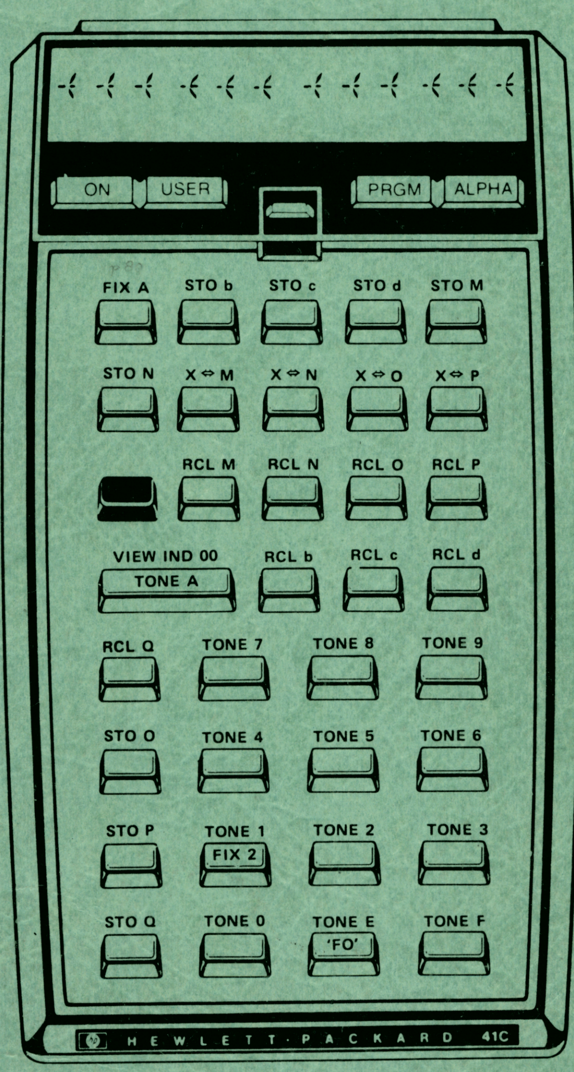
SYNTHETIC PROGRAMMING

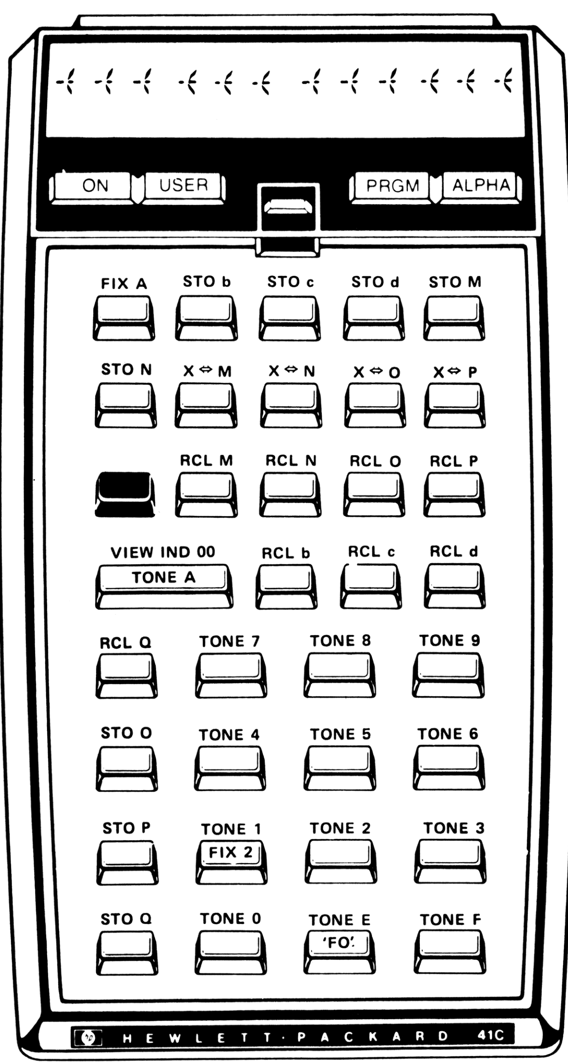
ON THE HP-41C



by
W.C. Wickes

SYNTHETIC PROGRAMMING

ON THE HP-41C



by
W.C. Wickes

Acknowledgements:

It is a pleasure to acknowledge the crucial contributions of the many people who have helped me in the preparation of this book. The cover art was provided by William Kolb, who also did an exhaustive editing of the manuscript, as well as providing encouragement and support throughout the whole project. Tom Cadwallader, Keith Jarett, and John McGechie, each an important synthetic programming pioneer, also provided invaluable editing and suggestions, along with Tom James and Lee Vogel. Charles Close has been an important contributor to our detailed knowledge of program line coding and other subtle HP-41C properties. The discovery of enhanced byte jumping by Roger Hill was a major advance in making synthetic programming a 'friendly' process. Jacob Schwartz developed the Barcode Character Table in Appendix 3. Richard Nelson, through his fine efforts in founding and maintaining the PPC, has fostered the world-wide communication necessary for the collective development of synthetic programming.

Professor Carroll Alley of the University of Maryland has been a principal backer of my synthetic programming efforts right from the arrival of our virgin, buggy HP-41C's.

This book is dedicated to my wife, Susan, the world's most supportive Calculator Widow. She handled many of the typing and proofreading chores, and kept all of the wolves from the door while I was closeted with my HP-41C and typewriter. Also, I am grateful to my children Kenny and Lara, who allow me use 'their' HP-41C whenever I want to play with it.

Cowardly Disclaimers:

The programs and descriptions in this book have been developed and tested on a number of HP-41C's. With the exception of the Text Enabler process (Section 5H), all of the HP- 41C's performed identically. No program in this book has ever failed to run on any HP-41C tried. But since the material deals with 'unsupported' features of the HP-41C, I cannot guarantee that every technique and every program will perform exactly as described under all circumstances. Hewlett-Packard naturally reserves the right to modify its products within the limits of the descriptions of the Owner's Handbook--the material herein is not covered by the Handbook, so we cannot expect that future modifications will preserve the unsupported features.

HP-41C 'crashes', i.e., where the display freezes or blanks and the keyboard becomes inactive, are an operational hazard of 'synthetic programming'. No program or technique described in this book will cause a crash if the directions are followed exactly--but occasional mistakes are inevitable. If you crash your calculator--don't worry, no harm will result to the HP-41C. Try removing then replacing the battery pack. If that fails (try it a few times), remove all peripherals and plug-in modules prior to removing the battery pack. As a last resort (I have never had to resort to this), remove the battery pack overnight.

The barcodes in the Appendices were produced by offset printing. The process was tested prior to publication, and worked successfully. If you have difficulty entering the barcode, check for poorly reproduced bars and black them in by hand.

To borrow some well-known words:

The material contained in this book is supplied without representation or warranty of any kind. The publisher, Larken Publications, and the author assume no responsibility and shall have no liablility, consequential or otherwise, of any kind arising from the use of this material or any part thereof.

© 1980 William C. Wickes

Published by:
Larken Publications, Post Office Box 987, College Park, Maryland 20740

TABLE OF CONTENTS

CHAPTER 1. WHY'S AND WHEREFORE'S
1A. Synthetic Programming? 1
1B. Purpose and Organization 2
1C. The Origin of Synthetic Programming 2
1D. No Risk to the HP-41C 3
1E. Some Conventions 3
1F. Prerequisites 4
1G. References 5

CHAPTER 2. INSIDE THE HP-41C
2A. Calculator Language: Bits, Nybbles and Bytes 6
2B. The Byte Table 9
2C. Register, Please 16
2D. Memory Partitioning 18
2E. The Key Assignment Registers 20

CHAPTER 3. EXOTIC EDITING WITH THE BYTE JUMPER
3A. Normal Editing 23
3B. The Byte Jumper 25

CHAPTER 4. THE STATUS REGISTERS
4A. Strange Postfixes 29
4B. The Alpha Register 31
4C. Register Q 33
4D. The Flag Register 33
4E. The Key Assignment Flags 34
4F. The Address Pointer and the Return Stack 35
4G. Register c and Memory Partitioning 36

CHAPTER 5. PROGRAMS FOR PROGRAMMING
5A. Unseemly Displays 38
5B. Register Exchanges and Normalization 39
5C. Getting Started: "CODE" 40
5D. Direct Access to Program Registers 42
5E. Synthetic Key Assignments 44
5F. Creation of Synthetic Program Lines 49
5G. Enhanced Byte Jumping 52
5H. The Text Enabler 53
5I. The Q-Loader 55
5J. Backtalk from the HP-41C 57
5K. Code Storage 58

CHAPTER 6. APPLICATIONS
6A. Getting to the .END. 60
6B. SIZE-finding and Other Tricks 61
6C. Fun and Games in the Alpha Register 63
6D. Character Recognition 66
6E. Synthetic Text Lines and the Printer 68
6F. Non-Normalized Numbers and Mass Flag Control 70
6G. Raising the Curtain 73
6H. Application Pacs: Sneaking in the Back Door 75

CHAPTER 7. AMUSING ANOMALIES
7A. 128 Tones 77
7B. Tricks with System Flags 79
7C. Flying the Goose Backwards 80

APPENDIX 1. NUMBER SYSTEMS 82

APPENDIX 2. PROGRAM BARCODE
"CODE" 84
"REG" 85
"KA" and "EF" 86
"DECODE" 88
"HM" 89

APPENDIX 3. THE BAR CODE CHARACTER TABLE 91

FIGURES
2-1 Three Levels of Coding in the HP-41C 7
2-2 HP-41C Barcode 8
2-3 Display Logic 10
2-4 Sample Byte Table Box 11
2-5 HP-41C User Memory Partitioning 19
2-6 Key Assignment Bytes 21
4-1 The Status Registers 32
4-2 Key Assignment Flag Bits 35
5-1 A Synthetic Programming Keyboard 50
6-1 A Special Graphics Character 69

TABLES
1-1 Symbols for Status Registers 4
2-1 The HP-41C Byte Table 12
2-2 Assignment of Non-Programmable HP-41C Functions .. 22
5-1 Non-Programmable Peripheral Functions 45
5-2 "KA" Entries for the Keyboard of Figure 5-1 51
7-1 TONE Frequencies, Numbers, and Durations 78

PROGRAM LISTINGS

	Global Label	Description
1.	"AD"	Address Finder 58
2.	"AL"	Alphabetize Alpha Data 67
3.	"BYTE"	Determine Byte Number 62
4.	"CA"	Clear All Key Assignments 47
5.	"CD"	Character-to-Decimal Conversion 66
6.	"CODE"	7-Byte Encoder 40
7.	"CR"	Code Recall 59
8.	"CS"	Code Store 59
9.	"CU"	Move Program/Data 'Curtain' 73
10.	"DC"	Decimal-to-Character Conversion 67
11.	"DECODE"	7-Byte Decoder 57
12.	"DI"	Turn on Display Annunciators 80
13.	"EF"	.END. Encoder 46
14.	"EN"	Find Last User Program 60
15.	"FL"	Set Any Flag 79
16.	"HM"	Hangman Game 65
17.	"ISO"	Isolate a Single Alpha Character 64
18.	"KA"	Synthetic Key Assignments 46
19.	"KP"	Pack Key Assignment Registers 47
20.	"MANT"	Find Mantissa of X 68
21.	"RE"	Reset All Flags 79
22.	"REG"	Store/Recall into any User Register .. 43
23.	"REV"	Reverse 6-Character Alpha String 57
24.	"ROM"	Direct Access to ROM Module Programs . 75
25.	"S"	Automatic Size-Finder 61
26.	"SAVE"	Save All Flags 79
27.	"SUB"	Substitute One Alpha Character 64
28.	"TONE"	Create All Synthetic TONE lines 77

CHAPTER 1

WHY'S AND WHEREFORE'S

"There are more things in the heavenly HP-41C, Hewpackio,
Than are dreamt of in your philosophy."

--apologies to W. Shakespeare

1A. SYNTHETIC PROGRAMMING?

No one, from the serious student of computer science, to the occasional user of four-function calculators, can fail to be impressed with the HP-41C calculator. This machine combines amazing computing power with the convenience of complete portability. The prospective buyer is attracted by the long list of computing functions built into the calculator; the experienced owner finds that the HP-41C becomes an ever more important part of his problem solving techniques as he masters programming and integrates his own ingenuity with the built-in functions.

And yet, even when an HP-41C user has learned everything the Owner's Handbook can teach him, he is in for another treat: the list of HP-41C functions and programming capability is not limited to the properties catalogued in that Handbook. There exists, in fact, a whole class of functions and programming applications that can be used to enhance greatly the power of the calculator, even though the new functions cannot, at first, be executed or programmed with normal, simple keystrokes. The new functions, which are 'synthesized' by creating new combinations of normal program bytes, are called 'synthetic functions'; their application in programs gives rise to the expression 'synthetic programming', and hence, to the title of this book.

To whet your appetite, here is a sample of some of the typical applications of synthetic programming that are impossible or impractical without the techniques described herein:

***Addition of twenty-one 'new' display characters for routine use.

***Transformation of the alpha register into four additional data registers. These registers can provide a 'scratch pad' for a program to use without disturbing data stored by other programs in numbered data registers. Furthermore, the contents of these registers can be input and output with the Card Reader operation 'WSTS'.

***Enhanced user control over the 56 user and system flags. Example: Two keystrokes can clear all 56 flags simultaneously.

***Automatic 'SIZE-finding' in less than 2 seconds.

***Rapid alphabetizing of alphanumeric data.

***Alphanumeric character-string processing

***Addition of six new TONE frequencies, plus variation of TONE duration.

***Interchange of program lines and stored data.

***Improved key assignment control, including two-byte function assignments (e.g., assignment of 'STO 65' to a key), automatic clearing of all assignments, and assignment register packing.

A simple exercise will introduce you to the world of synthetic programming, and perhaps motivate you to expend the effort to read the rest of this book. Try the following hocus-pocus:

1. Insert one memory module into the HP-41C.
2. Execute a 'Master Clear'.
3. Set SIZE 063 (if your module is double density, set SIZE 127).
4. Switch to PRGM mode.
5. Key in these program lines:

```
01 12345
02 STO IND 17
03 RDN
```

6. Turn the HP-41C off.
7. Remove the memory module; wait 60 seconds; replace.

8. Turn the HP-41C on.
9. Press RTN.
10. Key in '1.435245455 EEX 59'.
11. Press SST
12. Switch ALPHA on.

Where did that creepy little 'man' come from? Switch PRGM on, press BST once, and you will see the program line '01 STO M'. This 'synthetic' program line is the combination of the 'IND 17' program byte and the 'RDN' byte that resulted when you eliminated the 'STO' byte from 'STO IND 17' by removing the memory module. No hint of the existence of a 'STO M' function is given by the Owner's Handbook, but you will come to know and love 'STO M' and its friends as you master synthetic programming.

1B. PURPOSE AND ORGANIZATION

This book is designed to impart the joys and explain the mysteries of synthetic programming to any HP-41C user, from the novice programmer to the expert. It is a compendium of the theory of calculator operation that makes synthetic programming possible, the mechanical procedures for implementing the synthetic program lines, and a set of application programs that serve practical purposes and also illustrate the use of exotic programming techniques.

Chapter Two describes the inner workings of the HP-41C, from a conceptual point of view that will probably make computer engineers cringe. You will obtain there a working knowledge of calculator programming at a deeper level than is possible from the Owner's Manual alone. Besides laying the groundwork for synthetic programming, the information in Chapter Two will give you a picture of HP-41C operation that will help you optimize all of your programming work.

Chapter Three introduces the first and most important of the synthetic functions, the 'byte jumper'. This single keystroke function opens the door to simple procedures for creating the entire set of synthetic program lines. We will 'create' the byte jumper using a 'module pulling' trick like we used to make the 'STO M'; once that is done, we will never again need to resort to module removal.

In Chapter Four, a new set of HP-41C registers, the 'status registers', is introduced. Access to these registers, which include the alpha register, the 56 flags, memory allocation information, and the program address pointer and subroutine return stack, results in a host of practical applications like the examples given in Section 1A.

'Programming programs', a package of HP-41C programs and techniques, are described in Chapter Five. The principal use of these programs is to enable the writing and deciphering of other programs.

Chapter Six is a 'standard applications' chapter, in which we find a set of synthetic programs that in themselves are sufficient justification for the study of the material in the preceding chapters. But more than that, the programs illustrate general synthetic programming techniques that have application to a wide range of problems, limited only by the motivation and ingenuity of the enlightened user.

Finally, in Chapter Seven, we learn a few 'fun' tricks-of-the-trade that aren't particularly practical but will gladden the heart of the confirmed calculator nut. Included, of course, is a supreme example of the expenditure of enormous research effort to discover a totally useless result, namely, how to make that doggoned goose turn around and fly backwards!

Three appendices are included. Appendix 1 is a brief review of the decimal, binary, octal, and hexadecimal number systems. If you are unfamiliar or perhaps a little rusty with these notation systems, you will find it useful to study Appendix 1 before tackling Chapter Two. Appendix 2 contains the Wand barcode for the important programs of Chapter Five, "CODE", "REG", "KA", and "DECODE", plus the long 'Hangman' program of Chapter Six. Appendix 3 contains special barcode for specific synthetic programming purposes.

1C. THE ORIGIN OF SYNTHETIC PROGRAMMING

It all came about by accident! Early models of the HP-41C had an unintentional flaw, or 'bug', in their internal coding, that allowed execution of the operation 'STO IND 01', for example, with values from 719 to 999 in data register R_{01}. This operation caused the contents of Register X to be stored into program memory. I wondered what would happen if I used this 'feature' to synthesize new program lines by storing numbers into program that would link together normally impossible combinations of program 'bytes'. To make a long

story short, it worked. Once the new functions started popping up in memory, whence they could be recorded on a magnetic card and then merged into any program, the practical applications started coming in droves.

Following the discovery of the synthetic functions, the major advances in synthetic programming were 1) the development of synthetic key assignments, which permit single keystroke execution of the new functions, and also eliminate the need for the hardware 'bug'; and 2) the discovery of the byte jumper, which is perhaps the most fundamental synthetic function. Since the byte jumper can be created on any HP-41C, and since it can be used to generate almost any other synthetic program line, we can start from 'scratch' in this book and show how to 'bootstrap' an HP-41C to have complete synthetic program capability.

1D. NO RISK TO THE HP-41C

Synthetic functions, when keyed into the HP-41C with the methods described in this book, are 'proper' calculator operations. As such, they constitute no physical threat to the HP-41C. The only risk, which really should be considered a potential annoyance rather than a danger, is that certain operations with synthetic functions can cause either 'MEMORY LOST' or a 'crash', i.e., a state where the display freezes or blanks and the keyboard becomes disabled. The first disaster causes a lot of teeth gnashing and hand wringing, but certainly doesn't harm the HP-41C. The second problem can virtually always be corrected (99.9% of the time) by simple removal and immediate replacement of the battery pack, followed by one or two on-off presses. I have heard of only one case where a crash required overnight removal of the batteries for recovery, but the cause of that crash is unknown. I can't guarantee anything, of course, but in the course of developing synthetic programming, I have accidentally cleared the memory or crashed my calculator literally dozens of times, yet the HP-41C keeps ticking along.

Because of the risk of accidental memory loss, however, it is not appropriate for Hewlett-Packard itself to 'support' the use of synthetic functions. Therefore, you should not submit programs containing synthetic program lines to the User's Library.

1E. SOME CONVENTIONS

The following is a list of special notational conventions I have adopted for this book, to simplify the description of calculator programs, characters, numbers, instructions, etc.:

1. You may have already noticed the use of the single quotation marks in place of the usual double, as in 'example', instead of "example". Double quotation marks are reserved to indicate HP-41C alphanumeric characters and text lines, much as the Printer identifies characters in program listings. This convention is used also in listing of programs in this book, most of which are reproductions of Printer listings. Thus you should be aware that a program line enclosed in quotation marks will show in the HP-41C display as preceded by the text symbol "⊤".

In addition, I am deliberately violating the punctuation rule that requires commas and periods to be included within quotes when they are adjacent, i.e., "ABCD", rather than "ABCD,". In addition to defying logic, this rule would lead to unacceptable ambiguities in this book, since the comma and period are also standard HP-41C alpha characters. Their inclusion within quotation marks could suggest that they are part of the adjacent alpha string.

2. Whenever possible, to provide neater copy, standard typewriter symbols will be used to represent HP-41C display and printer characters. The identification is usually obvious, with a possible exception the use of the semicolon ";" to represent the HP-41C symbol "↗".

3. <u>Numbers</u> in the display will be represented in the same form that they would assume if entered into the alpha register with 'ARCL'. Thus, numbers in SCI or ENG format will be listed using "E" to indicate the exponent, e.g., '1.23 E10' rather than '1.23 10' to eliminate the possibly confusing spaces.

4. Entries such as 'STO mn', 'DEL lmn', or 'SF mn' are to be understood as typical operations with the listed function, where the letters l, m, n, etc., represent digits that can take any of the normal values associated with the function. Thus, for 'STO mn', 'm' and 'n' can each take the values 0 through 9.

5. In the listing of long hexadecimal or binary numbers, it is often convenient to group the digits for explanatory purposes. The grouping is indicated by spaces or bars "|" placed in the numbers, which of course are not contained in the HP-41C's actual coding of the numbers.

6. A basic unit of HP-41C user memory is the 'register', a block of seven bytes of code which may be used for storage of one number or of seven program bytes. Registers will be identified as follows: a) 'Register lmn' indicates the register found at memory address 'lmn' (see Section 2C), where 'lmn' is a 3-digit hexadecimal number. b) 'Register α', where 'α' is a single alpha character, refers to any of the 16 'status' registers discussed in Chapter 4. Registers X, Y, Z, T, and L are the usual RPN stack registers. The remaining 11 status registers, Registers M, N, O, P, Q, ⊢, a, b, c, d, and e, are so named because of the way the synthetic 'status register access functions' are displayed in program lines, e.g., 'STO M', 'RCL ⊢', 'ISG d', etc. c) 'R_{mn}' indicates data Register number 'mn', where 'mn' is a two (occasionally three)-digit decimal number.

7. Programs are reproduced in this book directly from 82143 Printer listings, to help avoid transcription errors. Unfortunately, the Printer lists status register access functions for Registers M, N, O, P, Q, and ⊢ using different symbols than the HP-41C display. The reader will need to become familiar with the following Table:

TABLE 1-1

Symbols For Status Registers

Display Symbol	Printer Symbol
M	[
N	\
O	]
P	↑
Q	_
⊢	⊤

8. Short program routines without alpha labels will be identified by a number shown in parentheses to the right of the routine. The format is '(Section number-routine number)'.

9. To simplify the instructions for keying in special program lines or operating programs, the instructions will be shown in most cases in three columns. Entries in the left column are codes to be entered into memory: either numbers to be keyed into Register X, alpha characters (enclosed in double quotation marks) to be keyed into the alpha register, or program lines (shown with program line numbers) to be keyed into the current program. The center column lists keystroke sequences, such as 'GTO .123' or 'DEL 005', that are not recorded in the program. The right column will show, when appropriate, the HP-41C display resulting from each 'center column' instruction. These 'displays' will be enclosed in square brackets [ ]. Example:

(Key in)	(Operation)	(Display)
	GTO .000	[00 REG 123]
01 STO 01		
	SST	[02 X<>Y]

indicates that you should press 'GTO .000' (to see '00 REG 123'), key in the line '01 STO 01', then 'SST' once to see line '02 X<>Y'.

1F. PREREQUISITES

In order to reach as wide a range of HP-41C users as possible, this book is designed around a minimal HP-41C 'system'. All that you will require are (1) an HP-41C, (2) temporary use of one memory module, and (3) a few hours' time to wade through all of the material. The peripheral Card Reader, Printer, and Wand are not necessary, although they are valuable accessories for synthetic programming just as for any normal use of the HP-41C. The card reader, for example, provides a great way to store your new programs and key assignments to ensure against accidental memory loss. The printer is invaluable for listing the programs as you key them in, and keeping a running record of everything you do. If you are fortunate enough to have a wand available, you can save a lot of keying by using the barcodes provided in Appendices 2 and 3.

Chapter 2 is the major 'stumbling block' for beginning synthetic programmers, since it contains a lot of detailed descriptions without any of the fun of pushing calculator

keys. I suggest that you read Chapter 2 rather rapidly the first time through, just carefully enough to get a general grasp of its contents to prepare you for the more interesting keystroking starting with Chapter 3. Then as you continue through the later chapters, you will wish to refer back in more detail to various sections of Chapter 2.

1G. REFERENCES

Most of the discoveries and techniques described in this book were first published in various issues of the PPC Calculator Journal. The PPC (the initials do not stand for anything in particular) is an independent, world-wide club of calculator enthusiasts with a common interest in the study and application of Hewlett-Packard programmable calculators. The Journal is the principal medium of information exchange among the developers of synthetic programming, who are scattered all over the world. Any serious HP-41C user, particularly if he is interested in building upon what he learns from this book, will find it a worthwhile investment to join the club and subscribe to the Journal. Having contact with several thousand other programmers can save you a lot of work! Inquiries should be directed to:

Richard Nelson
2541 W. Camden Place
Santa Ana, CA 92704

Here is a list of the articles relevant to the development of synthetic programming (the format is Volume, Number, Page):

Cadwallader, T., 'Improved Synthetic Key Assignments' V7N3P8
Close, C., 'Bug 2: A Practical Application' V7N3P8
Hewlett-Packard ('Corvallis Column') 'HP-41C Function Table V6N4P11
______________ 'HP-41C Postfix Table' V6N5P11
______________ 'HP-41C Data & Program Structure' V6N6P19
Istok, G. 'Pseudo XROM's on the HP-41C' V7N2P32
Kennedy, J. 'The HP-41C Combined Hex Table' V6N5P27
McGechie, J. 'HP-41C Synthetic Key Assignments' V7N2P34
Nelson, R. 'Bugs in the Box' V6N5P27
Wickes, W. 'Direct Status Register Access on the HP-41C' V6N7P31
________ 'Through the HP-41C with Gun and Camera' V6N8P27
________ 'HP-41C Black Box Programs' V6N2P35
________ 'Freedom From Bugs' V7N2P35
________ 'Synthetic Key Assignments' V7N2P30
________ 'Improved Black Box Programs' V7N2P35
________ 'HP-41C Synthetic Function Routines' V7N4P26
________ 'Byte-Jumping, or The Poor Man's Black Box' V7N4P26
________ 'Direct Addressing of ROM Routines' V7N5P55
________ 'Understanding BLDSPEC' V7N5P56

It should be stressed that the development of synthetic programming is a continuing process. Even as this book is being written, curious programmers are turning up new tricks and extending our understanding of the HP-41C. Even writing this book has increased my own understanding of the subject, and led to a few new discoveries, such as the 'text enabler' operation described in Chapter 5.

CHAPTER TWO

INSIDE THE HP-41C

This chapter will be, to some extent, an excursion into fantasyland. In order to give you a useful conception of the working of the HP-41C, I will introduce certain fictional, almost personified, systems to represent important operations of the calculator. These systems may or may not have exact electronic counterparts inside the HP-41C case; such details are of interest only to electronic engineers and are outside the scope of this book. The important thing is that the HP-41C behaves as if these systems were present. First and foremost, you are asked to conceive of the 'brain' of the calculator as a device called the 'processor'. This processor is responsible for reading data and programs stored in the memory, then instructing the other systems in the calculator what to do with what it has read. Depending on your imagination, you might think of the processor as an almost human taskmaster, busily reading one set of instructions provided by the user, then issuing its own instructions to the various 'workers' that make up the HP-41C system.

2A. CALCULATOR LANGUAGE: BITS, NYBBLES, AND BYTES

Riddle: what do the number '1.435245455 E59', the alpha string "⊼CREEPY", and the program

```
01 LBL 00
02 /
03 SQRT
04 X>Y?
05 X>Y?
06 LN
07 SIN
```

have in common? Answer: all three are stored identically in the HP-41C user memory! To understand this apparently obscure concept is to grasp the basis of the entire user memory organization and coding. By 'user memory', we mean that portion of the HP-41C memory under user control: the data and program registers, the key assignment registers, the RPN stack registers, the alpha register, etc.

At the elementary level, a calculator is really a very simple device. It can store and recall numbers, add them if desired, and that's about it. In order to carry out instructions which may seem elementary to the user, such as '+', or 'LN', the processor must initiate sequences of dozens of elementary steps. The real power of a calculator lies in its ability to allow a user to initiate this internal processing by means of a simple sequence of keystrokes. A programmable calculator is one which can also encode the keystroke sequence, and store the code for repeated automatic execution.

The HP-41C represents a major advance over previous hand-held calculators in that the user program codes are displayed to the user in directly readable alphanumeric characters and words. We might imagine the 41C as containing an invisible 'translator', which takes a section of stored code and translates it into a displayed number or word. But, in fact, there must also be another translator, to call up the proper sequence of elementary steps, called 'microcode', for the calculator to execute. There are, in effect, three levels of 'interpretation' of the same stored code, illustrated in Figure 2-1.

'Level one' is the rendering of codes into a form directly visible to the user. User input consists of generating codes by pressing keys; the resulting codes are read back by the 'user translator' in the form of numbers, characters, or program lines shown in the display. At 'level two', the codes are realized in a form that permits them to be stored in memory registers, or to be written into or read from external devices such as the card reader or wand. Finally, a 'machine translator' is necessary to translate the codes into 'level three', i.e. the set of elementary machine instructions required to carry out an operation.

We are primarily interested in levels one and two. 'Synthetic programming' is the process of creating new level two codes by bypassing the keyboard logic that restricts input to the set of instructions listed in the Owner's Handbook. The resulting synthetic codes can then be interpreted by both translators, often yielding practical results such as new display characters and program functions. To achieve this, the user must learn to 'speak' the level two language so that he can interact directly with the stored codes without depending on

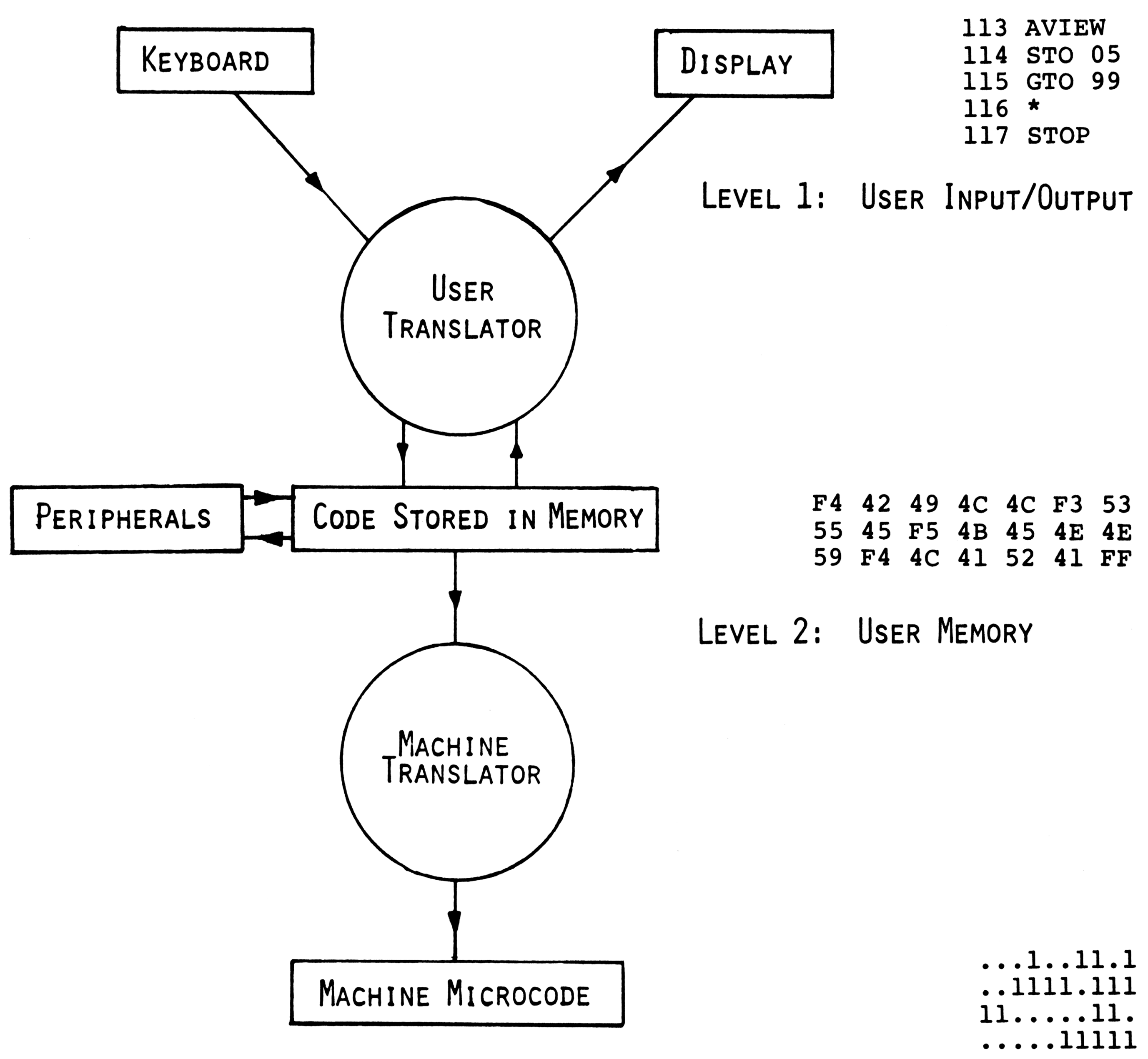


FIGURE 2-1. THREE LEVELS OF CODING IN THE HP-41C

the user translator.

The requirement that the memory codes be storable in HP-41C memory and also be able to be read to or from peripherals dictates the general form the codes must take. The card reader, for example, uses magnetic cards for code storage; the card itself can only contain information stored as regions of ordered magnetic fields in the oxide film of the card. To provide reproducible, reliable storage, the ordering must be the simplest possible: the code is represented by a string of magnetized or non-magnetized bar-shaped regions. This concept is directly analogous to the barcodes used by the wand. As the wand is scanned along the line of bars, it 'sees' either wide or narrow bars. The bar pattern can be considered as a long binary number, with the wide bars representing 'ones' and the narrow bars 'zeros'. Figure 2-2 is a sample of the barcode, which we can use to visualize how codes are stored on a magnetic card, or indeed, in the HP-41C itself.

