that corresponds to the function you desire. You will then be prompted for input if necessary for the selected function, and the screen will again show the window. Then you can select other keys to edit the file. When you are finished, press the R/S key or the ON key when flag Ø is set to quit the edit mode.

The edit mode keyboard is set up as shown in the table below. Each key performs a different file operation.
NOTE: No USER mode key assignments are affected by this redefined keyboard.

row Ø	**quit**	**INS X**			
row 1	**ADD L**	**BEG L**	**CHA L**	**DEL L**	**INS L**
row 2	**BACK n**	**GOTO A**	**POINT**	**INS A**	**AHEADn**
row 3	**BACK 1**	**DEL A**	**VIEW L**	**CHA A**	**AHEAD1**
row 4	**ADD A**		**GTOCHR**	**POS FL**	**DELCHR**
row 5	**UP n**	**7**		**8**	**9**
row 6	**UP 1**	**4**		**5**	**6**
row 7	**DOWN 1**	**1**		**2**	**3**
row 8	**DOWN n**	**Ø**		**GOTO REC**	**STOP/CONT**

Each edit mode operation listed on the keyboard diagram above will now be described in detail. The operations will be identified by the mnemonics on the keyboard diagram, and listed from the top to the bottom of the keyboard.

(ADD L) Add a line of text at the end of the file.
This operation will create a new line of text that you specify at the end of the file. The window will be set to the newly created text, at the end of the file. If there is not enough room allocated for the new text, then "ERROR" will be displayed, and the operation will not be performed. You will be prompted for the new text to be added ("NEW TEXT?"). Key in a string of characters that can be 1 to 24 characters in length, and press R/S. The routine will return to the window display.

(BEG L) Move the window to the beginning of the line
This routine will position the window to the first 12 characters of the line that the window is presently positioned to. If the window is already at the beginning of the line, then nothing happens. The routine does not prompt, and returns immediately to the window display.

(CHA L) Change the contents of a line
This routine will change the text of a line to new text, thus erasing the old text. You are prompted for the text that is to overwrite the previous text on the current line ("NEW TEXT?"). Respond to the prompt by keying in a 1 to 24 character string that will replace the old text, and press R/S. If there is not enough room for the new text, then "ERROR" will be displayed. The window is positioned to the beginning of the newly created text. The routine will return to the window display.

(DEL L) Delete a line of text
This routine will erase an entire line of text from memory. All subsequent lines will move up one, so the window will be positioned to the next line in memory. This routine does not prompt, and returns immediately to the window display.

(INS A) Insert text in a line
This routine will enable you to insert text at the current window position. The text inserted will appear right before the text currently shown in the window. The window will be positioned to the beginning of the newly inserted text. The prompt is "NEW TEXT?". Respond with a 1 to 24 character string of text to be inserted, and press R/S. If there is not enough room for the text to be inserted, then "ERROR" will be displayed. The routine returns to the window display.

(AHEAD n) Move window ahead n characters
This routine will move the window forward through the current line of text n characters. The routine prompts for a "NUMBER?". Enter a 3 key sequence that represents a 3 digit number for n. The routine then shows the three digit number in the display, and returns to the new window display. If the window would be moved beyond the end of the current line then "ERROR" will be displayed.

(BACK 1) Move window back 1 character
This routine will backspace the window one character in the line. If you backspace past the first character, then "ERROR" is displayed. The routine then returns to the window display.

(DEL A) Delete specified text
This routine will enable the user to delete certain strings of text from the line. The routine prompts for the text to be deleted with "OLD TEXT?". Enter a 1 to 24 character string, and press R/S. If the text to be deleted is not found, then nothing happens. The search for the text is done from the current window position to the end of the file. The routine returns to display the window.

(VIEW L) View an entire line
This routine views the current line, 12 characters at a time. It goes from the beginning of the line to the end. First it displays the line number "LINE n", where n equals the line number. Then it views the line, and returns.

(INS L) Insert a line of text
This routine will insert a line of user-specified text in the file. The text will be inserted right before the current line. After the text has been inserted, the window will be positioned to the beginning of the new line of text. If there is not enough room for the new text, "ERROR" will show up in

the display. The prompt for the text to be inserted is "NEW TEXT?". Respond with a 1 to 24 character string of letters to be inserted, and press R/S. The routine then returns to the window display.

(BACK n) Move back n characters
This routine will move the window back through the line n characters, where n is selectable. If by moving back n characters the window would be past the beginning of the text, then the window will be positioned to the beginning of the text. The prompt for n is "NUMBER?". Respond by pressing a 3 key sequence representing a 3-digit number while the program runs. The three digit number will show up in the display, and then the routine will return to display the window at its new position.

(GOTO A) Go to text in line
This routine will search the current line for a match with user specified text, and if a match is found, it will move the window to the first character of the specified text. The routine will only search the current line, and if a match is not found, the window does not change position. The prompt is "TARGET TEXT?". Respond by keying the string that is to be located (1 to 24 characters) and press R/S. The routine will then return to display the window.

(POINT) View the current pointer values
This routine will show how many lines down and how many letters from the leftmost position the window is, relative to the beginning of the file. The routine does not prompt. It displays the message "LINE n CHR m" where n is the number of lines down from the beginning, and m is the number of characters from the beginning of the line.

(CHA A) Change text in a line
This routine will allow you to replace text in a line. You are prompted for the text that is to be changed ("OLD TEXT?"). Enter a 1 to 24 character string representing the text to be changed. This text is then deleted, if it can be found. The routine then prompts "NEW TEXT?". Type in the text that is to replace the old text (1 to 24 characters), and press R/S. The new text is then inserted where the old text used to be. If there is not enough room for the new text to be inserted, then "ERROR" is displayed. The routine returns to display the window.

(AHEAD 1) Move window 1 character ahead
This routine moves the window one character ahead through the line. If the window is moved out of the text, then "ERROR" is shown. The routine returns to display the window.

(ADD A) Add text at end of line
This routine will add user specified text at the end of the current line. If there is not enough room for the new text, then "ERROR" will be displayed. The prompt is "NEW TEXT?". Enter a 1 to 24 character string that will serve as the addition to the line, and press R/S. The routine returns to display the window.

(GTOCHR) Move window to absolute character number
This routine will move the window to a specified character number (counted from the beginning of the line). If there is not a character at the specified position, then "ERROR" will be displayed. At the prompt "NUMBER?", key in 3 digits representing the position of the character within the line. The number will then be displayed, and the routine will return to display the new window.

(POS FL) Position window to specified text

This routine will search from the beginning of the file for a match with the specified text. If a match is found, then the window is positioned to the first character of the text being sought after. If a match is not found, then the window is not changed in position. The prompt for text to be located is "TARGET TEXT?". Your response is a 1 to 24 character string, followed by R/S. The routine returns to display the window at its new position.

(DELCHR) Delete n characters

This routine deletes n characters from the current line. The deleting starts at the first character displayed in the window, and proceeds n characters to the right. If n is larger than the number of characters from the beginning of the window to the end of the current line then everything from the first character of the window to the end of the line is deleted. The prompt for n, the number of characters to be deleted, is "NUMBER?". Key in a 3-digit number n, with leading zeros if necessary. This number will then be displayed briefly, and the routine will return to show the updated window display.

(UP n) Move window up n lines

This routine will move the window display up a specified number of lines. If the window would be moved past the beginning of the file (first line), then the window will be positioned to the first line. The prompt for the number of lines to be moved up is "NUMBER?". Respond with a 3 digit sequence, and the 3 digit number will be displayed. Then the routine will return to view the window display.

(UP 1) Move window up 1 line

This routine will move the window up one line of text. If the window is positioned at the first line, then nothing will happen. This routine does not prompt, and returns directly to the window display.

(DOWN 1) Move window down 1 line
This routine moves the window down one line of text in the file. If you try to move the window past the last line of text in the file, then "ERROR" will be displayed. This routine does not prompt, and returns directly to the window display.

(DOWN n) Move window down n lines
This routine will move the window display down a specified number of lines. If the window would be moved past the end of the file (last line), then the window position will be unchanged, and "ERROR" will be displayed. The prompt for the number of lines to move down is "NUMBER?". Respond with a 3 digit sequence, and the 3 digit number will be displayed. Then the routine will return to view the window display.

(GOTO REC) Move the window to a specified line
This routine will move the window to a specified line number n. Lines are numbered starting with line Ø (the first line in the file). The prompt for n is "NUMBER?". Respond with a 3 digit number for the line number n, with leading zeros if necessary. The 3 digit number will then be displayed, and the routine will return to the window display. If you try to move the window to a line of text that has not yet been created, then the display will show "ERROR" and the window position will not change.

(STOP/CONT) Exit the edit mode
This routine exits the edit mode, freeing you to use any other local label routines (A)-(H). It restores the original flag status, and ends with "OK" in the display. Either the R/S or the ON key will cause this exit routine to be executed.

(INS X) Insert a special character
This routine will insert a not normally keyable character into the file, at the current window position. The input to the routine is the ASCII code (a number between Ø-255) of the letter. A listing of these special characters and their corresponding codes can be found on pages 6Ø and 61. The routine will prompt for the character code with "NUMBER?" in the display. Key in a three digit numeric sequence that represents the character code. The code will be displayed, and then the routine will return to the window display. If there is not enough room for the new character in the file, then "ERROR" will be displayed.

If any of the preceding function descriptions were not completely clear to you, you should create a small ASCII file in which you can try out the edit mode functions. You will find that the descriptions make an excellent reference after you have actually used "TE".

CUSTOMIZATION OF "TE"

The information that follows is provided in case you want to add your own editing routines to "TE" or change an existing routine. If you do not intend to modify "TE", you can skip to the next chapter.

To add or change a routine:

1) Choose the key that your new routine is to be assigned to. Press (XEQ) (ALPHA)GETKEY(ALPHA) and that key.
2) The row/column keycode in X from step 1 is the numeric label number that will start your new routine. So, for example, if you saw 41 in the display from example 1, then your routine would start with LBL 41.
3) After the LBL at the start of your routine, you can add anything you please, followed by a RTN statement.

4) Your routine must observe the following constraints:

FLAG/REG	PRE-ROUTINE	POST-ROUTINE
Flag 29	clear	clear
Flag 25	set	set
FIX	Ø	Ø
Flag 28	set	set
Flags Ø-3,6	clear	clear
X	rrr.ccc	rrr.ccc
Y	not used	can be used
Z	not used	can be used
T	keycode	can be used
L	not used	can be used
ALPHA	window display	can be used
REG ØØ	initial flag sts	cannot be used

When your routine comes to the RTN statement, it must have the new window position in X in the form rrr.ccc, where rrr is the record number, and ccc is the character number of the first character of the window. If your routine deletes any text from the file, it should not conclude with RTN, but rather GTO 26. This calls an error handling routine.

5) To call various prompting routines, do the following:

PROGRAM STEP	DESCRIPTION	USES
XEQ Ø9	"NUMBER?" # in X	T,Z,L,ALPHA
XEQ Ø6	" TEXT?" txt in A	
XEQ Ø7	"NEW TEXT" txt in A	
XEQ Ø8	"TARGET TEST?" txt in A	

EXAMPLE OF AN ADDED ROUTINE

Add a routine that will erase all text that matches the text supplied by the user in the file. Have the prompt for the text to be deleted to be "TARGET TEXT?". This new routine is assigned to the (ALPHA) key.

ROUTINE LISTING

```
  01 LBL 04          start routine with label assignments
  02 ENTER↑          put rrr.ccc in Y
  03 XEQ 08          call "TARGET TEXT?' prompt
  04 LBL 88          start deleting loop
  05 RDN             now rrr.ccc is in X
  06 POSFL           search the file for text in ALPHA
  07 X<0?            was text found?
  08 CLA             if not, then clear alpha
  09 RDN             put rrr.ccc in X
  10 ALENG           get the length of text
  11 DELCHR          delete that many characters
  12 X≠0?            was alpha clear?
  13 GTO 88          if not, loop back to search again
  14 RDN             put rrr.ccc in X
  15 SF 25           set error ignore flag
  16 SEEKPT          try to position window to old position
  17 FS? 25          were you successful?
  18 RTN             then return
  19 GTO 26          if not, then call error handler
```

To add this routine press:
(GTO) (ALPHA)TE(ALPHA) (SHIFT)(RTN) (PRGM)
and key in the routine.

Now, every time that (ALPHA) is hit while in the edit mode, you will see a prompt for "TARGET TEXT?". Key in the text that is to be deleted throughout the file, and press (R/S). The routine will then return to the window display.

NUMERIC LABEL DESCRIPTION

LBL	Description	LBL	Description
ØØ	Edit mode input loop	24	INS A
Ø2	INS X	25	AHEAD n
Ø6	"TEXT?" prompt	26	error handler for deletion
Ø7	"NEW TEXT?" prompt	31	BACK 1
Ø8	"TARGET TEXT?" prompt	32	DEL A
Ø9	"NUMBER?" prompt	33	VIEW L
1Ø	loop for VIEW L	34	CHA A
11	ADD L	35	AHEAD 1
12	BEG L	41	ADD A
13	CHA L	42	GOTO CHR
14	DEL L	43	POS FL
15	INS L	44	DEL CHR
16	"FILE NAME?" prompt	51	UP n
17	"NAME?" prompt for LBL A	61	UP 1
18	"OK" prompt	71	DOWN 1
19	loop for LBL B	81	DOWN n
2Ø	escape loop for LBL B	83	GOTO REC
21	BACK n	84	exit edit mode
22	GOTO A	99	window display loop
23	POINT		

ERROR SUMMARY

Function	Meaning of ERROR message
ADD L	Not enough room for new text
BEG L	No error situations
CHA L	Not enough room for new text
DEL L	No error situations
INS A	Not enough room for new text
DEL A	No error situations
VIEW L	No error situations
INS L	Not enough room for new text
GOTO A	No error situations
POINT	No error situations
CHA A	Not enough room for new text
ADD A	Not enough room for new text
GTO CHR	New position exceeds text limits
POSFL	No error situations
DELCHR	No error situations
GTO REC	New position exceeds text limits
STOP/CONT	No error situations
INS X	Not enough room or illegal ASCII code
AHEAD n	New position exceeds text limits
AHEAD 1	New position exceeds text limits
BACK n	No error situations
BACK 1	No error situations
UP n	No error situations
DOWN 1	New position exceeds text limits
DOWN n	New position exceeds text limits
UP 1	No error situations

"TE" Program listing

```
01♦LBL 31
02 1 E-3
03 -
04 X<0?
05 CLX
06 RTN

07♦LBL 35
08 1 E-3
09 +
10 RTN

11♦LBL 71
12 INT
13 1
14 +
15 RTN

16♦LBL 61
17 INT
18 1
19 -
20 X<0?
21 CLX
22 RTN

23♦LBL 41
24 XEQ 07
25 APPCHR
26 RTN

27♦LBL 34
28 SF 10
29 XEQ 32
30 SF 25
31 GTO 24

32♦LBL 11
33 XEQ 07
34 APPREC
35 RCLPT
36 INT
37 RTN

38♦LBL 13
39 XEQ 07
40 DELREC
41 INSREC
42 GTO 26

43♦LBL 32
44 "OLD"
45 XEQ 06
46 POSFL
47 X<0?
48 CLA
49 ALENG
50 DELCHR
51 GTO 26

52♦LBL 24
53 XEQ 07
54 INSCHR
55 RTN

56♦LBL 43
57 0
58 SEEKPT
59 RDN
60 XEQ 08
61 POSFL
62 X<0?
63 RDN
64 RTN

65♦LBL 22
66 INT
67 XEQ 08
68 POSFL
69 INT
70 X<>Y
71 X=Y?
72 LASTX
73 RTN

74♦LBL 08
75 "TARGET"
76 GTO 06

77♦LBL 07
78 "NEW"

79♦LBL 06
80 "├ TEXT?"
81 AVIEW
82 CLA
83 AON
84 STOP
85 AOFF
86 RTN

87♦LBL 42
88 INT
89 GTO 25

90♦LBL 44
91 XEQ 09
92 DELCHR
93 GTO 26

94♦LBL 51
95 INT
96 XEQ 09
97 -
98 X<0?
99 CLX
100 RTN

101♦LBL 25
102 XEQ 09
103 1 E3
104 /
105 +
106 RTN

107♦LBL 21
108 XEQ 09
109 1 E3
110 /
111 -
112 X<0?
113 INT
114 RTN

115♦LBL 81
116 INT
117 XEQ 09
118 +
119 RTN

120♦LBL 02
121 XEQ 09
122 CLA
123 XTOA
124 FS? 25
125 INSCHR
126 RDN
127 RTN

128♦LBL 83
129♦LBL 09
130 "NUMBER?"
131 AVIEW
132 "RHIJ>?"
133 64
134 XTOA
135 RDN
136 "├456"
137 SF 01
138 GETKEY
139 SF 02
140 GETKEY
141 SF 03
142 GETKEY
143 CF 03
144 CF 02
145 CF 01
146 POSA
147 RDN
148 POSA
149 RDN
150 POSA
151 RDN
152 CLA
153 ARCL T
154 ARCL Z
155 ARCL Y
156 ANUM
157 X<0?
158 GTO 09
159 AVIEW
160 RTN

161♦LBL 14
162 DELREC
163 XEQ 26
164 RTN

165♦LBL 15
166 XEQ 07
167 INSREC
168 RTN

169♦LBL 33
170 FS? 55
171 SF 21
172 INT
173 "LINE "
174 ARCL X
175 AVIEW
176 SEEKPT

177♦LBL 10
178 CLA
179 ARCL 00
180 ARCL 00
181 ARCLREC
182 ASHF
183 ASHF
184 AVIEW
185 FS? 17
186 GTO 10
187 LASTX
188 CF 21
189 RTN

190♦LBL 23
191 ENTER↑
192 INT
193 "LINE"
194 ARCL X
195 "├CHAR"
196 LASTX
197 FRC
198 1 E3
```