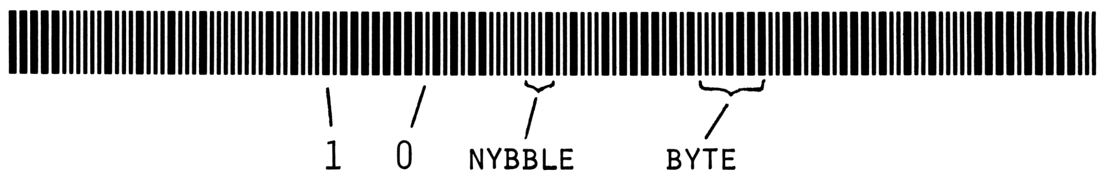


FIGURE 2-2. HP-41C BARCODE

In the calculator, the 1's and 0's are represented by the states of microscopic transistors, but the central idea is the same as for the wand or the card reader: user codes are stored as sections of a long string of binary 'bits'. To make sense of the code, the processor must know how to break the string into intelligible sections.

Consider again the number '1.435245455 E59'. A count reveals that there are 14 'pieces' of information required to represent the number in decimal form: ten mantissa digits, two exponent digits, a mantissa sign and and exponent sign. The basic unit or building block of storage code must be able to represent one of the pieces, i.e., it must be able to assume at least ten different values so that it can represent a single decimal digit. The decimal numbers 0 through 9 are represented in binary as 0000 through 1001 respectively, so we conclude that the unit must consist of four consecutive binary bits. This unit is called a 'nybble'--it's half a 'byte' as we shall see (computer jocks can't spel too wel). We shall also refer to a nybble as a 'digit', referring to its role in number storage. The decimal number above is thus coded as 14 nybbles, like this:

```
0000 0001 0100 0011 0101 0010 0100 0101 0100 0101 0101 0000 0101 1001
  +    1    4    3    5    2    4    5    4    5    5    +    5    9
```

(The spaces are provided for clarity.) The 'E' and the '.' do not require explicit coding, since their positions and 'values' never change. For the sign digits, the first and twelfth nybbles counting from the left, the HP-41C uses '0000' for '+' and '1001' for '-'.

You may already have observed that the four bits required to represent a decimal digit could take values up to binary 1111, or decimal 15. The coding is thus sufficient to represent 'hexadecimal' numbers, as well as decimal. This capacity would be wasted if the HP-41C only dealt with decimal numbers. However, even a four-bit nybble is not adequate as a basic unit for coding program lines.

The HP-41C uses 2 consecutive nybbles, 8 bits, as its elementary unit of program code, called a 'byte'. Binary 11111111 (hex FF) is decimal 255, so there are 256 possible elementary program codes, which is sufficient even for a calculator with the HP-41C's capability. As the 41C runs a program, we can imagine it 'byting' off successive 8-bit chunks of code for processing. The meaning of the riddle at the beginning of this section should now start to become clear. The number '+1.435245455 E+59' is stored as seven bytes, '01 43 52 45 45 50 59'. When these same bytes are in program memory, they represent the program lines 'LBL 00', '/', 'SQRT',

X>Y?, X>Y?, 'LN', and 'SIN' respectively. The HP-41C's user translator is even more sophisticated than we might have thought. The translation to the display depends not only on the code read by the translator, but also on the current mode (PRGM, ALPHA, etc.) of the calculator. The riddle suggests yet a third way of translating the same code--if the sample code were in the alpha register, it would be displayed as the seven alpha characters "∓CREEPY".

The output medium of the user translator is the display. Whatever you see in the HP-41C display is a rendering of the contents of some register into an alphanumeric display. When you first turn the calculator on, it is in a 'default' mode, in which the contents of Register X are copied into the display as a number, with each number character representing one digit from Register X. If the HP-41C is switched to ALPHA mode, the alpha register is copied, with one displayed character for each alpha register byte. If a 'VIEW mn' is executed, the Message Flag 50 is set to indicate a non-default display, with the contents of R_{mn} displayed. This display scheme permits the viewing without disturbing the contents of Register X. Similarly, we can view the alpha register by using AVIEW. A 'CLD' clears Flag 50, restoring the default display. In PRGM mode, the display shows a program line made from program bytes. In a running program, the 'flying goose' is the default display, which can be replaced by means of a 'VIEW' or 'AVIEW' instruction. Figure 2-3 illustrates the logic involved in the display process.

We have seen that user programs and data stored in the HP-41C are coded as a long string of 1's and 0's called bits. In data memory, each successive group of 4 bits, called a nybble, can represent a single decimal digit, or a sign for the mantissa or exponent. Since a number requires 14 nybbles, successive 14-nybble sections (56 bits) of code are stored and recalled together from a fundamental storage location called a register. The operation 'RCL 01', for example, instructs the calculator to copy the 56 bits of code found in the section of memory designated as R_{01} into another section, Register X. In program memory, the code is recalled and stored one or more bytes at a time. As described in the next section, each of the 256 possible bytes represents a unique set of program instructions. The division of memory into registers is less apparent in program memory than in data memory, but the addressing scheme described in Section 2C, nevertheless, is organized by 7-byte registers, so that registers can be used interchangeably for data or program storage.

2B. THE BYTE TABLE

Before continuing with a discussion of the addressing scheme used in the HP-41C, let's consider in more detail the coding of program lines. The element of program coding is the byte; each byte has 256 possible values, from hex 00 through FF. However, there are many more than 256 different program lines--this variety is achieved by allowing program lines to use one or more bytes, up to a maximum of sixteen. Thus even though the display shows what appears to be just one instruction, a program line, that single line may actually consist of several bytes of stored code.

Table 2-1, the 'HP-41C Byte Table', shows the 256 possible bytes in a 16x16 grid. This table is a powerful tool, indispensable for synthetic programming, so it is important for the new programmer to understand its use. It is, in effect, the dictionary used by the user translator. The numbers 0-F labeling the horizontal rows of the table represent the first nybble or digit of a byte code. The numbers labeling the vertical columns give the second digit of the byte. The box in a particular row and column lists a number of 'features', i.e., the various ways that the corresponding byte can be interpreted, depending upon its position in memory. Figure 2-4 shows a sample box, using fictitious entries to illustrate all of the possibilities.

The first number in the box, in the upper left-hand corner, is the decimal equivalent of the 2-digit hexadecimal byte value. This number is also the value used with the printer function 'ACCHR' to obtain the printer character shown in the box to the right of the decimal number. (The decimal equivalents will also be used as inputs for the key assignment program "KA" described in Chapter 5.) For example, in box 34 (row 3, column 4), we see decimal number 52 (3x16+4=52), and the corresponding printer character "4".

The next entry in each box is the name of an HP-41C function. For the bytes in rows 0-8 (except for bytes 1D and 1E), each byte by itself constitutes an entire program line. Byte 34 displays and executes as 'STO 04', byte 5C is 'ASIN', etc. These bytes can be called 'one-byte functions', or 'stand-alone' bytes, since they cause operations that are independent of any succeeding bytes in the program.

The HP-41C departs from its Hewlett-Packard predecessors by allowing multi-byte program 'lines' rather than only single-byte 'steps'. The bytes in row 9, bytes A8-AE, and bytes CE and CF are 'prefix' bytes for two-byte program lines. When the processor encounters one of

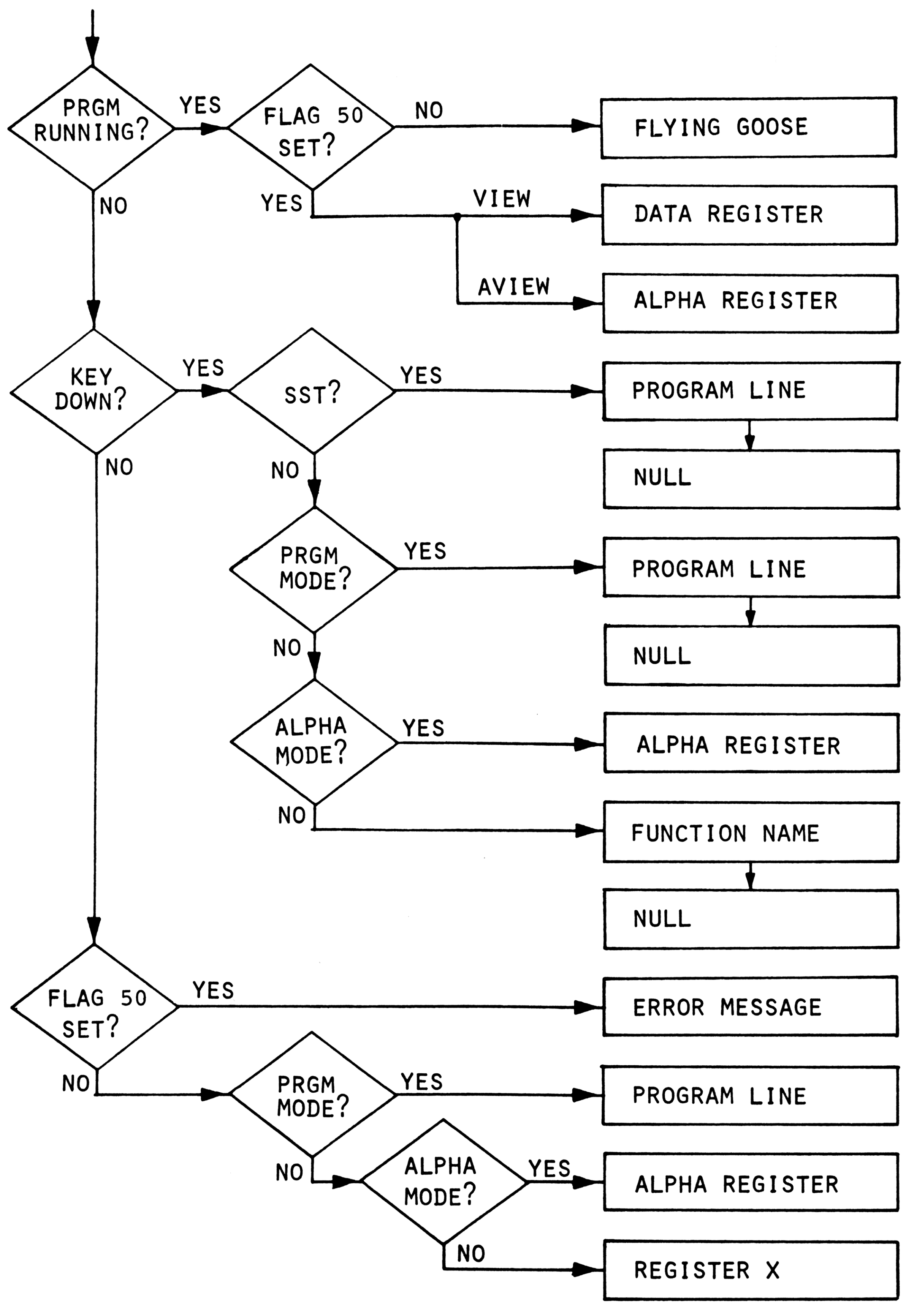


FIGURE 2-3. DISPLAY LOGIC

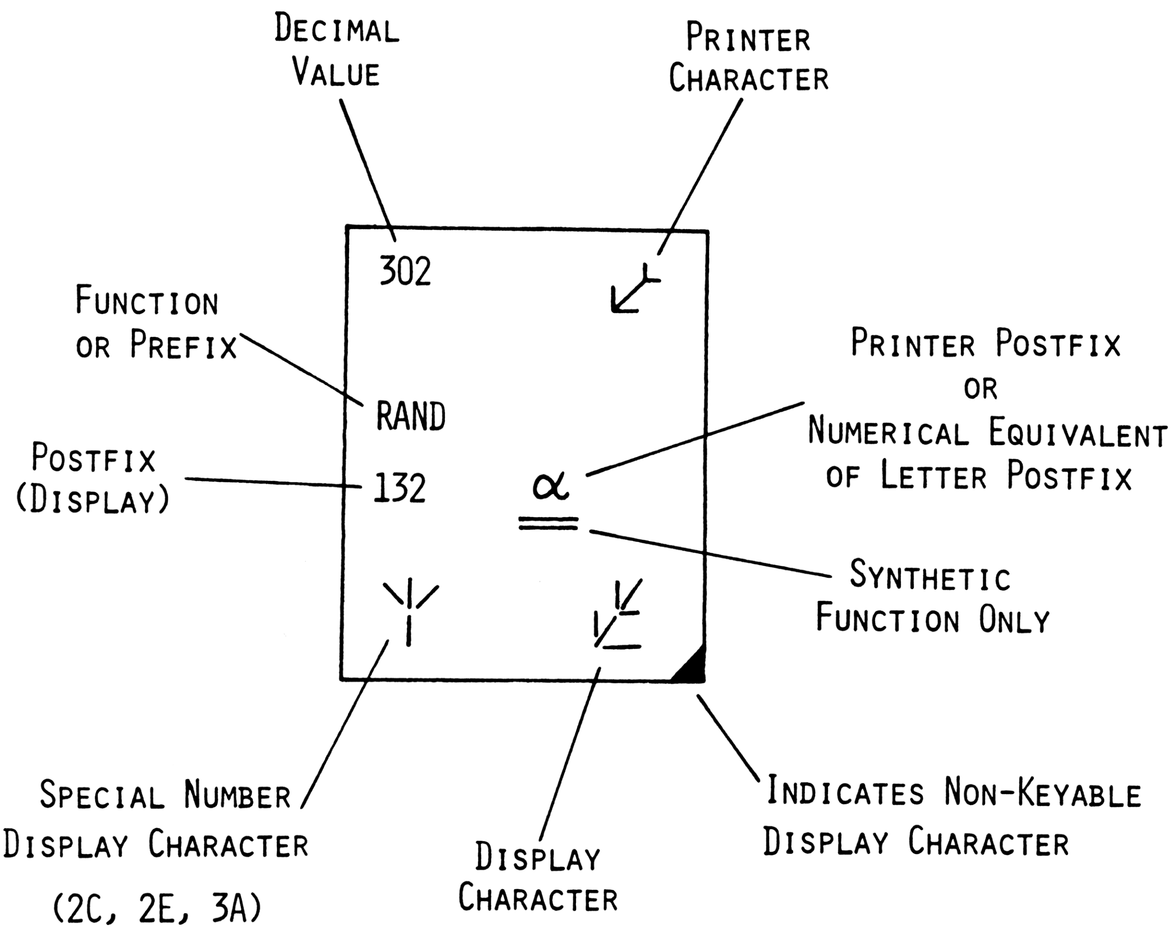


FIGURE 2-4. SAMPLE BYTE TABLE "BOX"

these bytes, it also must look at the following byte to complete the program instruction. For example, byte 90 is the prefix 'RCL', which requires a second, or 'postfix', byte to identify the register to be recalled. The postfix value of each byte is given by the number or letter listed immediately below the function name in a Byte Table box. To decipher the bytes '90 4C', for example, we observe from the Table that the first byte, 90, is the 'RCL' prefix, so we must look at the next byte, 4C, as a postfix, specifically '76'. Hence bytes 90 4C constitute the line 'RCL 76'. Similarly, 92 1D is 'STO+29', A8 03 is 'SF 03', etc. Notice that for bytes 00-63, the postfix value for the byte is the same as the decimal equivalent of the byte value. The first 5 bytes of row 7, when used as postfixes, access the stack registers T, Z, Y, X, and L (Last X), so that 91 70 is 'STO T', 98 73 is 'VIEW X', and so forth.

Some of the bytes in rows 6 and 7 are shown with two postfix values, one or both of which is double-underlined. The underlines indicate a postfix value that is only accessible with synthetic program techniques. These alternate values will be explained in Chapter 4.

The postfix values found in rows 0-7 are duplicated in rows 8-F, apparently shortchanging us by 127 possible postfixes. However, this is not a real duplication: the postfixes in the lower half-table enable indirect execution of the prefix functions. For example, 91 52 is 'STO 82', but 91 D2 is 'STO IND 82'. This feature allows use of any data register from R_{00}-R_{99} for indirect addressing. Other examples are AA AA = 'FS?C IND 42'; 9D 8F = 'SCI IND 15'; 9F 86 = 'TONE IND 06'.

Byte 'AE' has a dual role when used as a prefix. If the postfix is from the upper half-table, AE executes as 'GTO IND'; if the postfix is from the lower half-table, AE becomes 'XEQ IND'. For example, AE 2A is 'GTO IND 42', whereas AE AA is 'XEQ IND 42'.

Bytes A0-A7 have 'XROM' as their 'function' names. These bytes are prefixes, but not in quite the same sense as described in the preceding. Each peripheral function, such as 'WDTA' or 'ACSPEC', including the non-programmable functions like 'WALL' or 'LIST', has a unique two-byte code associated with it, found in the Table in the range between A0 00 and A7 FF.

Table 2-1. The HP-41C Byte Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	0 ◆ NULL 00	1 × LBL 00 01	2 x̄ LBL 01 02	3 ← LBL 02 03	4 α LBL 03 04	5 β LBL 04 05	6 Γ LBL 05 06	7 ↓ LBL 06 07	8 Δ LBL 07 08	9 σ LBL 08 09	10 ◆ LBL 09 10	11 λ LBL 10 11	12 μ LBL 11 12	13 ∡ LBL 12 13	14 τ LBL 13 14	15 Φ LBL 14 15	0
1	16 Θ 0 16	17 Ω 1 17	18 δ 2 18	19 Å 3 19	20 å 4 20	21 Ä 5 21	22 ä 6 22	23 Ö 7 23	24 ö 8 24	25 Ü 9 25	26 ü . 26	27 Æ EEX 27	28 æ CHS 28	29 ≠ GTO α 29	30 £ XEQ α 30	31 ▒ SPARE 31	1
2	32 RCL 00 32 (space)	33 ! RCL 01 33	34 " RCL 02 34	35 # RCL 03 35	36 $ RCL 04 36	37 % RCL 05 37	38 & RCL 06 38	39 ' RCL 07 39	40 (RCL 08 40	41) RCL 09 41	42 * RCL 10 42	43 + RCL 11 43	44 , RCL 12 44	45 − RCL 13 45	46 . RCL 14 46	47 / RCL 15 47	2
3	48 0 STO 00 48	49 1 STO 01 49	50 2 STO 02 50	51 3 STO 03 51	52 4 STO 04 52	53 5 STO 05 53	54 6 STO 06 54	55 7 STO 07 55	56 8 STO 08 56	57 9 STO 09 57	58 : STO 10 58	59 ; STO 11 59	60 < STO 12 60	61 = STO 13 61	62 > STO 14 62	63 ? STO 15 63	3
4	64 @ + 64	65 A - 65	66 B * 66 n	67 C / 67	68 D X<Y? 68	69 E X>Y? 69	70 F X ≤ Y? 70	71 G Σ+ 71	72 H Σ- 72	73 I HMS+ 73	74 J HMS- 74	75 K MOD 75	76 L % 76	77 M %CH 77	78 N P-R 78	79 O R-P 79	4
5	80 P LN 80	81 Q X^2 81	82 R SQRT 82	83 S Y^X 83	84 T CHS 84	85 U e^X 85	86 V LOG 86	87 W 10^X 87	88 X e^X-1 88	89 Y SIN 89	90 Z COS 90	91 [TAN 91	92 \ ASIN 92	93] ACOS 93	94 ↑ ATAN 94	95 _ DEC 95	5
6	96 ` 1/X 96	97 a ABS 97	98 b FACT 98	99 c X≠0? 99	100 d X>0? 00 100	101 e LN1+X 01 101	102 f X<0? A 102	103 g X=0? B 103	104 h INT C 104	105 i FRC D 105	106 j D-R E 106	107 k R-D F 107	108 l HMS G 108	109 m HR H 109	110 n RND I 110	111 o OCT J 111	6
7	112 p CL Σ T	113 q X<>Y Z	114 r PI Y	115 s CLST X	116 t R↑ L	117 u RDN M	118 v LASTX N	119 w CLX O	120 x X=Y? P	121 y X≠Y? Q	122 z SIGN ⊢	123 π X≤0? a	124 | MEAN b	125 → SDEV c	126 Σ AVIEW d	127 ⊦ CLD e	7
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8	128 ✦ DEG 00	129 × RAD 01	130 x̄ GRAD 02	131 ← ENTER 03	132 α STOP 04	133 β RTN 05	134 Γ BEEP 06	135 ↓ CLA 07	136 Δ ASHF 08	137 σ PSE 09	138 ◆ CLRG 10	139 λ AOFF 11	140 μ AON 12	141 ∡ OFF 13	142 τ PROMPT 14	143 Φ ADV 15	8
9	144 Θ RCL 16	145 Ω STO 17	146 δ STO+ 18	147 Å STO- 19	148 å STO* 20	149 Ä STO/ 21	150 ä ISG 22	151 Ö DSE 23	152 ö VIEW 24	153 Ü ΣREG 25	154 ü ASTO 26	155 Æ ARCL 27	156 æ FIX 28	157 ≠ SCI 29	158 £ ENG 30	159 ▒ TONE 31	9
A	160 XROM 32	161 ! XROM 33	162 " XROM 34	163 # XROM 35	164 $ XROM 36	165 % XROM 37	166 & XROM 38	167 ' XROM 39	168 (SF 40	169) CF 41	170 * FS?C 42	171 + FC?C 43	172 , FS? 44	173 – FC? 45	174 . GTO IND XEQ IND 46	175 / SPARE 47	A
B	176 0 SPARE 48	177 1 GTO 00 49	178 2 GTO 01 50	179 3 GTO 02 51	180 4 GTO 03 52	181 5 GTO 04 53	182 6 GTO 05 54	183 7 GTO 06 55	184 8 GTO 07 56	185 9 GTO 08 57	186 : GTO 09 58	187 ; GTO 10 59	188 < GTO 11 60	189 = GTO 12 61	190 > GTO 13 62	191 ? GTO 14 63	B
C	192 @ GLOBAL 64	193 A GLOBAL 65	194 B GLOBAL 66	195 C GLOBAL 67	196 D GLOBAL 68	197 E GLOBAL 69	198 F GLOBAL 70	199 G GLOBAL 71	200 H GLOBAL 72	201 I GLOBAL 73	202 J GLOBAL 74	203 K GLOBAL 75	204 L GLOBAL 76	205 M GLOBAL 77	206 N X < > 78	207 O LBL 79	C
D	208 P GTO 80	209 Q GTO 81	210 R GTO 82	211 S GTO 83	212 T GTO 84	213 U GTO 85	214 V GTO 86	215 W GTO 87	216 X GTO 88	217 Y GTO 89	218 Z GTO 90	219 [GTO 91	220 \ GTO 92	221] GTO 93	222 ↑ GTO 94	223 _ GTO 95	D
E	224 ` XEQ 96	225 a XEQ 97	226 b XEQ 98	227 c XEQ 99	228 d XEQ 00 100	229 e XEQ 01 100	230 f XEQ A 102	231 g XEQ B 103	232 h XEQ C 104	233 i XEQ D 105	234 j XEQ E 106	235 k XEQ F 107	236 l XEQ G 108	237 m XEQ H 109	238 n XEQ I 110	239 o XEQ J 111	E
F	240 p TEXT 0 T	241 q TEXT 1 Z	242 r TEXT 2 Y	243 s TEXT 3 X	244 t TEXT 4 L	245 u TEXT 5 M [	246 v TEXT 6 N \	247 w TEXT 7 O]	248 x TEXT 8 P ↑	249 y TEXT 9 Q =	250 z TEXT 10 ⊢ T	251 π TEXT 11 a	252 | TEXT 12 b	253 → TEXT 13 c	254 Σ TEXT 14 d	255 ⊢ TEXT 15 e	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

More properly, we might consider the leading nybble 'A' as the prefix, and the remaining 3 nybbles as a postfix identifying a specific peripheral function.

The 'XROM' codes that display when a peripheral is absent are derived directly from the byte values for the peripheral functions. The three nybbles following an 'A' prefix nybble are broken up into two 6-bit sections. The two numbers displayed with 'XROM' are just the decimal equivalents of the two sections. For example:

```
'PRX' = hex A7 54 = binary 1010|0111 01|01 0100 = 'XROM 29,20'
                                     29       20

'WSTS'= hex A7 8A = binary 1010|0111 10|00 1010 = 'XROM 30,10'
                                     30       10
```

Some confusion might arise from the one-byte functions in rows 0, 2, and 3. In order to allow direct storage and recall from as many as 100 registers, the 'STO' and 'RCL' functions must be two bytes; otherwise too much of the hex table would be used up with explicit functions such as STO 99 or RCL 50. On the other hand, a lot of two-byte functions in a program uses up memory rapidly. As a compromise, the HP-41C allows one-byte access to R_{00}-R_{15} by reserving one-byte codes for STO and RCL 00-15. For R_{16}-R_{99}, a two-byte prefix/postfix combination is necessary. At the same time, there could have been direct access to R_{100}-R_{255}, by assigning different numerical postfix numbers to each byte in the table, but that would have left no room for the versatile indirect addressing properties of the existing scheme.

The same choice between function versatility and program conservation appears in the availability of 'short form', or one-byte, labels, as well as two-byte labels. Labels 00-14 are explicitly coded in row 0 of the Table, whereas labels 15-99 require two bytes each: the prefix CF and a postfix from the upper half-table. The two-byte labels may also use postfixes 66-6F and 7B-7F, generating the so-called 'local alpha labels', 'LBL A' through 'LBL J' and 'LBL a' through 'LBL e'.

Matters begin to look a little more mysterious as we continue to progress downwards through the table, entering the regions beginning with row B. Look, for example, at row E. Why are there 16 different 'XEQ' prefixes? In this region of the table, the function beginning with each byte consists of two or more bytes, so the Byte Table as drawn becomes inadequate to list every detail. Consider first the 'two-byte GTO's' in row B contrasted with the 'three-byte GTO's' found in row D--again we have the compromise between versatility and memory use similar to that of the one- and two-byte STO's and RCL's.

When 'GTO 05', for example, is keyed into program, these two bytes are coded into memory:

1011 0110|0000 0000

The first byte is 'B6', which the table tells us corresponds to 'GTO 05'. What, then, is the second byte for? Here we encounter one of the many invisible but wondrous features of the HP-41C: 'rapid branching'. The first time that a running program encounters the 'GTO 05' line, the processor must search through the current program until it finds 'LBL 05', a (relatively) slow process. Once it finds its destination, it then records the distance between the 'GTO 05' and the 'LBL 05' in the second byte of the 'GTO 05' line code, so that in all subsequent executions of the GTO it can jump directly to the LBL. We will cover the details of how this information is coded in the next section--suffice it to say that the one byte reserved for the jump information allows jumps of up to 16 registers in length, either forward or backward. The three-byte GTO's of row D have an extra 5 bits for the distance record, permitting jumps of up to 512 registers. A programmer's choice of two- or three-byte GTO's thus amounts to a choice between program speed and program length: if the jump is less than 16 registers, the two-byte GTO and its corresponding one-byte LBL save two bytes without loss of speed. For longer jumps, however, the short forms will require considerably more execution time.

The XEQ's found in row E of the table are structured the same as row D three-byte GTO's. They also execute the same way, with the additional feature that the address of the XEQ line is recorded as well as the length of the jump to the LBL. The return addresses are stored in two special registers, as part of a 'return stack' (see Section 4F).

Take heart--we're almost done with the table! Row F next: these bytes identify alpha text program lines. When the processor encounters an F (binary 1111) as the first nybble of a byte, it is alerted that the program line contains alpha text. The number of characters

in the text, from one to fifteen, is indicated by the second nybble of the text byte. In PRGM mode, an 'Fn' byte results in a display with the text symbol "ᵀ" followed by n characters derived from the next 'n' bytes of program memory. In a running program, or with SST, the processor simply copies the next 'n' bytes of program into the alpha register, then resumes execution with the byte following the last of the 'n' text bytes. Examples:

ᵀA = F1 41
ᵀBIG = F3 42 49 47
ᵀTHRILL = F6 54 48 52 49 4C 4C

Text lines require explanation of the final entry in each Byte Table box. The character in the lower right-hand corner shows the alpha character displayed if the corresponding byte is either in the alpha register, in a program text line, or in a global alpha program line . The display mechanism is capable of generating 83 different characters. Of these, 59 constitute the normal character set and may be keyed in directly. Two more, the text symbol "ᵀ" and the append symbol "⊢", can be 'keyed in', but not in arbitrary positions.

Nineteen characters, identified by the black triangle in the lower right corner of a box, cannot be keyed in directly. They do appear in displays resulting from use of the printer function 'BLDSPEC'. The 'flying goose' "≻" is seen doggedly making its rounds during a running program, but its counterpart "≺" requires extraordinary effort to flush from its nest since the code 2C normally displays as the comma character. The bytes 2C, 2E and 3A are shown with two characters. The normal character is shown on the right-notice that these three characters, ",", ".", and ":", are displayed using the special LCD dot/comma segments between the main "starburst" segments. The left character can be user controlled in special number displays, as described in Section 7C. Finally, if a byte is not assigned one of the 82 characters mentioned so far, it 'defaults' to the full 'starburst' character "▩". Except for byte 3A, the starburst characters are not shown in the Table.

Two notes: First, the 'append' operation is coded with byte 7F. If this byte appears by itself, it is 'CLD', but if is the second byte of a text line, it causes the remaining bytes in the line to be appended to the current contents of the alpha register. The second nybble 'n' of the 'Fn' text byte will have a value one greater than the number of characters actually appended. "LEG" is 'F3 4C 45 47' but "⊢LEG" is 'F4 7F 4C 45 47'. Second, byte F0, or 'text 0', normally does not appear in user programs except as the 'IND T' suffix, but does play a role in the coding in key assignment registers (Section 2E).

We have come to the 'END'. The bytes C0-CD, 'GLOBAL', play a dual role--they identify both 'END' lines and global alpha labels. If the third byte of a line starting with 'Cn' ($0 \leq n < E$) is a text byte 'Fn', then the line is a global alpha label. Otherwise, it is a three-byte 'END'. For both types of lines, the second, third and fourth nybbles give the distance from the current line to the next 'END' or alpha label preceding in memory. The distance is coded as in the three-byte GTO's (Section 2C). Thus, all the global lines are linked together; a GTO-alpha or XEQ-alpha starts searching the global chain from the end of program memory, the permanent .END., backwards to the first global line in memory, which is identified by its first two bytes 'C0 00'. 'CAT 1' shows the labels and END's in order forward from the first global line.

In 'END' lines, the third byte is used to provide information about the current program --whether it has been packed and whether it is the last program in memory, i.e., if the END is the permanent .END. In the third byte, a first nybble '0' indicates a normal END; a '2' identifies the permanent .END. For the second nybble, '9' means that the program file is packed; 'D' indicates that the file needs packing.

The global alpha labels are the most complicated of the HP-41C program lines. The third byte is an 'Fn', where 'n' is the hex number one greater than the number of characters in the label name. The fourth byte in the line, the extra byte reserved by the 'Fn', contains a code for the key assigned to the label. '00' indicates no key assignment. The remaining 'n-1' bytes of the line spell out the name of the label. Example: LBL "ABC" = 'C1 mn F4 ab 41 42 43', where 'lmn' is the distance to the next label, and 'ab' identifies an assigned key.

There are a few mavericks in the Byte Table remaining to be examined. Bytes 1D and 1E, alien prefixes in the land of one-byte functions, are the prefixes for GTO (alpha) and XEQ (alpha), respectively. When one of these bytes starts a line, it is followed by an 'Fn' byte, reserving the next 'n' bytes for the name of the label called. For example:

GTO "BLAZES" = 1D F6 42 4C 41 5A 55 53
XEQ "SPY" = 1E F3 53 50 59

Next, we have the 'invisible man', byte '00'--the 'null' function. These bytes are normally invisible to the programmer, but are used by the HP-41C to facilitate editing and as place holders for future coding. As an example of its use, a null is automatically inserted in front of the first digits of a number entry line. The null serves to isolate this line from the previous line in case that also is a number line; the null is equivalent to an invisible 'ENTER' in this context. Upon execution of 'PACK', such a null is removed if it is found to be unnecessary, along with all other superfluous nulls in current program memory.

Finally, the bytes 1F, AF, and B0 are 'spare' function codes; that is, they have no prefix or stand-alone use, showing up in memory only as postfixes.

2C. REGISTER, PLEASE

We have seen that the HP-41C user memory, and its replicas on magnetic cards or in bar code, can be viewed as a long string of binary bits, like a machine-gun belt with a pattern of missing bullets. To make sense of the string, as the processor scans along it groups the bits into nybbles and bytes for decoding, and into 7-byte registers for data storage and retrieval. But, in order for the processor to know which bits to group, there must be an addressing scheme to identify each section of memory. The scheme must permit both 'absolute' addressing so the processor can retrieve information in permanent locations such as the stack registers, and also 'relative' addressing, to ensure that program jumps such as used by 'GTO' or 'XEQ' will remain unchanged by the 'SIZE' operation.

Since the smallest element of program storage is the byte, and since data registers are an integral number of bytes, it is sufficient to have individual addresses only down to the byte level, rather than for each nybble or even every bit. There should also be an address for each register, to facilitate data handling, and to speed up the process of finding an address --what we want is something like a street address, with the register and byte numbers analogous to the street name and house number, respectively. These simple ideas lead us right to the actual addressing system used in the HP-41C. Each byte in user memory has an address of the form:

[n a b c].

'abc' is a three-digit hexadecimal number designating a particular register. We should make a distinction between the absolute address of a data register and its data register number, which is a relative address. The memory location of the number stored in data register R_{00}, for example, is not fixed. When a new 'SIZE' is executed, the contents of memory are moved around to change the allocation between program and data storage. For convenience to the user, the original contents of R_{00} will still be accessed by 'STO 00', etc., even though the location, or absolute address, of the contents may have changed (see Section 4G).