"TE" Program listing (continued)

```
199 *
200 ARCL X
201 AVIEW
202 RDN
203 RDN
204 RTN

205♦LBL 12
206 INT
207 RTN

208♦LBL 26
209 RCLPT
210 SF 25
211 SEEKPT
212 FC?C 10
213 FS? 25
214 RTN
215 INT
216 SF 25
217 SEEKPT
218 FC? 25
219 XEQ 61
220 RTN

221♦LBL 01
222♦LBL 04
223 RCL 00
224 STOFLAG
225 GTO 18

226♦LBL A
227 "LINES?"
228 PROMPT
229 "CHAR?"
230 PROMPT
231 +
232 7
233 /
234 1
235 +

236♦LBL 17
237 "NAME?"
238 AON
239 CLD
240 STOP
241 AOFF
242 SF 25
243 RCLPTA
244 FS? 25
245 GTO 17
246 SF 25
247 CRFLAS
248 FC? 25
249 GTO A
250 GTO 18

251♦LBL B
252 XEQ 16
253 .
254 SEEKPT
255 FLSIZE
256 7
257 *
258 1
259 -

260♦LBL 19
261 SF 25
262 GETREC
263 FC?C 25
264 GTO 20
265 ALENG
266 -
267 FS? 17
268 GTO 19
269 1
270 -
271 GTO 19

272♦LBL 20
273 "CHAR LEFT="
274 RCLFLAG
275 FIX 0
276 CF 29
277 ARCL Y
278 STOFLAG
279 AVIEW
280 RDN
281 STOP
282 GTO 18

283♦LBL C
284 XEQ 16
285 SF 25
286 CLFL
287 FC? 25
288 GTO C
289 GTO 18

290♦LBL D
291 XEQ 16
292 SF 25
293 PURFL
294 FC? 25
295 GTO D
296 GTO 18

297♦LBL E
298 XEQ 16
299 RCLFLAG
300 STO 00
301 RDN
302 .
303 SEEKPT
304 FIX 0
305 CF 29
306 CF 27
307 GTO 99

308♦LBL G
309 XEQ 16
310 GTO 18

311♦LBL 16
312 "FILE NAME?"
313 AON
314 AVIEW
315 CLA
316 STOP
317 AOFF
318 SF 25
319 RCLPTA
320 FC?C 25
321 GTO 16
322 SF 25
323 POSFL
324 FC?C 25
325 GTO 16
326 RTN

327♦LBL F
328 EMDIR
329 7
330 *
331 "FREE="
332 ARCL X
333 PROMPT
334 GTO 18

335♦LBL H
336 5
337 "PROBLEM KEY?"
338 AVIEW
339 "DIRED DELCLRFRE"
340 "⊢ADDGTO"
341 GETKEY
342 23
343 X<=Y?
344 GTO 18
345 RDN
346 10
347 -
348 X>Y?
349 +
350 10
351 MOD
352 -3
353 *
354 AROT
355 ASHF
356 ASHF
357 ASHF
358 AVIEW
359 GTO H

360♦LBL "TE"
361 SF 27
362 CF 21
363 CF 10
364 SIZE?
365 1
366 X>Y?
367 PSIZE

368♦LBL 18
369 CF 25
370 CLST
371 "OK"
372 PROMPT
373 GTO 18

374♦LBL 99
375 RCLPT
376 SF 25
377 CLA
378 ARCL 00
379 ARCL 00
380 ARCLREC
381 ASHF
382 ASHF
383 SEEKPT
384 AVIEW
385 SF 25

386♦LBL 00
387 SF 00
388 GETKEY
389 CF 00
390 RDN
391 XEQ IND T
392 SEEKPT
393 "ERROR"
394 FC? 25
395 AVIEW
396 GTO 99
397 END

803 BYTES
```

CHAPTER NINE
AN HP-16 SIMULATOR PROGRAM

This chapter presents a program that simulates some of the functions of the HP-16C calculator. Since the example program performs base conversions, a little background on number bases is in order.

A decimal number wxyz has the value

$$wxyz_{10} = w \cdot 10^3 + x \cdot 10^2 + y \cdot 10 + z,$$

where w, x, y, and z are any digits from 0 to 9. The subscript 10 indicates base 10. Hexadecimal (base 16) notation works the same way. A hexadecimal number $qrst_{16}$ has the value

$$qrst_{16} = q \cdot 16^3 + r \cdot 16^2 + s \cdot 16 + t,$$

where q, r, s, and t are any hexadecimal digits from zero to fifteen. Since there are no ordinary digits that correspond to the numbers ten through fifteen, it is standard notation to borrow them from the alphabet: $A_{16} = 10$, $B_{16} = 11$, $C_{16} = 12$, $D_{16} = 13$, $E_{16} = 14$, and $F_{16} = 15$. For example $C5_{16} = 12 \cdot 16+5 = 197$, and $FF_{16} = 15 \cdot 16+15 = 255$.

These same principles apply to number bases other than 10 or 16. Each digit in the representation represents a coefficient of a power of the base.

Base conversion is a frequent application of programmable calculators. In fact, the HP-16C specializes in base conversion and operations in base 2 (binary), base 8 (octal), base 10 (decimal), and base 16 (hexadecimal). Keys labeled A through F are provided for easy entry of hexadecimal numbers.

A base 16 number can be entered in HEX mode, then converted to decimal simply by pressing the DEC key. The calculator then interprets all further entries as decimal numbers until the mode is changed.

The program "HP-16" listed on page 164 simulates the base conversion functions of the HP-16. Just press

XEQ ALPHA H P shift - shift 1 shift 6 ALPHA

to execute the program, and the keyboard is redefined as shown in the accompanying table. Keys that do not appear in the table do nothing when pressed, because there is no corresponding numeric label in the program.

"HP-16" GETKEY keyboard

row 1:	**A**	**B**	**C**	**D**	**E**
row 2:	**F**	**HEX**	**DEC**	**OCT**	**BIN**
row 3:	---	**X<>Y**	**STO**	**RCL**	---
row 4:	**ENTER↑**	---	---	**backspace**	
row 5:	**-**	**7**	**8**	**9**	
row 6:	**+**	**4**	**5**	**6**	
row 7:	*****	**1**	**2**	**3**	
row 8:	**/**	**Ø**	---	**quit**	

Although the program is somewhat sluggish, it is meant to simulate an HP-16C with a two-level stack (X and Y registers only). The mode is indicated by flag annunciators. Flag 1 denotes hexadecimal, flag 2 decimal, flag 3 octal, and flag 4 binary. The flag Ø annunciator, when lit, indicates that the HP-41 is ready for you to press a key. When <u>not</u> lit, the flag Ø annunciator indicates that a calculation is in progress. When you press a key, the disappearence of the flag Ø annunciator signifies that your input was recognized. Wait for the flag Ø annunciator to reappear before pressing another key.

A series of examples will make the operation of this program more clear. First, execute "HP-16" to start the program. The flag 1 annunciator signifies that you are in HEX mode.

Press the "C" key (row 1, column 3). The flag Ø annunciator disappears briefly, then it reappears and a "C" appears in the display. Press the "2" key and "C2" will appear. If you make a mistake, press the backarrow key and the rightmost digit of the displayed number will be removed.

Now convert this number to base 8 by pressing the "OCT" key (row 2, column 4). After a short wait, the result "3Ø2" will appear. The flag 3 annunciator indicates octal mode.

Let's add 7 to this number. This is done just the way you would expect. Simply press 7, wait for the "7" to appear in the display, then press +. The two numbers $3Ø2_8$ and 7_8 will be added and the octal result, 311_8, will appear in the display.

You can convert this number to binary by pressing the BIN key (row 2, column 5). This takes a little while because of the number of digits that need to be decoded, but the result is $11ØØ1ØØ1_2$. The flag 4 annunciator indicates binary mode.

To find the decimal equivalent, press the DEC key (row 2, column 3). The flag 2 annunciator indicates decimal mode, and the result 2Ø1 appears.

"HP-16" program listing

```
01♦LBL "HP-16"
 02 2
 03 X<>F
 04 16
 05 STO 00
 06 CLST
 07 CLA
 08 CF 21

 09♦LBL 05
 10 RDN
 11 AVIEW
 12 SF 00
 13 GETKEY
 14 CF 00
 15 X=0?
 16 GTO 05
 17 RDN
 18 SF 25
 19 XEQ IND T
 20 R↑
 21 GTO 05

 22♦LBL 82
 23 0
 24 GTO 10

 25♦LBL 72
 26♦LBL 73
 27♦LBL 74
 28 R↑
 29 71
 30 -
 31 GTO 10

 32♦LBL 62
 33♦LBL 63
 34♦LBL 64
 35 R↑
 36 58
 37 -
 38 GTO 10

 39♦LBL 52
 40♦LBL 53
 41♦LBL 54
 42 R↑
 43 45
 44 -

 45♦LBL 10
 46 48
 47 GTO 06

 48♦LBL 11
 49♦LBL 12
 50♦LBL 13
 51♦LBL 14
 52♦LBL 15
 53 R↑
 54 1
 55 -
 56 GTO 10

 57♦LBL 21
 58 15

 59♦LBL 10
 60 55

 61♦LBL 06
 62 FS?C 05
 63 GTO 10
 64 X<>Y
 65 +
 66 FS? 07
 67 GTO 09
 68 0
 69 X<>Y
 70 CLA
 71 SF 07

 72♦LBL 09
 73 XTOA
 74 X<> L
 75 X<> 00
 76 ST* Y
 77 X<> 00
 78 +
 79 RTN

 80♦LBL 10
 81 RDN
 82 SIGN
 83 RDN
 84 FC? 06
 85 RCL IND L
 86 STO IND L
 87 FS?C 06
 88 RTN
 89 GTO 07

 90♦LBL 44
 91 RCL 00
 92 /
 93 INT
 94 1
 95 CHS
 96 AROT
 97 RDN
 98 ATOX
 99 RDN
100 RTN

101♦LBL 01
102♦LBL 84
103 0
104 X<>F
105 RDN
106 CF 25
107 CLD
108 STOP
109 RTN

110♦LBL 22
111 2
112 16
113 GTO 10

114♦LBL 23
115 4
116 10
117 GTO 10

118♦LBL 24
119 8
120 ENTER↑
121 GTO 10

122♦LBL 25
123 16
124 2

125♦LBL 10
126 STO 00
127 RDN
128 X<>F
129 RDN

130♦LBL 07
131 CF 07
132 CLA
133 ENTER↑

134♦LBL 08
135 ENTER↑
136 X<> 00
137 ST/ 00
138 MOD
139 9
140 ST- Y
141 RDN
142 7
143 X<>Y
144 X>0?
145 ST+ Y
146 X<=0?
147 X<>Y
148 RDN
149 57
150 ST+ Y
151 RDN
152 XTOA
153 X<> L
154 X<> 00
155 1
156 CHS
157 AROT
158 RDN
159 INT
160 X>0?
161 GTO 08
162 RDN
163 RTN

164♦LBL 41
165 ENTER↑
166 CF 07
167 RTN

168♦LBL 32
169 X<>Y
170 GTO 07

171♦LBL 51
172 CHS

173♦LBL 61
174 X<>Y
175 ST+ Y
176 X<>Y
177 ABS
178 GTO 07

179♦LBL 81
180 1/X

181♦LBL 71
182 X<>Y
183 ST* Y
184 X<>Y
185 INT
186 GTO 07

187♦LBL 33
188 SF 06

189♦LBL 34
190 SF 05
191 END

297 BYTES
```

Press the X<>Y key (row 3, column 2, not row 2, column 1) and you will see that the number $C2_{16}$ = 194 was duplicated into the Y register when the 7 was added. This automatic duplication is similar to the way the T register duplicates itself when an operation like addition is performed normally.

The ENTER↑ key also works as expected, duplicating X into Y and terminating digit entry.

The "HP-16" program allows you to store and recall numbers from data registers Ø1 to 15. You simply press the STO or RCL key, wait for the flag Ø annunciator to reappear, and press a key from 1 to 9 or A to F to designate the register. Do not store anything in register Ø, because that register is reserved for holding the number base. The availability of STO and RCL operations alleviates the limitations of a two-level stack. The RCL operation does raise the stack, so you do not need to ENTER↑ before doing a RCL.

When you are done using the program, just press the R/S key or the ON key to quit and clear the flags. The X and Y register contents will still be in X and Y as decimal numbers, regardless of what mode you were in when you pressed R/S.

The "HP-16" program is an example of how completely you can change the personality of your HP-41 with just the GETKEY function and a program of moderate size. If you have an application that needs this degree of user convenience, GETKEY may be just what you need.

Line-by-line analysis of "HP-16"

The "HP-16" program is composed of many small pieces, each of which obeys a few basic rules. First, the number base is held in data register ØØ. The steps 16, STO ØØ at the top of the program have the effect of setting hexadecimal (base 16) mode. The program's "X" and "Y" register contents are held in the stack, in X and Y both before and after the XEQ IND T instruction (line 16). Numbers in the stack are always in decimal. The displayed number is actually a string in the ALPHA register. If the number being displayed changes other than by addition or removal of a digit, the LBL Ø7 subroutine is called to reconstruct the ALPHA representation from the decimal number in X. LBL 1Ø is used repeatedly for short forward (downward) jumps. A label number can only be re-used this way if none of the jumps cross each other.

The LBL Ø5 loop is the main loop of this program. It uses GETKEY to read the keyboard, then it sets flag Ø and executes the proper subroutine for the key that was pressed. Flag Ø is then cleared, the new contents of ALPHA are displayed, and another GETKEY is attempted. Flag 25 is set to avoid error stops when an invalid key is pressed accidentally. Incidentally, clearing flag 21 at line Ø8 prevents the presence of a turned-off printer from halting the program at the AVIEW instruction. If you want printout, delete the CF 21 instruction.

Digit entries are handled by placing the decimal value in Y, and the ASCII offset from that value in X. For example, when you press "C" (key 13), the value 12 is placed in Y, and 55 in X. For the letters A through F, the ASCII code is 55 plus the arithmetic value of 1Ø to 15. The LBL Ø6 sequence appends the correct ASCII character to ALPHA. Flag Ø7 indicates that a digit entry is in progress. If a digit is pressed when a digit entry was not already in progress, the stack will be raised. This is accomplished by the sequence Ø, X<>Y, CLA. This sequence is bypassed by the GTO Ø9 instruction if a digit entry was in progress. The LBL Ø9 sequence then updates the decimal number in X by multiplying by the base (lines 75 and 76) and adding the value of the new digit (lines 77 and 78).

If the digit is pressed as part of a STO or RCL, flag 5 will be set and the LBL 1Ø section at line 8Ø performs the necessary operation (STO if flag 6 is set, RCL if flag 6 is clear). For RCL operations, the LBL Ø7 sequence is used to reconstruct an ALPHA string corresponding to the new X register.

The LBL 44 routine simply divides the current X register contents by the number base, effectively performing a single-digit shift, then it removes the rightmost character from ALPHA.

The termination sequence (LBL Ø1 or LBL 84) uses X<>F to clear flags Ø to 7, then it clears the previously AVIEWed display before stopping. You can restart by pressing R/S again, but the number base will not show in the flag annunciators until you press a number base mode key.

The mode selection labels, 22 through 25, put a number in Y corresponding to the flag to be set, and a number in X corresponding to the new base. The base is then stored in register ØØ, and the X<>F function is used to set the proper flag and clear any others.

Once the base has been changed, the ALPHA register needs to be updated to agree. The LBL Ø7 sequence does this. After ALPHA is cleared, the digits are computed and placed in ALPHA, working from right to left. The ENTER↑ immediately preceding LBL Ø8 serves to keep a copy of the number being encoded. This is necessary because the ALPHA encoding destroys the number in X.

Near the top of the LBL Ø8 loop, the MOD function gives the arithmetic value of the current digit. The next 13 lines convert this arithmetic value to an ASCII equivalent, then an XTOA function creates the character. The AROT function rotates the new character to the front of the string. The current decimal value is divided by the base (this was the reason for the earlier ST/ ØØ instruction), and the integer part is taken. This procedure effects the arithmetic equivalent of a one-digit shift. If the result is still not zero, more digits remain to be decoded, and the LBL Ø8 sequence is performed again.

The ENTER↑ sequence, LBL 41, simply pushes a zero onto the stack and clears the ALPHA register in preparation for digit entry. The X<>Y sequence, LBL 32, interchanges X and Y and goes to the LBL Ø7 sequence to reconstruct ALPHA. The -, +, /, and * sequences also branch to LBL Ø7 for ALPHA reconstruction. The two X<>Y instructions and the ST+Y or ST*Y allow the previous value of Y to remain unchanged in Y. Since the LBL Ø7 routine cannot handle negative numbers or nonintegral numbers, INT is used after division and ABS after subtraction.

The STO and RCL sequences both set flag 5. That flag is used to indicate that the next digit entry is really a register number. Flag 6 is used to determine whether a STO or RCL is to be performed.

I hope this example convinces you of the tremendous power of GETKEY. Your applications may be simpler or more complex, but the same principles apply.

CHAPTER TEN

SYNTHETIC PROGRAMMING

10ØA. What is Synthetic Programming?

Synthetic instructions are those which cannot be entered from the keyboard by normal means. The creation and use of synthetic instructions is called synthetic programming. Thousands of synthetic instructions can be created, ranging from non-standard TONEs to powerful instructions that access system scratch registers. Synthetic programming will not harm your HP-41 in any way, although "crashes" (temporary keyboard lock-up and/or MEMORY LOST) can occur if you are experimenting in unfamiliar territory. Refer to section K of this chapter for tips on how to recognize and recover from a "crash" condition.

Synthetic programming will work on all calculators in the HP-41 family, 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.

The programs presented in this chapter use synthetic techniques to get into areas of memory that are not normally accessible. These include the registers that hold key assignment information and header registers in extended memory.

Since some of the instructions are not directly keyable, **barcode is provided in Appendix D.** If you do not have access to a wand, SYNTHETIX can provide you with magnetic cards for any or all of the programs in this book. The charge is $4.ØØ (USA) or $5.ØØ (elsewhere), plus $1.ØØ per magnetic card. To find out how many magnetic cards will be needed for each program you want, divide its byte count by 224 and round any fractional part up to the next integer. Mail your order to SYNTHETIX at the address on the top of the next page.

SYNTHETIX
P.O. Box 113
Manhattan Beach
CA 90266   USA

Checks must be payable through a US bank. Cash is also acceptable if you find it more convenient, but you should wrap it well.

Another alternative is to learn enough about synthetic programming so that you can key these programs in yourself. The easiest way to get started with synthetic programming is to buy a copy of the book "HP-41 Synthetic Programming Made Easy". If your dealer does not sell this book, it is available by mail from SYNTHETIX at the above address. See Appendix C for price information.

If you like to program your HP-41, you really should learn about synthetic programming. It is almost like finding a brand-new machine, hidden inside your familiar HP-41.

I hope you enjoy the programs presented in this chapter. If you are an experienced synthetic programmer, you will appreciate their power and versatility. If you are a novice, they offer a glimpse of the capabilities of synthetic programming.

10B. Single-key execution of extended functions

The program presented in this section allows you to execute any function from the set of extended functions, simply by specifying a numeric code for the function. This program was written by Clifford Stern, a "grand master" of synthetic programming. Clifford specializes in keystroke-efficient utility routines and intricate synthetic programs.

It is a simple matter to use the built-in ASN function to assign several of the commonly used extended functions to keys. You have probably already assigned EMDIR, the extended memory directory function, to a convenient USER mode key. Perhaps you have also assigned SAVEP and GETP. Or, if you have used data files, you may have assigned RCLPT, SEEKPT, SAVERX, and GETRX to keys. These key assignments are handy, but they can quickly use up a significant portion of your USER mode keyboard.

How would you like to be able to assign all the extended functions to a single key? Impossible? Not with synthetic programming! All you need is a copy of the "XF" (eXtended Functions) program.

To execute any extended function, just put the numeric code of the function in X, and execute "XF". The numeric codes are listed in front of each function name in the table on the next page. A short digression should make the numeric code equivalence more clear.

If you key in program lines using extended functions while the Extended Functions module is connected, the display

XEQ^{T}function name

will change to

XROM^{T}function name .

If you later remove the XFunctions module, the program lines will be displayed and printed as

XROM 25,xx .

The designation XROM indicates that the function resides in an eXternal Read-Only Memory. The number 25 identifies the Extended Functions module. The two-digit number xx identifies the specific function within the module. This is the same two-digit function code that the "XF" program uses.

Incidentally, as the table on the next page shows, "XF" also allows you to execute Time module (XROM 26) and optical wand (XROM 27) functions. Inputs 49-62 and 95-99 are only valid for the HP-41CX.

Numeric function codes for "XF"

(XROM numbers are also included for reference)

	-EXT FCN 2C	
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

	-X EXT FCN	
49	ASROOM	25,49
50	CLRGX	25,50
51	ED	25,51
52	EMDIRX	25,52
53	EMROOM	25,53
54	GETKEYX	25,54
55	RESZFL	25,55
56	ΣREG?	25,56
57	X=NN?	25,57
58	X≠NN?	25,58
59	X<NN?	25,59
60	X<=NN?	25,60
61	X>NN?	25,61
62	X>=NN?	25,62

	-TIME 2B	
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

	-X TIME	
95	CLALMA	26,31
96	CLALMX	26,32
97	CLRALMS	26,33
98	RCLALM	26,34
99	SWPT	26,35

	- 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

"XF" program listing

```
            01♦LBL "XF"     10 RDN          19 X<> a        28 STO c
            02 X<>Y         11 XTOA         20 RCL [        29 X<> L
            03 SIGN         12 RDN          21 STO IND \    30 SAVEP
            04 X<> \        13 501          22 X<> ]        31 CLD
            05 STO a        14 SIZE?        23 CLA          32 END
            06 RDN          15 ST- Y        24 X<> [
            07 64           16 RDN          25 RDN          74 BYTES
            08 ST+ Y        17 X<> \        26 X<> \
    09 "├♦♦xi♦δ♦ɏ+t"       18 X<> c        27 X<> a
```

Barcode for "XF" can be found in Appendix D.

Synthetic lines and their decimal byte equivalents:

line Ø4 = 2Ø6, 118 ; line Ø5 = 145, 123

line Ø9 = 254, 127, Ø, Ø, 1, 1Ø5, Ø, 18, Ø, 123, 145, 125,
2Ø6, 116, 166

line 17 = 2Ø6, 118 ; line 18 = 2Ø6, 125 ; line 19 = 2Ø6, 123

line 2Ø = 144, 117 ; line 21 = 145, 246 ; line 22 = 2Ø6, 119

line 24 = 2Ø6, 117 ; line 26 = 2Ø6, 118 ; line 27 = 2Ø6, 123

line 28 = 145, 125

(Use this information if you are keying in the program using a byte-loader, byte-grabber, or any other synthetic technique.)

Setting up "XF":

The "XF" program must be the first program in Catalog 1. This means that you must clear all other programs from main memory before loading in "XF". You can use SAVEP to copy some of the programs into extended memory, or you can use magnetic card or tape storage. To clear out program memory, just load the ALPHA register with the name of the first program in main memory (Catalog 1) and execute PCLPS.

Next load the "XF" program into your calculator using the barcode, magnetic cards, or using synthetic programming techniques described in Chapter 3 of "HP-41 Synthetic Programming Made Easy".

If you have an HP-41C, you may need to change line 13. This number, normally 5Ø1, should be 263+64n, where n is the number of single-density memory modules plugged in. This refers to main memory modules, not extended memory modules. For example, if you are using two single-density memory modules, line 13 should be 373. If you have a quad memory module, which is the equivalent of 4 single-density modules, the number 5Ø1 is correct. **WARNING:** Failure to put the right number in line 13 can result in MEMORY LOST. If you unplug a main memory module without changing this number, you are just one keystroke away from disaster. Of course it is OK to unplug extended memory modules.

For maximum convenience, assign "XF" to your favorite key. To do this, press

```
          shift ASN ALPHA X F ALPHA
```

followed by the key (or "shift" followed by the key for a shifted location) to which you want "XF" assigned. You should probably avoid assigning "XF" (or any other function) to a digit entry key or to the XEQ key. If you have the SIZE function assigned to a key, you can use that key for "XF". With "XF", you can conveniently resize by keying in the desired size, ENTER↑, 3Ø (the numeric code for PSIZE), and executing "XF".

"XF" is a self-modifying program which works by constructing and storing a short instruction sequence containing the requested extended function (line 3Ø) in the first program in memory. The first program will be changed, regardless of whether it is "XF" or not. If "XF" is the first program, as it should be, the stored sequence of instruction will fit right in, so only the extended function (line 3Ø) will change. Line 31, CLD, can be deleted if you have an HP-41CX. Its purpose is to clear the display after EMDIR.

WARNING: Do not change the "XF" program unless you make sure that you keep the same number of bytes between the top of the program and line 30. If you change this byte count, the stored instruction sequence will end up in the wrong place.

"XF" is an example of the power of synthetic programming. Self-modifying programs are usually rather complicated, but this shows how a simple one can do a job that cannot be reasonably done without synthetic programming.

<u>Example 1 for "XF"</u>:

The most frequently used extended memory function is probably the extended memory directory function, EMDIR. The table of "XF" inputs shows that the corresponding numeric code is 14. So if you press

```
        14
        XEQ "XF"   (just press the assigned key)
```

the extended memory directory will be displayed.

<u>Instructions for using "XF"</u>

1. Make sure that "XF" is the first program in main memory by executing Catalog 1. The first thing you should see is LBL⊤XF. You do not need to check Catalog 1 every time, but you should be certain that "XF" is at the top of it.
2. Load the X, Y, Z, and ALPHA registers with whatever contents they will need at the time the function is executed. The string in ALPHA is limited to 14 characters. If more characters are put in ALPHA, only the rightmost 14 will be used and the rest of the characters will be lost.
3. Press ENTER↑ and put the numeric code of the function in X. The numbers that were in X, Y, and Z are now in Y, Z, and T. <u>Do not use code zero</u>. Code zero has no effect on the HP-41CX, but on the HP-41C or CV it will destroy all

global label assignments, leaving only "phantom" assignments that act like the ABS function.

4. Press the assigned key to execute "XF". The designated function will be executed. "XF" actually builds a sequence of bytes in the ALPHA register, transfers the sequence into lines 27-30 and then executes the sequence. If you interrupt "XF" and do not restart it immediately, you run a risk of MEMORY LOST. Some safeguards have been provided, but if you stop between lines 18 and 28 and you do not allow "XF" to finish normally, MEMORY LOST can eventually result.
5. When the function is complete, the "flying goose" will disappear. If an error occurs, you will see the corresponding error message. For example if you use "XF" to execute GETX (function number 23) when the working file is not a data file, you will get the message FL TYPE ERR.
6. By checking line 30 of "XF" (GTO."XF" and GTO .030), you can find out what extended function was last executed using "XF". This can be quite helpful.

The "XF" program eliminates the need for key assignments of extended functions to the extent that you use these functions in RUN (non-PRGM) mode. If you are keying in a long program that uses extended functions, you may still want to temporarily assign a few of the more frequently encountered functions to your USER mode keys.

Two cautions apply to "XF". First, do not use "XF" to execute PCLPS (function number 27) with the ALPHA register empty. This will clear all main memory programs, including "XF" itself. Unless this is the result you want, you should name a program before executing PCLPS. For example, if you want to clear all programs except "XF", put the name of the second Catalog 1 program in ALPHA, put 27 in X, and execute "XF".

The second caution is not to call "XF" from a second-level or deeper subroutine. That is, "XF" must not be called when two or more RTNs are already pending. The "XF" program clears operating system register **a**, which holds the information used for third- through sixth-level RTNs.

An alternative version of "XF", also written by Clifford Stern, saves a couple of keystrokes over "XF". This version, called "EFTW" (extended functions / time module / wand) pauses in ALPHA mode for an entry of up to 7 characters. The operating instructions are otherwise the same as for "XF". Line 14 must be 246+64n, where n is the number of single-density memory modules present. Additionally, XYZALM (function number 93) cannot be used where a nonzero Z input is needed, because the Z register is changed to zero by the time XYZALM is executed.

"EFTW" program listing

```
01♦LBL "EFTW"          09 64              17 X<> \          25 STO c
  02 RCL [             10 +               18 X<> c          26 RDN
  03 CLA      11 "├♦♦×i♦δ♦uu→u            19 RCL [          27 SAVEP
  04 STO [             12 XTOA           20 STO IND \       28 CLD
  05 AON               13 CLX             21 X<> ]          29 END
  06 PSE               14 502             22 CLA
  07 AOFF              15 SIZE?           23 X<> [         67 BYTES
  08 CLX               16 -               24 RDN
```

Barcode for "EFTW" can be found in Appendix D.

Synthetic lines and their decimal byte equivalents:

line Ø2 = 144, 117 ; line Ø4 = 145, 117

line 11 = 254, 127, Ø, Ø, 1, 1Ø5, Ø, 18, Ø, 117, 117, 145, 125, 117, 166

line 17 = 2Ø6, 118 ; line 18 = 2Ø6, 125 ; line 19 = 144, 117

line 2Ø = 145, 246 ; line 21 = 2Ø6, 119 ; line 23 = 2Ø6, 117

line 25 = 145, 125

10C. The internal structure of extended memory

This section outlines the general arrangement of files in extended memory, showing what areas are affected by the card reader's VER and 7CLREG functions. Then, for advanced synthetic programmers, the details of file header structure and ways to avoid data normalization (see page 25 of "HP-41 Synthetic Programming Made Easy") are covered.

Extended memory is made up of one, two, or three blocks of registers, depending on whether zero, one, or two extended memory modules are plugged in. The Extended Functions/Memory module contains 128 registers, while each Extended Memory module contains 239 registers. The advertised sizes of these modules are 127 and 238, respectively, because the last register in each module is reserved. This last register contains a pointer to the beginning of the next module and another pointer to the end of the previous module. These pointers are needed for proper file linkage because the order in which extended memory modules are used can vary if the two modules are not installed at the same time.

Figure 10.1 on the next page shows the organization of extended memory in more detail, with absolute register addresses given for those adventurous enough to poke around.

Within the unreserved areas of extended memory, files are stored in the same order in which they appear in the extended memory directory. Each file has two header registers, as shown in Figure 2.1. A special **partition code** (hexadecimal FF,FF,FF,FF,FF,FF,FF for synthetic programmers) is stored just below the last register of the last file. This code separates used from unused portions of extended memory. It tells the calculator that the rest of extended memory is available for new files.

When you start with an empty extended memory directory, the partition code is at the top of the extended functions/-memory module. The first file you create will occupy the top-

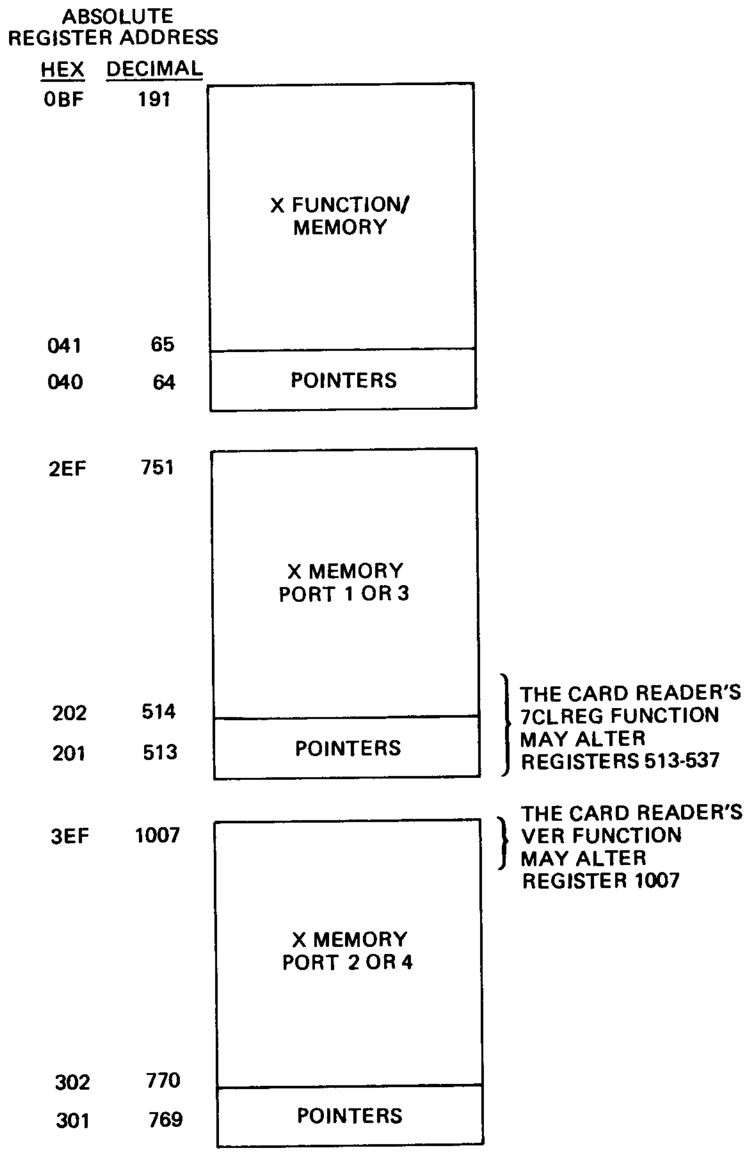


Figure 10.1 Overall Structure of Extended Memory.

most registers of the Extended Functions/Memory module, and will move the partition code down. As you create files, new files will always be added just below the last file, and the partition code will be moved down.

Eventually you will use up all 127 available registers in the first block of extended memory. When this happens, the file will spill over into an Extended Memory module. Usually the Extended Memory module in port 1 or 3 will be used before the module in port 2 or 4. The only exceptions are:

1) if there is no Extended Memory module in port 1 or 3, or
2) if the module in port 2 or 4 was partially filled before the other module was installed.

After MEMORY LOST, the natural order of use (port 1 or 3 first) will be restored.

Detailed structure of header and pointer registers:

Each file header consists of two registers at the top of the file. The first of these registers contains the file name, up to 7 characters. If the file name is fewer than 7 characters, spaces (hexadecimal 2Ø) are added on the right to fill the 7 bytes of the register.

The second file header register contains several pieces of information about the file. The structure will be described here in terms of nibbles, which are hexadecimal digits. Two nibbles make one byte; seven bytes make one register. The leftmost nibble of the second file header register indicates the file type. This nibble is 1 for program files, 2 for data files, and 3 for ASCII files.

For program files, the 14 nibbles of this register are:

```
1Ø,ØØ,ØØ,ØØ,BB,BS,SS ,
```

where BBB is number of bytes in the saved program (including the END) and SSS is the FLSIZE, in registers. Both of these numbers are in hexadecimal, not decimal. A program in an extended memory file has the same form as a program in main memory, including the END. The END is followed by a single **checksum** byte that contains the modulo 256 sum of all the bytes in the program. This represents a single byte of "overhead" in addition to the two program file header registers. Thus if a program's byte count is 49 bytes (7 registers), an 8-register file will be created by SAVEP because a 50th checksum byte must be included.