The remaining digit of the 4-digit address, 'n', is the 'byte number'. Each register is 7 bytes, so 'n' can assume one of the 7 values 0 through 6. We now expand our conception of the processor to include an address 'pointer', which always contains the 4-digit address of the program byte currently being processed. The convention used by the HP-41C is that 'forward' in program memory in the direction of increasing program line numbers corresponds to <u>decreasing</u> address (see Figure 2-5). Upon execution of 'SST', the pointer decrements the byte number by the number of bytes in the program line, with byte 6 as the first byte of a register, and 0 as the last. When a register boundary is crossed, 'n' starts over at 6, and 'abc' is decremented by 1. The data registers are numbered in the opposite direction, so that if R_{10}, for example, is (absolute) register '123', then R_{11} will be '124', R_{12} will be '125', etc. If we could place the pointer in a data register, then single-step in PRGM mode, we would see seven program bytes for each register, starting with a byte consisting of the mantissa sign nybble and first mantissa digit, and finishing with the two exponent digits.

It was stated that program branch jumps caused by 'GTO' and 'XEQ' store the jump lengths, rather than the absolute addresses of the labels, in the initiating program lines. This is so that a shift of program register contents such as caused by 'SIZE' or by inserting a new program file at a higher address will not require changing the stored jumps. The distance of a jump is expressed as a number of whole 7-byte registers plus remaining bytes. The distance is measured from the byte containing the (first part of the) jump distance code, to the byte immediately preceding the designated label. To clarify this coding, let's look at a few examples. First, take the routine:

```
01 GTO 05                B6 22
02 "ABCDEFGHIJKLMNO"     FF 41 42 .... 4E 4F
03 LBL 05                06
04 "ABCDEFGHIJ"          FA 41 42 .... 49 4A
05 GTO 05                B6 82
```

(2C-1)

where the numbers to the right of the program lines are the byte codes for the lines. Prior to the first execution of the routine, the code for lines 01 and 05 would have been 'B6 00'. The 'B6' identifies 'GTO 05'; the '00' indicates that the jump distance is unknown. Following execution, the codes are as shown above, with each '00' replaced by a distance code. Writing out the bytes in binary, we can see how the bits are interpreted:

	Direction	# Bytes	# Registers
(line 01) 22 =	0	010	0010
(line 05) 82 =	1	000	0010

If the first bit is zero, the jump is forward (to a lower address); if the bit is a one, the jump is backward. For two-byte GTO's, the jump information is entirely in the second byte of each 'GTO 05', so we count the jump distance from there. From the '22' in line 01, we count off 2 registers + 2 bytes = 16 bytes, starting with the FF in line 02, so that the pointer ends up at the "O" character byte of line 02. The pending instruction is then the 'LBL 05'. For the GTO in line 05, we count backwards 2 registers + 0 bytes = 14 bytes, starting with the B6. Again, the pointer goes to the "O". The maximum length of such jumps is F registers + 7 bytes = 112 bytes, or 16 registers. The 3-byte GTO's and XEQ's are similar to the 2-byte GTO's, but with a different ordering of the jump information. Substituting the longer forms in Routine 2C-1:

```
01 GTO 45                D8 02 2D
02 "ABCDEFGHIJKLMNO"     FF 41 42 .... 4E 4F
03 LBL 45                CF 2D
04 "ABCDEFGHIJ"          FA 41 42 .... 49 4A
05 XEQ 45                E0 02 AD
```

(2C-2)

Again breaking the codes into bits, and grouping:

	Type	# Bytes	# Registers	Direction	Label
(line 01) D8 02 2D =	1101	100	000000010	0	0101101
(line 05) E0 02 AD =	1110	000	000000010	1	0101101

Only 7 bits are required for label postfixes up to decimal 99; 3 more bits are needed for the number of bytes, 0-6. So with 4 bits for the line type (1101 for 'GTO', 1110 for 'XEQ'), and 1 bit for the direction, 9 bits remain for the number of registers. The jumps can therefore be up to 2^9 = 512 registers, which is larger than the memory.

The first byte of the GTO or XEQ line starts the jump coding, so we count off the jump from that first byte. For the 'GTO 45' in line 01, we count 4 bytes + 2 registers = 18 bytes from the D8 byte, which places the pointer on the "O" as before. Line '05 XEQ 45' moves the pointer backwards from the E0 byte, 0 bytes + 2 registers = 14 bytes.

Just as it is desirable for a data register number to be a relative rather than an absolute address to facilitate shifts of memory contents, there is no absolute program line number associated with any memory locations. The line number is a quantity that is recomputed each time it is required, i.e., for each program step displayed in PRGM mode or by a held SST or BST key. You may have noticed that the first time you switch to PRGM mode after running a program, or press 'BST', near the end of a long program, there is a noticeable pause before the current line is displayed. This 'dead time' is used for the processor to compute the line number, which it can only do from scratch by going back to the top of the current program file and chugging forward through the program, incrementing the line counter (stored in a special register) by one for each complete program line. It would be superfluous and time-consuming for the processor to keep track of line numbers during a running program, so it must do the full line number computation once when the user next switches to PRGM mode. Subsequent SST's are fast, but a BST can be slow because the processor has no way of knowing whether the preceding byte is a stand-alone byte or a postfix in a multi-byte function. It must again return to the top of the file and count forward by lines until it reaches a number one less

than the starting line.

2D. MEMORY PARTITIONING

Figure 2-5 is a pictorial representation of the HP-41C user memory, where we visualize all of the memory registers as stacked one on top of the other. The chart shows the 'mainframe' plus all four possible memory modules. The top of the chart is the 'top of memory', the highest numbered available data register. Going down the chart corresponds to decreasing data register number and absolute address, or increasing program line number. The horizontal direction represents the byte number, with the first byte, '6', of each register at the left, and the last, '0', at the right. Single-stepping moves the address pointer to the right through the bytes of a register, then back to the left to byte '6' of the next register lower.

The first data register, R_{00}, and the first program line of the first user program are immediately adjacent in memory, with no physical boundary between them. The current absolute address of R_{00} is stored by the HP-41C, so that the processor always knows which registers are allocated for data (those above R_{00}) and which are reserved for program (those below). The current 'size' is the number of registers between the top of memory and R_{00}. When a memory module is added, its 64 registers are added at the top of memory, so that the 'size' automatically increases by 64 (hex 40). When 'SIZE abc' is executed, the contents of data and program registers are moved upwards or downwards until the original contents of R_{00} are in Register 'mno' (mno is a three digit hexadecimal number), 'abc' registers from the top.

Register 'mno-1' is the first register of program memory. If we start with no programs in memory, the last three bytes of Register 'mno-1' automatically contain the permanent .END. line. This .END. is always present in user memory, necessarily since it is the first link in the global address chain connecting all global labels and END's in memory. When we start keying in a program, the first four bytes overwrite the null bytes remaining in Register 'mno-1'. If more bytes are added, the .END. is automatically shifted to the last 3 bytes of the next program register, providing 7 more bytes for program. This process is repeated until the program is complete, or until all available program registers are full. If we key in an 'END' at some point, we erect a 'barrier' in memory, serving to divide the previously keyed program lines into a self-contained program file. The 'END' line itself is the barrier, for when it is encountered using SST, or during a program search for a local label, it causes the address pointer to jump back to the next 'END' up the label chain, or to byte '0abc', if the current program is the first in memory.

If we have keyed in a total of 'def' registers of program (including the .END.), the address of the register containing the .END. will be [pqr = mno - def]. Remember that all such register address arithmetic is done in hexadecimal. In the HP-41C, 'pqr' can never be less than hex 0C0 (decimal 192). The choice of '0C0' for the bottom of programmable memory makes addresses in the 'mainframe' range from 0C0 to 0FF. If the first digit of a register number is a '1', the register is in a memory module: module 1--registers 100 through 13F; module 2--140 through 17F; module 3--180 through 1BF; module 4--1C0 through 1FF.

At any time, there are [pqr - 0C0] registers available for program, less the current number of registers used for key assignments. The user function key assignments are encoded in a block of registers starting at 0C0 and going upwards in memory (the details of the coding are given in Section 2E). If 'jkl' registers are used for key assignments, then there are [ghi = mno - def - jkl - 0C0] registers still available for new program lines or assignments. Overall we have

$$(N+1)*40 = abc + def + ghi + jkl$$

registers in the system, where 'N' is the number of memory modules currently inserted.

Below Register 0C0, there is a gap in the chart, meant to represent a void in the addressing scheme, since no registers exist that would correspond to addresses in the region. Between addresses 000 and 00F, however, there is a highly interesting block of 16 registers. We will call these registers the 'status registers', since their contents are recorded by the card reader on Track 1 of the cards generated with the 'WSTS' ('Write Status') function. Access to these registers is the basis of synthetic programming; their study merits an entire chapter, Chapter 4.

BYTE NUMBER

6 5 4 3 2 1 0

REGISTER NUMBER

Register	Contents	Module	Span
1FF	TOP OF MEMORY	MODULE 4	abc
	DATA		
1C0			
mno	R_{00}	MODULE 3	
	FIRST USER PROGRAM		def
180			
		MODULE 2	
140			
pqr	LAST USER PROGRAM .END.	MODULE 1	
	"AVAILABLE"		ghi
100		"MAINFRAME"	
	KEY ASSIGNMENTS		jkl
0C0			
	(NON-EXISTENT ADDRESSES)		
00F	STATUS REGISTERS		
000			

FIGURE 2-5. HP-41C USER MEMORY PARTIONING

2E. THE KEY ASSIGNMENT REGISTERS

The key assignment registers extend from Register 0C0 up to, but not including, the register containing the .END. The registers contain codes that tell the processor which functions are assigned to which keys (recall that user global label assignments are recorded in the label itself). Consider the following key sequence:

ASN ALPHA "LN" ALPHA 8 (Assign 'LN' to the 8 key)

If we were able by some sleight-of-hand to place the address pointer into Register 0C0 in PRGM mode and list the contents, we would see the following (with arbitrary line numbers, and the byte codes listed to the right):

01 ""	F0
02 LBL 03	04
03 LN	50
04 RCL 05	25

(Line 01, byte F0 or 'TEXT 0', shows in the display as '01 T'.) Four bytes are not enough to fill a register--there are three invisible null bytes between lines 01 and 02. The nulls disappear when we make a second assignment:

ASN ALPHA "LOG" ALPHA SHIFT 8 (Assign 'LOG' to the shifted 8 key)

Now Register 0C0 contains:

01 ""	F0
02 LBL 03	04
03 LOG	56
04 RCL 13	2D
05 LBL 03	04
06 LN	50
07 RCL 05	25

As a program, the sequence of lines doesn't mean anything, although we recognize the 'LOG' and 'LN' that were assigned. Rather, the bytes make up a special code. The first byte, 'F0', identifies the register as a key assignment register and divides it off from adjacent assignment registers. The next three bytes are a code for the 'LOG' assignment, the second assignment made. The last three bytes encode the 'LN' assignment. In both sets of three bytes, the first two bytes identify the assigned function and the third designates the key. For assignments of HP-41C functions, only one byte is required to identify the function, so that the '04' byte (LBL 03) is stuck in as a filler. If a peripheral function is assigned, both function bytes are required to represent the function. For example, if we had assigned 'PRP' and 'WSTS' instead of 'LN' and 'LOG', Register 0C0 would contain:

01 ""	F0
02 WSTS	A7 8A
03 RCL 13	2D
04 PRP	A7 4D
05 RCL 05	25

The code for a designated key is as follows: Suppose we assign key 'MN', i.e., the key in row M and column N of the keyboard. Then the byte representing that key will be hexadecimal 'XY', where [X = N-1], and [Y = M]. The '8' key assigned above is key '53', so that [X = 2] and [Y = 5], yielding the 'RCL 05' line in the assignment register. Other examples: the 'COS' key, key 24, is represented by byte 32, or 'STO 02'; 'R/S', key 84, is coded with byte 38 = 'STO 08'.

The code for the shifted key '-MN' is obtained from [X = N-1], [Y = M+8]. Thus the assignment of the shifted '8' key, key -53, is coded with byte 2D = 'RCL 13'. For shifted keys in row 8, where N=8, we carry the '1' resulting from [8+8=10] into X. For example, the 'VIEW' key, key -84, is coded with byte 40 = '+'.

The keys ordinarily numbered 42, 43, and 44, i.e., 'CHS', 'EEX' and '←', and their

shifted counterparts, are physically in columns 3, 4, and 5, respectively, and must be so numbered for use in the key assignment byte formula. It is as if the 'ENTER' key covers an imaginary key 42. Figure 2-6 shows the key assignment codes in a keyboard chart form for easy reference. The number on each key is the key designation byte for assignment of that key. The codes for the shifted keys are shown above the keys.

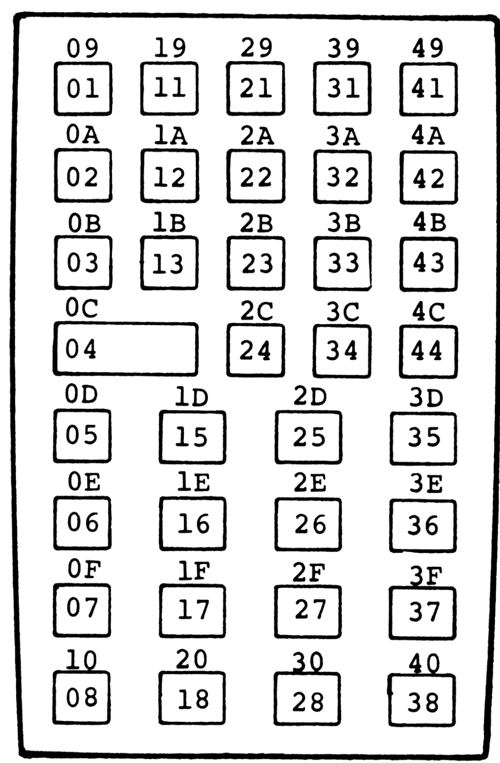


FIGURE 2-6. KEY ASSIGNMENT BYTES

If the assigned function is a prefix, such as 'STO', 'ISG', 'GTO', etc., the listing of the assignment register will show the function byte and the key designation byte merged into a single program line. The key designation byte acts as a postfix for the assigned prefix.

When a non-programmable HP-41C function is assigned, the function byte is found in row 0 of the Byte Table, so that the corresponding program line is one of the short-form labels, or a null. Table 2-2 shows the correspondence.

Although most of the entries in Table 2-2 correspond to normal assignments, bytes 01, 05, 0B, 0C, 0D, and 0E represent 'functions' that are not normally assignable. Using the key assignment programs in Chapter 5, however, we can place these bytes into assignment registers with amusing results. The 'functions' '@c' and '2_ _' are so named because pressing the key to which one is assigned produces the display indicated by the name. Execution of '@c' sometimes does nothing; at other times a 'GTO..' executes. '2_ _', upon entry of a two-digit number, causes the HP-41C to 'lock up' for some time. The more practical use of bytes 05, 0B, and 0E allows us to reassign the 'R/S', '←', and 'SHIFT' functions, respectively. Pressing the reassigned correction key always deletes the current program line, regardless of whether or not the HP-41C is in PRGM mode. Finally, the byte 0C reassigns the 'rocker key' functions 'ALPHA', 'PRGM' and 'USER'. The choice of function depends upon the designated key (!): if the key is in row 1 or 5, 'ALPHA' is assigned; keys in rows 2 and 6 will be assigned to 'PRGM'; 'USER' results from assignments to keys in the remaining rows 3, 4, 7, and 8.

TABLE 2-2

Assignment of Non-Programmable HP-41C Functions

Function	Byte	Program Line
CAT	00	null
@c	01	LBL 00
DEL	02	LBL 01
COPY	03	LBL 02
CLP	04	LBL 03
R/S	05	LBL 04
SIZE	06	LBL 05
BST	07	LBL 06
SST	08	LBL 07
ON	09	LBL 08
PACK	0A	LBL 09
[←]	0B	LBL 10
ALPHA/PRGM/USER	0C	LBL 11
2__	0D	LBL 12
SHIFT	0E	LBL 13
ASN	0F	LBL 14

CHAPTER 3

EXOTIC EDITING WITH THE BYTE JUMPER

3A. NORMAL EDITING

Every HP-41C programmer knows the simple rules governing normal editing: (1) In PRGM mode, a program line keyed in is inserted immediately following the program line initially shown in the display. All subsequent lines have their line numbers increased by one. (2) When the correction key is pressed, the displayed program line is deleted, and subsequent lines have their lines numbers decreased by one. The display will show the line preceding the deleted line. (3) Execution of 'DEL lmn' causes 'lmn' program lines to be deleted, including the line displayed. (4) 'PACK' does some kind of housekeeping, deleting invisible nulls to maximize the available program space.

These operations provide a simple, fast editing capability for the HP-41C. But for our purposes, the information in steps (1) through (4) above is inadequate; we need to know exactly what is going on in memory at the byte level, not at the program line level. So let us rewrite the rules as follows:

(1) The program line displayed in PRGM mode is the program line starting with the first non-null byte following the byte where the address pointer is currently situated. When a new program line is keyed in, the bytes constituting the new line are placed immediately following the last byte of the initially displayed line, by overwriting null bytes. If no null bytes are available, i.e., if the byte at the insertion location is not '00', the processor automatically inserts 7 nulls (or multiples of 7 nulls if required) before entering the new program bytes. The new line then overwrites as many of the new nulls as it requires, leaving the rest (invisibly) in the program. Insertion of exactly seven nulls makes the process of moving subsequent lines down in memory simple--each register containing user programs is just copied into the next register down, starting at the .END. and working back up to the register where the insertion is occurring.

A manual 'RTN', 'GTO.000', or 'GTO.001', moves the address pointer to the last byte of the preceding program. The first two operations, by setting the line number to '00', result in a display of '00 REG lmn' instead of a program line. When the line number is '00', program bytes are keyed in immediately after the current pointer byte rather than after the pending program line.

(2) When the correction key is pushed in PRGM mode, the bytes of the displayed line are replaced by an equal number of null bytes. The program pointer moves back one line.

(3) The 'DEL lmn' operation replaces all of the bytes in the next 'lmn' program lines with nulls. You might observe that an 'SST' following the deletion of a large number of lines requires a noticeable pause, which is actually the time required for the processor to scan through all the null bytes resulting from the deletion until it finds a non-null byte for display.

(4) An editing session can introduce a substantial number of superfluous nulls into a program file. The PACK operation removes all unnecessary nulls by moving non-null program bytes upwards in memory. Nulls found within multi-byte program lines are not removed.

When program bytes are shifted around in memory, either by editing (inserting or deleting bytes) or by packing, various jump-distance codes may become invalid. Hence, following any of these operations, the jump-distance bytes in all local labels in the file being edited are set to zero, so that they will have to be recomputed the next time the program is run. Furthermore, the relative addresses in the global label chain must be updated. Finally, the END line terminating the edited file is recoded to indicate that the file needs packing.

An example of program editing should clarify the rewritten rules. Starting from 'MEMORY LOST', we key in this simple program:

```
01 LBL 00
   .END.
```

If we write out all the bytes in the file, the program looks like this:

Address	Line Number	Line	Byte Code
60EE	01	LBL 00	01
50EE			00
40EE			00
30EE			00
20EE		.END.	C0
10EE			00
00EE			29

The addresses follow from the consideration that the 41C 'wakes up' with 47 (hex 2F) registers of program space, starting with Register 0C0: [0C0+02F-1=0EE]. So Register 0EE is the highest program register. In the .END. line, the '000' nybbles indicate that this is the topmost global label in memory; the '29' indicates a packed file, permanent .END. Now suppose we insert three '+' lines following line 01:

Address	Line Number	Line	Byte Code
60EE	01	LBL 00	01
50EE	02	+	40
40EE	03	+	40
30EE	04	+	40
20EE		.END.	C0
10EE			00
00EE			29

The nulls have been replaced by the '40' bytes. Now delete line 03:

Address	Line Number	Line	Byte Code
60EE	01	LBL 00	01
50EE	02	+	40
40EE			00
30EE	03	+	40
20EE		.END.	C0
10EE			00
00EE			2D

The '40' at address 40EE has been replaced by a null; the last byte of the .END. has changed to a '2D' to indicate an unpacked file. If we packed at this point, the '40' at 30EE would move up to 40EE, but then another '00' would be inserted at 30EE to keep the .END. in the last three bytes of the register. If we were to insert a one-byte line after line 02, it would simply overwrite the null at 40EE. But if we insert a two-byte line, e.g. 'STO 65', we get:

Address	Line Number	Line	Byte Code
60EE	01	LBL 00	01
50EE	02	+	40
40EE	03	STO 65	91
30EE			41
20EE			00
10EE			00
00EE			00
60ED			00
50ED			00
40ED			00
30ED	04	+	40
20ED		.END.	C0
10ED			00
00ED			2D

Since there was only one null byte available between lines 02 and 03, 7 more nulls were inserted. Two were then overwritten by the 'STO 65' bytes. The '40' at 30EE was moved down to 30ED. Finally, the .END. was reinserted, at the end of Register 0ED. Following a 'PACK', the program becomes:

60EE	01	LBL 00	01
50EE	02	+	40
40EE	03	STO 65	91
30EE			41
20EE	04	+	40
10EE			00
00EE			00
60ED			00
50ED			00
40ED			00
30ED			00
20ED		.END.	C0
10ED			00
00ED			29

The user program lines have been pushed together, but since there is not room for the .END. in Register 0EE, it remains in Register 0ED.

3B. THE BYTE JUMPER

Armed now with sufficient knowledge of normal HP-41C operation, we can boldly sally forth into brand-new territory. It should be re-emphasized at this time that even if some of the procedures we are about to use seem strange, there is no risk for the HP-41C. Follow me through the following procedure:

1. Insert one memory module into the HP-41C.
2. Master clear. (HP-41C off; hold down correction key; HP-41C on.) A clean break with the past!
3. Execute 'SIZE 000'. This places the '.END.' in the module.
4. ASN "X<>" +; ASN "Σ+" Σ+. This fills Register 0C0 with two assignments.
5. HP-41C off; remove memory module; wait 60 seconds or so; replace module; HP-41C on. If you a have second module available, you can save the 60 seconds by plugging the 'dead' module in in place of the one removed. Now the .END., which we placed within the module has 'evaporated'. If you had turned the calculator on <u>before</u> replacing the modulé, 'MEMORY LOST' would have resulted. Evidently the processor checks to see if the register where the .END. is supposed to be exists, but not whether the .END. bytes are actually present in that register.
6. Switch to PRGM mode; you should see '00 REG 126' (190 if your module is double density). Now press SST once. After a few seconds' wait, you will see '01 ᵀ'. The address pointer is now in Register 0C0, the first assignment register! With the .END. absent, there was nothing to stop the pointer from rolling merrily through empty memory until it encountered the first non-null byte, which in this case is the 'F0' from the key assignments we made in step 4. If you SST 5 more times, the following should appear in sequence:

```
02 LBL 03
03 Σ+
04 LBL 00
05 LBL 03
06 X<>06
```

(If you press SST again, the pointer will end up in the status registers.) You will recognize this set of lines as the code for the key assignments made in Step 4.

7. Use BST to return to line 03. Don't be distressed if some of these SST's and BST's take a few seconds. Now push the correction key twice to delete lines 03 and 02.
8. Now key in ALPHA "A" ALPHA, resulting in line '02 ᵀA'. (Actually, any single character will work as well as "A".)
9. Press 'GTO..'; the display 'GTO..' will persist for a few seconds, followed by a quick 'PACKING'. SST once, and delete the line '01 LBL 01'. For the second and last time in this book, you have carried out a synthetic programming operation by the trick of 'module pulling'. From now on, we will be able to achieve all our goals without having to resort to such unpleasant tactics.
10. Press and hold the 'Σ+' key in USER mode. You should see a display of 'XROM 05,01'. If this does not occur, you must have made a mistake, so repeat steps 1 through 9. By direct editing of an assignment register, you have created a brand new key assignment,

called the 'byte jumper'.

The best explanation of the operation of the byte-jumper is that it is, in effect, a manually executed program text line. To understand this, recall from Section 2B what happens when the automatic execution of a text line occurs: the processor looks at the second nybble of the current program byte, i.e., the 'Fn' byte that signals the text instruction, copies the next 'n' bytes of program code into the alpha register, and advances the pointer by 'n' bytes. The byte jumper is the manual equivalent of this operation, not to be confused with the single stepping of a text line. The 'F' nybble that starts the process is 'provided' by pressing the USER key to which we have assigned "A" ('F1 41'). To see why this operation is of interest, key in these lines:

```
01 STO 04         34
02 "ABCDEFG"      F7 41 42 43 44 45 46 47
```
(3B-1)

With line 02 still showing in the display, switch PRGM off, USER on, and press the byte jumper key (Σ+). Switch to PRGM mode again, and you will see by single stepping:

```
02 X<Y?     44
03 X>Y?     45
04 X<=Y?    46
05 Σ+       47
```

Where did these program lines come from? As you can see by looking at the byte values of the 'new' program lines, the lines are simply the stand-alone functions corresponding to the characters "D", "E", "F", and "G" in the original text line "ABCDEFG". We started with the display showing the line '02 "ABCDEFG"', i.e., the address pointer was positioned on the '34' byte, line 01. Then we executed the byte jumper, which made the processor think it was executing a text line (don't confuse this imaginary text line with the real line 02). So it looked at the second nybble of the current byte, '34', copied the next 4 bytes into the alpha register, and advanced the pointer by 4 bytes to the '43' byte. With the pointer there, the display will show the next program line, which is the one-byte line '02 X<Y?' corresponding to the '44' byte. If you now press PRGM (off), ALPHA, you will see the four characters "✱ABC", which are the four bytes copied from the program. The starburst character is the 'F7' byte. Since the byte jumper is a manually executed function, it does not change the current program line number, even though the pointer moves, so that the line '02 X<Y?' has the same line number as the '02 "ABCDEFG"' line from which the jump was executed.

While the pointer is 'inside' the text line, 'SST' operates normally, but a 'BST' from any of the bytes sends the pointer back to the 'STO 04' line. Remember that 'BST' sends the processor back to the start of the program file, whence it counts forwards by lines, refusing naturally to jump into the middle of a multi-byte line.

Let's see what use can be made of byte jumping. To condense future instructions, I will introduce a new instruction, 'JUMP .lmn', which means 'byte jump from line lmn'. That is:

'JUMP .lmn' means:
1. GTO .lmn (do even if the displayed line number is already 'lmn')
2. PRGM off
3. Press the byte jumper key
4. PRGM on

'JUMP' without a line number means 'byte jump' from the current line, i.e., step 1 is omitted.

Now, using Routine 3B-1, try 'JUMP .002'. After step 4 of the jump you will see displayed '02 X<Y?'. Press the correction key once, then 'SST'. You should now see '02 "ABC‾EFG"'. The '‾' is the display character for a null, in this case the null with which the "D" byte was replaced by the deletion. You have modified a program text line, without having to delete the entire line! This is only the beginning--now change line 01 of the routine to 'STO 03'. (If you make any mistakes while editing that may introduce invisible nulls, eliminate them with 'PACK'. If, for example, a null preceded line 02, which we wouldn't know from the display, 'JUMP .002' would do nothing, since the second nybble of the null is '0'.) Next, 'JUMP .002' to see the line '02 /', i.e., the "C" byte. Key in '03 LBL 00', then press 'GTO .002'. The display will show '02 "ABC⊼EFG"'. You have replaced the null following the "C" with byte '01', from the 'LBL 00', which displays as the 'full man' character "⊼" when the full text line is displayed. The following sequence will produce the program steps shown to its right:

JUMP .002 Key in: 03 LBL 11 04 DEC 05 RCL 09 06 ACOS GTO .002	→

```
01 STO 03
02 "ABCμ_)]"
03 LBL 00
04 X>Y?
05 X<=Y?
06 Σ+
   .END.
```

This is your first serious example of 'synthetic programming', where non-keyable byte combinations are synthesized by extraordinary means. According to the byte-level editing rules of Section 3A, when we tried to insert the 'LBL 11', the processor had to insert 7 nulls to clear space for the new line. This also provided room for the 'DEC', 'RCL 09', and 'ACOS', but at the same time pushed the "DEFG" bytes down in memory, clear out of the 'range' of the 'F7' byte. Those four bytes thus 'become' stand-alone program lines 03-06.

Any such synthetic text line created with the byte jumper will execute normally, as you can verify by single-stepping the new line 02 (PRGM off), then checking the alpha register to see the resulting characters. We may use this technique to place in a program text line any of the 19 non-keyable HP-41C display characters, plus the append and text symbols (but not the geese). Each of these characters is found in the upper half of the Byte Table, so that each may be edited in as above using a directly keyable one-byte program line. Furthermore, any of the 128 printer characters can be placed in a text line (those without display equivalents will display as starbursts), for transfer to the print buffer using 'ACA' (see Section 6E).

Program lines that 'append nulls' to the existing character string in alpha are used frequently in synthetic programs. Here is an example of the creation of such a line--in this case, to append 5 nulls:

Key in:

```
01 ASTO 02
02 "⊦ABCDE"
```

```
JUMP
DEL 005
DEL 001
```

The line '02 "⊦ABCDE" is chosen to have the same number of characters as the number of nulls to be appended. The 'DEL 005' changes the characters to nulls. The 'DEL 001' removes the 'ASTO 02' used to control the byte jump.

The byte jumper editing described so far has the limitation that the first character in a text line cannot be changed, except to delete it to a null. In "ABCDEFG" for example, the "A" cannot be altered, since to key a new byte into the "A" position would require that the preceding byte be displayed prior to insertion of the replacement byte, but since the byte preceding the "A" is the 'F7', the display insists upon showing the entire text line. Inserted bytes will enter the program following the full text line.

However, arbitrary text lines, with non-keyable characters in any or all positions, can be created through an elaboration of the byte jumping procedure. Suppose we wish to make the text line "(#)", i.e. bytes 'F3 28 23 29'. In the previous example, the text bytes were created normally along with 'temporary' characters, which are then pushed out of the line by the desired characters. This time it will be the text byte itself that is created within yet another text line. The desired text characters will be first entered as normal one-byte instructions. Start with this routine:

```
01 STO 01       31
02 "BJ"         F2 42 4A
```

(3B-2)

The line '01 STO 01' is used to provide the '1' nybble to produce a byte jump of one byte; we shall refer to such a line as the 'controller', since it controls the length of a byte jump. Similarly the line '02 "BJ"' is a temporary text line we shall call the 'generator'. The 'controller' and the 'generator' are to be deleted once the byte jump editing is completed. Now execute 'JUMP .002' and key in the line '03 "ABC"'. The full program is now:

```
01 STO 01       31
02 "B■"         F2 42 F3
03 -            41
04 *            42
05 /            43
06 HMS-         00 00 00 4A
```

The entry of the line '03 "ABC"' places an 'F3' right after the '42', where it becomes the last byte of the generator line. The '41 42 43' ("ABC") do not 'fit' in the generator, so these three bytes will show up as three independent program lines 03-05. Finally, we have the '4A' byte, the original "J", which was pushed out of the generator by the insertion, so it becomes line '06 HMS-'. The three nulls left over from the insertion are, as usual, invisible.

Next, press 'GTO .002' and key in:

```
03 RCL 08
04 RCL 03
05 RCL 09
```

to place the bytes '28 23 29' in position immediately following the 'F3' byte. Then 'JUMP .002', key in '03 HMS-'. Now the program is:

```
01 STO 01       31
02 "BJ"         F2 42 4A
03 "(#)"        00 00 00 00 00 00 F3 28 23 29
04 -            00 00 00 00 41
05 *            42
06 /            43
07 HMS-         00 00 00 4A
```

The insertion of the 'HMS-' pushes the 'F3' out of the controller, whereupon it reassumes its role as a 'TEXT 3', grabbing the bytes '28 23 29' to complete the text line as the characters "(", "#", and ")", respectively. The various nulls are left over from the groups of 7 nulls placed into program for each insertion. To clean up, we delete lines 01, 02, and 04-07, then 'PACK'. With some practice, you will find that the whole procedure goes quite rapidly.

Byte jumper editing is in no way restricted to text lines, particularly using the 'controller-generator' method described last. As an amusing example, try the following: start again with Routine 3B-2 (recall the instruction format described in Section 1E):

```
                JUMP .002       [02 *         ]
03 TONE 1
                GTO .002        [02 "B■"      ]
03 LBL 09
                JUMP .002       [02 *         ]
03 HMS-
                SST             [04 TONE 0    ]
```

The 'TONE 0' looks normal, but try executing it by single-stepping with PRGM off. You should hear a new, low frequency tone lasting over 2 seconds!

With the controller-generator method, we can create almost any combination of prefix and postfix that we want, to make synthetic two-byte functions. The most important set of such functions are the status register access functions that we will investigate in Chapter 4.

CHAPTER 4

THE STATUS REGISTERS

4A. STRANGE POSTFIXES

By the end of Section 2B, we had examined the Byte Table in great detail, accounting for almost every entry in the Table. But there is still one important feature still to explore: Notice that bytes 64-65 have their postfixes shown with a double underline, and that each of bytes 66-6F has two postfix 'values', one or both of which are underlined. The underlined postfixes never arise in 'normal' HP-41C programming. Consider postfixes 66 through 6F, shown as the letters "A" through "F". These letter postfixes are only seen in program lines involving the 'local alpha labels', such as 'LBL C' or 'XEQ F'. The keyboard logic prevents us from attaching these postfixes to other prefixes. For example, when we press 'STO', the ALPHA key is disabled so that only numerical postfixes can be entered. But the byte jumper has freed us from keyboard constraints, so let's try making a 'STO A'.

```
01 STO 01
02 "BJ"
                    JUMP .002          [02 *          ]
03 STO 22 (arbitrary postfix)
                    GTO .002           [02 "B▓"       ]
03 X<0?
                    JUMP .002          [02 *          ]
```

At this point, we're ready to push the 'STO' prefix out of the generator. But while we're at it, let's also make a 'RCL A':

```
03 RCL 22 (pushes out the 'STO', puts 'RCL' in the
          generator)
                    GTO .002           [02 "B▓"       ]
03 X<0?
                    JUMP .002          [02 *          ]
03 HMS -
```

The program is now:

```
01 STO 01
02 "BJ"
03 RCL A
04 6                          (4A-1)
05 STO A
06 6
07 HMS-
```

The '6's in lines 04 and 06 are the stand-alone equivalents of the '22' postfix bytes. Now try:

```
             SIZE 103
             PRGM (off)
12345
             GTO .005
             SST
             CLX                  [0.0000        ]
```

At this point, the number '12345' has disappeared from the display. To retrieve it, press 'GTO .003', SST. The number reappears, showing that it was stored in Register 'A'. To see where the number actually went, store '102' in R_{00}, then press 'VIEW IND 00'. As you might

have expected, 'STO A' is equivalent to 'STO 102'. If the numerical postfixes were continued past decimal 99, the 'A' postfix would be '102'. Synthetic programming thus allows us to extend direct data register access up to R_{111}(postfix 'J'), leading to the underlined postfixes for bytes 66-6F in the Byte Table.