For data files, the second header register is:

2A,AA,00,00,RR,RS,SS ,

where AAA is the absolute address of this second header register, RRR is the register pointer, and SSS is the file size. Registers are numbered 0, 1, 2, etc., starting with the register immediately below the second header register.

For ASCII files, the second header register is

3A,AA,00,CC,RR,RS,SS ,

where CC is the character pointer, RRR is the record pointer, and AAA and SSS are the same as for data files.

The pointer registers at the bottom of each block of extended memory contain these 14 nibbles:

00,0W,WP,PP,NN,NT,TT ,

where WW is the number of the working file (01 and up), PPP is the absolute address of the bottom register of the previous block of extended memory, NNN is the address of the top register of the next block, and TTT is the address of the top register of this block. The WW field is used only in the Extended Functions/Memory module. The PPP field is not used in the Extended Functions/Memory module, but in the HP-41CX it indicates the previous working file. The NNN field is not used in the second Extended Memory module. All these pointer registers are initialized when a file is created that occupies part of the module in question. If an extended memory-related function has been used, but no files have been created yet, the TTT field will contain the address of the pointer register itself.

The nibbles of the header or pointer registers that are unused do not have to be zero. For example, it is often convenient for a synthetic program to change the first nibble of a pointer register to 1, so that the register can be recalled as ALPHA data.

This brings up the subject of normalization. If a numbered register contains a bit pattern that does not represent a number and which the HP-41 does not recognize as ALPHA data, the register contents may be altered when the register is recalled. This point is discussed further on page 25 of "HP-41 Synthetic Programming Made Easy". Operations that normalize the contents of a numbered register include RCL, ARCL, X<>, VIEW, and any INDirect operation.

Among the extended functions, several of the SAVE and GET operations can transfer data without normalization. This makes possible many advanced synthetic programming applications, including some of the programs in this chapter. GETX, SAVEX, GETR, SAVER, and GETRX do not normalize data at all. The SAVERX and REGSWAP functions normalize both upper and lower extremes of the data register block used. The REGMOVE operation normalizes only the topmost data register used.

When a file is purged from extended memory, all files below that file in extended memory are moved up to fill the space left by the purged file. If the file was the last one in extended memory, no files are moved. The partition code is then stored just below the last remaining file. The registers beyond this point are not cleared. They retain the same contents they had before PURFL was executed, but they are no longer accessible except through synthetic techniques.

All of these details are of interest primarily to advanced synthetic programmers, but they illustrate the number and variety of pointers that the calculator must maintain to keep things simple for the user of extended memory.

10D. A solution to the VERify "bug"

The program "VER" (verify) presented here takes the place of the card reader's built-in VER function, while ensuring that extended memory is not damaged. This is another masterpiece by Clifford Stern. Unless your card reader is very new (revision 1G or higher) or unless you do not have an Extended Memory module in port 2 or port 4, you need this program. The revision of your extended functions is irrelevant here.

Two versions of the "VER" program are provided in barcode in Appendix D. Normally you should use the first version. The second version is only to be used in two cases:

1) If you have a single Extended Memory module in port 2 or 4 with no Extended Memory module in port 1 or 3, or
2) If an Extended Memory module was plugged into port 1 or 3 after a module in port 2 or 4 was partially filled, and MEMORY LOST has not occurred since then. If you plugged in both modules at the same time, the first version is the one to use.

If you have only the Extended Functions/Memory module (or an HP-41CX) and no extended memory modules, or if you have only one Extended Memory module which is plugged into port 1 or port 3, you can use the card reader's built-in VER function. If you plan to add more extended memory, though, you might want to get into the habit of using the "VER" program.

WARNING: Before you use either version of "VER", make sure that either:

1) There is at least one key assignment which is not a global label assignment, or
2) There are no Time module alarms set and there is at least one free program register. To check the number of free program registers, press GTO .000 and look for the 00 REG nn display in PRGM mode. The number **nn** is the number of free program registers.

Neither of these conditions is difficult to ensure, and either one will ensure that "VER" will work properly.

To use "VER", press

XEQ ALPHA V E R ALPHA

and the prompt

CARD

will appear. At the same time, you will see the ALPHA mode annunciator. If the ALPHA annunciator is <u>not</u> on, the "VER" program is not present and you have accidentally executed the built-in VER function. In that case, do not insert any cards!

After you have verified the last card, press R/S or backarrow to clear the display of the CARD prompt. You will then see the message

PRESS R/S .

WARNING: <u>Be sure to press R/S</u> to restart the "VER" program. If you do not restart the program, the damage caused by VER will not be repaired. Even worse, important system pointers will be disrupted, probably causing keyboard lockup and MEMORY LOST. This is a common feature of this type of synthetic program. It is powerful and useful but quite unforgiving if you do not use it properly.

Cautions:

1) If you do not take your finger off the R/S key quickly enough, the calculator will pause several seconds or more while it tries to compute a line number. During this pause, the "PRESS R/S" message will remain in the display, and the PRGM annunciator will be off, just as if nothing is happening. Do not be fooled, and do not press R/S again. The program will restart itself. When the program finishes you will probably see some starbursts and other non-standard characters in the display.
2) "VER" cannot be called from a depth of more than one subroutine level (that is, when more than one RTN is pending).

"VER" program listing

```
01♦LBL "VER"     11 REGMOVE        21 STO IND Z      31 AOFF           41 DSE 10
 02 CF 25        12 "├*"            22 DSE 10        32 X<>Y         42 STO IND 10
 03 RCLFLAG      13 191            23 STO IND 10     33 SEEKPT          43 R↑
04 "ˣi♦Âˣ♦θ←♦x̄"  14 STO 10            24 R↑          34 RDN             44 STO 12
 05 X<> \        15 R↑              25 RCL \         35 SAVEX           45 END
 06 ENTER↑       16 STO 06          26 RCLPT         36 X<>Y
 07 X<> c        17 "├♦♦♦ā "         27 GETX       37 STO IND 10      112 BYTES
 08 RCL 04       18 X<> ]         28 "PRESS R/S"     38 LASTX
  09 "θ♦. "      19 STO IND Y       29 AON           39 X=Y?
 10 ASTO 63      20 X<> [           30 VER           40 STO 63
```

Barcode for "VER" is given in Appendix D.
Synthetic lines and their decimal equivalents (version 1):
 line Ø4 = 252, 1, 1Ø5, Ø, 19, 24Ø, 1, 137, Ø, 48, 3, Ø, 2
 line Ø5 = 2Ø6, 118 ; line Ø7 = 2Ø6, 125
 line Ø9 = 245, 16, Ø, 46, 24Ø, 191
 line 17 = 247, 127, Ø, Ø, Ø, 22, 191, 255
 line 18 = 2Ø6, 119 ; line 2Ø = 2Ø6, 117
 line 25 = 144, 118 ; line 3Ø = 167, 133 (not synthetic)
Version 2 differences:
 line Ø9 = 245, 16, Ø, 62, 24Ø, 191
 line 17 = 247, 127, Ø, Ø, Ø, 7, 223, 255

The concept used in the "VER" program is simple. The goal of the program is to recall location 1ØØ7 (see the extended memory map on page 179), execute the card reader's VER function, then restore location 1ØØ7. Lines Ø2 and Ø3 force an error stop if extended functions are not present. You may delete these lines if you will be using "VER" with an HP-41CX.

So that GETX can be used to recall location 1ØØ7, "VER" temporarily alters the second header register of the top file to simulate a 4Ø95-register data file. This powerful technique, invented by Clifford Stern, allows all of extended memory to be accessed, without normalization, by SAVEX and

GETX. The REGMOVE function (line 11) saves the two header registers of the original top file so that they can be restored before the program finishes.

Contrary to appearances, registers Ø4, Ø6, 1Ø, and 63 are not affected by "VER". Due to the program's manipulation of the operating system's pointer to register Ø1, the instructions like DSE 1Ø and STO Ø6 actually access internal operating system registers. The ASTO 63 function alters the bottom pointer register of the Extended Functions/Memory module in order to set the working file pointer to Ø1.

10E. A solution to the PURFL bug

The "PFF" (purge file fix) program presented here is provided especially for owners of revision 1B Extended Functions/Memory modules. It allows you to recover from an inadvertent use of a working file function when no working file exists. This situation can occur after a PURFL is executed, as explained on page 19. What actually occurs is that the top register of extended memory is overwritten by the partition code, which was mentioned in Section C of this chapter. This erroneous partition code tells the HP-41 that extended memory is empty.

The solution is simple. The "PFF" program just replaces the file name that belongs in the top register of extended memory, location 191 (decimal) on Figure 10.1 (page 179). Since no other extended memory registers are disturbed by the "bug", no further action is needed to restore the extended memory directory.

The "PFF" program that achieves this result was written by Clifford Stern. It uses synthetic techniques, since the affected register is not a normally accessible data register.

Because the PURFL bug is present only in the revision 1B Extended Functions/Memory module, owners of revision 1C or the HP-41CX can skip this section.

Instructions for "PFF":

1. First verify that your extended memory directory is indeed empty. If you did not empty it intentionally, and MEMORY LOST has not occurred, then you know that the top register of extended memory has been changed. The "PFF" program will then repair the damage, provided that you have not created any new files in extended memory. Once you create a new file, old files are overwritten and the damage cannot be fixed.
2. Put the name of the first extended memory file, up to seven characters, in the ALPHA register.

Note: If you do not remember the name, any string will do. For example, you could name it "TOP". Then, after "PFF" re-establishes your extended memory directory, you can GET the "TOP" file, check its contents to establish its identity, then use "PFF" again to give it the correct name.

3. Execute "PFF". The program finishes with an EMDIR instruction, both to establish a working file and to show you exactly what extended memory files were recovered. You may interrupt the extended memory directory if you like.

WARNING: "PFF" may be single-stepped, but never abandon "PFF" between lines Ø8 and 12, or MEMORY LOST will probably ensue. Under normal operation, "PFF" should be trouble-free.

"PFF" program listing

```
01♦LBL "PFF"
02 "⊢      "
 03 7
 04 AROT
 05 RCL c
 06 RCL [
  07 "ᶦiλ♦"
 08 ASTO c
 09 STO 00
 10 X<>Y
 11 STO c
 12 CLST
 13 EMDIR
 14 CLD
 15 END

41 BYTES
```

Line Ø2 is "Append 6 spaces"

Barcode for "PFF" can be found in Appendix D.

Synthetic lines and their decimal byte equivalents:
line Ø5 = 144, 125 ; line Ø6 = 144, 117
line Ø7 = 245, 1, 1Ø5, 11, 242, Ø
line Ø8 = 154, 125 ; line 11 = 145, 125

10F. Executing a program within extended memory

If a program file is contained entirely within the 127 registers of the Extended Functions/Memory module, it is possible to execute that program without doing a GETP first. Naturally, synthetic programming techniques are needed, but nothing too fancy. All that needs to be done is to alter the program pointer, two bytes of an internal register that designate what part of memory is displayed when you switch into PRGM mode.

WARNING: Before you try to execute a program in extended memory, make sure all its GTO's and XEQ's are compiled. That is, make sure that each and every GTO or XEQ instruction in the program has been executed at least once since the program was last edited or PACKed. This applies to local GTO and XEQ instructions (those that refer to labels 00-99, A-J, or a-e). If you fail to do this, executing the program in extended memory may invalidate the checksum, causing GETP to give a CHKSUM ERR message.

Note: If you do not care about the fact that GETP may not work on a program file that has been executed in extended memory, you may ignore this warning. In particular, if you plan to execute the program only in extended memory, the loss of GETP is not important. Moreover, the program "RPF" (Retrieve Program File) presented in section 10I can be used in place of GETP in case of checksum error.

If you will be needing to GET the program frequently or under program control, it will prove much more convenient just to make sure the GTO's and XEQ's are compiled before you save the program. A digression on the subject of compiled branch information should make things more clear.

The first execution of a local GTO or XEQ instruction causes the branch direction and distance to be stored within the GTO or XEQ instruction. The calculator thus remembers the location of the label the next time the GTO or XEQ is encountered. It is this storage of distance information within the instructions that invalidates the program file checksum. If the GTO's and XEQ's are already compiled before the program is saved, the checksum cannot be altered by execution of the program in extended memory. Indirect and global (Catalog 1) ALPHA GTO and XEQ instructions do not compile, so no special care is needed for them.

All compiled jump distance information is lost whenever you edit the program (make an insertion or deletion). It is also destroyed by PACKing, unless the program was already packed. Therefore, the following procedure is recommended to ensure that all GTO's and XEQ's are compiled before you save a program that you intend to execute in extended memory:

1. You need to have the program in main memory. If it is already saved in extended memory, you can just do a GETP.
2. Next compile all GTO's and XEQ's:

   GTO.. (use PACK if the program already has its own END)
   For each line that contains a local GTO or XEQ:
   GTO.nnn (go to the line containing the GTO or XEQ)
   SST in RUN (non-PRGM) mode to execute the instruction:
   Press and hold the SST key until the instruction appears in the display. Then release the SST key.
   When the instruction disappears, it has been executed.
   Repeat until you have SST'ed all local GTO's and XEQ's.
3. Now you can execute SAVEP with the program name in the ALPHA register to save the program.

The "EXM" (Execute eXtended Memory) program uses one synthetic instruction, an ASTO b. This instruction is used here to transfer a character from the ALPHA register into the rightmost two bytes of operating system register **b**, the location of the program pointer.

"EXM" Example: Suppose your first file in extended memory is the "JNX" program. You can use "EXM" to execute "JNX" without bringing it into main memory. Just load the Y and X registers with the two inputs needed (n and x), and press

XEQ ALPHA E X M ALPHA .

The result will appear in X when "JNX" is complete. The program pointer will remain in the "JNX" program unless you execute Catalog 1 or unless you GTO or XEQ a Catalog 1 label.

As it is listed here, "EXM" only allows the first file in extended memory to be executed. If, however, you know the absolute address of the second header register of the program file you want to execute, you can use that number in place of the number 190 (line 02) to execute a different program. But, to repeat, the program file must reside completely within the Extended Functions/Memory module. It must not spill over into an Extended Memory module.

"EXM" program listing

```
01♦LBL "EXM"
 02 190
 03 CLA
 04 XTOA
 05 RDN
 06 ASTO b
 07 END

19 BYTES
```

Barcode for the "EXM" program is given in Appendix D.

The decimal byte equivalents for line 06 are 154, 124.

Instructions for "EXM":

1. Make sure that the program file you want to execute is the first file in the extended memory directory. If it is not, compute the location of the file's second header register. This is 190 minus the number of registers used by the preceding files. Remember that the number of registers used by a file is 2 more than the number shown

in the extended memory directory. Replace line Ø2 of the "EXM" program with this computed number.

2. Make sure that the program file you want to execute lies entirely within the Extended Functions/Memory module. Add up the number of registers used by all files up to and including the file to be executed. Make sure to include the two header registers for each file that are not included in the extended memory directory display. This total should not exceed 127 registers.
3. As discussed in the warning above, all the GTO's and XEQ's in the saved program should be compiled if you expect to be able to use GETP to retrive the program file later. In emergencies, you can use the "RPF" (retrieve program file) program presented in section 1ØI.
4. Load the X, Y, and Z registers with any inputs needed by the function. The ALPHA register cannot be used for input, because it is cleared by "EXM".
5. Execute "EXM". Line Ø6 of "EXM" causes an immediate jump to the first line of the program immediately below the absolute register location designated in line Ø2.

CAUTION: Do not use "EXM" to execute a program containing a PSIZE instruction. Whenever PSIZE changes the SIZE, it revises the program pointer to compensate for the fact that all of the programs in main memory have been moved. Even though PSIZE does not move your program in extended memory, PSIZE will revise the program pointer as if the program had moved. This causes an unwanted jump. The only case in which this jump will not occur is when the PSIZE input happens to equal the current SIZE, so that the SIZE is unchanged.

10G. Suspending and Reactivating USER mode key assignments

As part of its compatibility with HP-67/97 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 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 assignment in your way? You press LOG to execute LBL D, but instead you get another function that you assigned to that key. Wouldn't it be nice if there were a way to eliminate the conflicting key assignment, then bring it back later?

Once again, synthetic programming comes to the rescue. A very short synthetic program called "SK" (Suspend Key assignments), written by Tapani Tarvainen, temporarily de-activates all USER mode function and global label key assignments. To suspend these key assignments, press

```
XEQ ALPHA S K ALPHA  .
```

A program called "RK" (Reactivate Key assignments), also written by Tapani Tarvainen, allows you to reactivate the dormant key assignments. When you execute "RK", a GETP will be performed on a special synthetic program file. This synthetic file must first be created in extended memory by using the "IN" (INitialize) program described on the next page. Actually, you could reactivate the key assignments by retrieving any program from extended memory using GETP. In the process of retrieving the program, the calculator will reactivate all dormant key assignments. This reactivation occurs whenever a program is brought into main memory, whether from magnetic cards, barcode, tape, or extended memory. The advantage of using "RK" is that the last program in Catalog 1 will not be disturbed. Any other type of GETP operation will overwrite the last Catalog 1 program (see page 16). Unless that is what you want, you should use "RK" rather than GETP.

1ØH. Saving key assignment status in extended memory

The HP821Ø4A magnetic card reader has a WSTS function that allows you to record key assignment information on magnetic cards. This makes it easy to keep several sets of function key assignments (global label assignments are not recorded). You just set up and record each set of function assignments, constructing a key assignment "library". Then when you want to use a particular key assignment configuration, you just read in the corresponding magnetic card.

Synthetic programming techniques let you use extended memory just as you would use magnetic cards to store key assignments. The programs "SAVEK" (SAVE Key assignments) and "GETK" (GET Key assignments), were written by Tapani Tarvainen and revised (LBL Ø4 section added) by Clifford Stern. These programs save key assignment information in extended memory and retrieve it on request. Unlike previous versions, they are fully compatible with Time module alarms and other I/O buffers. These programs also include "SK" (Suspend Key assignments) and "RK" (Reactivate Key assignments).

Caution: Before you use either "RK" or "GETK" for the first time, you need to use an initialization program called "IN", which is listed on page 198. Both the "RK" and "GETK" programs conclude with a GETP instruction, which has the effect of reactivating any new or dormant key assignments. A unique method, invented by Tapani Tarvainen, uses a synthetic program file as the object of the GETP instruction. This procedure eliminates the normal over-writing of the last program in main memory when GETP is executed. The synthetic program file has the name " " (a single space) and a length of zero bytes. The "IN" program, an esoteric creation of Clifford Stern, automates the procedure of creating this synthetic program file in extended memory.

Warning: Before you execute "IN", read the warning under "VER" on page 183. Then, if one of the two conditions listed there is satisfied, you can press

```
          XEQ "IN"
```

to create the synthetic program file needed by "RK" and "GETK". Once this is done, you do not need to use "IN" again as long as the synthetic file remains in extended memory. To make sure, run the extended memory directory. You should see one entry that displays as "   P001" .

<u>Instructions for "SAVEK" and "GETK"</u>

1. There must be an END above LBL"SAVEK in Catalog 1. Failure to observe this constraint will result in an eventual MEMORY LOST.
2. If you have not already done so, execute the "IN" program as described in the last two paragraphs to set up the synthetic zero-byte program file called " " in extended memory.
3. To save the current set of function key assignments, put a file name of up to 7 characters in the ALPHA register and execute "SAVEK". If you get a NO ROOM error message at line 43, there is not enough space left in extended memory to hold the key assignment information. You have the option of clearing extended memory space if you still want to save the key assignments. After clearing the space, start "SAVEK" from the beginning with the file name in ALPHA. When "SAVEK" finishes, the number in X indicates the size of the new key assignment data file.
4. To use "GETK", load the ALPHA register with the name of a key assignment data file that you created with "SAVEK". Then press XEQ "GETK". If you get a "NO ROOM" error message at line 72, there are no free program registers. "GETK" requires the initial presence of a free register, and the error trap at line 72 assures its existence. Thus, NO ROOM at line 72 indicates that you must decrease

the SIZE by 1 or delete a program to make space. If you get the "NO ROOM" message at line 82, there are not enough free program registers to hold the key assignments from the designated data file. The difference between the number in X and the current SIZE is the register deficiency. Again, you must decrease the SIZE or delete a program to eliminate this deficiency. If either one of these error stops occurs, you must either re-load ALPHA and XEQ "GETK" again, or XEQ "RK" to simply reactivate the global label key assignments and quit.

The "SAVEK" program saves key assignments of Catalog 2 or Catalog 3 functions, but it does not save assignments of Catalog 1 labels. The "GETK" program retrieves the function key assignment information from the designated extended memory file. These function key assignments are merged with any existing global label assignments. Previous function assignments are cleared. In case of a conflict, where a global label is already assigned to one of the keys used in the stored set of function assignments, the global label assignment will take precedence.

If you have a PPC ROM (see Appendix C), you can delete the LBL Ø4 section, lines 46-62, and replace the XEQ Ø4 instructions on lines Ø4 and 86 by XROM E? .

In case you were wondering, the RTN on line Ø1 is a necessary part of the "GETK" program. If "SAVEK"/"GETK" is the last program in main memory (that is, if it has .END. as its last line), the GETP at line 1Ø5 will transfer control to line Ø1.

"SAVEK"/"GETK"/"RK"/"SK" program listing

```
 01 RTN

02♦LBL "SAVEK"
 03 RCL [
 04 XEQ 04
 05 193
 06 X>Y?
 07 RTN
 08 SF 10

 09♦LBL 01
 10 -
 11  E3
 12 /
 13 "θ"
 14 RCL [
 15 XEQ 10
 16 SIGN

 17♦LBL 02
 18 RDN
 19 RCL IND Y
 20 ""
 21 X<> [
 22 "⊦*"
 23 X<> \
 24 X≠Y?
 25 GTO 03
 26 ARCL c
 27 X<> \
 28 STO IND Z
 29 FC? 10
 30 SAVEX
 31 ISG Z
 32 GTO 02

 33♦LBL 03
 34 X<> L
 35 X<> c
 36 R↑
 37 CLA
 38 STO [
 39 R↑
 40 INT
 41 FC?C 10
 42 RTN
 43 CRFLD
 44  E
 45 GTO 01

 46♦LBL 04
 47 RCL c
 48 "*"
 49 X<> [
 50 STO \
 51 ASHF
 52 RDN
 53 ALENG
 54 8
 55 Y↑X
 56 ATOX
 57 *
 58 512
 59 MOD
 60 ATOX
 61 +
 62 RTN

 63♦LBL 10
 64 RCL c
 65 "θ×iμx̄♦"
 66 ASTO c
 67 RTN

 68♦LBL "GETK"
 69  E
 70 SIZE?
 71 +
 72 PSIZE
 73 LASTX
 74 PSIZE
 75 X<>Y
 76 FLSIZE
 77 XEQ 10
 78 CLKEYS
 79 X<> c
 80 RDN
 81 +
 82 PSIZE
 83 RDN
 84 PSIZE
 85 X<> L
 86 XEQ 04
 87 RCL Y
 88 -
 89 192
 90 -
 91  E3
 92 ST/ Z
 93 X↑2
 94 /
 95 +
 96 X<>Y
 97 X<> c
 98 X<>Y
 99 REGMOVE
100 GETR
101 X<>Y
102 STO c

103♦LBL "RK"
104 " "
105 GETP
106 RTN

107♦LBL "SK"
108 .
109 STO '
110 STO e
111 END     212 BYTES
```

Barcode for this program can be found in Appendix D. The "IN" program is listed on the next page.

Synthetic lines and their decimal equivalents:

line Ø3 = 144, 117 ; line 11 = 27, 19 ; line 13 = 241, 16
line 14 = 144, 117 ; line 2Ø = 241, 24Ø ; line 21 = 2Ø6, 117
line 23 = 2Ø6, 118 ; line 26 = 155, 125 ; line 27 = 2Ø6, 118
line 35 = 2Ø6, 125 ; line 38 = 145, 117 ; line 44 = 27
line 47 = 144, 125 ; line 49 = 2Ø6, 117 ; line 5Ø = 145, 118
line 64 = 144, 125 ; line 65 = 246, 64, 1, 1Ø5, 12, 2, Ø
line 66 = 154, 125 ; line 69 = 27 ; line 79 = 2Ø6, 125
line 91 = 27, 19 ; line 97 = 2Ø6, 125 ; line 1Ø2 = 145, 125
line 1Ø9 = 145, 122 ; line 11Ø = 145, 127

Here is the "IN" program, to be executed before the first use of "RK" or "GETK":

```
     01♦LBL "IN"      10 REGMOVE      19 STO 01      28 X<> \       37 X>0?
     02 EMDIR         11 "├*"         20 X<> ]       29 STO 02      38 ASTO L
     03  E            12 RDN          21 STO 03      30 CLA         39 ASTO X
04 " ,ˣi♦Aˣ♦0←♦x̄"    13 RCL 12       22 RDN         31 STO [       40 SAVEX
     05 CRFLD         14 STO 06       23 EMDIR       32 R↑          41 X<> L
     06 +              15 "├♦ˣiλ"     24 CLD         33 X>0?        42 STO 01
     07 RCL \         16 RCL [        25 ST- T       34 E↑X         43 R↑
     08 X<> c         17 STO 12       26 X<>Y        35 SEEKPTA     44 X<> c
     09 RCL 04        18 X<>Y         27 STO 01      36 "ˣ"         45 END
                                                                      93 BYTES
```

Barcode for "IN" can be found in Appendix D.

Synthetic lines and their decimal byte equivalents:
line Ø3 = 27 ; line Ø4 = 254, 32, 44, 1, 1Ø5, Ø, 19, 24Ø, 1, 137, Ø, 48, 3, Ø, 2
line Ø7 = 144, 118 ; line Ø8 = 2Ø6, 125
line 15 = 247, 127, Ø, 1, 1Ø5, 11, 223, 255
line 16 = 144, 117 ; line 2Ø = 2Ø6, 119 ; line 28 = 2Ø6, 118
line 31 = 145, 117 ; line 36 = 241, 1 ; line 44 = 2Ø6, 125

If you have an HP-41CX, lines Ø2 and 23 can be replaced by EMROOM. As with "VER", despite appearances, no numbered data registers are disturbed by "IN".

Notwithstanding its brevity, "IN" is a <u>very</u> sophisticated program. In fact, if you think you are an expert in synthetic programming, you might try figuring out how it works. Clifford Stern is probably the only one who knows all the tricks it contains.

10I. Saving extended memory files on magnetic cards

Chapter 3 introduced the programs "WAS"/"RAS" which allow you to transfer ASCII file data to and from magnetic cards. The "WFL" (Write File) and "RFL" (Read File) programs presented in this section were written by Clifford Stern. They allow all types of files to be written onto magnetic cards, or just into data registers for more temporary storage. Furthermore, the absolute minimum number of registers is used. Seven bytes of data are saved per register, rather than the 6 or fewer bytes per register that "WAS" saves. A third program, "RPF" (Retrieve Program File), allows the retrieval of extended memory programs that have checksum errors. For example, a checksum error can result from running the program in extended memory (see Section 10F) if all the GTO's and XEQ's were not compiled before the program was saved.

Constraints common to "WFL", "RFL", and "RPF"

1. There must be an END above the "WFL"/"RFL"/"RPF" program in Catalog 1. Failure to observe this constraint will lead to an eventual MEMORY LOST.
2. Make sure that at least one of the conditions listed on page 183 is satisfied (no alarms and one free register, or at least one non-label key assignment).
3. The ALPHA register must contain a file name at the start of each of these programs. The sequence ALENG, 1/X (lines 14 and 15) cannot be deleted from the program, because the file name is required during the program's execution. The POSA instruction at line 17 causes an error stop at line 19 if the ALPHA register contains a comma. Commas are not allowed in the file name, because a comma is interpreted by the calculator as a name separator (see page 15). The comma and all the characters that follow it are ignored by all extended functions except SAVEP. If a comma were present, the ALENG error trap would be ineffective.

3. These programs may be called from another program, but not from a subroutine. In other words, no RTNs may be pending when one of these programs is called.

Instructions for "WFL"

1. Set flag 14 if you intend to write data onto protected cards.
2. Two modes of operation are available for "WFL". The first mode, obtained by clearing flag Ø1, writes the entire contents of the file. The second mode provides slower, but more economical, storage of ASCII files that are only partially filled. It first counts the number of characters in the file, then it transfers only those registers that are actually in use to the data registers. To activate this second mode, set flag Ø1. After you have checked the status of flag Ø1, put the name of the file in the ALPHA register, and execute "WFL".
3. If the PSIZE instruction at line 49 gives a NO ROOM error message, you will need to clear some programs or key assignments to make enough room for the file contents. The number in X indicates the required SIZE.
4. When the RDY Ø1 OF nn prompt appears, you can either feed in a magnetic card to record the file data or press R/S twice to bypass the writing of magnetic cards. Do not just press backarrow, or your file, which has been temporarily changed to a data file, will not be restored to its original file type. Do not change the contents of the stack before restarting "WFL". MEMORY LOST is the probable result. If the card reader is not present, the RDY Ø1 OF nn prompt will not appear at all. The file contents will simply be transferred to the data registers.
5. If you forgot to set flag 14 and you still want to write the data onto protected cards, press R/S twice to bypass the RDY Ø1 OF nn prompt. When the program finishes, you

can SF 14 and execute WDTA from the keyboard to write the cards.

6. When the last magnetic card is fed through, or after you press R/S twice, the program will conclude with an instruction sequence that leaves the new SIZE in X. This number, which represents the minimum required file size, should be written on the cards. It may be needed later for "RFL".

CAUTION: If your purpose is to immediately transfer the file data back into extended memory rather than recording it on magnetic cards, you must be very careful not to disturb it before using "RFL". The data is in a volatile, "non-normalized" form (see page 182). Any RCL, VIEW, or similar operation will alter the data. You must not try to move the data, or even look at it, until it has been written back into extended memory. This is the price for the efficient register-for-register storage in "WFL". The data format is the same as that used within extended memory, where all data is also non-normalized. If you accidentally recall or view a data register written by "WFL", you will have to execute "WFL" again to restore the correct data before using "RFL".

Instructions for "RFL"

1. As with "WFL", two modes of operation are available for "RFL". The mode is selected by the status of flag Ø1. Depending on the amount of space available in the calculator, the first mode (flag Ø1 clear) may work regardless of whether flag Ø1 was set for "WFL". To find out, clear flag Ø1, put the file name in the ALPHA register, and execute "RFL".
2. If a NO ROOM error message appears at line 49, and flag Ø1 was clear when you used "WFL" to write the cards, you will need to clear some programs to make more space available. The number in X is the required SIZE. If you

get NO ROOM but flag Ø1 was set when you used "WFL" to write the cards, then you have another option. You can set flag Ø1 to indicate that the SIZE does not need to be increased to the FLSIZE. Whenever you use "RFL" with flag Ø1 set, you must manually reSIZE to the number of registers written (this is the same number you wrote on the cards). This flag Ø1 option can also be used to read data file cards into a larger data file when the SIZE cannot be set to the new FLSIZE. Regardless of what action you take in response to the NO ROOM error message, you must reload the ALPHA register and execute "RFL" again.

3. If a card reader is present, the prompt CARD will appear. At this point you can feed in the data cards you made with "WFL". If the data is already in the registers, you can press R/S <u>twice</u> to bypass the card reading operation. As with "WFL", <u>do</u> <u>not</u> just press backarrow in response to the CARD prompt, or the file, which has temporarily been changed to a data file, will not be restored to its original type. Also, <u>do</u> <u>not</u> disturb the stack before restarting the program. If you do, MEMORY LOST is the likely result. [If no card reader is present, the card reading is automatically bypassed.]
4. The program will take the information from the data registers and transfer it into the designated extended memory file. It does not matter whether the file is a program, data, or ASCII (text) file.

One useful application of "WFL" and "RFL" is to minimize the number of registers needed for a "fixed" ASCII file, one that you will not be adding information to frequently. You can create a large ASCII file, fill it with the desired information, then use "WFL" with flag Ø1 set to write the records into data registers and possibly onto magnetic cards. Note the resulting SIZE (which appears in X at the conclusion of

"WFL"). Purge the ASCII file and create a new ASCII file of the same name with a FLSIZE equal to the "WFL" SIZE. Then execute "RFL", pressing R/S twice at the CARD prompt, to read the information back from the data registers into the new ASCII file, which is just the right size for the data.

Another "WFL"/"RFL" application is to deal with the problem of an ASCII file that has outgrown its original FLSIZE. Just clear flag Ø1, put the file name in ALPHA, and execute "WFL" to write the whole file to data registers (and cards if you like). Then purge the file, create a larger one with the same name, set flag Ø1, put the file name in ALPHA, and execute "RFL". As long as you have not disturbed the data registers since executing "WFL", you can safely press R/S twice at the CARD prompt to bypass it.

A third application of "WFL" and "RFL" is one-step recording of a data file, without any need for a GETR instruction. When used in place of GETR, "WFL" eliminates the need for resizing, since the program does it automatically. In addition, the data file's register pointer will be restored to its original value. This is a slight improvement over GETR. You can also use "WFL" to record a program file directly from extended memory, if you do not care that the recorded information is in a format that can only be used by "RFL". This might be the case, for example, if you wanted to record a program that you only execute in extended memory.

The "RPF" (retrieve program file) program retrieves a program from extended memory when a checksum error exists. This can be recognized by the CHKSUM ERR message when you try a GETP or GETSUB. If a checksum error does not exist, GETP or GETSUB is far preferable to "RPF", because "RPF" has the side-effect of changing the retrieved program file into a data file. If you suspect real damage to the program, use "RPF" as a "last resort", only if the program is not available on magnetic cards, tape, or barcode. Of course if the damage is due only to running the program in extended memory, "RPF" will retrieve a "clean" copy of the program.

The procedure used for "RPF" is a bit unusual, and the manual operations required preclude use of "RPF" as a subroutine. Follow the instructions carefully and precisely.

Instructions for "RPF"

1. GTO ..
2. Switch to PRGM mode. Check to make sure there is at least one free register. Then SST to get the .END. in the display, and insert any instruction (ENTER↑ will do), then delete it.
3. Put the name of the damaged program file in the ALPHA register and execute "RPF". If you get a NO ROOM error message at line 47, you will have to clear some programs out of main memory to make room for the program to be retrieved. Then start over at step 1.
4. PACK (press XEQ ALPHA P A C K ALPHA, not GTO ..)
5. At the conclusion of "RPF", the SIZE will be ØØØ and the program file will be changed to a data file.
6. Check the copy of the retrieved program for accuracy, in case more than GTO/XEQ compiling caused the CHKSUM ERR.
7. If the retrieved program was less than 35 bytes long, it may not be possible to use "RPF" again on the same file (which is now a data file). This condition will occur only if the byte count modulo 7 exceeds the FLSIZE. It is not likely that you will encounter this problem, because any program worth executing in extended memory should be much longer than 34 bytes.

Note: If you want to remove the "RPF" portion of this "WFL"/"RFL"/"RPF" program, delete lines 149-232, 122-123, 11-12, and Ø1-Ø4. This reduces the byte count to yield a 265-byte Write and Read Files program. Alternatively, if you have a PPC ROM (see Appendix C), you can replace lines 153-166 of the program by XROM "E?".