But why stop at 111? There is another row of postfixes available; could we not use them to access up to Register 127? Notice first that postfixes 70, 71, 72, 73, and 74 are already used to access the stack registers T, Z, Y, X, and L, respectively But the remaining bytes beckon to us: they turn out to be the keys to 'Pandora's Box' of synthetic programming! Let's try a dramatic example: Using Routine 4A-1:

```
                    JUMP .002      [02 *        ]
03 STO 22
                    GTO .002       [02 "B▓"     ]
03 AVIEW
                    JUMP .002      [02 *        ]
03 HMS-
                    GTO .003       [03 STO d    ]
```

Now switch PRGM off; press 'SF 00, SF 01, SF 02, SF 03, SF 04, FIX 9, SF 28, GRAD, USER, CLX, ALPHA', which turns on various display annunciators. Now press 'SST' once. As the resulting display makes quite clear, the simple operation 'STO d', with zero in Register X, clears all 56 HP-41C flags in one fell swoop! The implication is obvious--the two-byte code '91 7E', or 'STO d', that we made with the byte jumper, allows us to store directly into a special register, which we shall call 'Register d', that contains the 56 flags.

The 'status registers' (Registers 000-00F) mentioned at the end of Section 2D are recorded by the card reader 'WSTS' operation. The status of all user and system flags is part of the information stored on the status card, so we infer that Register d is one of the 16 status registers. The observation that the stack registers, also accessed by postfixes in row 7 of the Byte Table, are status registers, leads us, in a bold leap of inspiration, to guess that all of the postfixes in row 7 refer to status registers--16 postfixes, 16 registers. It only remains to identify the roles of the 16 registers. From the 'WSTS' and 'WALL' operations, we would expect the status registers to contain, in addition, the alpha register, the address pointer, the subroutine returns, the current 'size', and the location of the summation registers.

As shown in the Byte Table, postfixes 75 through 7F display as 'M', 'N', 'O', 'P', 'Q', '⊢', 'a', 'b', 'c', 'd', and 'e', respectively. The second postfix values shown for bytes 75 through 7A are the postfixes shown with printer listing of the status register functions. Thus the line 'STO M' shown in the display would print as 'STO ['. The correspondence is also shown in Table 1-1.

In order to study the properties and uses of each of the status registers, we will synthesize program lines that allow us to view or change the contents of these registers. The function 'X<>' serves this purpose admirably, since it can act both as 'STO' and 'RCL'. Since storage into Registers a, b, and c can produce occasionally unpleasant results ('0, STO c' causes 'MEMORY LOST', as the most unpleasant example), we will limit ourselves to verbal descriptions of those registers for the time being. As an exercise, you should try to use the byte jumper to create Routine 4A-2 without guidance. Refer to the instructions in the next paragraph if you get lost. As we did in the creation of 'STO A' and 'RCL A', we will save keystrokes by using each successive 'X<>' prefix to push the previous one out of the generator. If you have cleared the assignment of 'X<>' that you made in Chapter 3, you should restore it now.

Start with Routine 3B-2. As usual, the choice of a temporary postfix for the 'X<>' prefixes is arbitrary (just don't use 'X<>29, 30, or 31', since the corresponding postfix bytes are prefixes themselves). The best choice is 'X<>00', since the '00' is hex '00', the null. All the leftover postfixes are 'invisible', and will be eliminated by packing. Now try:

```
                    JUMP .002      [02 *        ]
03 X<>00
                    GTO .002       [02 "B▓"     ]
03 CLD    (e)
                    JUMP .002      [02 *        ]
03 X<>00
```

```
                         GTO .002          [02 "B▩"     ]
03 AVIEW   (d)
                         JUMP .002         [02 *        ]
03 X<>00
                         GTO .002          [02 "B▩"     ]
03 SIGN    (⊢)
                         JUMP .002         [02 *        ]
03 X<>00
                         GTO .002          [02 "B▩"     ]
03 X≠Y?    (Q)
                         JUMP .002         [02 *        ]
03 X<>00
                         GTO .002          [02 "B▩"     ]
03 X=Y?    (P)
                         JUMP .002         [02 *        ]
03 X<>00
                         GTO .002          [02 "B▩"     ]
03 CLX     (O)
                         JUMP .002         [02 *        ]
03 X<>00
                         GTO .002          [02 "B▩"     ]
03 LAST X  (N)
                         JUMP .002         [02 *        ]
03 X<>00
                         GTO .002          [02 "B▩"     ]
03 RDN     (M)
                         JUMP .002         [02 *        ]
03 HMS-
```

Following this sequence, we are left with Routine 4A-2 (delete line '11 HMS-'):

```
01 STO 01     06 X<>P
02 "BJ"       07 X<>Q
03 X<>M       08 X<>⊢          (4A-2)
04 X<>N       09 X<>d
05 X<>O       10 X<>e
```

Now, if we want to execute an 'X<>M', for example, then with PRGM off we press 'GTO .003', 'SST'. In the remainder of this chapter, we will use the program lines in the above routine to explore the status registers. The practical programming applications of the properties we discover will be discussed in Chapter 6. Figure 4-1 is a diagram summarizing the use of the various parts of the status registers, in a format similar to that of Figure 2-5.

4B. THE ALPHA REGISTER

A HP-41C programmable memory register is 7 bytes long. The 'alpha register' appears to be an exception to this rule, since we can store up to 24 bytes (characters) in alpha. In fact, the alpha register consists of four status registers, Registers M, N, O, and P. Four registers gives us a total of 28 bytes, but only 24 of these can be displayed, or accessed with 'ASTO' and 'ARCL'.

To illustrate the structure of the alpha register, we can use Routine 4A-2. Start by keying in 24 characters into alpha, such as "ABCDEFGHIJKLMNOPQRSTUVWX". If Flag 26 is set, you will hear the warning tone upon entering the "X" to inform you that the alpha register is full. Now press 'GTO .003', 'CLX', 'ALPHA(on)', 'SST', to see "ABCDEFGHIJKLMNOPQ‾‾‾‾‾‾‾". The overline "‾" is the character corresponding to a null byte '00'. The 'CLX' filled Register X with nulls; the 'SST' executed an 'X<>M', moving the nulls to Register M, which is revealed to be the rightmost 7 bytes of the alpha register. If you switch the HP-41C out of ALPHA mode, then set 'FIX 9', you will see '-2.5354555 E-42' in X. The hex code for the character string "RSTUVWX" is '52 53 54 55 56 57 58'. These bytes are now in Register X, where the processor is trying valiantly to display a decimal number. There is a '5' in the mantissa sign digit, and a '7' for the exponent sign, resulting in the minus signs (see Section 5A). The mantissa digits are all normal decimal digits, so the mantissa shows as '2.5354555565',

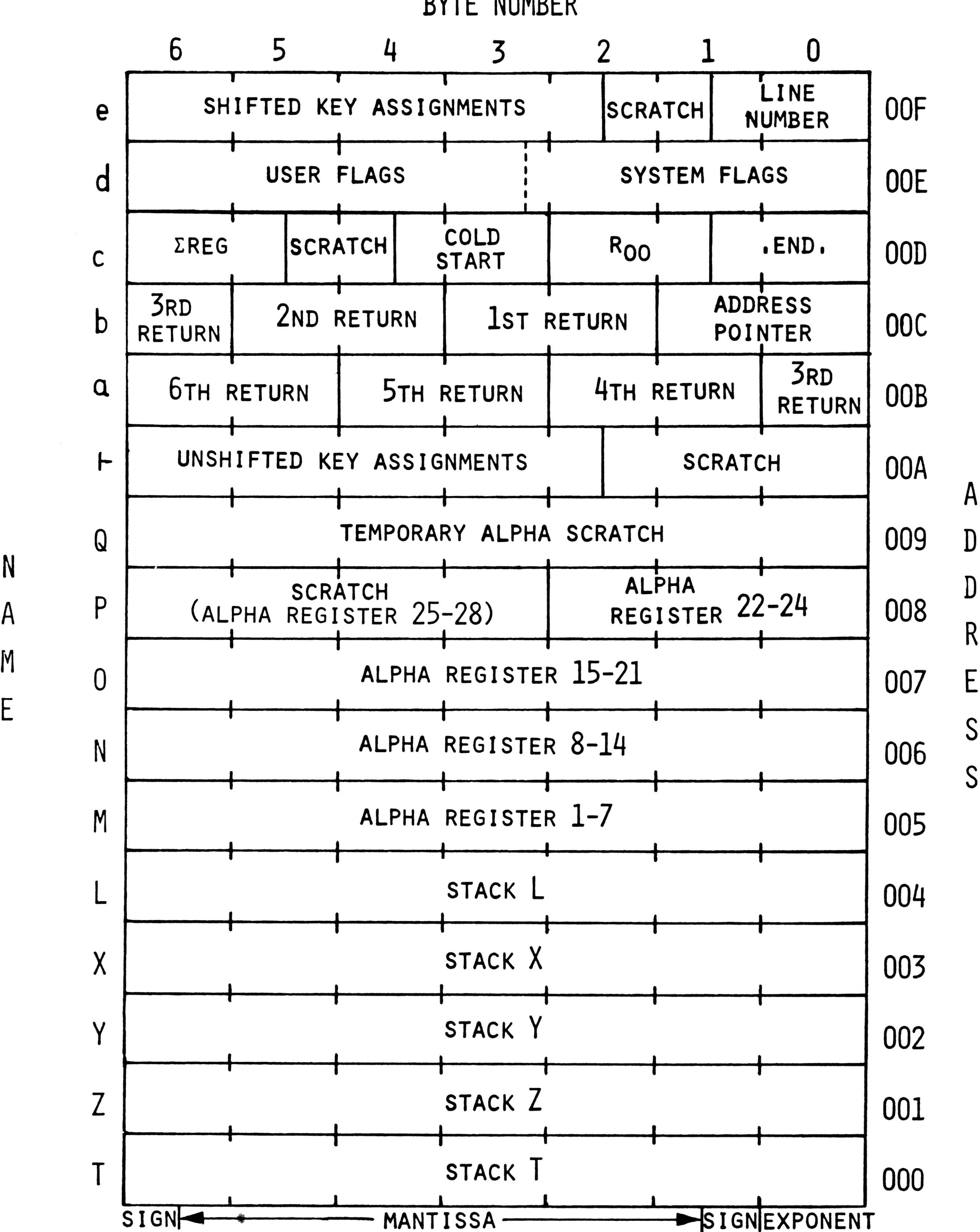


FIGURE 4-1. THE STATUS REGISTERS

the last two digits of which are suppressed to make room for the exponent. The byte code for a negative exponent '-xy' is the complement of the exponent, '100-xy'(decimal); in this case we see an exponent [100-58=42].

To continue your explorations, try 'ALPHA(on)', 'SST', to see "ABCDEFGHIJRSTUVWX-------". The 'SST' executed line '04 X<>N', so that now the original contents of Register M, "RSTUVWX", have been moved to Register N, the next 7 bytes of the alpha register. To complete the exercise, key 'ALPHA(off)', 'GTO .003', 'SST', 'ALPHA'; the display is "ABCDEFGHIJRSTUVWXKLMNOPQ" --the original contents of M and N are interchanged. All told, the original 24-character alpha string divided up like this:

```
                xxxxABC|DEFGHIJ|KLMNOPQ|RSTUVWX
     Register:     P       O       N       M
```

When a fresh character string is keyed into the alpha register, the first character keyed enters the last byte (the exponent byte) of Register M, address 0005. The next character also goes to 0005, pushing the previous character leftwards in the alpha register, i.e., to the next-to-last byte of Register M, byte 1005. Subsequent characters continue the process; when M is full, the next character entry will push the first into the last byte of Register N, byte 0006, and so on up into Registers O and P. When a character enters the third-to-last byte of Register P, 2008, the warning tone sounds. Appending further characters shoves the leading characters into the first four bytes of P(shown above as "xxxx"), where they vanish from the display.

Now press 'GTO .006', turn ALPHA on, and append four more characters "YZ=?" to the original 24. "ABCD" disappears, but surprisingly, it is still present in Register P. Press 'SST' once, to execute 'X<>P', then 'ALPHA(off)'. You will see '-1.4243444 E-53'. To translate this into characters, try 'GTO .003', 'ALPHA', 'CLA', 'SST'('X<>M'). The characters "ABCDEFG" reappear--these were the initial contents of Register P, which we moved into Register M via Register X. However, if we repeat the whole process starting with the keying in of the 28 characters, but, this time, switching out of ALPHA mode <u>before</u> the 'SST' that executes the X<>P, we end up with "▒▒CDEFG". The processor <u>occasionally</u> uses the first four bytes of Register P for 'scratch' purposes, wiping out part or all of the original contents. Evidently, pressing the 'ALPHA' key requires processor use of Register P. Studies by Charles Close have revealed that, during a program execution, only 'VIEW', 'AVIEW', and number entry program lines cause loss of the leading bytes of P. If these steps are avoided, we can use a full 28-byte ALPHA register for character string manipulations. As an example of the processor use of Register P, bytes 1, 2, and 3 are used to record the current size and summation registers location on a magnetic card with the 'WSTS' operation.

There are two important areas of application of direct access to Registers M, N, O, and P. First, as demonstrated in the above examples, we have obtained a new class of alpha string manipulations, which when added to the conventional 'ASTO', 'ARCL', 'APPEND', and 'ASHF', provides efficient, fast character sorting useful for displays, games, word sorting, etc. The second application is the use of the alpha register as an additional three (or four, if Register P is included) data registers, with the same capabilities as ordinary data registers, but with the advantage of fixed memory locations and non-normalizing recall(Section 5B). These applications will be explored in detail in Chapters 5 and 6.

4C. REGISTER Q

Register Q is primarily a scratch register for the processor. It is used so frequently that it is virtually useless as an additional data register. Of primary interest for synthetic programming is the use of Register Q for temporary storage of alpha strings that don't directly enter the alpha register. Such strings are obtained during execution of functions or programs that are 'spelled out' by the user, or during the entry of program text lines. For example, using Routine 4A-2, try 'XEQ' 'ALPHA' "GTO" 'ALPHA' '.007', 'SST', 'GTO .003', 'ALPHA', 'CLA', 'SST'. The "OTG" now in the alpha register is the reverse of the letters "G", "T", "O" that you used to spell out "GTO". This feature can be used to simplify the creation of non-keyable program text lines (Section 5I).

4D. THE FLAG REGISTER

We discovered at the beginning of this chapter that Register d 'contains' all 56 HP-41C user and system flags. When we recall that a register consists of exactly 56 bits, it is

obvious that each of the flags is just one of the bits of Register d. The 'first' (or leftmost, or highest, or most significant, depending on how you like to visualize a register) bit is Flag 00, the second Flag 01, and so forth to the 56th bit, Flag 55.

As a sample of the behavior of Register d, configure your HP-41C as follows: SF 04, SF 09, SF 17, SF 18, SF 26, USER (on), SF 28, FIX 9, RAD; all other user flags clear. Now, using Routine 4A-2, call up the contents of Register d by pressing: 'GTO .009', 'SST', 'ENTER', 'BST', 'SST', 'RDN', 'ALPHA', 'CLA', 'ARCL X'. This sequence allows us to view the contents of d without changing them. We use the 'ARCL X' so that we can view all 10 mantissa digits as well as the exponent. Recalling that the positive exponent and mantissa correspond to zeros in the sign digits, we conclude from the alpha display that the bytes of Register d are '08 40 60 38 09 90 10', which is, writing out all 56 bits (grouped by nybbles):

```
Flags:      4    9         17 18     26 27 28      36 39 40 43       51
            |    |          \ /       \ / /         \  \ / /          /
Bits:  0000 1000 0100 0000 0110 0000 0011 1000 0000 1001 1001 0000 0001 0000
```

Each '1' in the string corresponds to one of the flags we set. The first '1' from the left, in the second nybble, is Flag 04, for example. The next '1' is Flag 9, and so forth, over to the last '1', in the second nybble from the right, which is Flag 51, the 'SST' Flag, which was set momentarily because we used an 'SST' to execute the 'X<>d'.

To give yourself a taste of what can be done through use of the flag register, multiply the existing number '8.405038099 E10' in Register X by '1 E30'. Then execute 'GTO .009', 'SST'. No--you don't have a low battery, you just set Flag 49, the low battery flag. (To clear it, turn the HP-41C off, then on.) This is an example of the use of user-controlled bytes, i.e., the number you placed in Register X, to control system flags through exchanges with Register d. Now conceive of the reverse process--using explicit control over the user flags to create arbitrary bytes in Register d, whence they can be transferred to Register X and elsewhere with status register access functions (see Chapter 5). It was, in fact, the implementation of this concept that originally led to the development of serious synthetic programming.

4E. THE KEY ASSIGNMENT FLAGS

When a key is pressed USER mode, if that key is assigned to other than its default function, the processor must check the user global labels and the key assignment registers to discover what program or function the key is intended to execute. To save a lot of fruitless searches, the HP-41C keeps a set of 72 'key assignment flags', one for each key and shifted key (counting the imaginary key under the ENTER key). When a user key is pressed, the processor first checks the corresponding assignment flag. Only if the flag is set does the assignment search begin.

As in Register d, a memory bit is used for each assignment flag. Since 72 bits are too many for a single register, the assignment flags are divided between Register ⊢ and Register e. The 36 unshifted key flags are the first 36 bits of Register ⊢; the shifted key flags are similarly situated in Register e. Figure 4-2 shows the correspondence between bit number and key location.

To see one of these registers 'in action', we will use Register ⊢ as an illustration. The only key assignments we have made so far were made in Chapter 3, i.e., the assignment of the byte jumper to the 'Σ+ ' key, and the assignment of the 'X<>' function to the '+' key. If you have made additional assignments in the meantime, your results in the following will differ from what is shown here, so you might want to delete those extra assignments.

To view the contents of Register ⊢ using Routine 4A-2, press 'GTO .008', 'CLX', 'SST', 'FIX 7'. You should see '0.0000021'. The 6 zeros plus the positive sign show that the first [7x4=28] bits of the number (which is the original content of Register ⊢) are zeros. The '2' digit, which is 0010 binary, indicates that assignment bit 31 is set; the '1', binary 0001, comes from setting assignment bit 36. Referring to Figure 4-2, we see that those bits correspond to the '+' key and the 'Σ+ ' key respectively, which are just the keys we had assigned.

Inspection of the contents of Registers ⊢ and e thus provides a quick means of finding which USER keys are assigned, without requiring use of the printer. We used 'FIX 7' because we were only interested for this purpose in the first 9 nybbles of the register. This trick is not completely general in its application--if too many keys are assigned, the numbers derived from a 'RCL ⊢' or 'RCL e' may contain hex digits A-F, which can be hard to decipher in the display (see Section 5A). You must be sure to restore the original values to the key assignment registers; otherwise, you will lose the user key assignments, including the portion

of user memory used to encode the assignments. If you now press the '+' key in USER mode, the HP-41C will execute '+' rather than 'X<>', because the zero we stored into Register ⊢ with 'X<>⊢' cleared the key assignment flags. To recover, press ('LASTX', if you executed the '+') 'GTO .008', 'SST'. If you accidentally lose the contents of Register ⊢ or e while playing such games, execution of a card reader 'WSTS' followed by reading back the resulting status card will restore the original key assignments.

The last three nybbles of Register e are the storage location for the program line number (coded in hexadecimal). If the line number nybbles are '000', as when following a 'GTO .000', a manual 'RTN', or program execution that terminates with an 'END', the next program display shows '00 REG lmn'. When program execution halts at a position other than an 'END', the line number is set to 'FFF'. When the processor 'sees' that mythical line number, it knows that it must recompute the actual line number before showing the current program line upon the next activation of PRGM mode or 'SST'.

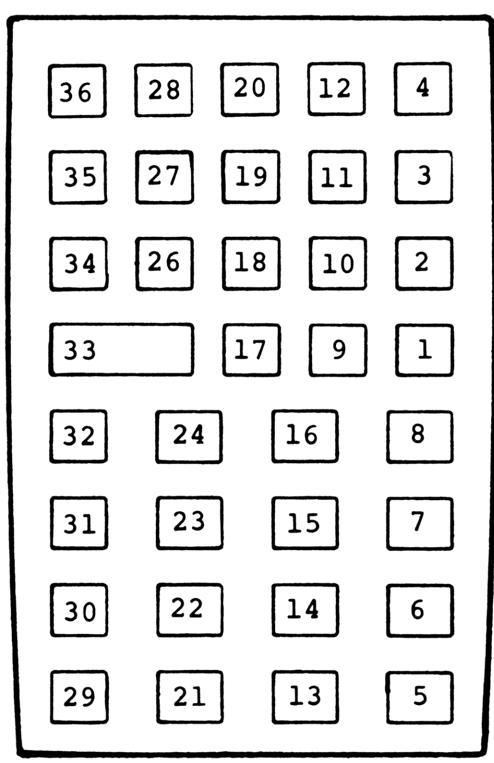


FIGURE 4-2. KEY ASSIGNMENT FLAG BITS

4F. THE ADDRESS POINTER AND THE RETURN STACK

In Chapter 2 we learned that the address pointer uses a 4-digit address consisting of the byte number plus three register number digits. The address pointer itself is the last four nybbles of Register b. That is, if the address pointer is positioned at byte 'n' of Register 'abc', then Register b will contain the number '0000000000nabc'. If a subroutine is called, the return address is recorded in the next four nybbles to the left of the address pointer, with the new current address, following the jump to the subroutine, entered into the pointer nybbles. As further subroutine calls occur, previous return addresses are pushed to the left. When Register b is full, the addresses continue on into Register a, so that there is sufficient room in Registers a and b for the current address plus six pending return addresses. At any time, the contents of Registers a and b look like this:

```
|yzab uvwx qrst mn|op ijkl efgh abcd|

|    Register a   |   Register b    |
```

where 'efgh' is the first pending return address, 'ijkl' the second, and so forth. When a 'RTN' or 'END' is executed, the 'stack' of return addresses shifts to the right, so that the first return address becomes the new pointer address, the second return address becomes the first, etc. The only complication in this nifty scheme is that the return addresses are written in a slightly different format than the pointer addresses. For example, if the return is to address '3160', the address would be stored in the return stack as '0760'. The '0760' is actually a compressed form of '3160'. If we write out all the bits of '3160' we get:

3160 = 0011 0001 0110 0000

For user program addresses (i.e. locations in the HP-41C or memory modules) the first, fifth, sixth, and seventh bits are always zero, since the first nybble only takes values up to 7 (0111) and the second is always 0000 or 0001. So in the return addresses the three 'used' bits of the first nybble are moved to the three 'unused' bits of the second. The '3160' becomes:

0760 = 0000 0111 0110 0000

This compression of the address code frees the first nybble to contain additional information. Specifically, if the first nybble is zero, the return is to a user program address; whereas if it is nonzero, the processor knows to return to an address contained in some peripheral device. For example, all addresses in the printer memory start with the nybble '0110'.

The most obvious application of knowledge of the structure of the return stack and of synthetic function access to the stack through functions such as 'STO b' or 'RCL a' is to allow the user to move the address pointer to arbitrary positions in memory, including to the key assignment registers or even into the status registers themselves. Also, using a 'STO b', we can move the pointer directly to any byte of a multi-byte program line for editing just as we did with the byte-jumper. For full control of such operations, we first need to develop a facility that allows us to generate arbitrary 7-byte hex codes, and to decipher such codes. This facility will be developed in Chapter 5.

4G. REGISTER c AND MEMORY PARTITIONING

The last unexplored section of user memory in the HP-41C is Register c, which is full of interesting nybbles and bytes. The 14 hexadecimal digits of Register c are laid out as follows:

stu|vw|169|mno|pqr

where the letters representing the digits are grouped as the digits are used by the processor. The letters are chosen to correspond to the memory partitioning shown in Figure 2-5. The first three digits, 'stu' are the absolute register address of the first of the six summation registers as specified by the function 'ΣREG'. This address is changed each time either 'ΣREG' or 'SIZE' is executed. When 'Σ+', 'Σ-', 'CLΣ', 'SDEV', or 'MEAN' is executed, the processor refers to digits 'stu' to find out which registers are currently designated as the summation registers.

The new two digits, 'vw' are used for scratch purposes by the printer. Digits 6, 7, and 8, shown with the explicit value '169', are interesting in a perverse sort of way, since they are the so-called 'cold-start constant'. At various times during routine operation, particularly when the HP-41C is turned on, the processor checks the value of the three digits and compares it with the fixed value '169'. If the digits are other than '169', the processor assumes that something drastic has occurred to the memory, and so takes the irreversible step of clearing the user memory and displaying 'MEMORY LOST'. This explains why '0, STO c' causes memory loss--that operation clears the '169'.

Digits 'mno' and 'pqr' are three-digit register numbers that correspond to the Registers 'mno' and 'pqr' shown in Figure 2-5. 'mno' is the absolute address of the lowest numbered data register, R_{00}. The processor refers to 'mno' for each use of a data register. Data register R_{ab} is the memory register with absolute address [mno+ab] (with 'ab' converted to hexadecimal).

Digits 'pqr' indicate the current location of the permanent '.END.'. Since the global label/END chain starts with the '.END.', a 'GTO (alpha)' or 'XEQ (alpha)' instruction starts the label search in Register 'pqr'.

Despite the risk of memory loss when dealing with Register c, there are many important applications of the manipulation of its contents. Foremost among these are control of the program/data memory dividing line, and the capability of storing data into program registers, as will be described in Chapter 5.

PROGRAMS FOR PROGRAMMING

The central theme of this chapter is to develop a set of HP-41C programs that allows us to create, decipher, store and recall arbitrary 7-byte hexadecimal codes, giving us great power for synthetic programmming. The 'programming' programs and techniques are intended to perform specific synthetic programming tasks, while acting as examples of the uses of synthetic functions. Even if you do not use every program from this chapter, you will find it instructive to study the techniques and programming philosophy embodied in the routines. It is assumed at this point that you have mastered the byte jumping techniques of Chapters 3 and 4 sufficiently to be able to 'key in' any two byte synthetic function, such as 'RCL M' or 'X<>d', whenever it is required in a program. As we did when creating Routine 4A-2, you will find it convenient, when byte jumping a series of functions, to start at the bottom of the program and work upwards, using each prefix entered into the byte jumper generator to 'eject' the previous one. After all the synthetic functions are keyed in, you can insert normal functions with ordinary keystrokes.

Three crucial synthetic programming techniques are featured in the programs:

1. Multiple use of flags. Direct access to Register d allows a set of user flags to be used for several purposes simultaneously, simply by saving a current flag status with 'RCL d' or 'X<>d', using the flags for a second purpose, then restoring the original status of all flags with a 'STO d' or 'RCL d'. This permits, for example, a program to employ different display or trigonometric statuses yet always return the HP-41C to the user's favorite flag status at the end of the program.
2. Use of flags as bits. User control and testing of Flags 0-29 allow programmable conversion of 30-bit binary numbers to and from decimal or octal. This technique is the indispensable key to creating and deciphering memory bytes and multiple-byte codes automatically.
3. Alpha string manipulations. The powerful synthetic alpha register access functions, used in this chapter's programs to assemble individual bytes into 7-byte codes, and vice-versa, are encountered again and again in practical synthetic programming, as we will see in Chapter 6.

The HP-41C is designed to process only program- or user-generated data that are normal decimal numbers. We might, therefore, anticipate that asking the processor to deal with numbers containing digits 'A' through 'F' will at least cause some unexpected behavior. Before exploring the promised programming programs, we should first learn how various 7-byte codes are 'interpreted' by the display, and also study how the codes are affected by register exchanges such as caused by 'STO' or 'RCL'.

5A. UNSEEMLY DISPLAYS

When presented with a sequence of bytes to be displayed (PRGM off) the processor must first decide whether to display the bytes as a number or as a string of alpha characters. If ALPHA mode is on, or if 'AVIEW' is executed, there is no choice: all bytes in the alpha register are displayed as characters. But if ALPHA is off, so that the display is showing a data register, the choice is determined by the value of the mantissa sign digit. We already know that if the sign digit is '1' (0000) or '9' (1001), the register contents are displayed as a positive or a negative number, respectively. The only other 'normal' sign digit is '1' (0001), for which the register is assumed to contain a six character alpha data string, such as might result from an 'ASTO' operation. Each of the last six bytes in the register is displayed as a character; the first byte is not shown. Any sign digit other than the normal '0', '1', or '9' will cause the register contents to display as a negative number, but to be treated as 'alpha data' in many arithmetic operations.

When the displayed register contains alpha data, as indicated by the sign digit '1', the display is simplified through suppression of null bytes. All null bytes, rather than just leading nulls as in alpha register displays, are not only blanked but are also 'eliminated' by moving the non-null characters into adjacent positions. For example, the alpha data coded '10 00 41 00 00 42 00', which would show as "A--B-" if viewed in the alpha register following an 'ARCL', would display simply as "AB" in Register X.

Number displays have the additional feature of user choice of the number of digits that will be shown, including appropriate rounding, using the display format functions 'FIX', 'ENG' and 'SCI'. We should remember that this choice is only a display feature--the number is still stored as a full seven bytes.

In many situations numbers containing any of digits 'A' through 'F' will display using only ordinary decimal characters '0' through '9'. Since A through F are greater than 9, each such digit found in the mantissa of a number 'carries' a '1' into the next digit to the left. For example, the bytes '01 03 B0 0F 00 00 00', which 'should' be the number '1.03B00F', will display in FIX 6 format as '1.041015'. The F digit shows as '15', using two positions. The B becomes an '11', but since there was already a 3 in the next digit, the combination '3B' displays as '41'.

An exception to this general rule occurs when the displayed number has a non-negative exponent, and is displayed in a format that shows all ten mantissa digits. These requirements can only be satisfied by 'FIX' format displays of numbers with exponents between 00 and 09. If we switch to 'FIX 9' format, the number '1.03B00F' will be displayed as '1.03;00?000'. The digits B and F are represented by single characters ";" and "?" respectively. These characters are found in row 3 of the Byte Table, just as are the decimal number characters "0" through "9". Similarly, digits 'C', 'D', and 'E' display as characters "<", "=", and ">", respectively. An 'A' digit is represented by the starburst character shown in the lower left corner of the table box for byte 3A. If a number is 'copied' into the alpha register using 'ARCL', the special characters are preserved, except that the starburst will 'change' to the alpha character ':'.

Numbers with an exponent zero as '1.03B00F' are displayed with ten mantissa digits only in FIX 9 format. If the number were '10.3B00F' ('1.03B00F E01'), the special display would result with FIX 8 as well as FIX 9. In general, we obtain the special display characters with formats FIX 'n' through FIX 9, where 'n' is nine minus the exponent. However, when the exponent itself contains any digits A-F, the exponent will display using the special characters (but not the mantissa, since 10 mantissa digits cannot be displayed with an exponent).

5B. REGISTER EXCHANGES AND NORMALIZATION

As we prepare to deal with data registers containing arbitrary 7-byte codes, let us introduce the following classification of codes: first, an 'alpha data string' is any 7-byte code for which the first nybble is '0001'. A 'number' is any code for which the first and twelfth nybbles are either '0000' or '1001', and the remaining twelve nybbles are any of the decimal digits '0' through '9'. Any other code will henceforth be called a 'Non-Normalized Number', or simply an 'NNN'.

That this classification is useful follows from consideration of the treatment of 7-byte codes by register exchange functions 'STO', 'RCL', 'X<>', and 'VIEW'. 'STO' is the most straightforward--'STO mn', where 'mn' refers to <u>any</u> register whether addressed directly or indirectly, exactly copies the content of Register <u>X</u> into the designated register. Unfortunately for synthetic programming, the other three exchange functions are not so benign. 'RCL pq', 'X<>pq', or 'VIEW pq', if 'pq' refers to a numbered data register, causes the content of R_{pq} to be 'normalized' before it is copied. (Exchanges between status registers do not cause normalization.) By 'normalized', we mean that NNN's are changed, either into ordinary decimal numbers (containing no heretical digits greater than 9) if the original mantissa sign digit was 0 or 9, or into alpha data with a sign digit of 1 if the original sign was other than 0 or 9. For example, an NNN coded with bytes '01 0C 00 D0 0E 00 FF', which displays as '1.1201301 E??', is normalized following 'STO 01', 'RCL 01' to the number '1.120130140 E-35'. The bytes have actually changed to '01 12 01 30 14 09 65'. The NNN '21 0C 42 34 7E 40 DD', which has an abnormal sign digit, displays as '-1.1242348 E=='. If normalized, it will change to the alpha data "μB4Σ@▩", i.e., bytes '11 0C 42 34 7E 40 DD'.

'ASTO' and 'ARCL' perform still different kinds of register exchanges. 'ASTO' takes the first six bytes in the alpha register, starting with the first non-null byte, adds a '10' alpha-identifier byte to the 'front' to make a total of seven bytes, then stores the resulting bytes in the addressed register. 'ARCL' reverses the process if the addressed register contains alpha data, dropping the '01' byte and appending the alpha characters to the right end of the existing string in the alpha register. Leading nulls in the alpha data are dropped; trailing or intermediate nulls are retained. For example, start with "ABC" in alpha. Then 'ARCL X', where Register X contains '10 44 45 46 47 48 49' ("DEFGHI") will result in "ABCDEFGHI". If Register Y contains '10 00 4A 00 00 4B 00' (displayed as "JK"), 'ARCL Y' yields the string "ABCDEFGHIJ--K-" in the alpha register. Finally, if the register addressed by 'ARCL' contains

a number or an NNN, the operation does not copy the bytes of the number into alpha, but rather changes each digit, as shown in a number display, into the corresponding row 3 character for storage in the alpha register. 'ARCL' also normalizes the contents of the addressed register.