"WFL"/"RFL"/"RPF" program listing

```
01♦LBL "RPF"
 02 CF 01
 03 SF 05
 04 GTO 02

05♦LBL "WFL"
 06 CF 06
 07 GTO 01

08♦LBL "RFL"
 09 SF 06

 10♦LBL 01
 11 CF 05

 12♦LBL 02
 13 CF 25
 14 ALENG
 15 1/X
 16 44
 17 POSA
 18 CHS
 19 LN
 20 FLSIZE
21 "├      "
 22 7
 23 AROT
 24 X<> [
 25 X<>Y
 26 FC? 01
 27 GTO 04
 28 SF 25
 29 FS? 06
 30 GTO 04
 31 CLX
 32 SEEKPT

 33♦LBL 03
 34 CLA
 35 GETREC
 36 ALENG
 37 +
 38 FS? 25
 39 GTO 03
 40 RCLPT
 41 +
 42 8
 43 +
 44 7
 45 /

 46♦LBL 04
 47 PSIZE
48 "×i♦Ȧ×♦0←♦x̄"
 49 X<> \
 50 X<> c
 51 STO 10
 52 X<> [
 53 REGMOVE
 54 RCL c
 55 "├*"
 56 STO 06
  57 "├♦×iλ"
 58 CLX
 59 STO 07
 60 RCL \
 61 R↑
 62 CF 07
 63 X=Y?
 64 SF 07
 65 X<> [
 66 STO 12
 67 STO 01
 68 X<> ]
 69 STO 03
 70 2
 71 CHS
 72 LASTX
 73 FS? 07
 74 GTO 06
 75 .

76♦LBL 05
77 STO ]
78 LASTX
79 STO 01
80 R↑
81 FLSIZE
82 ST+ Y
83 2
84 ST+ Z
85 CLX
86 STO ]
87 RCL c
88 STO 01
89 R↑
90 SEEKPTA
91 RCL [
92 GETX
93 X≠Y?
94 GTO 05
95 X<> \
96 STO [
97 RCLPT
98 GETX

 99♦LBL 06
100 STO ]
101 "├♦"
102  E20
103 STO ↑
104  E
105 CHS
106 AROT
107 R↑
108 SEEKPT
109 R↑
110 .
111 X<> ]
112 FC? 07
113 SAVEX
114 FS? 07
115 SIGN
116 RDN
117 LASTX
118 STO 01
119 RCL a
120 X<> c
121 SF 25
122 FS? 05
123 GTO 07
124 TONE 8
125 FS? 06
126 RDTA
127 FS? 06
128 SAVER
129 FC? 06
130 GETR
131 FC? 06
132 WDTA
133 CF 25
134 STO c
135 STO 01
136 CLX
137 STO \
138 R↑
139 SEEKPTA
140 R↑
141 FC? 07
142 SAVEX
143 FC? 07
144 LASTX
145 STO 01
146 X<> a
147 STO c
148 SIZE?
149 RTN

150♦LBL 07
151 RCLPTA
152 STO 00
153 RCL c
154 "*"
155 X<> [
156 STO \
157 ASHF
158 ALENG
159 8
160 Y↑X
161 ATOX
162 *
163 512
164 MOD
165 ATOX
166 +
167 ENTER↑
168 "×i"
169 16
170 *
171 2
172 +
173 ENTER↑
174 XEQ 10
175 RCL 00
176 SIGN
177 CLX
178 XTOA
179 SEEKPT
180 PSIZE
181 X<> [
182 X<> c
183 FLSIZE
 184 "♦-"
185 STO ]
186 X<>Y
187 STO \
188 R↑
189 X<> [
190 STO 00

191♦LBL 08
192 GETX
193 STO IND ]
194 DSE ]
195 GTO 08
196 X<> [
197 RCL \
198 LASTX
199 7
200 MOD
201 SEEKPT
202 CF 25
203 LASTX
204 -
205 AROT
206 CHS

207♦LBL 09
208 "├♦"
209 DSE X
210 GTO 09
211 X<>Y
212 X<> [
213 STO 01
214 2
215 CHS
216 AROT
217 R↑
218 ENTER↑
219 XEQ 10
220 X<> [
221 X<> c
222 BEEP
223 STOP

224♦LBL 10
225 256
226 MOD
227 X<>Y
228 LASTX
229 /
230 XTOA
231 RDN
232 XTOA
233 END

419 BYTES
```

Barcode for this program can be found in Appendix D.

Note that line 21 is "Append 6 spaces".

Synthetic lines and their decimal byte equivalents:

```
line 24 = 206, 117
line 48 = 252, 1, 105, 0, 19, 240, 1, 137, 0, 48, 3, 0, 2
line 49 = 206, 118 ; line 50 = 206, 125 ; line 52 = 206, 117
line 54 = 144, 125
line 57 = 247, 127, 0, 1, 105, 11, 223, 255
line 60 = 144, 118 ; line 65 = 206, 117 ; line 68 = 206, 119
line 77 = 145, 119 ; line 86 = 145, 119 ; line 87 = 144, 125
line 91 = 144, 117 ; line 95 = 206, 118 ; line 96 = 145, 117
line 100 = 145, 119 ; line 101 = 242, 127, 0
line 102 = 27, 18, 16 ; line 103 = 145, 120 ; line 104 = 27
line 111 = 206, 119 ; line 119 = 144, 123
line 120 = 206, 125 ; line 134 = 145, 125
line 137 = 145, 118 ; line 146 = 206, 123
line 147 = 145, 125 ; line 153 = 144, 125
line 155 = 206, 117 ; line 156 = 145, 118
line 168 = 242, 1, 105 ; line 181 = 206, 117
line 182 = 206, 125 ; line 184 = 243, 192, 0, 45
line 185 = 145, 119 ; line 187 = 145, 118
line 189 = 206, 117 ; line 193 = 145, 247
line 194 = 151, 119 ; line 196 = 206, 117
line 197 = 144, 118 ; line 208 = 242, 127, 0
line 212 = 206, 117 ; line 220 = 206, 117
line 221 = 206, 125
```

10J. Key assignments of synthetic functions

If you do any synthetic programming, it is quite helpful to assign some frequently used two-byte synthetic functions to your USER mode keyboard. Chapter 4 of "HP-41 Synthetic Programming Made Easy" contains two of the most efficient programs to make synthetic key assignments.

The program "ASG" (assign) presented here was conceived by Tapani Tarvainen and optimized by Tapani and Gerard Westen. This program represents a major step forward in synthetic programming. Previous synthetic key assignment programs required the user to specify the function to be assigned in terms of its two decimal byte equivalents. "ASG" lets you simply spell out the function to be assigned.

WARNING: An END must precede LBL$^{\mathsf{T}}$ASG in Catalog 1. If you have "XF" as the first program in Catalog 1, this is already taken care of. If you do accidentally execute "ASG" when it is the first program in Catalog 1, keyboard lockup is likely and MEMORY LOST is possible.

<u>"ASG" Example 1</u>: Suppose you want to assign RCL b to a key. First, execute "ASG" (press XEQ ALPHA A S G ALPHA). The following message will appear in the display:

ASN

just as for the real ASN function, except that the ALPHA mode annunciator will be on. Now you fill in the name of the function to be assigned, in this case RCL b. The calculator is already in ALPHA mode, so you need only press

R C L (space) shift b

Then press R/S to restart the program. Wait about half a second, then press the key to which you want RCL b assigned. To assign RCL b to a shifted location, press the shift key, wait for the minus sign to appear (indicating a shifted location), then press the key to which you want the assignment made.

Once you have entered the function name and pressed the key to which the assignment is to be made, you need only wait for the "ASG" program to complete its work. The procedure is strikingly similar to the use of the built-in ASN function.

The "PASG" (programmable assign) entry point provides a synthetic key assignment capability similar to PASN. Spell out the synthetic function in ALPHA, put the row/column keycode in X, and execute "PASG". The keycode is the same one you would use for PASN.

The "PASG" portion of "ASG" accomplishes the amazing feat of decoding the function name into its decimal equivalents. This is done in two steps. In the RCL b example, "PASG" first assigns the function RCL and extracts the decimal code from the operating system registers, then the program decodes the suffix **b** into its decimal equivalent. These two values are used as input to a more standard key assignment program, "MKX", also a Tarvainen creation with optimization by Westen.

For adventurous novices: If you are unfamiliar with synthetic programming and you have been puzzling over the synthetic program listings, you may want to use "ASG" to create some synthetic instructions. A few basic points will help you avoid some of the simpler pitfalls. First and foremost, do not alter the contents of register **c** unless you know exactly what you are doing. The likely result is MEMORY LOST. Second, you should be aware that some synthetic lines in printer listings appear differently in the display. Most notable are text lines, in which characters with decimal codes 128-255 disappear, and instructions that access some of the operating system registers. The equivalence is:

display	printer listing
STO M	STO [
STO N	STO \
STO O	STO]
STO P	STO ↑
STO Q	STO _
STO ├	STO ᵀ

The STO prefix could just as well be RCL, X<>, or any other two-byte prefix. The M, N, O, and P registers make up the ALPHA register. These four, plus Q and a, are the safest to experiment with.

Of course, full details of the operating system registers and their uses can be found in the book "HP-41 Synthetic Programming Made Easy". Also given are techniques for creating synthetic text lines, which cannot be made with "ASG".

Instructions for "ASG"

1. Make sure there is an END above LBL^{T}ASG in Catalog 1, and that there is no LBL^{T}ANUM in Catalog 1. Failure to observe these restrictions will lead to MEMORY LOST.
2. Execute "ASG". The prompt ASN will appear, and the ALPHA mode annunciator will be lit.
3. Using the ALPHA mode keys, spell out the function to be assigned. If the suffix is a synthetic character, you may spell out a decimal number, Ø to 255, instead. If you like, the prefix can also be a decimal number. For indirect functions like GTO IND X, you do not actually need to spell out "IND". As long as there are two spaces between the GTO and the X, the function GTO IND X will be assigned.
4. Press R/S to restart the program.
5. Wait half a second and press the key to which the function is to be assigned. For a shifted assignment press the shift key, wait a moment until "-" appears in the display, then press the desired key. It is not necessary to press R/S again.
6. The program will proceed to make the synthetic function assignment. Should an error stop occur, do not attempt to restart the program. Instead, start over with step 2 above. If the error was NO ROOM at line 5Ø, decrease the SIZE or clear a program.
7. To make another assignment, execute "ASG" again.

Instructions for "PASG"

1. Make sure there is an END somewhere above LBL^{T}PASG in Catalog 1, and that there in no LBL^{T}ANUM in Catalog 1. As for "ASG", the penalty for failing to heed these restrictions is MEMORY LOST.
2. Load the ALPHA register with a string that spells out the function to be assigned. See item 3 in the "ASG" instructions for an explanation of the various types of strings that are allowed.
3. Put the row/column keycode in X. "PASG" works just like PASN in this respect.
4. Execute "PASG". The program will make the synthetic assignment. As with "ASG", do not attempt to restart after an error stop occurs. Start over with step 2.
5. To make another assignment, load ALPHA and X and execute "PASG" again.

Instructions for "MKX"

1. Put the decimal prefix code in Z, suffix code in Y, and keycode in X.
2. Execute "MKX" to make the desired assignment.

General cautions for "ASG", "PASG", and "MKX":

1. Do not interrupt or SST these programs. If you accidentally interrupt or SST the program between lines 158 and 161, you will have to re-execute the program. If you interrupt the program after line 176, you must restart the program to avoid an eventual MEMORY LOST.
2. Make sure there is no global label present that has the same name as a function you want to assign. For example, if you have a LBL "STO" in Catalog 1, "ASG" and "PASG" will not be able to assign a synthetic STO instruction.
3. **WARNING:** Make sure there is no global label "ANUM" in Catalog 1. If you have a LBL^{T}ANUM in Catalog 1, memory will be completely trashed.

The "ASG", "PASG", and "MKX" programs are fully compatible with Time module alarms and other I/O buffers. "ASG" and "PASG" also allow you to assign Catalog 1 labels and nonsynthetic functions, as well as synthetic functions. In fact, "ASG" and "PASG" are essentially direct replacements for the ASN and PASN functions. They will accept any input that ASN or PASN accepts, plus many more that correspond to synthetic functions.

Examples of "ASG" and "PASG"

The following list shows typical key assignments, both nonsynthetic and synthetic, and the "ASG"/"PASG" ALPHA inputs needed to obtain them. Several variations on the ALPHA input are usually possible, as the list shows. Any functions that show only decimal inputs are more easily assigned with "MKX" by putting the decimal inputs in the stack.

Function	ALPHA input
"ASG"	"ASG" (any global label can be assigned
"VER"	"VER" using "ASG" or "PASG")
BST	"BST"
SIGN	"SIGN"
SF 14	"SF 14" or "168 14"
STO N	"STO N", "STO 118", "145 N", or "145 118"
X<> M	"X<> M", "X<> 117", or "2Ø6 117"
GTO IND X	"GTO IND X", "GTO X" (note 2 spaces), "GTO I 115", "GTO 243", "2Ø8 243", etc.
RCL IND X	"RCL IND X", "RCL X" (2 spaces), "RCL I X", "RCL 243", "145 243", etc.
XROM 29,Ø8	"XROM 29,Ø8" or "X 29,Ø8" (=PRA)
TONE 1Ø	"TONE 1Ø" or "159 1Ø"
TONE 89	"TONE 89" or "159 89"
FIX 1Ø	"FIX 1Ø" or "156 1Ø"
XROM 28,35	"XROM 28,35" or "X 28,35" (=OUTA)

(continued on page 213)

"ASG"/"PASG"/"MKX" program listing

```
01♦LBL "ASG"
02 RCLFLAG
03 SIGN
04 ""
05 ASTO d
06 "ASN "
07 STOP
08 CF 21
09 "├ "

10♦LBL 01
11 31
12 AVIEW
13 GETKEY
14 X=0?
15 GTO 01
16 X≠Y?
17 GTO 02
18 "├-"
19 FS?C 03
20 GTO 01
21 2
22 CHS
23 AROT
24 ATOX
25 ATOX
26 SF 03
27 GTO 01

28♦LBL 02
29 ARCL X
30 AVIEW
31 FC? 03
32 CHS
33 X>0?
34 XTOA
35 LASTX
36 STOFLAG
37 ATOX
38 ATOX
39 LN
40 CHS
41 AROT
42 ASHF
43 R↑

44♦LBL "PASG"
45 AOFF
46 32
47 POSA
48 X>0?
49 GTO 03
50 RDN
51 PASN
52 CLD
53 RTN

54♦LBL 03
55 "├♦"
56 AROT
57 "├@0"
58 ATOX
59 POSA
60 ISG X
61 AON
62 AROT
63 44
64 POSA
65 ISG X
66 GTO 04
67 AROT
68 R↑
69 ANUM
70 ASHF
71 ANUM
72 640
73 +
74 64
75 *
76 +
77 RCL X
78 256
79 ST/ Z
80 MOD
81 GTO 06

82♦LBL 04
83 R↑
84 ANUM
85 ATOX
86 84
87 -
88 X<=0?
89 GTO 04
90 CHS
91 7
92 +
93 X>0?
94 GTO 05
95 CHS
96 31
97 MOD
98 2

99♦LBL 04
100 X<0?
101 9
102 X>0?
103 +
104 X>0?
105 3
106 X>0?
107 +

108♦LBL 05
109 17
110 +
111 X>0?
112 95
113 X>0?
114 +
115 X>0?
116 X<>Y
117 CLX
118 POSA
119 X>0?
120 AROT
121 CLX
122 X<> d
123 R↑
124 SF 25
125 PASN
126 RCL _
127 STO ↑
128 RDN
129 X<>Y
130 X<> d
131 X≠0?
132 ATOX
133 ASHF
134 X=0?
135 ANUM
136 R↑
137 FS? 48
138 GTO 06
139 X<>F
140 R↑
141 208
142 R↑
143 X≠Y?
144 SF 07
145 X<=Y?
146 X=Y?
147 RDN
148 X>Y?
149 174
150 R↑
151 X<>F

152♦LBL 06
153 AOFF
154 R↑

155♦LBL "MKX"
156 "ANUM"
157 PASN
158 RCL '
159 CLA
160 STO ↑
161 "├      B"
162 ASTO \
163 ARCL \
164 R↑
165 XTOA
166 R↑
167 XTOA
168 RCL \
169 ATOX
170 SIGN
171 AROT
172 RCL [
173 RCL c
174 "@×iλX̄♦"
175 ASTO c
176 R↑

177♦LBL 07
178 RCL IND L
179 "**"
180 STO \
181 X<> [
182 STO ]
183 ASTO [
184 ASHF
185 X<> \
186 ASHF
187 X≠Y?
188 X<> [
189 X=Y?
190 R↑
191 "├****"
192 STO \
193 "├"
194 X<> [
195 STO ]
196 ARCL c
197 X<> ]
198 STO IND L
199 RDN
200 X≠Y?
201 ISG L
202 X≠Y?
203 GTO 07
204 X<> Z
205 STO c
206 CLST
207 CLA
208 CLD
209 END
372 BYTES
```

Barcode for this program can be found in Appendix D. Line 146 is a text line "ANUM", not the instruction ANUM.

Synthetic lines and their decimal byte equivalents:

line Ø4 = 242, 132, 128 ; line Ø5 = 154, 126
line 55 = 242, 127, Ø ; line 57 = 243, 127, 64, 48
line 122 = 2Ø6, 126 ; line 126 = 144, 121
line 127 = 145, 12Ø ; line 13Ø = 2Ø6, 126
line 158 = 144, 122 ; line 16Ø = 145, 12Ø
line 161 = 243, 127, 166, 66 ; line 162 = 154, 118
line 163 = 155, 118 ; line 168 = 144, 118
line 172 = 144, 117 ; line 173 = 144, 125
line 174 = 246, 64, 1, 1Ø5, 11, 2, Ø ; line 175 = 154, 125
line 18Ø = 145, 118 ; line 181 = 2Ø6, 117
line 182 = 145, 119 ; line 183 = 154, 117
line 185 = 2Ø6, 118 ; line 188 = 2Ø6, 117
line 192 = 145, 118 ; line 193 = 245, 127, 132, 132, 132, 24Ø
line 194 = 2Ø6, 117 ; line 195 = 145, 119
line 196 = 155, 125 ; line 197 = 2Ø6, 119
line 2Ø5 = 145, 125

For faster operation with an HP-41CX or Time module use:
line 156 "SW" ; line 161 = 243, 127, 166, 154

(continued from page 211)

Fancier synthetic functions:

GTO.ØØØ	"199 133"	(works in PRGM mode, only when the card reader is attached)
eGOBEEP	"4 167" or "Ø 167"	(gives mass storage or printer functions; experiment in PRGM mode)
Q-Loader	"27 Ø"	(experiment in PRGM mode)
byte grabber	"247 63"	(not for novices; can give MEMORY LOST if inserted above an END)

The "ASG" program is perhaps the most advanced synthetic program ever written, in that it makes use af a wide variety of synthetic techniques to provide a very high degree of user convenience. I hope you enjoy using it.

10K. "Crash" recovery tips

A "crash" is a condition in which the keyboard is "locked up" and fails to respond, or in which Catalog 1 is damaged. There is usually no problem recognizing a crash, but recovering from one is another story. Unfortunately, MEMORY LOST is necessary to recover from many types of crashes.

If the keyboard "locks up" and you cannot get any response from the R/S key or the ON switch, there are several techniques that may help you regain control:

1. Press and hold the backarrow key, press the R/S key, release the R/S key, and release the backarrow key.
2. Newer HP-41's (1982 or later, approximately) have a reset feature. Check your Owner's Manual before trying this, because on older HP-41's it gives MEMORY LOST. It will also give MEMORY LOST on a newer HP-41 if the keyboard is not locked up. Press the ON key to turn the calculator off, then press and hold the backarrow key. Press and release the ON key, then release the backarrow key.
3. Remove the batteries for a few seconds and then replace them. This will clear all but the most serious crashes.
4. The next thing to try, if you have a card reader, is to insert a card (any type). If the card is not pulled through, remove the batteries for a few seconds and reinsert them, with the card still in place. The card should be pulled through and the display should respond, without MEMORY LOST. This technique was developed by Clifford Stern.
5. Remove the batteries and reinstall them with each cell reversed (this cannot be done with the HP rechargeable battery pack). Press and hold the ON key for 10 seconds. Replace the batteries in the normal polarity and press the ON key. You should get MEMORY LOST.
6. Simply removing the batteries overnight will usually not clear a serious crash. Older HP-41's can retain their

memory without batteries overnight, and newer HP-41's can retain their memory for a week or more.

If the keyboard does respond, but Catalog 1 is not normal, you will usually have to clear the calculator (using the ON and backarrow keys in the sequence described in your Owner's Manual). However, if you have a PPC ROM (see Appendix C), you may be able to restore Catalog 1 along with all of your programs. Try this sequence, developed by Clifford Stern:

```
     ALPHA C Ø Ø Ø 2 D ALPHA   (You can use spaces in place of
                                 the zeros to save a few keystrokes.)
     XEQ "HN"
     XEQ "E?"
```

If this result is less than 192 or more than 511, stop here. Otherwise, continue:

```
     XEQ "SX"
     PACK
```

Check Catalog 1 to see what programs were recovered.

This sequence will not deal with cases in which the pointers to the .END. or register ØØ have been altered. For these cases you need a PPC ROM, knowledge of the structure of system scratch register **c** (see page 11Ø of "HP-41 Synthetic Programming Made Easy"), persistence, and some luck.

SOLUTIONS TO PROBLEMS

3.1. "*" APPREC DELREC RCLPT

3.2. Here is one solution:

```
01 LBL "PAS"     (print ASCII file)
02 CLX
03 SEEKPTA
04 SF 25
05 LBL 01
06 GETREC
07 FC? 25
08 RTN
09 ACA
10 FS? 17
11 GTO 01
12 PRBUF
13 ADV
14 GTO 01
15 END
```

4.1. One solution is:

```
XTOA      add the designated character.
SIGN
CHS
AROT      rotate it to the front of ALPHA.
```

The sequence SIGN, CHS, is much faster than a digit entry -1.

4.2. Here Is a typical sequence to ASTO a string:

```
(start register number)
ENTER↑
LBL 01
ASTO IND Y
```

```
          RDN
          ALENG
          X>Ø?
          GTO Ø1
```

4.3a. To delete n characters from the left:

```
          n             a digit entry line
          X=Ø?
          RTN           quit if n=Ø
          LBL Ø1
          ATOX          delete a character
          RDN
          DSE X         decrement n
          GTO Ø1
```

4.3b. To delete n characters from the right:

```
          n
          CHS           rotate n characters from the right
          AROT          end to the left end of ALPHA.
          CHS
          continue using the sequence from 4.3a.
```

4.4. Starting with "firstname initial lastname" or "firstname lastname" in the ALPHA register, the following sequence will produce "lastname, firstname initial" or "lastname, firstname":

```
          32
          POSA       find the first space character
          "├, "      append a comma and a space
          AROT       rotate firstname behind lastname
          ATOX       remove space that followed firstname
          POSA       find the next space
          44
          POSA       find the comma
          X<>Y
          X<Y?       if space is in front of comma
          X<Ø?       and if space was found
          RTN        then skip the RTN and continue
          "├ "       append a space after firstname
          AROT       rotate middlename behind firstname
          ATOX       remove space that followed middlename
```

4.5.
```
          PI
          *
          RCLFLAG
          X<>Y
          SIN
          X<>Y
          STOFLAG
          X<>L
          /
```

4.6. The program "FE" (FIX/ENG) listed here preserves the status of flags 36-39, while setting flags 4Ø (FIX) and 41 (ENG). The approach is similar to "FEX", except that another RCLFLAG is needed at the beginning to save the status of flags 36-39:

```
          Ø1 LBL "FE"
          Ø2 RCLFLAG        Save status of flags Ø-39
          Ø3 ENG Ø          Set flag 41
          Ø4 RCLFLAG        Save status of flag 41
          Ø5 FIX Ø          Set flag 4Ø
          Ø6 X<>Y
          Ø7 .39
          Ø8 STOFLAG        Restore flags Ø-39
          Ø9 R↑         This sequence is faster than
          1Ø R↑           the alternative: RDN, RDN
          11 41
          12 STOFLAG        Set flag 41
          13 R↑
          14 R↑
          15 END
```

4.7. The program "BR" (block rotate) listed below is one possible solution. The first 10 lines of this program form the sum 1.001*sss+.000001*(nnn-1). Lines 11-21 add 1 if rrr<0 or .001 if rrr>0. At this point the number in x is

sss.(sss+1)(nnn-1) if rrr>0, or
(sss+1).sss(nnn-1) if rrr<0.

The absolute value of rrr (line 14) ends up in Y, where it can be used as a DSE counter in the LBL 01 loop.

"BR" program listing

```
01♦LBL "BR"     08 1 E-6       15 X<> T       22♦LBL 01
02 .1           09 *           16 SIGN        23 REGSWAP
03 %            10 ST+ Y       17 CHS         24 DSE Y
04 +            11 X<> L       18 X>0?        25 GTO 01
05 X<>Y         12 SQRT        19 X<>Y        26 END
06 1            13 R↑          20 RDN
07 -            14 ABS         21 +          43 BYTES
```

APPENDIX A
The VER and 7CLREG bugs

If your card reader is a revision 1G or higher, you may skip to the discussion of the 7CLREG bug on the next page. To find out which revision you have, run Catalog 2. If you see one of these headers:

```
CARD READER
CARD RDR 1D
CARD RDR 1E
CARD RDR 1F
```

then your card reader has the VER bug. If you see

```
CARD RDR 1G
```

then your card reader does not have the VER bug.

Here is the full story on the VER bug (for card readers up to 1F). When the card reader's VER (verify) function is executed with an extended memory module plugged into port 2 (**port numbers** are shown on the bottom of your HP-41 next to the serial number), the first register of that module will be altered. The same warning applies to having an extended memory module plugged into port 4 of a port extender or built into a dual Extended Memory module (see Appendix C).

When you use VER under these conditions, one register (decimal location 1ØØ7) of your data or program information in extended memory will be incorrect, unless there was no data in the port 2 module. It is even possible that the altered register will be a file header register, disrupting the extended memory directory.

If this discussion is not completely clear to you, come back to it after you read Chapters 2 and Section 1ØC. For the present, just refrain from executing the card reader function VER if you have an extended memory module in port 2. If you must have an XMemory module in port 2, at least make sure that the module in port 1 or 3 will be filled before the one in port 2 is used. This is easy to do:

1) If you have only one XMemory module, put it in port 1 or 3.
2) If you have two XMemory modules, install them at the same time (while the calculator is turned off, of course).

If you follow this procedure, the register affected by VER will be the 366th register of extended memory. If you add up the file sizes shown in the extended memory directory and add 2 more registers for each file header, you can figure out which file contains the 366th register. That file should be checked or purged after a VER operation. If the 366th register is the second of the two header registers for a file, that file and all the following files will probably be lost. This paragraph will become clear after you read Section 1ØC.

Now for the good news. It is possible to completely eliminate the destruction of the 366th extended memory register. The synthetic program "VER" introduced in Section 1ØD does the job, in less time than it takes to press XEQ "VER". If you have a port extender (Appendix C), another technique is almost as handy. Just switch off all XFunction and XMemory modules before executing VER.

All card readers have the 7CLREG bug. The card reader's 7CLREG function is intended to simulate the HP-67/97 CLREG function. This 7CLREG function can ruin an entire module of extended memory. If you execute 7CLREG when the SIZE is less than 25, some of the data near register 36Ø of extended memory will be lost. This assumes that the recommended module plug-in procedure was used, so that the module in port 1 or 3 contains register 365 as its last register. In addition, 7CLREG is likely to cause all extended memory data starting at register 366 to become inaccessible. The solution is to avoid using 7CLREG, or to precede it with the sequence SIZE?, 25, X>Y?, PSIZE (see pages 74-75) to ensure a SIZE of at least 25.

APPENDIX B
EXECUTION TIMES FOR EXTENDED FUNCTIONS

Whenever you write a program, you face choices between different ways of obtaining the same result. A table of execution times for various functions is a helpful tool in making these choices. For example if you want to put the value 1 in X, you might not be aware that the sequence CLX, SIGN is almost 40 milliseconds (65%) faster than the usual digit entry 1. In a loop which will be executed many times, this difference could be worth the extra byte of program space used.

A table of execution times for important built-in (Catalog 3) functions can be found on pages 145 and 146 of "HP-41 Synthetic Programming Made Easy". Execution times, in milliseconds, are presented in this appendix for most of the extended functions, including the ones for which you are likely to have alternatives, so that you can make the best choice of functions when writing your own programs.

These execution times were measured by Clifford Stern, using his Time module application program that was presented in "HP-41 Synthetic Programming Made Easy". Although each timing run was automated, the entire process was still quite an effort because of the large number of variables that affect execution time.

Execution times for the extended functions

All times are listed in milliseconds (thousandths of a second)

ALENG	168-3.5C-5.8INT((C-1)/7), where C is the number of characters in the ALPHA register.
ANUM	95 to 39∅ (unpredictable)
APPCHR	237+1∅.2C+12.1R+file increments* where C is the number of characters appended, and R is the record number. This formula assumes R is the last record of the file; otherwise APPCHR will be slower and less predictable.
APPREC	211+6.9C+12.1R+file increments
ARCLREC	458+12R+file increments
AROT	X>∅: 286-7.77C-6.6INT((C-1)/7)-1∅.9X
	X<∅: 287-7.77C-6.6INT((C-1)/7)-1∅.9(C-|X|)
ATOX	X=∅: 145
	X>∅: 167-4.28C
CLFL	199+3.24·FLSIZE+file increments
CLKEYS	326+12.2G, where G is the number of global (Catalog 1) LBLs present.
CRFLAS	246+3.6·FLSIZE
CRFLD	246+3.6·FLSIZE
DELCHR	unpredictable, but slow
DELREC	unpredictable, but slow
FLSIZE	72.9+file increments
GETP	1∅99+115·FLSIZE
GETR	58.5+12.5·FLSIZE+2.5(SIZE-FLSIZE)+file increments
GETREC	35∅+12.1R+file increments

*File increments are: 12.9(N-1)+9E for the working file, or 136+12.3(N-1)+9E for a named file, where N is the file number in the extended memory directory (1 and up) and E is the number of the extended memory block in which the file resides. E can be ∅, 1, or 2(see Figure 1∅.1).

GETRX	110+9.9D+file increments,
	where D is the number of data registers retrieved.
GETX	63.5+file increments
INSCHR	unpredictable, but slow
INSREC	unpredictable, but slow
PASN	depends on search time of Catalogs 1, 2, and 3 until the named label or function is found.
PCLPS	700+1.0P+16G,
	where P is the number of program registers cleared.
POSA	83.5 to 281
POSFL	190+17.0C+24.5R+file increments
	where C is the number of characters scanned, and R is the number of records scanned.
PSIZE	746+4.1X
PURFL	270+file increments
	This formula assumes you are purging the last file in the extended memory directory. Otherwise PURFL will be slower and less predictable.
RCLFLAG	36.8
RCLPT or **RCLPTA**	102+file increments
REGMOVE	77+6.5D,
	where D is the number of data registers in the block.
REGSWAP	75.6+7.4D
SAVEP	350+90·FLSIZE+file increments
SAVER	56+12.5·SIZE+file increments
SAVERX	INT(X)=0: 102.6+9.9D
	INT(X)≠0: 109.3+9.9D
SAVEX	59.6+file increments
SEEKPT or **SEEKPTA**	Data files X=0: 69.6+file increments
	.001≤X<1: 65.2+file increments
	ASCII files X=0: 102.3+file increments
	.001≤X<1: 96.2+file increments
	X≥1 Slower and less predictable.

SIZE?	56+2.6X	
STOFLAG	X=alpha data:	35.1
	X=bb.ee:	approx. 58+20F,
	where F is the number of flags restored.	
X<>F	100	
XTOA	X=numeric:	approx. 48
	X=alpha data	47+3.7C,
	where C is the number of characters appended, plus 1 or 2 ms if ALPHA is not clear.	
ASROOM	unpredictable	
CLRGX	67.6+1.19D	
EMROOM	91+file increments	
RESZFL	unpredictable	
ΣREG?	53	
XcompareNN?	45 to 50 ms, minus about 4 ms if Y=0.	

APPENDIX C

HP-41 BOOKS, PUBLICATIONS, AND MODULES

This appendix lists several excellent sources of further information about your HP-41 system. These range from the introductory to the very advanced.

1. An Easy Course in Programming the HP-41, a book by Ted Wadman and Chris Coffin. This is by far the best book for anyone who has trouble getting through the HP-41 Owner's Manual. If you know a calculator novice who needs a very easy to read, simple, introductory book on the HP-41 calculator, this is it. If your dealer does not carry this book, you can order it from:

Grapevine Publications, Inc. , Dept. X
P.O. Box 25724
Portland, OR 97225 U.S.A.

The price per copy is $15.00 plus shipping, which is $2.00 (USA), $3.50 (Canada), or $6.00 (elsewhere). Checks must be payable through a U.S. bank.

2. HP-41 Synthetic Programming Made Easy, a book by Keith Jarett. 192 pages, plastic spiral bound. The most up-to-date and readable introduction to the fascinating subject of synthetic programming. Contains application programs for the Extended Functions and Time modules. Works equally well with the HP-41C, CV, or CX. Includes a plastic Quick Reference Card for Synthetic Programming, a $3.00 value. If your dealer does not have this book, you may order it directly from:

SYNTHETIX, Dept. X
P.O. Box 113
Manhattan Beach
CA 90266 U.S.A.

The price per copy, is $16.95 plus shipping, which is $1.00

(USA, book rate), $2.00 (USA, United Parcel), $3.00 (USA or Canada, air mail), or $5.55 (elsewhere, air mail). California residents add sales tax. Checks must be payable through a U.S. bank. This same price and shipping schedule applies to "HP-41 Extended Functions Made Easy".

3. The HP-41CX Owner's Manual, in two volumes. An excellent general HP-41 reference. These are the best, most complete calculator manuals HP has ever produced. Order them through your dealer or direct from HP (call 800-538-8787). The part numbers are 00041-90474 (Vol. I) and 00041-90492 (Vol. II).

4. Synthetic Programming on the HP-41C, a book by William C. Wickes. 92 pages, softbound. The first book on synthetic programming, and an excellent follow-up to "HP-41 Synthetic Programming Made Easy" (above). Contains many useful details needed to complete your knowledge of synthetic programming. You can order it from:

Larken Publications
4517 NW Queens Ave.
Corvallis, OR 97330 U.S.A.

The price is $11.00 postpaid, by surface mail. For airmail, add: $1.00 (USA, Canada, Mexico), $2.00 (Europe, South America), or $3.00 (elsewhere). Checks must be payable through a U.S. bank.

5. Calculator Tips and Routines (Especially for the HP-41), a book edited by John Dearing. 130 pages, spiral bound. This book contains many listings of routines from the PPC ROM (see item 6), plus a great number of other short, useful instruction sequences and tips. This book is available from dealers or directly from:

Corvallis Software, Inc.
P.O. Box 1412
Corvallis OR 97339-1412 U.S.A.