5C. GETTING STARTED: "CODE"

The byte jumper is so far the only tool we have for creation of non-standard program lines and NNN's (from 'RCL M', etc.) However, the byte jumper is strictly a manual operation, and has some limitations in the byte codes it can generate (bytes from the lower half of the Byte Table are difficult to handle). We are now going to write a program called "CODE" which will automatically generate <u>any</u> 7-byte code specified by the user, placing the code into both Registers X and M. This program will, through use of an accompanying routine "REG" (which allows storage of the code into any register), enable generation of any program sequence, NNN, or non-standard alpha character string. Furthermore, we will be able to make 'synthetic key assignments' that assign any two-byte functions to user keys. This ability will finally place the synthetic functions on an equal footing with any normal HP-41C or peripheral function: namely, they will be able to be executed manually or inserted anywhere into a program with single keystrokes.

"CODE" is designed to satisfy several requirements. Upon execution, the program should halt and prompt the user for entry of an easy-to-key code that specifies a 14-digit hexadecimal number. Following entry, the program should run with no further user intervention, yielding the desired number coded with the proper bytes. Even though the flag register will be required for the byte creation, the program should restore the original flag status at the finish. Finally, for reasons which will become clear later (Section 6G), "CODE" should use no numbered data registers. This set of requirements is rather difficult to implement--the version of "CODE" described here is many times revised from the original (see 'HP-41C Black Box Programs', <u>PPC Calculator Journal</u>, V6 N8 P27).

```
01♦LBL "CODE"
02 "CODE=?"
03 AON
04 STOP
05 AOFF
06♦LBL "CO"
07 "⊢ABCDEFG"
08 .006
09 STO L
10♦LBL "HB"
11 1
12 RCL ]
13 X<> d
14 X<>Y
15 CF 00
16 CF 01
17 CF 02
18 FS?C 03
19 GTO 01
20 SF 04
21 FC?C 07
22 SF 07
23 FS? 07
24 CHS
25 FS? 06
26 CHS
27 SF 06
28 X<0?
29 CF 06
30 X<0?
31 SF 05
32♦LBL 01
33 ST* X
34 FS?C 04
35 SF 00
36 FS?C 05
37 SF 01
38 FS?C 06
39 SF 02
40 FS?C 07
41 SF 03
42 FS?C 11
43 GTO 01
44 SF 12
45 FC?C 15
46 SF 15
47 FS? 15
48 CHS
49 FS? 14
50 CHS
51 SF 14
52 X<0?
53 CF 14
54 X<0?
55 SF 13
56♦LBL 01
57 FS?C 12
58 SF 04
59 FS?C 13
60 SF 05
61 FS?C 14
62 SF 06
63 FS?C 15
64 SF 07
65 RDN
66 X<> d
67 RCL ]
68 RCL \
69 RCL [
70 STO \
71 R↑
72 STO [
73 "⊢*"
74 X<> \
75 R↑
76 STO ]
77 R↑
78 STO \
79 "⊢**"
80 RCL Z
81 STO [
82 ISG L
83 GTO "HB"
84 RCL [
85 CLA
86 STO [
87 AVIEW
88 TONE 9
89 END
```

"CODE"
191 BYTES

Instructions for use of "CODE":

1. XEQ "CODE".
2. At the prompt "CODE=?", key in 14 alpha characters to represent the desired code, using the characters "0" through "9" and "A" through "F" for the nybbles '0' through 'F',respectively.
3. R/S.
4. At the beep, the requested code will be in Registers X and M (shown with 'AVIEW'). an 'AVIEW'.

To test that your version of "CODE" is correct, try these examples:

Input		Output (AVIEW)
"41 42 43 44 45 46 47"	⟶	"ABCDEFG"
"00 01 28 29 00 7F 7E"	⟶	"ж()‾⊢Σ"

[The remainder of this section is a detailed discussion of the operation of "CODE", and may be skipped at a first reading.]

The task for "CODE" is to convert 14 characters, entered by the user following the halt at line 04, into the corresponding bytes. (The global labels "CO" and "HB" are for use by other programs calling portions of "CODE" as subroutines.) To minimize program length, this task is handled by a routine that converts one pair of input characters into one output byte. The routine is run 7 times, using the alpha register to append the successive output bytes together. The problem is complicated by the requirement that we use no data registers, leaving only the alpha register(s) and the stack for juggling the input code, the output code, the original contents of the flag register, and any arithmetic that might be required for the conversions.

The basic conversion routine is found in lines 10-64. To understand how it works, let's start by assuming that one pair of input characters has been moved into the first two bytes of Register d. For example, take the characters "49", which are to be changed into the single byte '49'. The initial characters are actually bytes '34 39'--what we must do is ignore the '3's and move the '4' into the first nybble, and the '9' into the second:

34 39 ⟶ 49 39

after which we will concentrate only on the first of the two bytes. The shifting of digits is done with ordinary flag operations--recall that in Register d, the first four bits are Flags 0-3, the next four are Flags 4-7, etc. The program lines to do the copying are as follows (to the right of each line is shown the effect of execution of the line on the first 16 bits of Register d):

Program Line	Flags 00-15
(initial value '34 39')	0011 0100 0011 1001
15 CF 00	"
16 CF 01	"
17 CF 02	0001 0100 0011 1001
18 FS?C 03	0000 0100 0011 1001
19 GTO 01	
32 LBL 01	
34 FS?C 04	"
35 SF 00	"
36 FS?C 05	0000 0000 0011 1001
37 SF 01	0100 0000 0011 1001
38 FS?C 06	"
39 SF 02	"
40 FS?C 07	"
41 SF 03	"
42 FS? 11	"
43 GTO 01	"
56 LBL 01	"
57 FS? 12	"
58 SF 04	0100 1000 0011 1001
59 FS? 13	"
60 SF 05	"
61 FS? 14	"

```
62 SF 06                  "
63 FS? 15                 "
64 SF 07              0100 1001 0011 1001 = 49 39
```

The simplicity of the preceding set of program lines arises from the circumstance that the second nybble of each character "0" through "9" is the same as the numerical equivalent of the character. Unfortunately, this is not true for characters "A" through "F", requiring a more complicated conversion process. The program must test each input character to determine whether it is 'greater' than "9" and thus needs extra processing. The testing is done for the first of the pair of input characters in line 18. The additional conversion is performed in lines 20-31 (and uses line 11). The second input character of the pair is tested in line 42; lines 33 and 44-55 provide the corresponding conversion.

The remainder of "CODE" is centered around the basic routine described so far: the program must place two input character bytes in the first two bytes of Register d, run the routine, then extract one output byte from Register d. Registers N and O are used to store the input characters; Register M contains the 'growing' output code. You may find it of interest to single-step through lines 66-81 to see how the leading byte from Register d is appended to the last character position in Register M, while simultaneously the user input code is moved left by <u>two</u> positions in Registers N and O. After seven iterations (Register L is used as a counter) M contains the final output bytes.

When the input characters are stored into Register d (line 13), numerous system flags are set or cleared (half the fun of "CODE" is watching the various annunciators blink on and off), including possibly Flag 52, the PRGM mode flag. If this flag is set during a running program, certain operations will cause the HP-41C to go berserk and start to program itself! (See Section 7B.) Included among these operations are ordinary number entry lines, so to avoid such a catastrophe, the entry of a '1' in line 11 precedes line '13 X<>d', necessitating the otherwise wasteful inclusion of line '14 X<>Y'.

5D. DIRECT ACCESS TO PROGRAM REGISTERS

The program "CODE" developed in the last section is a powerful synthetic programming tool, but so far there's not much we can do with it, beyond creating strings of non-keyable alpha characters, since the output codes can only be transferred into other data registers. But consider this: the division of the HP-41C memory into program and data registers is entirely controlled by one number--namely, the absolute address of R_{00}, stored in Register c. Changing that address is normally done only with the 'SIZE' function, which also shifts the contents of memory registers so that program stays program and data stays data. As ambitious programmers, we will not let this minor detail deter us--synthetic functions give us access to Register c, and now "CODE" allows us to create any bytes we might wish to store there. By placing the proper code into Register c, we can move the 'curtain' separating program and data registers anywhere we want without using 'SIZE', and thereby transform the contents of data registers into program lines, or vice-versa!

Furthermore, to maintain the proper spirit, we will write a program to perform the whole process of storing into program registers automatically. The general approach is this: we use "CODE" twice, once to create a temporary value to store into Register c that will specify as R_{00} the program register into which we wish to store; and again, to create the special code to be stored. Then we swap the new value for Register c with the original, using 'X<>c', execute 'STO 00' with the special code in X, then restore the original contents of Register c so that access to existing programs and data is preserved.

The specific value that we wish to store into Register c is '10 00 01 69 xy z1 00', where 'xyz' is the 3-digit absolute address of the program register we wish to access. The '169' is the 'coldstart' constant needed to prevent 'MEMORY LOST'. '100' is chosen for simplicity for both the ' REG' address and the '.END.' address--since this value for Register c is only temporary, it is not too important which values we use. Having the byte '10' starting the code is convenient because it makes the string 'alpha data', which can be stored and recalled without normalization. The only variable in the new Register c value are the digits 'xyz', so we can save execution time by only using "CODE" to create two bytes (in this case 'xy z1') rather than a full seven. The next program, 'REG', is an implementation of these ideas.

```
01♦LBL "RREG"        14 STO \              26 X<> c
02 SF 10             15 "⊢♦♦♦♦♦♦8♦×i"      27 RCL [
03 GTO 01            16 .001               28 X<> 00
04♦LBL "REG"         17 STO L              29 FS?C 10
05 CF 10             18 XEQ "HB"           30 STO 00
06♦LBL 01            19 "⊢♦"               31 X<>Y
07 "REG?"            20 RCL [              32 STO c
08 AON               21 STO 00             33 X<>Y
09 STOP              22 CLD                34 "REG-"
10 AOFF              23 FC? 10             35 ARCL 01
11 ASTO 01           24 XEQ "CODE"         36 AVIEW
12 "⊢1"              25 RCL 00             37 END
13 RCL [
```

"REG"

100 BYTES

SIZE 002

Instructions for "REG":

1. XEQ "REG". (If you only wish to recall the contents of a register, XEQ "RREG".)
2. At the prompt "REG?", enter three alpha characters to identify the absolute address of a register, then R/S.
3. At the prompt "CODE=?', enter 14 alpha characters to specify the code to be stored, then R/S.
4. The display "REG-abc" (abc is the register address) announces that program execution is complete.

Line 15 of "REG" is a synthetic program line, code 'FA 00 00 00 00 00 00 10 00 01 69'. It can be created by byte jumping as follows:

```
15 STO 07
16 STO 02
17 "⊢ABCDEFGHIJ"
                    JUMP .016       [16 X>Y?         ]
17 0
18 /
19 LBL 00
20 FRC
                    JUMP .016       [16 X>Y?         ]
                    SST, SST        [18 /            ]
                    DEL 001         [17 STO 02       ]
                    JUMP .017       [17 -            ]
                    DEL 005         [16 STO 02       ]
                    GTO .018        [18 X<=Y?        ]
                    DEL 005         [17 "⊢--------▩-天▩"  ]
                    GTO .015        [15 STO 07       ]
                    DEL 002         [14 STO N        ]
```

The label "RREG" is provided in case you want only to recall the contents of a register. Remember, however, that the recalled register will be normalized by the recall.

As an example of the use of "REG", execute 'SIZE 010' (if no memory modules are inserted --use 'SIZE 074' for one module, 'SIZE 138' for two, 'SIZE 202' for three, or 'SIZE 266' for four). Now use 'CAT 1' to place the pointer in the first program in memory, then press 'RTN' to put the pointer at the start of the program. The address of the first program register in this configuration is 'OF5'. In PRGM mode, key in seven 'ENTER' lines, which just fill Register OF5, pushing the already existing programs down in memory. To create the synthetic text line "######':

```
                    XEQ "REG"       ["REG?"          ]
"OF5"
                    R/S             ["CODE=?"        ]
"F6232323232323"
                    R/S             ["REG-OF5"       ]
```

GTO first program (use CAT 1)
GTO .001
PRGM (on) [01 "######"]

If you repeat the above sequence, replacing the "F6232323232323" with "0191759F0A9676", the following program lines will be placed at the top of program memory:

```
01 LBL 00
02 STO M
03 TONE 0
04 ISG N
```

(The 'TONE 0' in line 03 is actually 'TONE 10'--see Section 7A.)

Any synthetic program lines can be created in this manner. To make a line requiring more than 7 bytes, we simply use "REG" twice (or even three times), storing 7 bytes of the code at a time into adjacent registers.

5E. SYNTHETIC KEY ASSIGNMENTS

A powerful application of "REG" is for the generation of 'synthetic key assignments', i.e., assigning synthetic two-byte functions to user keys so that they can be executed manually or entered directly into programs. To achieve the assignments, we simply use "REG" to store special codes into the key assignment registers. Each such use of "REG" can assign two user keys. The procedure is best illustrated by use of an example: we will assign the functions 'RCL M' and 'STO M' to the 'TAN' (25) key and the 'ATAN' (-25) key respectively.

1. Clear all existing key assignments except the byte-jumper. (This drastic step is not usually necessary, but you should do it this time.) 'PACK', then assign any function to any key.

2. Assign any HP-41C function to the two keys to be assigned (for the example, assign the 'TAN' key and the 'ATAN' key). This puts a 'dummy' code into Register 0C0, the lowest assignment register, and also sets the proper key assignment flags in Registers ⊢ and e.

3. Determine the required code to overwrite the dummy code in Register 0C0. This code follows the format described in Section 2E. The byte codes for the functions to be assigned can be found in the Byte Table; the key assignment bytes are shown in Figure 2-6. In this case the code is:

```
F0       starts a key assignment register
90 75    'RCL M'
42       assignment of 'TAN' key
91 75    'STO M'
4A       assignment of 'ATAN' key
```

4. Use "REG" to store the assignment register code into Register 0C0. For this example, we enter "0C0" at the "REG?" prompt, and "F090754291754A" at the "CODE=?" prompt. After "REG-0C0" displays, we will find that pressing the 'TAN' key executes 'RCL M'; the 'ATAN' key executes 'STO M'.

This synthetic assignment technique is by no means limited to synthetic functions. The whole point of key assignments is to allow the user to replace frequently-used multi-key sequences with single keystrokes. Ordinarily, a single instruction like 'ST+IND X' takes five keystrokes; the best we can do normally is reduce it to four by assigning 'ST+' to a user key. But with synthetic assignment techniques, there is no reason not to assign the entire function 'ST+IND X' to a key (the function code in this case would be '92 F3'.)

In addition, synthetic assignments are not limited to HP-41C functions; we can also assign peripheral functions to keys. We simply obtain the 'XROM' code for the desired function from the appropriate peripheral manual, convert the 'XROM' code to hexadecimal using the conversion described in Section 2B, then store the resulting byte codes into an assignment register. This method allows a user to enter programs containing peripheral functions into memory even when the peripheral is unavailable. Also, we can enter 'nonprogrammable' functions, such as 'LIST' or 'WALL', into programs. The codes for nonprogrammable card reader and printer functions are shown in Table 5-1.

When a normal HP-41C function is assigned to a key, only one of the two function bytes in one half of an assignment register is used to identify the function. The first of the two

TABLE 5-1

Non Programmable Peripheral Functions

Function	XROM Number	Byte Code	Execution?
CARD READER	30,00	A7 80	Clears digits flags
VER	30,05	A7 85	Normal
WALL	30,06	A7 86	Normal
WPRV	30,09	A7 89	Normal
-PRINTER-	29,00	A7 40	Crash
LIST	29,07	A7 47	Lists from next line
PRP	29,13	A7 4D	NONEXISTENT

bytes is always '04' (LBL 03). If the first byte is other than '04', the processor knows that the assignment corresponds to a two-byte peripheral function. If the peripheral is absent pressing the assigned key produces the appropriate 'XROM' code. In the case of two-byte synthetic assignments, the first function byte differs from '04', again with the result that pressing and holding the key displays an 'XROM' code. In our example of assigning 'RCL M' to the 'TAN' key, pressing the TAN key gives a display 'XROM 01,53'. Releasing the key prior to the 'NULL' display causes 'RCL M' to execute.

Synthetic XROM codes can be deciphered in the same manner as normal XROM's, as described in Section 2B. For example, to determine the XROM code corresponding to 'RCL M', we write out the hex code '90 75' for 'RCL M' in binary, then group the last 12 bits into two 6-bit numbers, which we convert to decimal:

```
hex:          9     0     7    5
binary:    1001 |0000 01|11 0101|
decimal:             01      53
```

Hence 'RCL M' = 'XROM 01,53'. Similarly, 'STO M' = 'XROM 05,53'.

To facilitate the construction of a large set of key assignments, it is convenient to automate the synthetic assignment procedure more than is possible through use of "REG". Use of the 'Key Assignment' program, "KA", listed next, eliminates preliminary manual key assignments, use of Figure 2-6 to determine key codes, and any necessity for worrying about the current contents of the key assignment registers. "KA" will refuse to overwrite an existing assignment unless the user manually clears the assignment when prompted.

Three additional utility routines are listed with "KA". "KP" 'packs' the key assignment registers. Although two key assignments use only one key assignment register, clearing two key assignments will not recover one register for further use unless both assignments were recorded in the same register. It is thus possible to have a number of half-filled assignment registers. "KP" compresses the code in the assignment registers, leaving at most one half-filled register when there are an odd number of active key assignments. The half-filled register will be Register 0C0, so that a new assignment will fill that register.

The "Clear Assignments" program "CA" automatically clears all function and user program assignments. However, "CA" does not clear the key assignment bytes in global labels, so that if a program is assigned to a key that was first assigned to another program lower in memory then cleared by "CA", that first assignment will be reactivated.

"EF", or 'End Finder', is a routine used by "KA", "CA", and "KP" to locate the '.END.'. If 'xyz' is the number of registers separating Register 0C0 and the '.END.' register, then "EF" places the number '0.xyz' into Register X, to control iterations involving recalling each of the assignment registers in turn. Line 31 of "EF" places the code 'F0 00 00 00 0C 0C 0C' into Register M. This NNN is stored by the other programs into Register c so that Register 0C0 temporarily becomes data register R_{00}. When this value is in Register c, a program halt will cause 'MEMORY LOST'. To avoid this, do not halt the execution of "KA", "CA" or "KP" while they are running. Furthermore, always be sure that none of the four routines is the first program file in memory. There must be at least one 'END' preceding the routines so that their backward-jumping 'GTO's will function properly. When a GTO causes a jump to a label higher in memory, the label search proceeds downwards to the 'END' of the file, then resumes at the top until the label is found. With the program/data 'curtain' moved to Register 0C0, the search will jump into data memory unless there is an 'END' in a program file higher

in memory.

```
01♦LBL "KA"
02 XEQ "EF"
03 CF 00
04 CF 01
05 SF 03
06 RCL [
07 ENTER↑
08 X<> c
09 X<>Y
10 X<> 00
11♦LBL 00
12 ""
13 X<> [
14 "⊦♦"
15 X<> \
16 X=0?
17 GTO 01
18 "⊦-------"
19 X<> \
20 ISG Z
21 GTO 03
22 STO 00
23 RDN
24 STO c
25 GTO 15
26♦LBL 03
27 X<> IND Z
28 GTO 00
29♦LBL 01
30 RDN
31 STO c
32 CLA
33♦LBL 05
34 CF 22
35 ASTO L
36 "PRE↑POST↑KEY"
37 TONE 8
38 PROMPT
39 CLA
40 ARCL L
41 FC? 22
42 GTO 06
43 X<> Z
44 XEQ 02
45 XEQ 02
46 36
47 STO a
48 RDN
49 ENTER↑
50 X<0?
51 SF 00
52 ABS
53 .1
54 *
55 LASTX
56 -
57 INT
58 ST- a
59 LASTX
60 FRC
61 80
62 *
63 ST- a
64 2
65 *
66 +
67 8
68 FC? 00
69 CLX
70 +
71 X<> a
72 24
73 X<=Y?
74 SF 01
75 FC? 01
76 CLX
77 -
78 FS? 00
79 RCL e
80 FC? 00
81 RCL '
82 ASTO L
83 STO [
84 FS? 01
85 "⊦♦♦♦"
86 X<> [
87 X<> d
88 FC? IND Y
89 GTO 01
90 X<> d
91 RCL \
92 FIX 0
93 "CLEAR "
94 ARCL T
95 TONE 4
96 PROMPT
97 STO \
98 RDN
99 X<> d
100♦LBL 01
101 SF IND Y
102 X<> d
103 STO [
104 "⊦♦♦♦♦"
105 FC?C 01
106 "⊦♦♦♦"
107 RCL \
108 FS? 00
109 STO e
110 FC?C 00
111 STO '
112 CLA
113 ARCL L
114 RCL a
115 XEQ 02
116 FS?C 03
117 GTO 05
118♦LBL 06
119 ASTO X
120 ""
121 FC?C 22
122 "⊦♦♦♦"
123 ARCL X
124 RCL [
125 "μμμ"
126 RCL [
127 X<> c
128 X<>Y
129 STO 00
130 X<>Y
131 X<> c
132 "DONE"
133 BEEP
134 PROMPT
135♦LBL 02
136 X=0?
137 "⊦♦♦♦♦"
138 OCT
139  E3
140 /
141 10
142 +
143 X<> d
144 FS?C 19
145 SF 20
146 FS?C 18
147 SF 19
148 FS?C 17
149 SF 18
150 FS?C 15
151 SF 17
152 FS?C 14
153 SF 16
154 CF 07
155 SF 03
156 X<> d
157 ARCL X
158 "⊦ABC"
159 0
160 X<> \
161 STO [
162 RDN
163 RDN
164 END
```

"KA"

334 Bytes

```
01♦LBL "EF"
02 RCL c
03 STO [
04 "⊦♦♦♦♦x̄"
05 RCL [
06 X<> d
07 CF 00
08 CF 01
09 CF 02
10 CF 03
11 FS?C 07
12 SF 05
13 FS?C 08
14 SF 06
15 FS?C 09
16 SF 07
17 FS?C 10
18 SF 09
19 FS?C 11
20 SF 10
21 FS?C 12
22 SF 11
23 X<> d
24 DEC
25 193
26 -
27 X<0?
28 GTO 15
29 1 E3
30 /
31 "♦♦♦μμμ"
32 END
```

"EF"

79 Bytes

Instructions for "KA". (Should not be first program in memory.) To make one or two assignments:

1. XEQ "KA". The display will show 'NONEXISTENT' if no registers are available for assignments. DO NOT ATTEMPT TO HALT EXECUTION; IF STARTED, MAKE AT LEAST ONE ASSIGNMENT.
2. At the prompt "PRE POST KEY", key in three numbers:

'prefix', 'ENTER↑', 'postfix', 'ENTER↑', 'keycode', R/S

The 'prefix' and 'postfix' are the decimal values of the prefix and postfix bytes of the function to be assigned, which can be determined from the Byte Table. For example,

the function 'STO N' ('91 76') would be entered as prefix '145', postfix '118'. The 'keycode' is the same row-column code that is displayed during ordinary assignments. Thus the '1/X' key has keycode '12', the 'FS?' key has code '-54', etc. The 'CHS', 'EEX', and [←] keys (and their shifted counterparts) should be entered with keycodes '43', '44', and '45' (negative for shifted keys), respectively, as if the 'ENTER↑' key covers an imaginary key '42'.
3. If the key designated in Step 2 is already assigned, TONE 4 will sound, and the display will shown "CLEAR nm", where 'nm' is the keycode. The user should manually clear the assignment, then press 'R/S' to continue.
4. Step 2 will repeat automatically for the second assignment. If only one assignment is desired, press 'R/S' at the prompt. At the completion of "KA", the BEEP will sound, and "DONE" will be displayed. For further assignments, return to Step 1.

```
01♦LBL "CA"
02 0
03 STO e
04 STO '
05 XEQ "EF"
06 X<> [
07 X<> c
08♦LBL 00
09 0
10 ENTER↑
11 X<> IND [
12 X=Y?
13 GTO 01
14 RCL Z
15 ISG [
16 GTO 00
17♦LBL 01
18 RCL Z
19 STO c
20 END
```

"CA"

41 BYTES

Instructions for "CA":
1. XEQ "CA". If no function assignments are present, "NONEXISTENT" will be displayed. DO NOT HALT EXECUTION.

```
01♦LBL "KP"
02 XEQ "EF"
03 STO Y
04 RCL [
05 X<> c
06 ENTER↑
07 CLA
08 CF 03
09♦LBL 14
10 FS?C 03
11 GTO 07
12 CLX
13 X<> IND Z
14 SF 25
15 X=0?
16 FS?C 25
17 GTO 07
18 ASTO L
19 CF 01
20 STO [
21 ASHF
22 X<> [
23 X=0?
24 SF 01
25 CLX
26 "⊢♦"
27 STO \
28 "⊢♦♦"
29 X<> \
30 "⊢♦"
31 X<> \
32 X=0?
33 GTO 01
34 X<> \
35 ASTO [
36 "⊢♦"
37 STO \
38 "⊢♦"
39 X<> \
40♦LBL 01
41 STO \
42 FC?C 01
43 "⊢♦♦♦"
44 X<> \
45 CLA
46 STO [
47♦LBL 09
48 ASTO X
49 CLA
50 ARCL L
51 ARCL X
52 SF 03
53 ISG Z
54 CF 03
55 GTO 05
56♦LBL 07
57 X<> [
58 XEQ 00
59 X<> 00
60 SIGN
61 X≠0?
62 GTO 03
63 CLX
64 LASTX
65 XEQ 00
66 STO IND T
67♦LBL 03
68 RDN
69 STO c
70 TONE 9
71 RTN
72♦LBL 05
73 X<> \
74 STO \
75 X=0?
76 GTO 14
77 ASTO X
78 ASHF
79 ASTO L
80 ""
81 ARCL X
82 CLX
83 X<> [
84 STO IND T
85 ARCL L
86 ISG T
87 GTO 14
88 RTN
89♦LBL 00
90 ""
91 X<> [
92 "⊢↑"
93 X<> \
94 "⊢↑↑↑↑↑↑"
95 X<> \
96 END
```

"KP"

190 BYTES

Instructions for "KP":
1. XEQ "KP". "NONEXISTENT" will display if the key assignment registers are empty. DO NOT HALT EXECUTION.

Certain lines in the key assignment programs require careful byte-jumping. Step-by-step procedures for keying the lines into program are listed next.

Lines 12 and 120 in "KA", and lines 80 and 90 in "KP" all have the byte code 'F1 F0'. To make one of these, e.g., "KA" line 12, use this procedure:

```
12 STO 01
13 "BJ"
                    JUMP            [13 *            ]
14 GTO 07
15 STO IND T
                    PACK            [14 STO IND T    ]
                    JUMP .013       [13 *            ]
14 "A"
                    GTO .014        [14 -            ]
                    DEL 002         [13 "B▩"         ]
                    PACK
                    JUMP            [13 *            ]
14 X<>Y
                    GTO .015        [15 HMS-         ]
                    DEL 001         [14 "▩"          ]
                    GTO .012        [12 STO 01       ]
                    DEL 002         [11 LBL 00       ]
```

Line 125 in "KA" is code 'F3 0C 0C 0C' :

```
125 STO 01
126 "BJ"
                    JUMP            [126 *           ]
127 "ABC"
                    GTO .127        [127 -           ]
                    DEL 004         [126 "B▩"        ]
127 LBL 11
128 LBL 11
129 LBL 11
                    JUMP .126       [126 *           ]
127 X<>Y
                    GTO .125        [125 STO 01      ]
                    DEL 002         [124 RCL M       ]
```

The code for "EF" line 04 is 'F6 7F 00 00 00 00 02':

```
04 STO 02
05 "⊢ABCDE"
                    JUMP            [05 -            ]
                    SST 3 times     [08 X<Y?         ]
09 LBL 01
                    JUMP .005       [05 -            ]
                    DEL 004         [04 STO 02       ]
                    DEL 001         [03 STO M        ]
                    GTO .005        [05 X>Y?         ]
                    DEL 001         [04 "⊢----▩"     ]
```

Finally, line 31 of "EF", 'F7 F0 00 00 00 0C 0C 0C':

```
31 STO 01
32 "BJ"
                    JUMP            [32 *            ]
33 GTO 10
34 STO IND T
                    PACK            [33 STO IND T    ]
                    JUMP .032       [32 *            ]
33 "ABCDEFG"
                    GTO .033        [33 -            ]
```

```
                        DEL 008          [32 "B■"          ]
                        PACK
                        GTO .033         [33 ""            ]
34 +
35 +
36 +
37 LBL 11
38 LBL 11
39 LBL 11
                        GTO .034         [34 +             ]
                        DEL 003          [33 ""            ]
                        JUMP .032        [32 *             ]
33 X<>Y
                        GTO .031         [31 STO 01        ]
                        DEL 002          [30 /             ]
                        GTO .032         [32 HMS-          ]
                        DEL 001          [31 "■---µµµ"     ]
```

As an example of the use of "KA", use it to assign the functions 'RCL b' (prefix 144/postfix 124) and 'STO b' (145/124) to keys. These functions are allow us to move the address pointer to usually inaccessible locations in memory. Try this (follow exactly; keying in extra lines while the pointer is in the status registers will cause 'MEMORY LOST'.)

```
                        XEQ "CODE"       [CODE=?           ]
"00000000000006"
                        R/S              ["ʈ"              ]
                        ALPHA            ["ʈ"              ]
"ABCDEFGHIJ"
                        ALPHA            [0,000,000.000 ]
                        STO b (use the assigned key)
                        PRGM, SST        [01 X<Y?          ]
                        DEL 003          [00 REG ---       ]
01 RCL 08
02 STO 15
03 RCL 09
                        PRGM (off)
                        ALPHA            [ABC(?)GHIJ       ]
```

You actually edited the alpha register--the 'STO b' sent the address pointer to address 0006, the last byte of Register N. The 'DEL 003' wiped out the first three bytes of Register M, which you then replaced with the characters "(?)" by inserting the corresponding program lines. Similarly, if you change the characters in the alpha register, you will see new program lines in PRGM mode.

5F. CREATION OF SYNTHETIC PROGRAM LINES

Synthetic program lines may be grouped into four general types: (1) two-byte synthetic 'functions', usually a combination of a status register postfix with a normal prefix; (2) synthetic text lines, containing at least one non-keyable character; (3) other non-standard multi-byte lines, principally global labels, 'GTO's and 'XEQ's, where the label name contains non-keyable characters; (4) 'En' lines, where a normal line '1 En', entering a power of ten such as '1 E3', is shortened to 'En' by removing the superfluous '1' byte. Of these four, types 1 and 2 are most common. Type 3 lines, given the abundance of normally available global labels, are primarily just curiosities which will interest only advanced program tinkerers. Generation of type 4 lines will appeal to the purist to whom a single wasted byte is offensive. We will discuss in detail methods of making types 1 and 2, and learn in passing how to make type 4 lines. Type 3 lines will be given little attention; in general they can be created using the same methods as for type 2.

We have studied three approaches to the task of creating synthetic program lines. The most powerful of these is use of a custom byte preparing routine, "CODE", to create arbitrary

byte sequences that can be stored into program memory using "REG". This 'programmed programming' method is unlimited in its application--with it, we can create all four types of synthetic program lines, with any combinations of bytes. The price of this power is the program 'overhead' of the nearly 300 bytes required for "CODE" and "REG". Furthermore, we need a means of determining the address of the register where the new lines are to be stored, requiring yet another program (see Section 5J).

The second program line-generating technique is to use "KA" to assign synthetic functions to keys so that they can be entered into programs at will. This method is limited to type 1 lines. Since "KA" is such a long program, it is best used to create several assignments in one session, followed by its removal from memory.

The availability of certain special key assignments, which depend upon quirks in HP-41C operation, provides a third approach to synthetic line generation. The prototype of such assignments is the byte jumper, which we used to start the whole synthetic programming process. As we shall see in the next section, the use of key-assigned 'RCL e' and 'STO e' results in an important improvement of the byte jumper process. Sections 5H and 5I describe two new special assignments for exotic editing, the 'text-enabler' and the 'Q-loader'. The text-enabler allows us to convert arbitrary existing program lines into text lines, and vice-versa. The Q-loader is used with "CODE" to store arbitrary 7-byte codes into program as text lines. The byte jumper and the text enabler require very little program overhead, but are limited somewhat in the byte combinations they can produce separately (used together, the byte-jumper and the text-enabler can make any byte combinations except those containing bytes E4 through EF). Any combination of seven or fewer bytes can be made using the Q-loader with "CODE".

From the above considerations, a recommended approach to synthetic programming is to use a 'standard synthetic programming keyboard', such as that shown in Figure 5-1, with key assignments for the most frequently used synthetic functions. When the need arises for other synthetic program lines, use the byte jumper and/or the text enabler. When these are insufficient, load in "CODE" and use the Q-loader. Finally, if necessary, use "REG" for any strange byte combinations that defy the other methods.

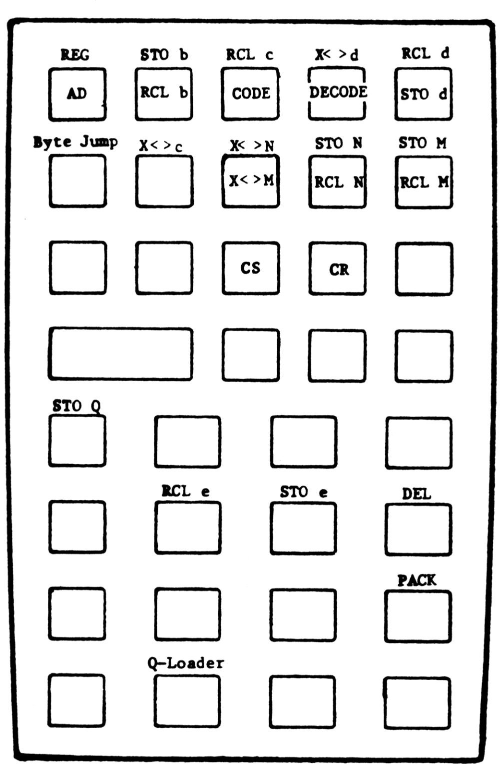


FIGURE 5-1. A SYNTHETIC PROGRAMMING KEYBOARD

A study of the various programs in this book will reveal that the following synthetic functions are used frequently enough to justify permanent key assignments: STO M, RCL M, X<>M, STO N, RCL N, and X<>N, for ready access to the alpha registers; STO b and RCL b to move the program pointer around; X<>c and RCL c for manipulating data register addresses; STO d, RCL d, and X<>d, for flag register access; STO e and RCL e for 'enhanced byte jumping' (Section 5G); STO Q and the Q-loader (see Section 5I); the byte jumper. If memory space permits, it is convenient to have "CODE", "REG", "CS" and "CR" (Section 5K), and "AD" (and even "DECODE"--see Section 5J) in memory and assigned to user keys. Figure 5-1 shows a convenient set of user keys assigned to make a 'synthetic programming keyboard'. Note that all 'STO' and most 'X<>' functions are assigned to shifted keys to reduce the risk of accidental storage into a sensitive register.