The price is $15 within the USA and Canada, $2Ø elsewhere, airmail prepaid. Checks must be payable through a US bank.

6. The 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 system. The PPC Calculator Journal is the most up-to-date and comprehensive source for such information.

To obtain PPC membership information and a sample Journal, send a 9" by 12" self-addressed stamped envelope with 3 ounces of postage to:

PPC, Dept. XF
2545 W. Camden Place
Santa Ana, CA 927Ø4 USA

7. The PPC ROM, an 8K custom ROM module designed by PPC members and manufactured by Hewlett-Packard. The PPC ROM contains 122 programs of general utility, and it comes with a 5ØØ-page User's Manual. It is an excellent value both for its utility and as a learning tool, because all the programs are fully documented and accompanied by line-by-line analysis.

Many calculator dealers now carry the PPC ROM. You may also write to PPC at the above address for price and ordering information. Mark the lower left corner of your outer envelope "PPC ROM ordering info" and enclose a self-addressed, stamped envelope if possible. A substantial discount on the PPC ROM is available to PPC members. This discount could almost pay for your first year's membership.

8. The AME Port-X-Tender, a flat, thin box that fits under the HP-41 and adds six more plug-in positions, for a total of ten. The 6 extra slots can be used for any modules or peripherals, including the HP-IL module. These extra slots are switchable,

allowing you to switch between two sets of extended memory if you have an HP-41C or CV (this will not work with the CX because some of the extended memory is internal and cannot be switched). A lithium battery maintains the contents of all modules, whether switched on or not. The Port-X-Tender plugs into port 3 with a short cable. The box is held in place with fabric fasteners. No modifications to your HP-41 are required. If your dealer does not carry the Port-X-Tender, mail your order to:

AME Design
2554 Lincoln Blvd. Suite 5ØØØ
Marina Del Rey, CA 9Ø291 U.S.A.
Telephone (213)-3Ø6-1249

The US price is $149.95 plus $5.ØØ for shipping. Elsewhere, please write for price information. California residents please add sales tax.

9. Double and Triple XFUNCTIONS and XMEMORY modules. These are multiple modules in a single package. This frees some of your 4 ports for other uses, so you don't have to resort to swapping modules in and out of the calculator. The prices are:

2 XMEMORY	$16Ø.ØØ	(for the HP-41C, CV, or CX)
XFUNCTIONS + 1 XMEMORY	$16Ø.ØØ	(for the HP-41C or CV only)
XFUNCTIONS + 2 XMEMORY	$25Ø.ØØ	(for the HP-41C or CV only)

These prices include manuals and shipping. California residents add sales tax. Mail your order to:

Software, Operations, and Systems Co.
945 Medford Rd.
Pasadena, CA 911Ø7 U.S.A.

Trade-in credit is available for any single modules you may have. Write for details. Other multiple modules are also available.

APPENDIX D
BARCODE FOR PROGRAMS

Barcode is provided here for all but the shortest 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 dark sheet of paper behind the barcode to improve the contrast.

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 or tape drive, you should record these programs in case your dog or cat finds this book. Extended memory should not be considered as permanent storage, since it is susceptible to MEMORY LOST.

BESSEL FUNCTION

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 12

ROW 1 (1 : 7)

ROW 2 (8 : 20)

ROW 3 (21 : 33)

ROW 4 (34 : 43)

ROW 5 (44 : 54)

ROW 6 (55 : 63)

ROW 7 (63 : 63)

VIEW ASCII FILE

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 13

ROW 1 (1 : 3)

ROW 2 (3 : 10)

ROW 3 (10 : 14)

ROW 4 (15 : 20)

ROW 5 (21 : 26)

ROW 6 (27 : 34)

ROW 7 (35 : 40)

WRITE, PARTIAL WRITE, READ,
PARTIAL READ ASCII FILE
PROGRAM REGISTERS NEEDED: 42

ROW 1 (1 : 5)

ROW 2 (5 : 14)

ROW 3 (15 : 23)

ROW 4 (24 : 28)

ROW 5 (28 : 37)

ROW 6 (38 : 44)

ROW 7 (45 : 53)

ROW 8 (53 : 62)

ROW 9 (62 : 70)

ROW 10 (71 : 80)

ROW 11 (80 : 88)

ROW 12 (89 : 93)

ROW 13 (94 : 101)

ROW 14 (101 : 105)

ROW 15 (106 : 113)

ROW 16 (114 : 120)

ROW 17 (121 : 128)

ROW 18 (128 : 136)

ROW 19 (137 : 145)

ROW 20 (146 : 152)

ROW 21 (153 : 161)

ROW 22 (161 : 169)

ROW 23 (170 : 171)

BLOCK CLEAR USING ΣREG

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 7

ROW 1 (1 : 4)

ROW 2 (4 : 12)

ROW 3 (12 : 21)

ROW 4 (22 : 23)

BLOCK ROTATE

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 7

ROW 1 (1 : 7)

ROW 2 (8 : 15)

ROW 3 (15 : 25)

ROW 4 (25 : 26)

VIEW REGISTERS

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 9

ROW 1 (1 : 4)

ROW 2 (4 : 12)

ROW 3 (13 : 22)

ROW 4 (23 : 31)

ROW 5 (32 : 33)

ALPHA SORT

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 11

ROW 1 (1 : 3)

ROW 2 (4 : 12)

ROW 3 (12 : 20)

ROW 4 (21 : 28)

ROW 5 (28 : 36)

ROW 6 (36 : 38)

COUNT BYTES WITH XMEMORY

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 8

ROW 1 (1 : 4)

ROW 2 (5 : 8)

ROW 3 (8 : 13)

ROW 4 (14 : 23)

PROGRAM REGISTERS NEEDED: 58

ROW 1 (1 : 4)

ROW 2 (5 : 10)

ROW 3 (10 : 15)

ROW 4 (15 : 20)

ROW 5 (20 : 26)

ROW 6 (26 : 30)

ROW 7 (31 : 40)

ROW 8 (41 : 49)

ROW 9 (49 : 54)

ROW 10 (54 : 57)

ROW 11 (58 : 63)

ROW 12 (64 : 71)

ROW 13 (71 : 77)

ROW 14 (78 : 89)

ROW 15 (90 : 99)

ROW 16 (100 : 103)

ROW 17 (104 : 114)

ROW 18 (115 : 118)

ROW 19 (118 : 127)

ROW 20 (128 : 132)

ROW 21 (133 : 140)

ROW 22 (141 : 148)

ROW 23 (148 : 156)

ROW 24 (157 : 168)

ROW 25 (168 : 179)

ROW 26 (179 : 187)

ROW 27 (188 : 195)

ROW 28 (195 : 202)

ROW 29 (203 : 210)

ROW 30 (210 : 217)

ROW 31 (217 : 219)

ROW 32 (219 : 219)

NAME–ADDRESS–PHONE
MAILING LIST PROGRAM
PROGRAM REGISTERS NEEDED: 64

PAGE 1
OF 2

ROW 1 (1 : 4)

ROW 2 (4 : 13)

ROW 3 (14 : 19)

ROW 4 (20 : 24)

ROW 5 (24 : 26)

ROW 6 (26 : 29)

ROW 7 (29 : 36)

ROW 8 (37 : 44)

ROW 9 (44 : 45)

ROW 10 (46 : 54)

ROW 11 (55 : 62)

ROW 12 (63 : 66)

ROW 13 (67 : 72)

ROW 14 (73 : 81)

ROW 15 (82 : 86)

ROW 16 (87 : 97)

ROW 17 (98 : 100)

ROW 18 (100 : 106)

ROW 19 (106 : 112)

ROW 20 (113 : 119)

ROW 21 (120 : 124)

ROW 22 (125 : 129)

ROW 23 (130 : 137)

ROW 24 (137 : 145)

ROW 25 (145 : 152)

ROW 26 (153 : 158)

ROW 27 (159 : 162)

ROW 28 (163 : 169)

ROW 29 (170 : 178)

ROW 30 (178 : 184)

ROW 31 (184 : 188)

ROW 32 (189 : 194)

ROW 33 (194 : 201)

ROW 34 (201 : 208)

ROW 35 (209 : 211)

PROGRAM REGISTERS NEEDED: 115

ROW 1 (1 : 8)

ROW 2 (8 : 16)

ROW 3 (17 : 25)

ROW 4 (26 : 31)

ROW 5 (32 : 39)

ROW 6 (39 : 44)

ROW 7 (44 : 51)

ROW 8 (51 : 57)

ROW 9 (58 : 65)

ROW 10 (66 : 75)

ROW 11 (75 : 78)

ROW 12 (79 : 84)

ROW 13 (85 : 91)

ROW 14 (92 : 98)

ROW 15 (99 : 106)

ROW 16 (107 : 114)

ROW 17 (115 : 122)

ROW 18 (123 : 130)

ROW 19 (130 : 132)

ROW 20 (132 : 137)

ROW 21 (138 : 144)

ROW 22 (144 : 152)

ROW 23 (153 : 160)

ROW 24 (161 : 167)

ROW 25 (167 : 173)

ROW 26 (173 : 180)

ROW 27 (181 : 189)

ROW 28 (190 : 195)

ROW 29 (195 : 201)

ROW 30 (202 : 211)

ROW 31 (211 : 218)

ROW 32 (219 : 225)

ROW 33 (226 : 229)

ROW 34 (229 : 237)

ROW 35 (237 : 244)

ROW 36 (244 : 249)

ROW 37 (250 : 255)

ROW 38 (256 : 264)

ROW 39 (264 : 271)

ROW 40 (271 : 273)

ROW 41 (273 : 279)

ROW 42 (280 : 286)

ROW 43 (286 : 291)

ROW 44 (291 : 296)

ROW 45 (297 : 304)

ROW 46 (304 : 309)

ROW 47 (310 : 312)

ROW 48 (312 : 320)

ROW 49 (320 : 325)

ROW 50 (326 : 331)

ROW 51 (332 : 337)

ROW 52 (337 : 339)

ROW 53 (339 : 339)

ROW 54 (340 : 343)

ROW 55 (344 : 352)

ROW 56 (352 : 360)

ROW 57 (360 : 365)

ROW 58 (366 : 373)

ROW 59 (373 : 379)

ROW 60 (380 : 388)

ROW 61 (388 : 393)

ROW 62 (393 : 397)

HP-16C SIMULATOR

PAGE 1
OF 2

PROGRAM REGISTERS NEEDED: 43

ROW 1 (1 : 4)

ROW 2 (4 : 13)

ROW 3 (14 : 21)

ROW 4 (22 : 29)

ROW 5 (29 : 36)

ROW 6 (37 : 44)

ROW 7 (45 : 54)

ROW 8 (55 : 62)

ROW 9 (63 : 71)

ROW 10 (72 : 79)

ROW 11 (80 : 88)

ROW 12 (89 : 98)

ROW 13 (98 : 107)

ROW 14 (108 : 115)

ROW 15 (116 : 122)

ROW 16 (123 : 131)

ROW 17 (132 : 141)

ROW 18 (142 : 151)

ROW 19 (152 : 160)

ROW 20 (161 : 169)

ROW 21 (170 : 178)

ROW 22 (178 : 186)

ROW 23 (187 : 191)

EXTENDED FUNCTIONS

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 11

ROW 1 (1 : 6)

ROW 2 (7 : 9)

ROW 3 (9 : 13)

ROW 4 (14 : 20)

ROW 5 (21 : 28)

ROW 6 (28 : 32)

EXTENDED FUNCTIONS–
TIME MODULE–WAND

PAGE 1
OF 1

PROGRAM REGISTERS NEEDED: 10

ROW 1 (1 : 4)

ROW 2 (5 : 11)

ROW 3 (11 : 14)

ROW 4 (14 : 20)

ROW 5 (21 : 29)

ROW 6 (29 : 29)

VERSION 1
VERIFY CARD
PROGRAM REGISTERS NEEDED: 16

PAGE 1
OF 1

ROW 1 (1 : 4)

ROW 2 (4 : 5)

ROW 3 (6 : 11)

ROW 4 (11 : 17)

ROW 5 (17 : 21)

ROW 6 (22 : 28)

ROW 7 (28 : 32)

ROW 8 (33 : 41)

ROW 9 (41 : 45)

PURGE FILE FIX

PROGRAM REGISTERS NEEDED: 6

PAGE 1
OF 1

ROW 1 (1 : 2)

ROW 2 (2 : 7)

ROW 3 (7 : 15)

ROW 4 (15 : 15)

VERSION 2
VERIFY CARD
PROGRAM REGISTERS NEEDED: 16

PAGE 1
OF 1

ROW 1 (1 : 4)

ROW 2 (4 : 5)

ROW 3 (6 : 11)

ROW 4 (11 : 17)

ROW 5 (17 : 21)

ROW 6 (22 : 28)

ROW 7 (28 : 32)

ROW 8 (33 : 41)

ROW 9 (41 : 45)

EXECUTE PROGRAM IN
EXTENDED MEMORY
PROGRAM REGISTERS NEEDED: 3

PAGE 1
OF 1

ROW 1 (1 : 4)

ROW 2 (5 : 7)

SAVE, GET, SUSPEND, REACTIVATE
KEY ASSIGNMENTS
PROGRAM REGISTERS NEEDED: 31

ROW 1 (1 : 4)

ROW 2 (4 : 11)

ROW 3 (12 : 19)

ROW 4 (20 : 26)

ROW 5 (26 : 32)

ROW 6 (33 : 41)

ROW 7 (42 : 49)

ROW 8 (50 : 58)

ROW 9 (58 : 65)

ROW 10 (65 : 68)

ROW 11 (68 : 76)

ROW 12 (77 : 84)

ROW 13 (84 : 90)

ROW 14 (91 : 99)

ROW 15 (100 : 104)

ROW 16 (105 : 110)

ROW 17 (110 : 111)

GETK INITIALIZATION

PROGRAM REGISTERS NEEDED: 14

ROW 1 (1 : 4)

ROW 2 (4 : 5)

ROW 3 (6 : 13)

ROW 4 (14 : 18)

ROW 5 (19 : 28)

ROW 6 (28 : 37)

ROW 7 (38 : 45)

ROW 8 (45 : 45)

WRITE FILE, READ FILE,
RETRIEVE PROGRAM FILE
PROGRAM REGISTERS NEEDED: 60

ROW 1 (1 : 4)

ROW 2 (5 : 8)

ROW 3 (8 : 13)

ROW 4 (14 : 21)

ROW 5 (21 : 26)

ROW 6 (26 : 33)

ROW 7 (34 : 41)

ROW 8 (42 : 48)

ROW 9 (48 : 52)

ROW 10 (52 : 57)

ROW 11 (57 : 64)

ROW 12 (64 : 73)

ROW 13 (74 : 82)

ROW 14 (83 : 91)

ROW 15 (91 : 98)

ROW 16 (98 : 104)

ROW 17 (105 : 113)

ROW 18 (113 : 121)

ROW 19 (122 : 128)

ROW 20 (128 : 134)

ROW 21 (135 : 143)

ROW 22 (143 : 151)

ROW 23 (152 : 159)

ROW 24 (160 : 168)

ROW 25 (168 : 176)

ROW 26 (177 : 183)

ROW 27 (184 : 190)

ROW 28 (191 : 197)

ROW 29 (198 : 207)

ROW 30 (208 : 215)

ROW 31 (216 : 223)

ROW 32 (224 : 232)

ROW 33 (233 : 233)

ASSIGN, PROGRAMMABLE ASSIGN, MAKE
KEY ASSIGNMENTS WITH XFUNCTIONS
PROGRAM REGISTERS NEEDED: 54

PAGE 1
OF 2

ROW 1 (1 : 4)

ROW 2 (5 : 9)

ROW 3 (10 : 18)

ROW 4 (18 : 25)

ROW 5 (25 : 33)

ROW 6 (34 : 41)

ROW 7 (42 : 46)

ROW 8 (47 : 55)

ROW 9 (55 : 60)

ROW 10 (61 : 67)

ROW 11 (68 : 75)

ROW 12 (76 : 83)

ROW 13 (84 : 92)

ROW 14 (93 : 103)

ROW 15 (104 : 114)

ROW 16 (115 : 124)

ROW 17 (124 : 132)

ROW 18 (132 : 140)

ROW 19 (141 : 149)

ROW 20 (149 : 155)

ROW 21 (155 : 160)

ROW 22 (161 : 167)

ROW 23 (167 : 174)

ROW 24 (174 : 179)

ROW 25 (179 : 185)

ROW 26 (186 : 192)

ROW 27 (192 : 196)

ROW 28 (197 : 204)

ROW 29 (205 : 209)

NOTES

NOTES

NOTES

NOTES

NOTES

INDEX

INDEX TO PROGRAMS

UNLEASH THE POWER OF YOUR EXTENDED FUNCTIONS!

The Extended Functions/Memory module, built into the HP-41CX and available separately for the HP-41C and CV, is the most powerful module that Hewlett-Packard sells for the HP-41. Unfortunately, the Owner's Manual barely hints at the true capabilities of the extended functions and extended memory.

HP-41 Extended Functions Made Easy is the definitive book on extended functions and extended memory, by a leading expert on the HP-41 system. The book assumes no prior knowledge of extended functions or extended memory. Instead, it leads you step by step from the basic concepts of extended memory through explanations of each of the extended functions and short examples of their use.

The second half of the book introduces over 30 utility programs, including a text editor, a mailing list manager, programs to store text files on magnetic cards, mathematical programs (solve, integrate, etc.), and much more. These programs make your extended functions more powerful, convenient, and fun to use. Barcode, included for all programs, makes the programs as easy to load as they are to use.

If you own an HP-41CX or an Extended Functions/Memory module, you need this book!

ISBN: 0-9612174-1-3