"KA" can be used to make the function key assignments required for the user keyboard shown in Figure 5-1. Table 5-2 lists the prefixes, postfixes and keycodes required, as well as the 'XROM' code that will display when one of the assigned keys is pressed and held.

TABLE 5-2

"KA" Entries for the Keyboard of Figure 5-1

Function	Prefix	Postfix	Keycode	XROM Code
STO b	145	124	-12	05,60
RCL b	144	124	12	01,60
RCL c	144	125	-13	01,61
X<>d	206	126	-14	57,62
STO d	145	126	-15	05,62
RCL d	144	126	15	01,62
X<>c	206	125	-22	57,61
X<>N	206	118	-23	57,54
X<>M	206	117	23	57,53
RCL N	144	118	24	01,54
STO N	145	118	-24	05,54
RCL M	144	117	25	01,53
STO M	145	117	-25	05,53
STO Q	145	121	-51	05,57
RCL e	144	127	-62	01,63
STO e	145	127	-63	05,63
Q-Loader	4	25	-82	"OD"
Byte Jumper	241	65	-21	05,01

The following sequence will also assign the functions from Table 5-2:

1. Assign any HP-41C function to the following keys:

1/X	y^x	X=Y?
10^x	x^2	ASIN
LN	e^x	ACOS
%	TAN	ATAN
SIN	CLΣ	P-R
COS	BEEP	π

2. Execute "REG" nine times, entering the codes as listed, at the corresponding prompts:

RUN	"REG?" Entry	"CODE=?" Entry
1	OCO	F091763A91754A
2	OC1	FOF1410ACE762A
3	OC2	FOCE7E39CE7522
4	OC3	F0917C19907C11
5	OC4	FOCE7D1A907D29
6	OC5	F0917E49907E41
7	OC6	F0907542907632
8	OC7	F091790D041920
9	OC8	F0907F1E917F2E

5G. ENHANCED BYTE JUMPING (eJUMP)

The principal defect of byte jumper editing is its inability to alter directly the second byte of a multi-byte program line. As a result, we have to resort to use of the 'generator', a dummy text line used to 'hide' prefix bytes so that postfixes can be inserted into program then attached to the prefix when it is 'pushed' out of the generator. Besides requiring a double use of the byte jumper, this method leaves a lot of leftover bytes that must be deleted, including the generator itself.

This inability to alter 'second bytes' arises from the fact that bytes are normally inserted in program following the last byte of the currently displayed line. But as was first pointed out by Roger Hill, this rule is not applied if the current program line number is '00'. In that case, inserted bytes enter program immediately after the current address pointer byte. 'Enhanced byte jumping' takes advantange of this feature to eliminate the need for the generator line.

The easiest method of setting the line number to zero is to press 'RTN', although this returns the pointer to the top of the current program file. But the simple sequence (PRGM off) 'RTN', 'RCL e', 'GTO .lmn', 'STO e', changes the line number of <u>any</u> program line from its normal value 'lmn' to '00'. Register e contains the line number--at any point, restoring to Register e the value it contained following a 'RTN', resets an arbitrary line number to zero. Switching to PRGM mode following this sequence always results in the display '00 REG lmn'. You can also achieve the same result with 'GTO .lmn', 'RCL b', 'RTN', 'STO b'.

As we did for the original byte jumper, it is convenient for simplification of programming instructions to introduce a new instruction 'eJUMP .lmn', which instructs the user to make the following keystrokes:

'eJUMP .lmn' means	1. PRGM off		1. PRGM off
	2. RTN		2. GTO .lmn
	3. RCL e		3. Press byte jumper
	4. GTO .lmn	OR	4. RCL b
	5. Press byte jumper		5. RTN
	6. STO e		6. STO b
	7. PRGM on		7. PRGM on

For an extended program editing session, the first method is preferred. If steps 2 and 3 are performed at the beginning of the session, then as long as the Register e contents are left undisturbed in Register X (and as long as no shifted key assignments are changed), these two steps can be omitted from the sequence.

To illustrate the use of enhanced byte jumping, let's create a 'RCL M'.

```
01 STO 01 (controller)
02 RCL 99 (any data register number greater than 15)

                      eJUMP .002     [00 REG lmn     ]
                      DEL 001*       [00 REG lmn     ]
01 RDN
                      GTO .001       [01 STO 01      ]
                      DEL 001
                      SST            [01 RCL M       ]
```

*In this case, DEL 001 is not equivalent to [←]. You can substitute 'SST', [←].

Two-byte functions, other than 'STO', 'RCL', and 'LBL' (which have one-byte forms for 'STO 00', 'RCL 00, and 'LBL 00'), are easiest to edit if they are initially keyed with '00' as their postfixes, for then the 'DEL 001' following the 'eJUMP' is not necessary:

```
01 STO 01
02 ISG 00
                      eJUMP .002     [00 REG lmn     ]
01 LASTX
                      GTO .001       [01 STO 01      ]
                      DEL 001
                      SST            [01 ISG N       ]
```

This method is not limited to two-byte functions--in the following, we create the 9-character text line found as line 90 of the 'Hangman' program in Section 6C. The code is 'F9 40 40 40 40 40 40 43 4C 5F'.

```
90 STO 01
91 "ABCDEFGHI"
                      eJUMP .091     [00 REG lmn     ]
                      DEL 009        [00 REG lmn     ]
01 +
02 +
03 +
04 +
05 +
06 +
07 /
08 %
09 DEC
                      GTO .090       [01 STO 01      ]
                      DEL 001
                      SST            ["@@@@@@CL_"    ]
```

5H. THE TEXT ENABLER

Creation of program text lines is an interesting process. When the user presses the initial character key, in PRGM-ALPHA mode, the processor writes an 'F1' text byte, followed by the character byte, into memory. For each subsequent character added to the string, the processor must update the text byte as well as adding the new character byte. The information required for the processor to keep track of this operation is recorded in Register Q during the text line entry. The first nybble of Q is the current number of characters in the string; the last four nybbles record the address of the last byte entered. Flag 45, the 'Data Entry Flag', is set during the line entry. Once Flag 45 is cleared by any keystrokes that terminate the entry of characters, it is normally not possible to resume adding characters to the line. Thus there should be no way to alter characters in an existing text line except by deleting the entire line and starting over.

But again, synthetic programming leads us into dark and uncharted territory. We can turn on any flag we want (see Section 6F) by storing the appropriate NNN into Register d. In particular, if we use 'STO d' to set simultaneously Flags 45 (Data Entry), 48 (ALPHA) and 52 (PRGM), we can actually add characters to an existing program text line. Furthermore, we will find that we can change any sequence of program bytes into a text line, and vice versa!

But a word of caution. These are 'deep waters', so that the instructions below must be carried out exactly, to avoid HP-41C 'crashes' that require battery pack removal for recovery. Furthermore, not all HP-41C's will behave exactly alike. So be prepared to experiment.

The act of storing the magic NNN into Register d will be referred to as the 'Text Enabler', and the NNN itself will be called the 'TEN' (Text Enabler Number). A suitable 'TEN' is any NNN that has '0' as its first nybble and '488' as its last three nybbles. The '488' comes from the three bits of Register d that correspond to Flags 45, 48, and 52. Since the storage into Register d affects all 56 flags, we might as well choose the unspecified nybbles of the 'TEN' to produce some 'nice' display format. The NNN '00 00 02 3C 04 84 88' used in the examples below acts as a 'TEN', while setting Flags 26 (audio), 27 (USER), 28 and 29, FIX 4 format, and DEG mode.

Use "CODE" to generate the 'TEN':

```
                      XEQ "CODE"     [CODE=?         ]
"0000023C048488"
                      R/S            [▩∠X▩▩          ]
                      ASTO 00
```

The 'ASTO 00' saves the 'TEN' for future use. If you wish to recall the 'TEN', however, use

'CLA', 'ARCL 00', 'RCL M', rather than 'RCL 00', to eliminate the alpha-identifier byte that the 'ASTO 00' adds to the NNN.

Now execute 'GTO .." to start a new program file. Switch to PRGM mode, and key in the text line "ABC". If you switch 'ALPHA' off following entry of the "C", the line is 'terminated'. But if you next turn 'PRGM' off, and press 'STO d' (with the 'TEN' in Register X), you will see the line '01 "ABC" in PRGM mode, with 'ALPHA' on; pressing any character key adds that character to the text line (plus the "_" to prompt for further characters). Pressing the correction key deletes characters from the string, including the original "ABC" if desired. However, this trick only works if the contents of Register Q are undisturbed since the original string entry was terminated. In general, returning to a text line by means of the text enabler after some intervening instructions have been executed will not enable further character addition but will most likely cause a crash. The processor needs the correct information in Register Q to continue with text line building, otherwise it gets hopelessly lost.

So we resort to an even stranger process. Rather than try to add to an existing text line from the end of the line, we deliberately set the first nybble of Register Q to zero by storing the 'TEN' into Q immediately before we store it into Register d. Remarkably, with the 'TEN' in Q, the processor will make text lines out of bytes that are already present in memory. If the starting byte doesn't happen to be an 'Fn' the processor simply replaces it with an 'Fn', changing 'n' whenever characters are added to or deleted from the text line.

All of this is best explained with examples. One more 'shorthand' notation is required:

'TE .lmn' means

1. PRGM off
2. Place the 'TEN' in Register X (e.g., CLA, ARCL 00, RCL M)
3. GTO .lmn
4. STO Q
5. STO d

Now clear the existing program, and key in:

01 "ABC"
02 X<Y?
03 X>Y?
04 X<=Y?

	PACK	
	TE .001	[01 "ABC"]
Press "A" key		[01 "A_"]
Press "A" key		[01 "AB_"]
Press "A" key		[01 "ABC_"]
Press "A" key		[01 "ABCD_"]
Press "A" key		[01 "ABCDE_"]
Press "A" key		[01 "ABCDEF_"]
	SST	[.END.]

Each pressing of the "A" key took an existing byte from program and incorporated it into the text line. Any 'character key' would have acted the same way as the "A" key, i.e., any key other than 'SST', 'BST', 'SHIFT', 'R/S', and the correction key. If we had continued beyond the six "A" keystrokes, the '.END.' itself would have been absorbed into the text. However, some HP-41C's will not continue beyond a 7-character text line; pressing an eighth key causes a crash. Unfortunately, the only way to determine which type of HP-41C you have is to try to make an eight character text line with the text enabler.

The correction key works in a unique manner with the text enabler. If it is pressed anytime after the text line has been started with at least one character, the rightmost character will be 'deleted' from the text line but remains in memory as a stand-alone line (or prefix). At present, you have the line '01 "ABCDEF"' in memory. Try:

	TE .001	[01 "ABCDEF"]
Press "A" six times		[01 "ABCDEF_"]
	←	[01 "ABCDE_"]
	SST	[02 X<=Y?]

As you see, the "F" byte was not deleted, only rejected from the text line to resume its role as line '02 X<=Y?'. If you had not pressed the 'SST' key, further use of the correction key would have removed the "E", "D", "C", and "B" from the line, leaving their bytes in memory. But one more deletion, with '01 "A_"' displayed, deletes the "A" and the 'F1' text byte from memory entirely.

If the correction key is the first key pressed following the 'STO d' of the text enabler, the first byte of the displayed line is replaced with an 'FF', and the next 15 bytes of program are incorporated into a text line. Subsequent presses of the correction key 'back up' the process, restoring each successive rightmost character to its original role and shrinking the text line.

The text enabler makes it quite simple to generate text lines containing non-keyable characters. Each desired character is entered into program as a one-byte program line. The string should be preceded by one 'dummy' one-byte line, which will be converted to the text byte. Once we are happy with the sequence of characters, we simply use the text enabler to transform the one-byte lines into a text line, like adding beads to a string:

```
01 X<>Y (dummy)
02 LBL 11
03 X<=Y?
04 RCL 08
05 E↑X-1
06 RCL 09
                         TE .001
Press 5 character keys
                         ALPHA           [01 "µF(X)"      ]
```

If you want a line of more than 7 characters but your HP-41C won't cooperate, you can press the correction key following the text enabler, then use it to delete characters until you are left with the desired line.

The text enabler can be used to generate synthetic functions, by using it to move an existing prefix into a text line, editing in a new postfix, then using the text enabler again to eject the prefix. This procedure is usually more complicated for two-byte functions than use of the enhanced byte jumper. But we shall see in the next section that a combination of the text enabler and the Q-loader provides a neat way of making synthetic lines directly from NNN's in Register X.

Making Type 4 program lines is a pleasant exercise with the text enabler:

```
01 1 E25
                         TE .001         [01 1 E25        ]
Press one character key                  [01 "▓"          ]
Press correction key, SST                [01  E25         ]
```

This works so cleanly because the null automatically inserted in front of the '1' byte acts as the 'dummy' byte to be turned into an 'F1', then deleted along with the '1'. If you want to delete the '1' from an already existing line, in a packed program file, you must edit in a dummy byte first, then PACK prior to using the text enabler.

5I. THE Q-LOADER

In Section 4C it was shown how Register Q is used for the temporary storage of alpha strings that are not stored either in program or in the alpha register. Through another quirk in HP-41C operation, we can exploit this behavior to assist in the creation of text lines up to seven characters in length. We need key assignments for 'STO Q' and the 'Q-loader' (prefix 4, postfix 25) that we made in Section 5F.

The 'Q-loader' is at first glance just the assignment to a user key of the number '9', i.e., byte '19'. Actually, the assignment of any byte from '10' to '1C' will work as a Q-loader, but the '19' assignment is easy to use, and gives an easily recognizable display "OD" when pressed and held. If pressed with PRGM off, the Q-loader enters a '9' into Register X. But in PRGM mode, the Q-loader enters two program lines: the first is just a '9' entry line, but the second is a text line, containing whatever characters were present in Register Q. To illustrate, execute 'PACK' (PRGM on) by spelling out 'XEQ "PACK"', then press the Q-loader. You will see the '9' line, followed by a text line "PACK". The letters "P-A-C-K" were placed

in Register Q by the 'XEQ "PACK"', then transferred to the text line by the Q-loader.

By using 'STO Q', we are not limited to sequences that we can 'spell out'--any NNN can be placed in Q. For complete versatility, we can use "CODE" to make the NNN. Hence, to make a text line of up to seven arbitrary bytes:

1. Use "CODE" to create a string of seven bytes corresponding to the characters you want in the text line, entered in reverse order. If you want only 'n' characters, where 'n' is less than seven, then the first [7 - n] bytes entered for "CODE" should be nulls (00).
2. 'GTO' the program line preceding the point where you want to insert the new text line.
3. With PRGM off, press 'STO Q'.
4. Switch PRGM on, and press the Q-loader key. Delete the resulting '9' line, then SST to see the new text line.

As an example, let us use the Q-loader to make line 81 from the 'Hangman' program in Section 6C. The code for this line is 'F5 60 06 04 05 01':

```
                          XEQ "CODE"      [CODE=?         ]
"00000105040660"
                          R/S             ["ҲҲҲ7ᴛ"        ]
                          GTO .000*
                          STO Q
                          PRGM (on)
                          Q-loader        [01 9           ]
                          DEL 001
                          SST             [01 "ᴛ7ҲҲҲ"     ]
```

*In the actual construction of the 'Hangman' program, this would be 'GTO .080'.

The Q-loader inserts its text line anywhere in a program, without any consideration of the register boundaries or addresses required for the use of "REG". By using the text enabler to transform Q-loader generated text lines into separate program lines, we can make arbitrary synthetic program lines of up to seven bytes in length, or several shorter lines simultaneously. Let us make the sequence '01 "(#)"', '02 ASTO M':

```
                          XEQ "CODE"      [CODE=?         ]
"0000759A292328"
                          R/S             ["▩▩)#("        ]
                          GTO ..
                          STO Q
                          PRGM on
                          Q-loader        [01 9           ]
                          DEL 001
                          PACK
                          PRGM off
                          TE .001
Press three character keys.               [01 "(#)"       ]
                          SST             [02 ASTO M      ]
```

If we want none of the seven characters in the text line to remain as text characters, we could use the text enabler to remove the text byte from the string, just as we eliminated the '1' from '1 En' lines in Section 5-H.

There are other types of Q-loader assignments available, corresponding to the other uses of program text strings. The following assignments have been developed by Tom Cadwallader:

Byte Code	"KA" Prefix/Postfix	Loads Q Contents Into:
04 1C	4/28	GTO (alpha)
04 1D	4/29	XEQ (alpha)
CD 00	205/00	LBL (alpha)

One minor annoyance in the use of the Q-loader is the requirement for encoding the byte sequences in reverse order, which is easy to forget. However, if the desired byte sequence is 6 bytes or fewer, the following routine will reverse the sequence automatically:

```
01 LBL "REV"
02 ASTO M
03 SF 25
04 GTO IND M
05 RCL Q
06 STO M
07 END
```

"REV"

20 BYTES

"REV" is designed to reverse a string of up to 6 characters in the alpha register. To reverse a string in Register X, execute a 'STO M' prior to 'XEQ REV'. Notice that "REV" will not work properly if there is a global label anywhere with the same name as the alpha string, but that is a rare occurrence. Also, "REV" will fail <u>if the printer is attached</u>.

5J. BACKTALK FROM THE HP-41C

At this point, we can 'tell' the HP-41C anything we want, by using "CODE" to translate user-readable characters into HP-41C internal byte codes. But for a complete dialogue with the machine, we also need a program to take existing codes and decipher them into readable characters. The program "DECODE", listed next, exactly reverses the operation of "CODE", taking an arbitrary 7-byte code in Register X and translating it into 14 characters in the alpha register. As an added touch, the output characters are grouped into pairs (bytes) by using colons as separators. The basic operation of "DECODE" is quite similar to that of "CODE". "DECODE" serves as an illustration of the use of Register P as a full 7-byte extension of the alpha register, as described in Section 4B.

```
01◆LBL "DECODE"
02 CLA
03 .006
04 STO L
05 X<>Y
06 GTO 14
07◆LBL 13
08 STO ↑
09 "⊦:"
10 RCL ↑
11◆LBL 14
12 ENTER↑
13 X<> d
14 CF 12
15 CF 13
16 CF 14
17 CF 15
18 FS?C 07
19 SF 15
20 FS?C 06
21 SF 14
22 FS?C 05
23 SF 13
24 FS? 04
25 SF 12
26 CF 08
27 CF 09
28 SF 10
29 SF 11
30 FS?C 04
31 XEQ 01
32 FS? 03
33 SF 07
34 FS? 02
35 SF 06
36 FS?C 01
37 SF 05
38 FS?C 00
39 SF 04
40 SF 02
41 SF 03
42 FS? 04
43 XEQ 02
44 FIX 5
45 ARCL L
46 X<> d
47 STO [
48 "⊦12"
49 X<> \
50 STO [
51 RCL ]
52 STO \
53 RCL ↑
54 STO ]
55 R↑
56 ISG L
57 GTO 13
58 0
59 STO ↑
60 TONE 9
61 AVIEW
62 RTN
63◆LBL 01
64 FS? 13
65 CF 10
66 FS? 14
67 CF 10
68 FS? 10
69 RTN
70 CF 12
71 CF 11
72 SF 09
73 FC?C 15
74 SF 15
75 FC? 15
76 RTN
77 FC?C 14
78 SF 14
79 FC? 14
80 RTN
81 CF 13
82 RTN
83◆LBL 02
84 FS? 05
85 CF 02
86 FS? 06
87 CF 02
88 FS? 02
89 RTN
90 CF 04
91 CF 03
92 SF 01
93 FC?C 07
94 SF 07
95 FC? 07
96 RTN
97 FC?C 06
98 SF 06
99 FC? 06
100 RTN
101 CF 05
102 RTN
103 END
```

"DECODE"

205 BYTES

Instructions for "DECODE":

1. Place the code to be deciphered in Register X.
2. XEQ "DECODE".
3. 'Output characters' are placed in alpha, shown with AVIEW.

Examples:

ALPHA "ABCDEFG" ALPHA, RCL M, XEQ "DECODE" ⟶ "41:42:43:44:45:46:47"
-1.234567891 E-56, XEQ "DECODE" ⟶ "91:23:45:67:89:19:44"

"DECODE" is designed to handle full 7-byte codes, which is a bit of 'overkill' for one particular application, namely, for determining the current program pointer address. If we have 'RCL b' assigned to a key, then 'RCL b', 'XEQ "DECODE"' will certainly decipher Register b for us. Usually, however, we are not interested in the first 5 bytes of Register b (subroutine return addresses). "AD" (for 'Address') is a quick and dirty routine that will decode a two-byte address, sacrificing the power and elegance of "DECODE" for execution speed.

"AD" is 'quick because it is so short, but 'dirty' because it uses the display properties of 'FIX 9' for rapid conversion of hex codes into characters. Notice that since the output is viewed in the alpha register, a digit 'A' will be represented by the colon ":" rather than the 'starburst'. Refer to Section 5A.

"AD" leaves the original address in Register X, so that a 'STO b' executed following "AD" will return the pointer to that address. This feature requires the otherwise superfluous line '15 STOP', to ensure that a switch to PRGM mode after the 'STO b' will give the correct line number (if the 'END' terminated execution of "AD" we would end up with line number '00').

```
01♦LBL "AD"        09 "  "
02 STO [           10 X<> [
03 "├♦♦♦"          11 STO \
04 RCL d           12 ASTO L
05 FIX 9           13 RDN
06 ARCL [          14 VIEW L
07 STO d           15 STOP
08 X<> [           16 END
```

"AD"

39 BYTES

Instructions for "AD":

1. Press RCL b (PRGM off).
2. XEQ "AD"
3. Output is four alpha characters (shown with AVIEW) corresponding to the four digit pointer address obtained with the 'RCL b'. The hexadecimal digits greater than '9' are represented as follows: 'A' = ":", 'B' = ";", 'C' = "<", 'D' = "=", 'E' = ">", and 'F' = "?".
4. To return to the original byte (where the 'RCL b' was executed), press 'STO b'.

5K. CODE STORAGE

A pair of short routines will round out our 'library' of special 'programming programs'. There are many occasions when it is desirable to store an NNN into a data register for future recall, but the normalization of NNN's during any recall from a numbered data register is a major obstacle.

One means of recalling an NNN without normalization is to use the byte jumper to transfer the NNN to the alpha register. For example, suppose the NNN of interest is in R_{00}, which we determine to be Register 123 by decoding Register c. Then we store '1 E7' into Register 124, i.e., R_{01}. Next we use "CODE" to make the 'address' '00 00 00 00 00 01 24', followed by 'STO b'. Pressing the byte jumper key will then copy the NNN in R_{00} into Register M. The '1 E7' in R_{01} placed the byte '07' at address '0124', providing for a byte jump of seven bytes.

This method is rather clumsy, and can only be used manually. For automatic storage and recall, we can use the routines "CS" and "CR". "CS" takes an NNN and breaks it into two pieces, each of which is converted to alpha data for ordinary storage. This requires two data registers to store the entire NNN code. "CR" reverses the process, recalling the two alpha data strings and reassembling them into a single 7-byte NNN. For convenience, the routines are designed to execute manually just like ordinary 'STO' and 'RCL': Assign "CS" and "CR" to user keys, then execute either by pressing the appropriate key followed (during the subsequent pause, which starts almost immediately) by the desired data register number. Each routine uses the designated data register plus the next higher-numbered register.

"CS"

40 BYTES

SIZE 002

```
01♦LBL "CS"
02 CLA
03 STO [
04 STO L
05 RDN
06 PSE
07 X<> L
08 "⊢♦♦"
09 .9
10 ST+ L
11 X<> \
12 ASHF
13 ASTO IND L
14 ISG L
15 CLA
16 STO [
17 ASTO IND L
18 CLA
19 RDN
20 END
```

```
01♦LBL "CR"
02 PSE
03 STO L
04 CLA
05 ARCL IND L
06 "⊢♦♦"
07 ISG L
08 CLD
09 CLX
10 RCL IND L
11 STO \
12 "⊢♦♦♦♦♦"
13 X<> \
14 CLA
15 END
```

"CR"

37 BYTES

SIZE 002

Instructions for "CS":

1. Manual use: XEQ "CS"; during the subsequent pause, enter a data register number 'mn' .The NNN will be stored in R_{mn} and R_{mn+1}. The contents of Registers X, Y, and Z are preserved.
2. Subroutine use: the calling program should have the NNN in X, and 'mn' in Y. Following execution, the contents of Registers T and Z will 'drop' to Z and Y, respectively.

Instructions for "CR":

1. Manual use: XEQ "CR"; during the subsequent pause, enter a data register number 'mn'. The NNN in R_{mn} and R_{mn+1} will be placed in Register X, replacing 'mn' and, in effect, 'raising' the stack contents present prior to execution of "CR"
2. Subroutine use: the calling program should place 'mn' in X prior to calling "CR". Following execution, the recalled NNN will replace 'mn' in X.

CHAPTER 6

APPLICATIONS

This chapter is intended to serve as a 'standard applications handbook' for synthetic programming. Included are numerous HP-41C routines, which, like the programs of Chapter 5, illustrate the creative use of synthetic functions as well as having powerful practical applications. The purpose and justification of synthetic programming is embodied in these routines. First, the use of synthetic functions enables the HP-41C to perform various important operations faster and with less program memory usage than is possible with the standard functions alone. Examples are the 'SIZE-Finder' (Section 6B) and the alpha string manipulations (Section 6C). Second, synthetic programming provides a new class of operations that cannot be carried out at all using only standard functions. Such operations include alpha character identification and comparison (Section 6D) and direct access to 'Application Pac' programs (Section 6H).

The set of routines described in this chapter by no means constitutes a complete list of the uses of synthetic functions--no list of programs can ever exhaust the capability of the HP-41C, especially as enhanced with synthetic functions. The development of the techniques and applications of synthetic programming is an ongoing process. (As an example, the discovery of the 'text enabler' described in Section 5H came from a typographical error in a preliminary draft of this book!) When you have completed studying the material in this book, you will be ready to use synthetic functions routinely in your own programs, as readily and with little more effort than you would use standard HP-41C functions. You might even discover a few new 'tricks of the trade' yourself. In this regard, an appropriate slogan is 'take nothing for granted'. If you get an idea, try it to see if it works, no matter how outlandish it might seem. It took months of widespread synthetic programming before anybody noticed, for example, that Register P acts as a full 7-byte continuation of the alpha register. Since the display shows a maximum of 24 characters, it was assumed that characters lost from the left side of the alpha register were gone forever. Yet, there they were, hiding in Register P.

6A. GETTING TO THE .END.

More often than not, a program under development is the last program file in memory, i.e., the file containing the '.END.'. If the address pointer is moved to some other file, there are only two ways to return it to that last file: use 'GTO' and spell out a global label within the program, or use 'CAT 1', running to the end of the catalog. If the new program does not have a global label, the first method is eliminated. If there are several memory modules and many programs in the HP-41C, the second method can be annoyingly slow. The program "EN" adds a third method, which you will find to be a great convenience during many editing sessions, particularly when you are programming using the routines of Chapter 5.

```
01♦LBL "EN"        10 SF 03
02 RCL c           11 X<> d
03 STO [           12 CLA
04 "├♦♦♦♦♦"        13 STO [
05 X<> [           14 "├♦♦"
06 X<> d           15 X<> \
07 CF 00           16 STO b
08 CF 01           17 END
09 SF 02
```

"EN"

45 BYTES

Instructions for "EN":

1. XEQ "EN".
2. At completion, the program pointer will be at the top of the program file containing the '.END.'

The '.END.' is situated in bytes 2, 1, and 0 of the register identified by the address recorded in the last three nybbles of Register c. "EN" takes that address 'lmn' (line 02), shifts it into the first two bytes of Register d (lines 03-06), and makes the code '3lmn' by clearing Flags 0 & 1, and setting Flags 1 & 2 (lines 07-10). That code is next shifted to the last two bytes of Register N (lines 11-15). When the code is finally transferred to

Register b in line 16, the address pointer immediately jumps to the byte immediately preceding the .END. The program continues to run, so that the .END. itself is executed, which stops execution with the pointer at the top of the file.

6B. SIZE-FINDING AND OTHER TRICKS

An elegant demonstration of how synthetic functions allow improved HP-41C performance is found in the following 'SIZE-Finder' routine, written by Keith Jarett (PPC Calculator Journal, V7 N5 P57):

```
01♦LBL "S"        14 FS?C 13        27 CHS
02 "AB"           15 SF 11          28 64
03 RCL c          16 FS?C 14        29 MOD
04 X<> [          17 SF 13          30 SF 25
05 STO \          18 FS?C 15        31♦LBL 14
06 ASHF           19 SF 14          32 VIEW IND X
07 "├♦♦♦"         20 FS?C 16        33 FC? 25
08 X<> [          21 SF 15          34 RTN
09 X<> d          22 X<> d          35 64
10 FS?C 11        23 1 E3           36 +
11 SF 09          24 *              37 GTO 14
12 FS?C 12        25 INT            38 END
13 SF 10          26 DEC
```

"S"

75 BYTES

Instructions for "S":

1. XEQ "S".
2. At completion, the current 'size' will be displayed in Register X.

There is, unfortunately, no direct way to determine the current program/data 'size' in the HP-41C. No 'top of memory' address is maintained in memory, since it changes with each insertion or removal of a memory module. The only way of determining the size is to try to access successively higher numbered data registers until a 'NONEXISTENT' message indicates that the last existing data register has been passed. This method can be automated in a program, taking advantage of the Error Ignore Flag 25, but if the number of data registers is large, the process can be very slow. Even the cleverest of such programs takes at least four seconds to run. The routine "S" takes a maximum of 1.5 seconds. The improvement arises from a partial decoding of Register c, which provides a starting value for the size that needs only to be increased by 64 times the number of memory modules present. Only a maximum of four registers must be tested to determine the number of modules.

The 'heart' of the size finder routine is found in lines 09-26, which constitute a 3-digit hexadecimal-to-decimal conversion routine developed by Roger Hill. The three hex digits of interest are digits 9, 10 and 11 of Register c--the absolute address of R_{00}. Lines 01-09 of "S" place those digits in Register d as the third, fourth, and fifth digits, i.e., as Flags 8-19.

Consider a typical R_{00} address, say '12A', which in decimal is [256+32+10 = 298]; use the 'OCT' function to find that 298_{10} equals 452_8. Let's write out both of the numbers '12A' and '452' as they are coded in the HP-41C:

Hexadecimal	12A = 0001 0010 1010	binary
Octal	452 = 0100 0101 0010	binary

Notice that the two numbers have the same number of bits with the value '1'. The difference between the two representations is that the first bit of each octal digit is always '0', since octal digits have a maximum value of 7 (0111). To convert the hexadecimal bit pattern to octal, we only have to shift the values of certain bits 'leftward' to 'make room' for the extra '0' bits. Here's a second example, with arrows showing the shifting of the bits from the hexadecimal pattern to octal:

```
Hexadecimal 1BC = 0001 1011 1100
                    ↙↙↙↓↓↙ ↓↓↓
Octal       674 = 0110 0111 0100
```

This shifting of bits is easily accomplished though explicit user flag operations, as seen in lines 10-21 of "S". Lines 22-26 complete the hex-to-decimal conversion, taking the three digits from Register d and converting them to a decimal integer in Register X.

The result 's' in X is still the absolute address of R_{00}, now expressed in decimal. If there are no memory modules in the calculator, the 'size' is [256-'s'], where the '256' is the decimal address of the top of memory. However, since [256-'s'] is always less than 256 (for no modules), and since 256 is an integral multiple of 64, [-s mod 64] (which, since '-s' is negative, is the smallest positive number obtained by adding multiples of 64 to 's') is the same as [256 + N*64 -s], where 'N' is the current number of modules. Thus, lines 27-29 yield the distance, in registers, from R_{00} to the next higher module boundary. The size is this number plus an unknown multiple of 64. Lines 30-37 are a trial-and-error method of determining 'N', by incrementing 's' in steps of 64 until 'VIEW IND X' causes an error that clears Flag 25.

The hexadecimal-to-octal-to-decimal conversion scheme in "S" can be used in a variety of programs. Only a slight modification of "S" would be required, for example, to yield the current location of the statistics registers block from the first three digits of Register c. A different example is provided by the next routine, "BYTE", which is designed to give the current address pointer location as a decimal number of bytes, counting from the bottom of user program memory. In this context, byte '1' is byte '00C0'.

```
01♦LBL "BYTE"     12 SF 15        23 LASTX
02 CLA            13 FS?C 18      24 FRC
03 STO [          14 SF 17        25 1 E3
04 "⊢♦♦♦♦"        15 FS?C 19      26 *
05 X<> [          16 SF 18        27 DEC
06 X<> d          17 FS?C 20      28 7
07 FS?C 15        18 SF 19        29 *
08 SF 13          19 X<> d        30 +
09 FS?C 16        20 10           31 1343
10 SF 14          21 *            32 -
11 FS?C 17        22 INT          33 END
```

"BYTE"

69 Bytes

Instructions for "BYTE":

1. Press 'RCL b'.
2. XEQ "BYTE".
3. Output is byte number in decimal.

Following the user-executed 'RCL b', which places the current pointer address into the last two bytes of Register X, lines 01-06 of "BYTE" move the two bytes into the second and third bytes of Register d (Flags 08-23). The first digit of the address is a byte number, which never exceeds six. The remaining three digits number registers of 7 bytes, taking a maximum value of hex 1FF or octal 777. Thus Flag 12 will always be zero. Lines 07-21 perform the hex-to-octal conversion, placing in X the number 'n.abc', where 'n' is the byte number, and 'abc' is the number of the register in three octal digits. Line 22 isolates 'n', whereupon lines 23-27 convert 'abc' into a decimal integer. Lines 28-30 compute the total bytes [n + 7*abc], measured from '0000'. Lines 31-32 subtract 1343, so that the output byte number is measured from the bottom of normal program memory, byte '00C0'.

There are two obvious applications for "BYTE", both of which require two executions. More useful than the byte number of any single address is the distance in bytes between two memory locations. "BYTE" saves the original value in Register X present prior to the manual 'RCL b', placing it in Register Y above the output value at the end of program execution. Thus the sequence

```
GTO 'point A'
RCL b
GTO 'point B'
RCL b
```

```
XEQ "BYTE"
X<>Y
XEQ "BYTE"
-
```

which can be executed manually or by a program, will give the distance in bytes between 'point A' and 'point B' in memory. The first application of this procedure is to have 'point B' be the first line of a program, and 'point A' the first line of the next program down in memory. Then the byte difference is the length of the program, as is also given by a 'CAT 1' using the printer. The second type of use is to have 'point B' be a two-byte 'GTO' program line and 'point A' the corresponding label, to determine whether the jump between the 'GTO' and the label is fewer than 112 bytes. The only other means of determining this result is by laborious counting of program bytes, line-by-line.

6C. FUN AND GAMES IN THE ALPHA REGISTER

Perhaps the single most useful group of synthetic functions are those which access the alpha register, such as 'STO M, 'RCL N' or 'X<>IND O'. The fact that the 'M', 'N', 'O', and 'P' postfixes can be attached to any normal data register function prefix means that the alpha register can be used as four (with some limitation on the use of P) extra data registers. This is obviously advantageous when memory space is limited--the use of the alpha register frees four ordinary registers for additional program or data storage. The best use of these 'extra' data registers is for 'scratch' purposes (almost like an extension of the RPN stack) that can be sandwiched in between normal uses of the alpha register for messages, etc. Examples are indirect function indexing (e.g., 'ISG M', 'DSE O', 'STO+ IND N'), accumulations ('STO+M', 'STO-N', etc.), and temporary storage of intermediate numerical results. Note that all four alpha registers are cleared simultaneously by the one-byte function 'CLA'. The alpha registers (and the stack registers) can only be addressed indirectly through the extraordinary step of adjusting the contents of Register c so that one of the status registers (any will work except Register T) becomes R_{00}, but that is seldom practical.

The alpha register access functions combine with the standard alpha functions 'APPEND', 'ASTO', 'ARCL', 'ASHF', and 'CLA' to provide alpha character string manipulations of a speed and flexibility greatly exceeding that possible with the standard functions alone. Consider, as a first example, the problem of isolating a particular character from an alpha string, such as might be required by a variety of word-guessing games. Here is a routine that will isolate (i.e., leave by itself in the alpha register) the 'nth' character (counting from the left) in a string of up to six characters, using only standard functions:

```
01♦LBL A          10♦LBL 01
02 7              11 "*"
03 -              12 ARCL Y
04 CHS            13 ASTO Y
05 1 E3           14 ASHF
06 /              15 ISG X
07 1              16 GTO 01
08 +              17 AVIEW
09 ASTO Y         18 .END.
```

(6C-1)

To use the routine, either the user or program places 'n' into Register X. Then 'XEQ "A"' isolates the 'nth' character in the alpha register. There are two problems with this routine that make it less than satisfactory: first, it is relatively slow, requiring from 0.9 to 2.1 seconds to execute, depending upon the value of 'n'; second, it is not directly extendable to strings of more than six characters. If the strings to be processed can be more than six characters, the program has no way of knowing where the 'first' character is situated in the alpha register. This latter problem can be overcome to some extent by numbering the characters from right-to-left so that 'n = 1' corresponds to the last (rightmost) character in the alpha string. Then the string can be broken into up to four 6-character strings, with the appropriate string, depending on 'n', being searched by Routine 6C-1 for the desired character. But, as advertised, the use of synthetic functions provides a better method.

The invisible boundaries between Registers M, N, O, and P simplify the task of chopping up alpha strings. All we have to do is find an automated procedure for shifting the strings

around so that the desired character is at one of the boundaries. Consider the sequence 'CLX', 'FIX 4', 'ARCL X'. Following execution of these steps, the alpha register will contain its original contents, now shifted left by the appending of the six characters "0.0000". If we had used 'FIX 6' instead of 'FIX 4', the original string would have been shifted by eight positions. This demonstrates a non-iterative (and hence, fast) method of shifting alpha strings by a variable amount, which is used in the following version of "ISO". If you 'single-step' through the program, with the HP-41C in ALPHA mode, you can see the characters shifted around and selectively cleared to leave only one character.

```
01♦LBL "ISO"        08 X<> ]
02 10               09 "⊢↑"
03 -                10 X<> ]
04 CHS              11 CLA
05 SCI IND X        12 STO [
06 ARCL X           13 AVIEW
07 CLX              14 END
```

"ISO"

30 BYTES

Instructions for "ISO"

1. Start with a string of up to 10 characters in the alpha register.
2. Place a number 'n' between 1 and 10 into Register X.
3. XEQ "ISO".
4. At completion, only the 'nth' character from the original string will remain. 'n' is counted from the right.

This routine is both shorter and faster than Routine 6C-1, requiring only 0.8 seconds for execution, independent of 'n'. 'SCI IND X' is used (line 05) rather than 'FIX IND X', to provide shifts of between 4 (for n=10) and 13 (n=1) characters. "ISO" has the disadvantage of changing the HP-41C display mode, but this can be corrected at an expense of 4 additional program bytes by replacing steps 05 and 06 with:

```
05 X<>d
06 SCI IND d
07 ARCL d
08 X<>d
```

Similar operations are found in the next routine, "SUB", which is used to replace a character in an alpha string, leaving the string otherwise intact:

```
01♦LBL "SUB"     12 "⊢↑"          23 X<> ]       34 ARCL X
02 10            13 X<> T         24 LASTX       35 R↑
03 -             14 X<> ]         25 X<> ↑       36 STO d
04 CHS           15 "⊢======"     26 X<> T       37 CLX
05 RCL d         16 CLX           27 9           38 X<> ]
06 SCI IND Y     17 X<> \         28 -           39 STO [
07 ARCL Y        18 STO [         29 CHS         40 CLX
08 RCL ↑         19 CLX           30 FIX 0       41 X<> ↑
09 STO L         20 X<> ]         31 RND         42 STO \
10 CLX           21 STO \         32 CF 29       43 AVIEW
11 X<> ]         22 X<> T         33 10↑X        44 END
```

"SUB"

86 BYTES

Instructions for "SUB"

1. Start with a string of up to ten characters in alpha.
2. Place one alpha character in Register Y.
3. Place a number 'n' between 1 and 10 in X.
4. XEQ "SUB"
5. Following execution, the character from Y will replace the 'nth' character in the alpha string. 'n' is counted from the right.

In lines 30-34 of "SUB" we see another type of variable character shift, using the function '10^X' to produce a number 'x+1' characters long in Register X.

The 'Hangman' game ("HM") listed next demonstrates a practical application of the string manipulations made possible with routines "ISO" and "SUB". Versions of these routines are found in lines 169-183 and 114-168 respectively.

```
 01♦LBL "HM"
 02 0
 03 STO d
 04 .009
 05 STO 07
 06 FIX 0
 07 SF 26
 08 "WORD?"
 09 AON
 10 STOP
*11 "├
 12 ASTO 00
 13 ASHF
 14 ASTO X
 15 CLA
 16 ARCL X
 17 "├↑↑↑↑"
 18 RCL \
 19 CLA
 20 STO [
 21 ASTO 01
 22 "---"
 23 ASTO 03
 24 ARCL 03
 25 ASTO 02
 26 CLA
 27 ASTO 04
 28 1.009
 29 STO 08
 30 STO 06
 31 SF 19
 32 " "
 33 ASTO 05
 34 GTO 02
 35♦LBL 01
 36 1.009
 37 STO 06
 38 CLA
 39 ARCL 02
 40 ARCL 03
 41 "├ "
 42 ARCL 04
 43 TONE 9
 44 CLD
 45 STOP
 46 ASTO 05
 47 RCL d
 48 AVIEW
 49 STO d
 50♦LBL 02
 51 FS? IND 06
 52 GTO 04
 53 RCL 05
 54 CLA
 55 ARCL 00
 56 ARCL 01
 57 RCL 06
 58 INT
 59 XEQ 08
 60 ASTO X
 61 X=Y?
 62 XEQ 03
 63♦LBL 04
 64 ISG 06
 65 GTO 02
 66 FS?C 19
 67 GTO 01
 68 ISG 07
 69 GTO 06
 70 "ARRRRGGH..."
 71 AVIEW
 72 TONE 0
 73 TONE 0
 74 PSE
 75 "WORD IS: "
 76 ARCL 00
 77 ARCL 01
 78 AOFF
 79 PROMPT
 80♦LBL 06
 81 "ᵀΓαβ×"
 82 10
 83 RCL 07
 84 INT
 85 -
 86 XEQ 08
 87 ASTO X
 88 RCL 07
 89 INT
 90 "@@@@@@CL_"
 91 XEQ 08
 92 ARCL Y
 93 ASTO 04
 94 GTO 01
 95♦LBL 03
 96 ISG 08
 97 GTO 05
 98 " **DONE**"
 99 AVIEW
100 TONE 3
101 TONE 4
102 TONE 5
103 TONE 8
104 TONE 7
105 TONE 8
106 CLA
107 PSE
108 RCL 07
109 INT
110 ARCL X
111 "├ WRONG."
112 AOFF
113 PROMPT
114♦LBL 05
115 SF IND 06
116 SF 19
117 RCL 06
118 INT
119 CLA
120 ARCL 02
121 ARCL 03
122 10
123 -
124 CHS
125 RCL d
126 SCI IND Y
127 ARCL Y
128 RCL ↑
129 STO L
130 CLX
131 X<> ]
132 "├↑"
133 X<> T
134 X<> ]
135 CLX
136 FIX 4
137 ARCL X
138 CLX
139 X<> \
140 STO [
141 CLX
142 X<> ]
143 STO \
144 X<> T
145 X<> ]
146 LASTX
147 X<> ↑
148 X<> T
149 9
150 -
151 CHS
152 FIX 0
153 RND
154 CF 29
155 10↑X
156 ARCL X
157 R↑
158 STO d
159 CLX
160 X<> ]
161 STO [
162 CLX
163 X<> ↑
164 STO \
165 ASTO 02
166 ASHF
167 ASTO 03
168 RTN
169♦LBL 08
170 10
171 -
172 CHS
173 X<> d
174 SCI IND d
175 ARCL d
176 X<> d
177 CLX
178 X<> ]
179 "├↑"
180 X<> ]
181 CLA
182 STO [
183 END
```

"HM"

386 BYTES

SIZE 009

* APPEND 11 SPACES

Instructions for "HM":

1. XEQ "HM".
2. First player keys in a word of up to nine letters; R/S.
3. At the tone, the display will show as many dashes "-" as there are letters in the unknown word. The second player guesses a letter by pressing the corresponding letter key, then R/S.
4. At the next tone, all occurrences of the guessed letter will be shown in the display. If the guessed letter is not present, one 'piece' will be added to the 'gallows' "⊏⊐" or to the 'man' "天" at the right of the display. Play resumes with step 3.
5. If the full word is guessed with fewer than 10 wrong guesses, "**DONE**" is displayed, followed by the total number of wrong guesses.
6. On the tenth wrong guess, the 'man' is 'hung', and the unknown word is displayed.

'Hangman' works with words of up to nine letters. If the first player enters fewer than nine letters, the program fills out the word with spaces (lines 11-21), then 'guesses' the 'space' character, the same way a player would, in order to display the correct number of unknown letters to the second player.

Some synthetic programming notes for "HM": Lines 72 and 73 are "TONE 10", hex '9F 0A', which can be created with the byte jumper. The construction of line 81, 'F5 60 06 04 05 01', was described in Section 5I; Line 90, 'F9 40 40 40 40 40 40 43 4C 5F', was made as an example of enhanced byte jumping in Section 5G. The trick used to make the guessed letters 'goose-step' around the display (lines 47-49) is described in Section 7B.

Data storage is allocated by "HM" as follows:

R_{00} & R_{01}	mystery word
R_{02} & R_{03}	current guessed word
R_{04}	'hangman'
R_{05}	current guessed letter
R_{06}	loop counter
R_{07}	wrong guess counter
R_{08}	right guess counter

6D. CHARACTER RECOGNITION

Although a user can simply look at an alphanumeric display to read its contents, the HP-41C itself has no means of determining what characters, if any, are present in the alpha register, except by laborious one-by-one comparisons with known characters. Thus, for example, alphabetizing a group of alpha data strings is a prohibitively slow, memory-expensive process. However, synthetic functions can extend the capability of the HP-41C into the domain of 'word-processing', by allowing conversions of characters into numbers and vice-versa.

Suppose we wish to identify or give a numerical value to a single alpha character. Since there are 256 characters (not all display differently, of course), the identification should consist of a decimal number in the range 0 to 255--i.e., the decimal equivalent of the byte code for the character. The same hexadecimal-to-octal-to-decimal conversion used in Section 6B can be used for this purpose, as shown in this 'Character-to-Decimal ("CD") program:

```
01♦LBL "CD"        10 SF 09
02 "├♦♦♦♦x̄"        11 FS?C 11
03 X<> [           12 SF 10
04 X<> d           13 FS?C 12
05 FS?C 08         14 SF 11
06 SF 06           15 X<> d
07 FS?C 09         16 DEC
08 SF 07           17 END
09 FS?C 10
```

"CD"

43 BYTES

(Line 02 is 'F6 7F 00 00 00 00 02', the same as in the program "EF" in Section 5E.)
Examples: "A", XEQ "CD" gives '65'; "$", XEQ "CD" gives '36'.

The reverse process, 'Decimal-to-Character' ("DC"), is only slightly more complicated. Lines 03-06 of "DC" ensure that the three octal digits of the input number always go into the same set of flags in Register d, even if the number is only a one- or two-digit decimal integer.

```
01♦LBL "DC"      11 SF 19         20 "├♦♦"
02 OCT           12 FS?C 17       21 CLX
03  E3           13 SF 18         22 STO \
04 /             14 FS?C 15       23 "├A"
05 10            15 SF 17         24 X<> \
06 +             16 FS?C 14       25 CLA
07 X<> d         17 SF 16         26 X<> [
08 FS?C 19       18 X<> d         27 AVIEW
09 SF 20         19 STO [         28 END
10 FS?C 18
```

"DC"

58 BYTES

Examples: '37', XEQ "DC" gives "%"; '64', XEQ "DC" gives "@".

The problem of alphabetizing a set of alpha data strings requires a more complicated character recognition scheme than provided by "CD". Since the only alpha comparison the HP-41C can make is 'X=Y?", we need numerical equivalents for entire alpha data strings in order to make the 'X<Y?' comparison necessary for alphabetizing. Once such a comparison is made, standard number sorting techniques can be used to alphabetize a list of alpha strings. A straightforward way of generating such an equivalent would be to use "CD" on each character of an alpha data string and combine the results into a single number. Notice that since the decimal equivalent of the character "Z" is 90, the maximum value of a six-letter string is 90^6 = 5.3 E11, which is greater than the largest integer the HP-41C can handle. Therefore, this conversion process should subtract 64 from the decimal value of each character (making "A" = 1, "B" = 2, etc.) before making the combination of the six values into a single number.

Synthetic programming offers a method of generating numerical equivalents for alpha strings that is much shorter and faster than a character-by-character conversion. The next routine, "AL", alphabetizes a single pair of alpha data strings. It should be combined with the user's choice of ordinary number-sorting routines to alphabetize a set of alpha data.

```
01♦LBL "AL"      12 RTN                           22 RCL [
02 XEQ 01        13 X<> IND T                     23 RTN
03 XEQ 01        14 X<> IND Z                     24♦LBL 01
04 X≠Y?          15 X<> IND T                     25 "xx" (F2 Ø1Ø1)
05 GTO 03        16 RTN                           26 ARCL IND Y
06 RDN           17♦LBL 02                        27 "├♦♦♦"
07 RDN           18 "xxx" (F3 Ø1Ø1Ø1)             28 ASTO [
08 XEQ 02        19 ARCL IND Y                    29 "├♦♦"
09 XEQ 02        20 ASHF                          30 RCL [
10♦LBL 03        21 "├♦♦"                         31 END
11 X>Y?
```

"AL"

70 BYTES

SIZE 002

Instructions for "AL":

1. Two alpha data strings to be ordered should be in numbered data registers. The strings can be 1 to 6 characters.
2. Place the number of one data register in X, the number of the other in Y.
3. XEQ "AL".
4. "AL" places the string that comes first alphabetically in the register originally designated in Y; the other string goes to the register originally designated in X.

"AL" first uses subroutine 01 (lines 24-31) to change the first four characters of the two strings into numbers for comparison. A 'number' is characterized by a first nybble of '0' or '9'; furthermore, numerical comparisons of alpha strings are only meaningful if the assigned numbers have common exponents. These two considerations restrict the alpha comparisons to four characters at a time, since we need to put a number identifier ("AL" uses byte '01') at the left of the string, and two bytes '00 00' on the right to standardize the exponents. This leaves only four 'free' bytes in a seven-byte register. Four characters are usually

sufficient to distinguish two strings; if the remaining two bytes of the original strings are required, subroutine 02 (lines 17-23) provides for that additional comparison.

The trick of changing an alpha data string into a number can be 'reversed'. The next routine, "MANT", shows how a number can be changed into alpha characters to use alpha instructions like 'APPEND' or 'ASTO' to change the number. In this case, we want to replace a number with its mantissa by chopping off its exponent. We could do this using the 'LOG' and '10^x' functions, but that occasionally introduces error into the last digit of the mantissa. "MANT" always gives an exact result. Upon execution of "MANT", the number in Register X is replaced by its mantissa (including the sign); Y and Z are undisturbed; T, L, and the alpha register are lost.

```
01♦LBL "MANT"      09 1 E50
02 STO [           10 *
03 CLX             11 LASTX
04 FIX 4           12 X>Y?
05 ARCL X          13 1/X
06 X<> [           14 /
07 "⊢↑"            15 FIX 9
08 X<> \           16 END
```

"MANT"

42 BYTES

6E. SYNTHETIC TEXT LINES AND THE PRINTER

Synthetic text lines, created by any of the methods described in Chapters 3 and 5, are particularly useful for printer applications. Any of the 128 standard printer characters can be included in a program text line by placing the corresponding byte (as found in the Byte Table) in the line. This eliminates the necessity for repeated use of the printer function 'ACCHR'. For example, consider a routine that would print the characters "Big Deal #7". Using 'ACCHR', we would need the following steps,* a total of 40 bytes: Exactly the same result can be obtained in only 14 bytes by writing a synthetic text line that contains the lower case letters and the "#" symbol explicitly:

```
01 ⊤B▩▩ Dea▩ #7
02 PRA
```

Line 01 is coded 'FB 42 69 67 20 44 65 61 6C 20 23 37'. It can be created easily with byte jumping, or with the text enabler:

```
01 +
02 *          ("B")
03 FRC        ("i")
04 X=0?       ("g")
05 RCL 00     (" ")
06 X<Y?       ("D")
07 LN1+X      ("e")
08 ABS        ("a")
09 HMS        ("l")
10 RCL 00     (" ")
11 RCL 03     ("#")
12 STO 07     ("7")
13 PRA
```

*

```
01 CF 13     07 CF 13     13 CF 13
02 "B"       08 "D"       14 35
03 ACA       09 ACA       15 ACCHR
04 SF 13     10 SF 13     16 "7"
05 "IG"      11 "EAL"     17 ACA
06 ACA       12 ACA       18 ADV
```

TE .001 [01"B▩▩ Dea▩ #7"]
Press correction key 5 times.

In a similar manner, we can replace the printer function 'BLDSPEC'. Consider the special graphics character shown in Figure 6-1, where we show the dot pattern and the corresponding 'values' and 'column print numbers' as called for by the 'BLDSPEC' instructions (<u>82143A Printer</u>

Owner's Handbook, pp. 64-66).

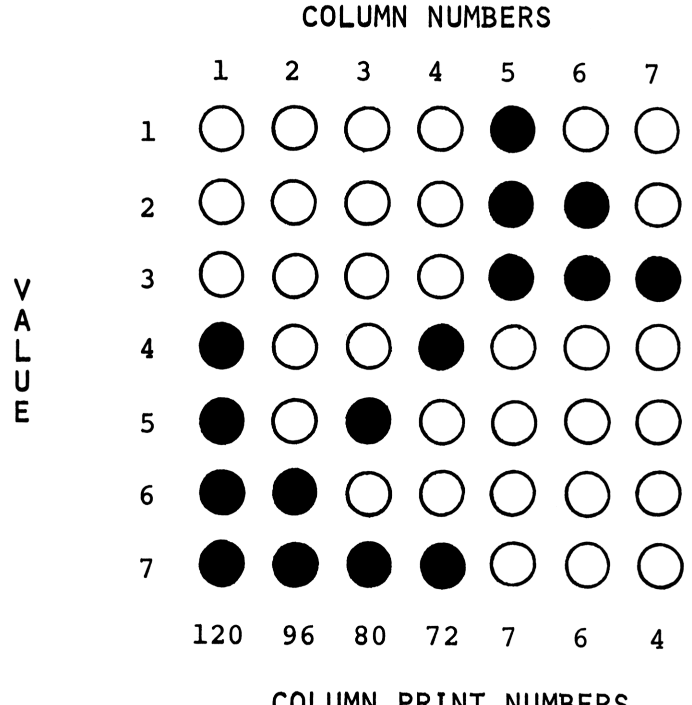


FIGURE 6-1. A SPECIAL GRAPHICS CHARACTER

The column print numbers are just the decimal equivalents of numbers made by treating each column as a 7-bit binary number, with dark dots as 1's and blank dots as 0's. The first time 'BLDSPEC' is executed, with zero in Register Y and the first column print number (120) in X, an alpha data string is created in X which uses the seven bits of the first column print number (1111000) as the last seven bits of the string:

```
X = 0001000000000000000000000000000000000000000000|1111000
```

When 'BLDSPEC' operates on the next column print number (96), its seven bits are copied into the last seven bits of X, with the previous entry 'pushed' to the left:

```
X = 000100000000000000000000000000000000000|1111000|1100000
```

After the last column print number is entered, Register X contains:

```
          |   120   |   96   |   80   |   72   |   7   |   6   |   4
X = 0001 000|1 1110 00|11 0000 0|101 0000 |1001 000|0 0001 11|00 0011 0|000 0100
(hex)   1    1    E    3    0    5    0    9    0    1    C    3    0    4
```

The successive column print numbers are shown above the binary representation of X. Below X, we show the byte codes corresponding to the bit pattern. This byte code is entered into program as a 7-character text line 'F7 11 E3 05 09 01 C3 04' that will place the byte code into the alpha register, whence it can be transferred to Register X using 'RCL M'. Then the printer function 'ACSPEC' can be executed normally.

In practice, the step of computing the column print numbers is unnecessary. All you have to do is write out a 49-bit binary number to represent the 7 x 7 dot grid, using 1's for

dark dots and 0's for blanks. Start with the lower left corner of the grid as the leftmost bit, then work up to the top of the first column, then bottom-to-top on the second column, etc. When all the dots are encoded, add the 7 bits '0001000' to the left of the number for a total of 56 bits. Group the 56 bits into 4-bit digits, then make a 7-character text line from the hexadecimal equivalents. Since these bytes may come from anywhere in the Byte Table, use of "CODE" plus the Q-loader is an ideal way of creating the desired text line.

This procedure may seem like a lot of trouble, but with practice, it's scarcely more difficult than the normal 'BLDSPEC' method. The savings in program memory are obvious from the following comparison:

'Normal' Program:

```
01 0          10 BLDSPEC
02 ENTER      11 7
03 120        12 BLDSPEC
04 BLDSPEC    13 6
05 96         14 BLDSPEC
06 BLDSPEC    15 4
07 80         16 BLDSPEC
08 BLDSPEC    17 ACSPEC
09 72
```

--30 bytes

Synthetic Program:

```
01 "▩▩天▩天▩大"
02 RCL M
03 ACSPEC
```

--12 bytes

Of course, you could execute the normal 'BLDSPEC' sequence manually and store the resulting special character alpha data string in a data register for use by a program, ending up with a total memory use (counting the data register) of only 10 bytes. But if the program is read from magnetic cards, a data card must also be read; furthermore, the data register used must be guarded against use by any other program as long as you desire to use the special character program.

If you are bothering to read this section, you are probably using a printer to list the programs. On printer output, line 01 of the synthetic program last described will print as:

```
01 "Ωβσ×α"
```

There are only five characters shown, because the program listing of a text line will only show characters from the top half of the Byte Table. Characters corresponding to bytes in the lower half of the table are invisible. Furthermore, the print buffer uses bytes from rows A, B, D, and E for internal purposes related to special character printouts, single and double width instructions, etc. Hence, text lines containing characters from those four rows may print out in very strange ways. For example, if a text line contains the character corresponding to byte 'D5', a program listing containing that line will have all printout following that character printed double-wide and lower case.

6F. NON-NORMALIZED NUMBERS AND MASS FLAG CONTROL

The use of synthetic text lines is by no means restricted to the programmed generation of non-standard character strings in the alpha register. A synthetic text line of up to seven characters, followed by a 'RCL M', will place an NNN into Register X. An important use of NNN's so created is for 'mass flag control' through storage of the NNN into Register d. We have already seen one application of mass flag control in the use of the text enabler.

The programs described in previous sections have contained numerous examples of the use of instructions such as 'X<>d' to restore an initial flag status after the flag register has been used as a 'binary encoder'. The ability to create any NNN allows us to set or clear all 56 flags in one operation. The basic sequence is:

```
01 "xxxxxxx"
02 RCL M          (6F-1)
03 STO d
```

where "xxxxxxx" represents the synthetic text line used to generate the NNN. Routine 6F-1 uses 12 program bytes, the same as would be required for six 'SF mn' or 'CF mn' program lines. In general, therefore, it is more efficient to use Routine 6F-1 rather than individual 'SF' or 'CF' lines whenever more than six flags are to be set or cleared. Such occasions arise frequently in program initialization routines, where various user flags are set or cleared, and desired display and trigonometric formats are established.

As an illustration, let us write a routine to set the HP-41C flags as follows: Flags 1, 2, 3, 26 (audio enable), 28 (radix), and 29 (separator) are set; 'FIX/ENG 3' display format (Flags 38, 39, 40, and 41 set); 'RAD' mode (Flag 43 set); continuous ON (Flag 44); all other flags clear. The 'FIX/ENG' display is chosen particularly because it is a number display format that is not available without synthetic programming. In ordinary 'FIX' format (Flag 40 set, Flag 41 clear), numbers which are too large or too small to display properly cause the display to default to the 'SCI' format. In the 'FIX/ENG' format, however, the default is to 'ENG' mode.

To determine the synthetic text line required to generate the desired flag status, we write out the states of all of the flags as a 56-bit binary number, with 1's for set flags and 0's for clear flags, then group the bits into eight-bit hexadecimal bytes:

```
                                 audio  radix      digits      RAD
1 2 3                            enable  | separator\ FIX/ENG /  ON
\ | /                                 \  | /         \\  ^   / /
0111 0000|0000 0000|0000 0000|0010 1100|0000 0011|1101 1000|0000 0000

  7    0 |  0    0 |  0    0 |  2    C |  0    3 |  D    8 |  0    0
```

We see that the required text line is 'F7 70 00 00 2C 03 D8 00'. This particular byte code is a challenge to each of the synthetic text line generating methods we have studied. For example, because the line is eight bytes long, it can't be created with a single operation of "REG". However, we can key into program a 'dummy' 7-character text line and jockey it around in memory by adding or deleting bytes higher in memory until the seven character bytes are positioned all in the same register (i.e., with the program display showing the dummy line, 'RCL b', XEQ "AD" should give an address starting with a byte number '1'). Then we can use "REG" to store the code '70 00 00 2C 03 D8 00' into the register containing the dummy characters, leaving the 'F7' byte intact.

It is tricky to use the byte jumper or the text enabler to edit in the text line because of the 'D8' byte, which cannot be keyed in as a stand-alone line. Also, the Q-loader won't work because of the '00' byte on the end. If we used the Q-loader on the code, we would obtain a six-character text line, since the Q-loader 'ignores' any leading nulls in the input code. But a combination of the byte jumper and the Q-loader will do the trick. We first use the Q-loader with the code '01 D8 03 2C 00 00 70', where we have replaced the '00' byte with an '01'. Then we byte-jump to the '01' and delete it.

```
                              XEQ "CODE"        [CODE=?__         ]
"01 D8 03 2C 00 00 70"        R/S               ["天▦▦,¯¯▦"       ]
                              GTO new program
01 STO 07                     PRGM off
                              STO Q
                              PRGM on
                              press Q-loader    [02 9             ]
                              DEL 001           [01 STO 07        ]
                              PACK, SST         [02 "▦¯¯,▦▦天"     ]
                              JUMP .002         [02 LBL 00        ]
                              DEL 001           [01 STO 07        ]
                              DEL 001           [00 REG abc       ]
```

We finish by adding the lines '02 RCL M', '03 STO d' to match Routine 6F-1.

Storing NNN's into Register d is the only way we can set many of the 'system' flags, i.e., Flags 30-35, 45-47, and 49-55. Although control of these flags usually leads to some amusing but not particularly useful effects (see Section 7B), an example of a practical application involves setting the system Data Entry Flag 45 (already encountered through use of the text enabler).

Often, especially during calculations using statistical accumulations, we are required to enter a string of numbers that differ only in the last digit or two, like '123456', '123457', '123460', etc. To save entering the '1234' each time, we could add a few program steps that add '123400' to our entries so that we only have to enter the final two digits of each number. But we can make the process even more 'friendly' by asking the HP-41C to enter and display the '1234' in such a way that when we key in the final two digits, we see the '1234' at the same time. Setting Flag 45 allows this kind of operation:

```
01 "▓-"   (F2 84 00)
02 RCL M
03 X<> d
04 1234
05 STOP
06 X<>Y
07 STO d
```

(6F-2)

The first character of line 01, byte '84', sets Flags 40 and 45 when stored into Register d. When the program halts at line 05, with Flag 45 set, the processor thinks it is still in the process of data entry. At the halt, the display will show '1234' (Register X). If we press a number key, '5' for example, the display will become '12345_', with the underline indicating that further digit entry is possible. Lines 06 and 07 are optional--they serve to restore the initial flag status, leaving the newly entered number in Register Y.

The same trick used in Routine 6F-2 works with alpha character entry, so that we can add alpha characters to a current alpha string while displaying the entire string. Or, to go one better, we can append alpha characters to a display message, yet have only the newly entered characters remain in alpha for processing. To illustrate, replace the first five lines of "CODE" with this sequence:

```
01 LBL "CODE"
02 "⊼▓"        (F2 04 80)
03 RCL M
04 X<>d
05 "CODE="
06 AVIEW
07 CLA
08 STOP
09 X<>d
```

Lines 02-04 set Flags 45 and 48 (ALPHA on). Lines 05-07 arrange that when the program halts at line 08, the display will show "CODE=", even though the alpha register has been cleared. When we key in the alpha characters for the code, they enter the alpha register normally, but also appear in the display appended to the "CODE=". If, during the halt, we clear the display by turning ALPHA off, then on, the phantom "CODE=" will disappear, leaving only the keyed-in characters in the display. This is not a profound achievement, but it does make the HP-41C even 'friendlier'. Unfortunately, there doesn't (yet) seem to be a way to add numerical entries to alpha prompts.

A cautionary note is in order regarding the use of NNN's in the HP-41C. The arithmetic routines in the calculator were designed to handle only normal decimal numbers. Their use with NNN's can have surprising and occasionally unpleasant results. For example, use "CODE" to generate the NNN '00 00 01 00 00 00 00', which will be displayed in 'SCI 5' format as '0.00010 E00'. Now execute '1/X', and watch what happens. The display will blank for about 5 seconds, during which time the keyboard is 'locked out', i.e., the HP-41C will not respond to any key presses, including the 'ON' switch. The NNN denominator causes the delay: the 'divide' process (or 1/X) assumes that both numerator and denominator are stored in proper scientific format. Dividing is carried out as a series of subtractions--the two exponents are subtracted, then the denominator is repeatedly subtracted from the numerator--in effect, a reverse of multiplication by repeated adding. The process doesn't take long if both numerator and denominator mantissas are of order unity, as they should be, but in our example, the denominator mantissa is only 0.0001, so that 10^4 times as many subtractions are necessary to complete the division. Five seconds is not long, but other NNN's could easily require 5000

seconds or more for divisions. Other functions may take even longer: 'LOG (0.0001 E0)' takes 45 seconds, compared to the 5 seconds for '1/X'. The 'normalization' of register contents that occurs when register recall functions are executed is specifically designed to eliminate the danger of calculator 'lockups' caused by NNN's that might be introduced into data registers when a memory module is inserted. So, be careful. As with most other crashes, battery removal and replacement will unlock the machine.

6G. RAISING THE CURTAIN

Because of the savings in program bytes associated with the use of the one-byte 'STO' and 'RCL' functions, it is desirable for programs to use data registers R_{00}-R_{15} whenever possible. Thus, it is quite common to have several programs in the HP-41C memory which each use the same block of data registers, so that execution of one program would destroy the data used by another. One solution is to write a data transfer program that moves the contents of a block of data registers to another block, clearing the first set of registers for use by another program. If the number of registers involved is large, this will be a slow process. The program "CU" (for 'Curtain') offers an alternate, faster solution.

```
01♦LBL "CU"
02 STO L
03 CLX
04 RCL c
05 STO [
06 "├♦♦♦♦♦"
07 11
08 X<> [
09 X<> d
10 STO ]
11♦LBL 11
12 RDN
13 CLX
14 LASTX
15 INT
16 X=0?
17 GTO 14
18 2
19 /
20 RCL [
21 X<>Y
22 FRC
23 X=0?
24 GTO 13
25♦LBL 12
26 FC?C IND Y
27 SF IND Y
28 FC? IND Y
29 CHS
30 X>0?
31 GTO 13
32 FC? IND Y
33 CHS
34 DSE Y
35 GTO 12
36♦LBL 13
37 DSE [
38 GTO 11
39♦LBL 14
40 X<> ]
41 X<> d
42 STO [
43 "├ABC"
44 X<> \
45 STO c
46 RDN
47 END
```

"CU"
87 BYTES

Instructions for "CU":

1. Enter an integer number 'n' into Register X.
2. XEQ "CU".
3. If n>0, R_n will become the new R_{00}. If n<0, R_{-n} will become the new R_{00}. All other data registers will shift accordingly.

"CU" takes an integer number 'n' from Register X (entered manually or by another program) and adds it to the address of R_{00} stored in Register c. If 'n' is positive, data registers R_{00} through R_{n-1} will be 'transformed' into program registers, by raising the imaginary 'curtain' that separates data and program memory from its initial position below R_{00} to a new position below R_n; R_n becomes the new R_{00}. If 'n' is negative, the curtain is lowered, so that 'n' registers of program memory are transformed into data registers. All of this occurs without alteration or moving of the contents of the registers involved.

Suppose 'Program 1' is executed, leaving data in R_{00}-R_{50} that is required for future use. But in the meantime we wish to execute 'Program 2', which uses registers R_{00}-R_{25} for its own purposes. In this case, we enter '51' into X and execute "CU" (the 'size' should be 77 or greater). Following execution of Program 2, we can prepare for a second run of Program 1 by pressing '-51', 'XEQ "CU"'.

**Warning: Raising the curtain above the top of memory, i.e., executing "CU" for 'n' greater than the current 'size', or lowering it to (hex) addresses from '010' through '0C0', or to '000', will cause 'MEMORY LOST'.

"CU" works by performing a binary addition of the number 'n' to the hexadecimal digits 9-11 of Register c that constitute the 'curtain' address. The corresponding Flags 32-43 in Register d cannot be controlled individually, so the contents of Register c are transferred to Register M and shifted 'left' by appending null bytes (line 06). Then lines 08-09 place the curtain address into Register d as Flags 00-11.

Binary addition is a very simple process. To add 1 to a binary number, we merely switch the value of the last bit, from 1 to 0 or vice versa. If the last bit becomes a 1, we stop.

If it becomes a 0, we switch the next bit to the left. If the next bit becomes a 1, we stop; if it becomes a 0, we go on to the next bit to the left, and so forth until we stop at a bit that changes from 0 to 1. Subtracting 1 is almost the same--we follow the same procedure, starting on the rightmost bit and working left until we encounter a bit that changes from 1 to 0. Adding 2 (binary 10) works the same way, only we start with the next-to-last bit. In general, to add 2^m, we start with the 'm+1' bit, counting from the right.

Binary addition is performed in lines 11-35 of "CU". The entered number 'n' is broken up into binary bits through repeated division by 2 (lines 18-19). The successive bits are added or subtracted to the address bits according to the test in line 30. Once the addition is complete, the three bytes of the Register c code that are in Register d are rejoined to the first four bytes waiting in Register N (lines 40-42). Then the full code is bumped into N (line 43), and finally restored to Register c in line 45. When "CU" is finished, the contents of stack Registers X and Y prior to entry of 'n' are restored.

The HP-41C will operate quite normally while the curtain is raised or lowered from the position last established by a 'SIZE' operation. However, if the curtain is raised, changing data into program, the memory should not be 'PACKed', since that will most likely change the data stored below the curtain irreversibly by removing all the null bytes in the data. This difficulty can be avoided if an 'END' is placed at the top of program memory, followed by execution of 'PACK'. If the curtain is subsequently lowered, the data registers transformed to program memory will be unaffected by the 'PACK'. They are protected by the 'END', which was coded to indicate a packed file.

A second important application of "CU" is to change data into program permanently, providing us with yet another means of generating synthetic program lines. This method is most useful when several consecutive registers of program, or perhaps an entire program, contains sufficient synthetic program lines to justify being written entirely with "CODE". In that case, we use "CODE" to generate the byte code for each seven bytes of program, storing the successive codes into adjacent data registers. The last seven bytes go into R_{00}, the next-to-last into R_{01}, etc. (This, incidentally, is a major justification for writing "CODE" so that it uses no numbered data registers.) When the coding is complete, we use "CU" to raise the curtain above the highest data register containing program. The synthetic codes will then appear as program lines, starting at the top of program memory. To access the new lines, we use 'CAT 1', stopping at the first global label or 'END', followed with a manual 'RTN'. Here's a sample:

```
                     XEQ "CODE"        [CODE=?          ]
"C000F400600401"
                     R/S               ["▩¯▩¯ ᵀ人天 "     ]
                     STO 01
                     XEQ "CODE"        [CODE=?          ]
"F32801297E8685"
                     R/S               ["▩(天)Σ▩▩"       ]
                     STO 00
2
                     XEQ "CU"
                     CAT 1--stop at first label or END
                     RTN
                     SST to see new program:

                    | 01 LBL " ᵀ人天 " |
                    | 02 "(天)"        |
                    | 03 AVIEW         |
                    | 04 BEEP          |
                    | 05 RTN           |
```

At this point, the LBL " ᵀ人天 " will not show up in the user catalog since it is not part of the global chain. This can be fixed by inserting then deleting a temporary program line anywhere among the new lines, followed by 'PACK'.

6H. DIRECT ROM ACCESS

The Application Pac 'Read-Only Memory (ROM)' modules for the HP-41C are an important means of extending the memory capacity of the calculator to include an extensive library of preprogrammed routines. Unfortunately, many of the routines suffer from the limitation that they cannot be called as automatic subroutines from user programs because of the various halts for manual input and output included in the routines. In many cases, this limitation can be overcome by using the following routine, which permits direct branching to any point in any ROM routine:

```
01♦LBL "ROM"      11 X<> [          21 X<> ]
02 SF 01          12 X<> \          22 X<> a
03 X<> a          13 X<> [          23 X<> \
04 X<> \          14 ARCL 00        24 STO b
05 CLX            15 "├-----"       25♦LBL 00
06 RCL b          16 X<> ↑          26 X<> \
07 FC?C 01        17 X<> ]          27 CLA
08 GTO 00         18 "├++"          28 END
09 STO [          19 STO [
10 "├↑↑↑↑↑"       20 "├**"
```

"ROM"

73 BYTES

SIZE 001

Instructions for "ROM":

Prior to execution of "ROM", store in R_{00} the absolute address of the point in the ROM routine where you wish execution to begin. Then, your program should call "ROM" as a subroutine rather than calling the ROM routine directly. The RPN stack and data registers should be configured as expected by the ROM program at the point of entry. "ROM" transfers execution to the specified address in the ROM, following which the routine executes normally, returning to the original main program upon encountering the final 'RTN' or 'END' in the ROM routine. Although ordinarily a ROM program called by its global label can be a sixth-level subroutine (i.e., during its execution, there can be up to six pending addresses in the return stack), "ROM" can only be called as a fifth level subroutine.

The loss of one subroutine level arises from the way that "ROM" works. Following execution of line 09, the alpha register will contain a replica of the return stack in Registers a and b:

R6 R5 R4 R3 R2 R1 A6

where 'A6' is the absolute address of the second byte of line 06 (that's where the RCL b was made); 'R1' is the return address of the program line from which "ROM" was called as a subroutine; 'R2' is the second pending subroutine return, etc. Lines 10-20 shuffle the characters in the alpha register around until Registers O and N contain a new return stack of the form:

R5 R4 R3 R2 R1 ER A6

where 'ER' is the address of the entry point in the external ROM, recalled from R_{00}. Notice that the 'R6' address has been lost, accounting for the loss of one subroutine level when executing "ROM". The new return stack is stored into Registers a and b by lines 21-24. At the execution of line '24 STO b', 'A6' becomes the pointer address, so that execution will resume with line 07. This time through, Flag 01 is clear, so the program jumps to line '25 LBL 00'. Lines 26-27 complete some 'housekeeping', restoring the RPN stack to its state at the point when "ROM" was called. The 'END' drops the return stack, so that 'ER', the address in the ROM, becomes the pointer address, whereupon execution transfers to the ROM program. At the final 'END' or 'RTN' encountered there, execution returns to the user program at the calling address 'R1'. Remember that the ROM routine itself may add subroutine levels, causing 'R5', 'R4', etc. to be lost.

The procedure for determining the correct ROM entry point address is quite simple. First, execute a 'GTO' to any global label within the ROM program of interest. Then press 'GTO .lmn', where 'lmn' is the line number of the line where you want program execution to transfer. Then press 'RCL b', 'CLA', 'STO M', 'ASTO 00', which places the address in R_{00}, in the proper form

for use by "ROM". The choice of R_{00} is arbitrary; if that register is required for some other purpose, any data register may be used if line 14 is altered accordingly.

As an example of the use of "ROM", suppose we want to use the "SSS" routine in the Math Pac. This routine prompts for manual entry of the lengths of three sides of a triangle, then outputs the angles, sides, and area of the triangle. If the printer is not attached to the HP-41C, the output requires manual 'R/S's to produce each of the seven outputs, precluding use of "SSS" as an automatic subroutine.

Use of "ROM" can eliminate this difficulty by allowing "SSS" to be 'called' at a point in the program after the input halts, and also after the 'SF 21' instruction (line 65) that causes the output halts in the absence of the printer. A good entry point is line '06 LBL 05'. At this point, the program assumes that the lengths 'S1', 'S2', and 'S3' are already in R_{00}, R_{02}, and R_{04}, respectively, so we should arrange the calling program to accomplish that storage:

```
01♦LBL "MAIN"
02 25
03 STO 00
04 35
05 STO 02
06 45
07 STO 04
08 XEQ "ROM"
09 "DONE"
10 AVIEW
11 END
```

In the sample program "MAIN", we put in explicit side lengths of '25', '35' and '45'. To run the program:

1. GTO ."SSS"
2. CLA
3. GTO 05 (or GTO .006)
4. RCL b
5. STO M
6. ASTO 00
7. XEQ "MAIN"

When "DONE" appears in the display, we will find the solutions

A1 = 95.74 in R_{01}
A2 = 33.56 in R_{03}
A3 = 50.70 in R_{05}
AREA = 435.31 in X.

Since "SSS" uses R_{00}, it would be preferable to change the register used to store the ROM address to some other register, say R_{10}, so that steps 1 through 6 would not have to be repeated for each subsequent execution of "MAIN".

CHAPTER 7

AMUSING ANOMALIES

The primary purpose of synthetic programming is to extend the programming capability of the HP-41C. The application programs in Chapter 6 are the results of straightforward use of synthetic functions, plus a lot of experimentation, wild ideas, diligent searching, etc. It should not be surprising that this exploration of the inner workings of the HP-41C has also turned up a number of oddities, which have no important practical uses, but are nevertheless amusing to play with. This chapter contains descriptions of several of these oddities.

7A. 128 TONES?

In normal operation, the HP-41C can execute the ten tones 'TONE 0' through 'TONE 9', corresponding to the byte codes '9F 00' through '9F 09'. We saw at the the end of Chapter 3, however, that the code '9F 0A' executes a new tone, longer in duration and lower in frequency than any of the standard tones. '9F 0A' displays in program mode as 'TONE 0', as we saw in the 'Hangman' program of Section 6C. We can use synthetic programming techniques to attach to the '9F' prefix any of the 128 postfixes from rows 0 through 7 of the Byte Table. It turns out that almost every combination gives a different tone. There are 16 frequencies, corresponding to the 16 possible values of the second nybble of the tone line postfix. Two tone codes differing only in the first nybble of the postfix produce tones of the same frequency, but usually of varying duration.

In program mode, any 'TONE' line with postfix less than hex '65' (decimal 101) will display as 'TONE n', where 'n' is the second digit of the decimal equivalent of the postfix byte For higher postfixes, the lines will display as 'TONE α', where 'α' is a single alpha character postfix, such as 'TONE D' for code '9F 69' or 'TONE P' for code '9F 78'. Table 7-1 shows the tone durations and frequencies for each of the 128 possible postfixes. The data in the table was assembled by Richard Nelson (PPC Calculator Journal, V7 N1 P21, 1980). There are a few cases of duplication, so that there are actually only 114 'different' tones.

In order to experiment with these tones, you can execute the program "TONE", which will automatically create 127 'TONE' program lines (all but 'TONE 0', '9F 00', which can be made normally). After execution of "TONE", the first 127 lines of the first program in memory will be 'TONE' lines, with the line number of each line being the same as the tone number, from 1 to 127. "TONE" calls "DC" (Section 6D) and "CU" (Section 6G). Prior to execution, set 'SIZE 046 (or greater)'. After execution the size will be reduced by 43 registers.

The 'synthetic tones' are no more 'musical' than the standard tones. Nevertheless, the additional frequencies and variety of tone durations allows much more interesting audible output from the HP-41C.

```
01♦LBL "TONE"        14 ASTO X        27 43
02 FS? 55            15 RCL 43        28 XEQ "CU"
03 CF 21             16 INT           29 BEEP
04 1.126             17 XEQ "DC"      30 RTN
05 STO 43            18 ASTO X        31♦LBL 03
06 42                19 CLA           32 ISG 44
07 STO 45            20 ARCL Y        33 GTO 01
08♦LBL 02            21 ARCL X        34 "├♦"
09 .002              22 ISG 43        35 RCL [
10 STO 44            23 GTO 03        36 STO IND 45
11 CLA               24 "├"           37 DSE 45
12♦LBL 01            25 RCL [         38 GTO 02
13 "├" (F2 7F 9F)    26 STO 00        39 END
```

"TONE"

88 BYTES

SIZE 046

TABLE 7-1. TONE FREQUENCIES, NUMBERS, AND DURATIONS (SEC)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
Freq. (Hz)	175	197	225	263	315	394	525	629	788	1051	105	113	121	131	143	158	
0	TONE 0 0.28	1 0.28	2 0.28	3 0.28	4 0.28	5 0.28	6 0.28	7 0.28	8 0.28	9 0.28	0 2.20	1 2.20	2 2.70	3 3.50	4 0.80	5 2.30	0
1	TONE 6 2.00	7 0.34	8 1.50	9 0.33	0 0.50	1 1.00	2 0.45	3 0.84	4 0.30	5 0.55	6 5.00	7 3.50	8 2.00	9 4.10	0 0.30	1 2.40	1
2	TONE 2 0.025	3 1.13	4 2.35	5 2.00	6 1.35	7 0.023	8 0.023	9 0.35	0 0.70	1 0.52	2 0.85	3 0.45	4 3.20	5 0.18	6 1.36	7 0.13	2
3	TONE 8 0.54	9 0.37	0 2.10	1 1.95	2 0.28	3 0.15	4 0.80	5 0.77	6 0.65	7 0.058	8 0.42	9 0.41	0 3.30	1 0.39	2 0.97	3 0.30	3
4	TONE 4 1.88	5 2.35	6 0.40	7 0.24	8 1.05	9 0.29	0 0.032	1 0.24	2 0.14	3 0.15	4 3.70	5 0.30	6 3.76	7 3.40	8 0.89	9 0.90	4
5	TONE 0 0.085	1 0.22	2 1.75	3 0.74	4 0.28	5 1.25	6 0150	7 0.14	8 0.58	9 0.050	0 2.70	1 0.42	2 3.21	3 2.95	4 0.30	5 2.40	5
6	TONE 6 0.65	7 2.32	8 0.43	9 1.25	0 0.12	1 1.00	A 0.99	B 0.84	C 0.70	D 0.52	E 0.23	F 0.45	G 3.62	H 0.33	I 2.10	J 0.35	6
7	TONE T 1.70	Z 0.65	Y 1.45	X 0.52	L 1.25	M 1.30	N 0.24	O 0.84	P 0.14	Q 0.33	+ 0.25	a 4.60	b 0.76	c 4.00	d 3.50	e 2.90	7
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

SECOND POSTFIX DIGIT

FIRST POSTFIX DIGIT

7B. TRICKS WITH SYSTEM FLAGS

In our development of synthetic programming, we have encountered several examples of the deliberate setting of normally inaccessible system flags to produce surprising (such as the setting of the Low Battery Flag 49 in Section 4D) or useful (the text enabler, Section 5H) results. Some further amusing effects can be produced by setting system flags. Here is a set of routines for use in flag register explorations:

"SAVE"
18 BYTES
SIZE 002

"RE"
16 BYTES
SIZE 002

"FL"
42 BYTES

```
01♦LBL "SAVE"
02 0
03 RCL d
04 XEQ "CS"
05 RTN
06♦LBL "RE"
07 0
08 XEQ "CR"
09 STO d
10 RTN
11♦LBL "FL"
12 24
13 -
14 X<> d
15 STO [
16 "├♦♦♦"
17 RCL [
18 X<> d
19 SF IND X
20 RCL d
21 STO [
22 "├ABCD"
23 X<> \
24 STO d
25 RTN
26 END
```

Prior to experimenting with the system flags, you should execute "SAVE", which recalls the current content of Register d and stores it (using "CS") in R_{00} and R_{01}. Then, at any time, you can return your calculator to its initial flags status by executing "RE" (which calls "CR").

"FL" provides a means of setting any system flag (up to Flag 53) by moving the content of Register d 'to the left', where control of user flags allows us to set the bit in the 'shifted' location of the system flag of interest. Then the Register d bytes are shifted back to their original position; when the program halts, the chosen system flag will be set. For example, '49 XEQ "FL"' turns on the BAT annunciator; '47 XEQ "FL"' halts execution with the SHIFT annunciator active (and the next key pressed will execute its shifted function).

Setting Flag 30 will produce some unusual 'catalog' displays. These phantom catalogs don't have any particular application, but it is interesting to watch the various 'entries' in the catalogs as they cycle through. According to Thomas Cadwallader, different catalogs are accessed by choosing various number display formats before setting the catalog flag. To see one such catalog, try 'FIX 9, 30, XEQ "FL", R/S'. Notice that these catalogs can be halted and single-stepped just as any normal catalog.

For whatever it's worth, we can turn on program mode with '52 XEQ "FL"'. You may have wondered how, in programs like "CODE" and "DECODE", we could be so cavalier about storing 'any old thing' into Register d--why, for example, doesn't the HP-41C switch to PRGM mode when Flag 52 is set during a running program? The answer lies in the fact that the processor only checks the statuses of the various flags at particular times, not continuously, so as long as potentially dangerous flags are cleared before they are checked, nothing untoward happens. But there certainly are pitfalls: to see what can happen, alter "FL" by inserting a line '25 1' following line '24 STO d', then press '52 XEQ "FL"'. The processor does turn on PRGM mode, but since the program is still running, the HP-41C starts to program itself, filling up available space with '1' lines until memory is full, at which point the display shows 'PACKING, TRY AGAIN'! Evidently, one of the times Flag 52 is 'checked' is at number entry program lines.

Flag 50, the Message Flag, is perhaps the most interesting of the system flags. At the time Flag 50 is set, whatever happens to be in the display is 'frozen in'. To see four different possibilities, key in (after cleaning up all the '1' program lines made in the last experiment) a line '24 STOP' following line '23 X<>N'. Then try '50 XEQ "FL"', and observe the display when the program halts. Press and release 'SST' quickly. The number displayed (in 'SCI 0' format) remains as it is, although various annunciators may change at the 'SST'. Pressing the correction key will return the display to its 'pre-"FL"' format. Now try '50, XEQ "FL", SST' again, only this time press and hold the SST key to see '25 STO d', then release --the '25 STO d' remains in the display. Next, run '50 XEQ "FL"', and execute a 'STO d' manually (you will have to turn on USER mode to access the 'STO d' key assignment). This time the display will freeze as 'XROM 05,62'. Finally, do one more '50 XEQ "FL", then press 'R/S'. The flying goose is brought to roost! To move the little fellow to a different position, insert a few 'LBL 01' lines into "FL", delete the 'STOP' line, and try another '50 XEQ "FL"'.

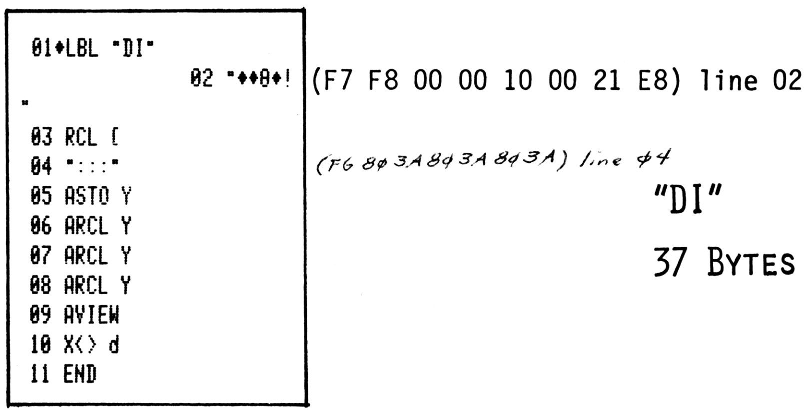
Judicious clearing of Flag 50 can also produce an interesting result. During a running program, if Flag 50 is clear, the display shows the flying goose. However, if a 'VIEW mn'

or an 'AVIEW' is executed, Flag 50 is set and the content of the specified register is displayed. A 'CLD' clears Flag 50, and returns the goose to the display. But if, during a 'VIEW', we clear Flag 50 without use of 'CLD', the processor restarts the default display, but steps the 'VIEW' display around instead of the goose. In effect, we can replace the flying goose with any other character, or string of up to twelve characters. The easiest way to accomplish this trick is to have the program execute an 'RCL d' just before the 'VIEW' line, i.e., at a time when Flag 50 is clear. Then immediately after the 'VIEW' is executed, the program does a 'STO d', restoring the pre-VIEW status of Flag 50:

```
01 "ABCD"     05 0
02 RCL d      06 LBL 01
03 AVIEW      07 SIN
04 STO d      08 GTO 01
```

When you run this routine, you will see "ABCD" stepping around the display. Lines 05-08 provide an endless loop to make our ersatz goose fly. Lines 01-04 can be included in any program, using any 12-character display to 'personalize' your running program. This trick is used in the 'Hangman' program of Section 6C to display the 'guessed letter' in a novel manner.

Here's a one question quiz to test your HP-41C ingenuity. There are 216 independent LCD segments in the HP-41C display--12 'starburst' characters times 14 segments = 168 segments; plus 12 'colon & comma' characters times 3 segments = 36 segments; plus 12 number and word segments in the annunciators; [168 + 36 + 12 = 216]. The question: How many segments can be 'on' simultaneously, with the 41C in 'standby' mode (i.e., on, but not running a program)? When you think you have the answer, and can demonstrate your number with an actual display, try running the routine "DI". If you can turn on more segments than "DI", you have learned more than this book can teach you!



7C. FLYING THE GOOSE BACKWARDS

To wrap up the exposition of synthetic programming, I'd like to give you one more example of 'they said it couldn't be done, but we did it (with synthetic programming, of course)!' As far as the user community was concerned, the best kept secrets of the HP-41C were the existence of the backward-facing goose character, and the means to display it. (Of course, there may be better-kept secrets, but they're still secrets.) To coax this shy creature into the display, we need one last new synthetic function, 'FIX 10'.

Numeric displays in the HP-41C are controlled by Flags 36-41. The 'FIX' format, for example, is established if Flag 40 is set and Flag 41 is clear. Digits Flags 36-39 control the number of digits that are displayed following the decimal point. The four flags constitute one hexadecimal digit; the number of digits displayed is equal to the value of that digit. Normally, the 'FIX' format is chosen from among 'FIX 0' through 'FIX 9', where the corresponding values of the four Digits Flags range from '0000' to '1001'.

With synthetic programming techniques, we can place any values we wish into the Digits Flags, extending the range of 'FIX' formats to include 'FIX 10' through 'FIX 15'. The display can't show more than ten digits, of course; in fact, 'FIX 11' through 'FIX 15' produce displays identical to that of 'FIX 0'. But 'FIX 10' does produce a new display format. For numbers with positive exponents, only the full ten mantissa digits are shown, with the exponent suppressed. Thus, for example, '1.234567891 E56', will show in 'FIX 10' as '1.234567891', whereas in 'FIX 9' it would display as '1.2345678 E56'.

There are several ways to place the HP-41C in 'FIX 10' format. As suggested above, we can directly set the value 10 (1010, or hex 'A') into the Digits Flags by storing an appropriate NNN into Register d. Another easy way is to set 'FIX 8', which sets Flag 36 and clears Flags 37-39, then use '38 XEQ "FL"'. Or, the synthetic function 'FIX 10' (code 9C 0A) can be used, either as a program line (it displays as 'FIX 0') or by assigning it to a user key (the prefix/postfix for "KA" is 156/10). Finally, we could place the NNN '0A 00 00 00 00 00 00' into Register X, and execute 'FIX IND X'.

'FIX 10' has a moderately useful practical application as a means to display only the mantissa of a number with a positive exponent. Unfortunately, the method isn't quite clean: if the exponent is 10, 11, 12, or 13, when taken modulo 14, some of the mantissa digits will be represented in the display by character from row 2 of the Byte Table rather than by the proper number characters from row 3. The most dramatic example is when the exponent is 13 (or 27, or 41, etc.) and Flags 28 and 29 are clear. In this case, only the first mantissa digit will display normally. '1.234567891 E13' will display in this mode as '1"#$%&'()¦ ' --still decipherable, if you have your Byte Table handy, but not very convenient.

The display of row 2 characters is by no means limited to the normal decimal digits '0' through '9'. Referring to the Byte Table, you will observe that bytes '2C', '2E', and '3A' each have two associated characters. In alpha displays, these bytes always show up as the characters ",", ".", and ":", respectively. But in number displays, the individual number digits are each represented by a character--lo and behold, in a number with the proper exponent, mantissa digits 'C' and 'E' are represented by the geese "≺" and "≻", respectively. The '3A' number character is the 'starburst'--as we found out in Section 5A.

To catch a couple of geese, set 'FIX 10' by any method, and clear Flags 28 and 29. Then:

	XEQ "CODE"	[CODE=?]
"0100E00C000013"		
	R/S	["⊼‾▩µ‾‾▩"]
	press ◄	[: ≻ ≺]

A maximum of nine goose characters can be made in a single display. The first character in the number display will always come from row 3--perhaps the most innocuous of these is the 'semicolon' character (byte '3B'). For example, the NNN '0B CC CC CC CC C0 13' will display as:

[> ≺ ≺ ≺ ≺ ≺ ≺ ≺ ≺ ≺]

At the risk of gilding the lily, let us return once more to "CODE", the quintessential synthetic program. What could be more fitting than making the goose fly backwards while "CODE" is running? Line 07 of "CODE" (line 11 if you modified "CODE" as suggested in Section 6F) is '07 "⊦ABCDEFG"'. The seven characters appended can be any characters--they might as well be a 7-byte NNN containing a backwards goose. Replace line 07 with:

```
07 "⊦▩▩‾‾‾‾▩"    F8 7F 0B C0 00 00 00 00 13
08 RCL d
09 FIX 0         9C 0A
10 CF 28
11 CF 29
12 CF 21         (necessary only with printer)
13 VIEW M
14 STO d
```

As the goose flies backwards around the display, he will be pursued by a ">". Perhaps that's a goose dropping???

* * *

On this charming note, we come to the end of this book. You've done a lot of work, and learned a lot about the HP-41C. Henceforth, 'synthetic programming' should mean 'normal programming' for you. You are now entitled to call yourself an 'NNU'--a 'Non-Normalized User'!

APPENDIX 1

NUMBER SYSTEMS

Every HP-41C user is familiar with the decimal number system, in which the fundamental quantity, or 'base' of the system is the number ten. Consider the following group of letters:

A B C D E F G H I J K L M N.

If we count the letters, we say we have 'thirteen' letters; as a shorthand notation for the number thirteen we write, in 'decimal':

13

The short notation possible with the decimal system arises from the repeated use of a limited set of symbols, i.e., the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, rather than having a different symbol for each possible number. When we write the double symbol '13', the value represented by each of the numerals depends upon its position in the '13'. That is, '13' means 'one times ten plus three'. Each numeral is multiplied by the base number raised to an integer power:

$$13 = (1 \times 10^1) + (3 \times 10^0)$$

Each numeral in a number is called a digit; we say '13' is a 'two-digit number'. For an 'N-digit number':

$$ab...de = (a \times 10^{N-1}) + (b \times 10^{N-2}) + ... + (d \times 10^1) + (e \times 10^0)$$

The digits a, b, ..., can take values from 0 through 9, i.e., up to one less than the base of the number system, ten.

There is nothing sacred to a mathematician, however, about the number ten: we can equally well choose any number as the base of a number system. For example, try eight: in 'base eight', usually called the 'octal number system' or just 'octal', the maximum value of a digit is seven. The number thirteen is represented by:

$$15_8 = (1 \times 8^1) + (5 \times 8^0) = 13_{10}$$

When more than one number system is in use, numbers with two or more digits should be written with subscripts to identify the base to which they are referred. We can write equalities like:

$$15_8 = 13_{10}$$

$$1295_{10} = 2417_8$$

The 'OCT' and 'DEC' functions in the HP-41C provide an easy means of converting numbers back and forth between the octal and decimal systems. It is important to remember that such conversions do not change number, but only the symbols used to represent the number. 7654_8 apples remain the same quantity of apples even if we write 4012_{10} apples.

Two other number systems are of interest in our study of the HP-41C. The first is the 'binary number system' in which the base is two. Only two symbols are needed, '1' and '0'. Our lucky number thirteen is represented in binary as

$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 13_{10}$$

The binary system is ideal for computer use, since each digit can take only two values, which is very easy to implement mechanically or electronically. Each digit, or 'bit' as binary digits are usually called, can be represented by the state of any kind of simple switch, where 'on' means '1', and 'off' means '0'. All calculator computations are carried out in

binary--the conversion to a decimal display is only for the convenience of the user. As a matter of fact, even the decimal numbers in the HP-41C are represented internally in 'binary-coded-decimal', or 'BCD', where each decimal digit is coded with four binary bits. For example, '13' is stored as

$$13 \longrightarrow 0001\ 0011$$

Each group of four bits can represent a number up to $15_{10} = 1111_2$, which leads us to the last number system that we need to consider--the 'hexadecimal' system, base sixteen. In hexadecimal notation, each digit can take a value from zero to fifteen, so the symbols '0' through '9' alone are insufficient. We add the symbols 'A' through 'F':

A = ten
B = eleven
C = twelve
D = thirteen
E = fourteen
F = fifteen

So, for example,

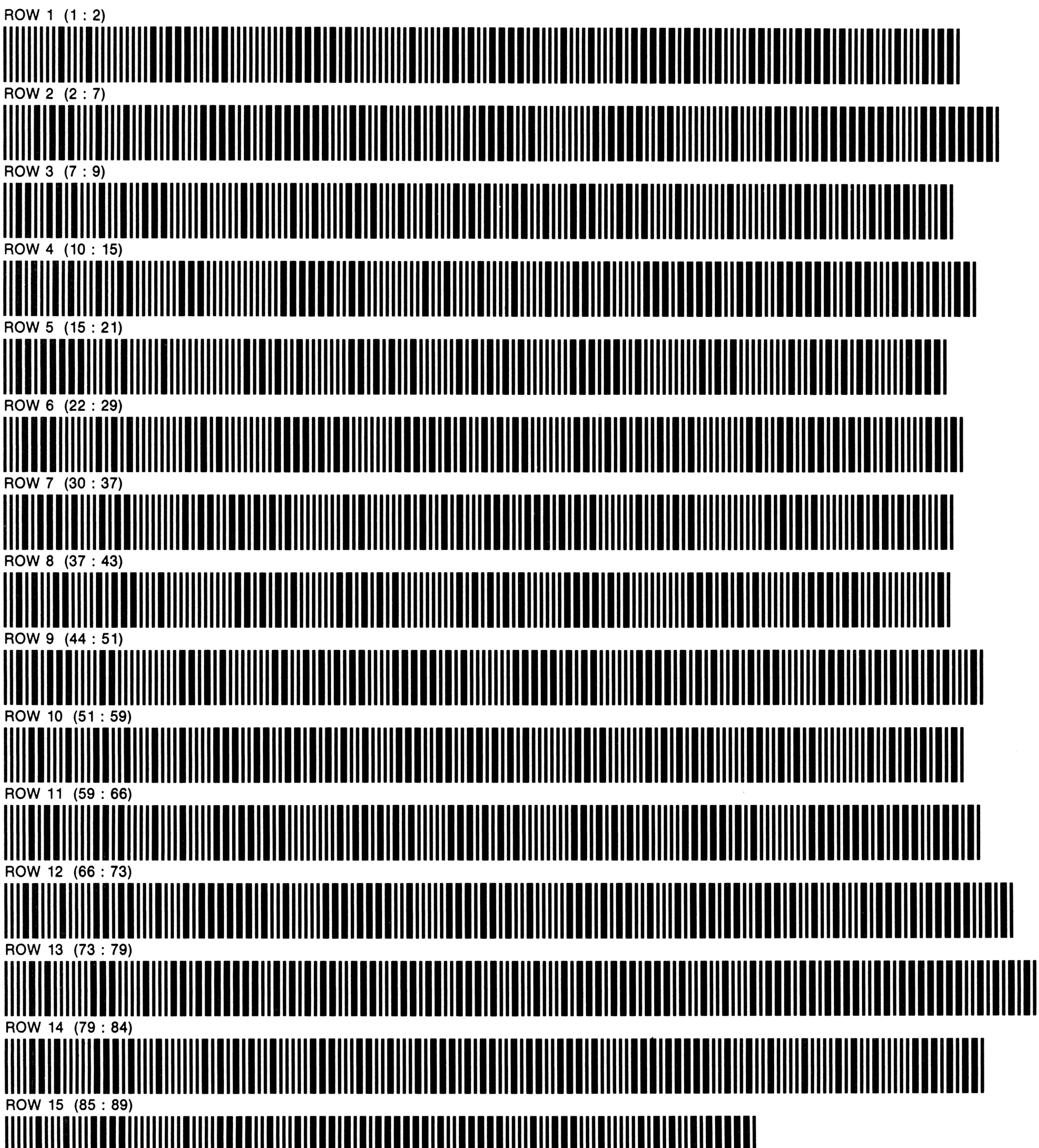
$$8A_{16} = (8 \times 16_{10}) + (10_{10}) = 138_{10}$$

$$1FF_{16} = 511_{10}$$

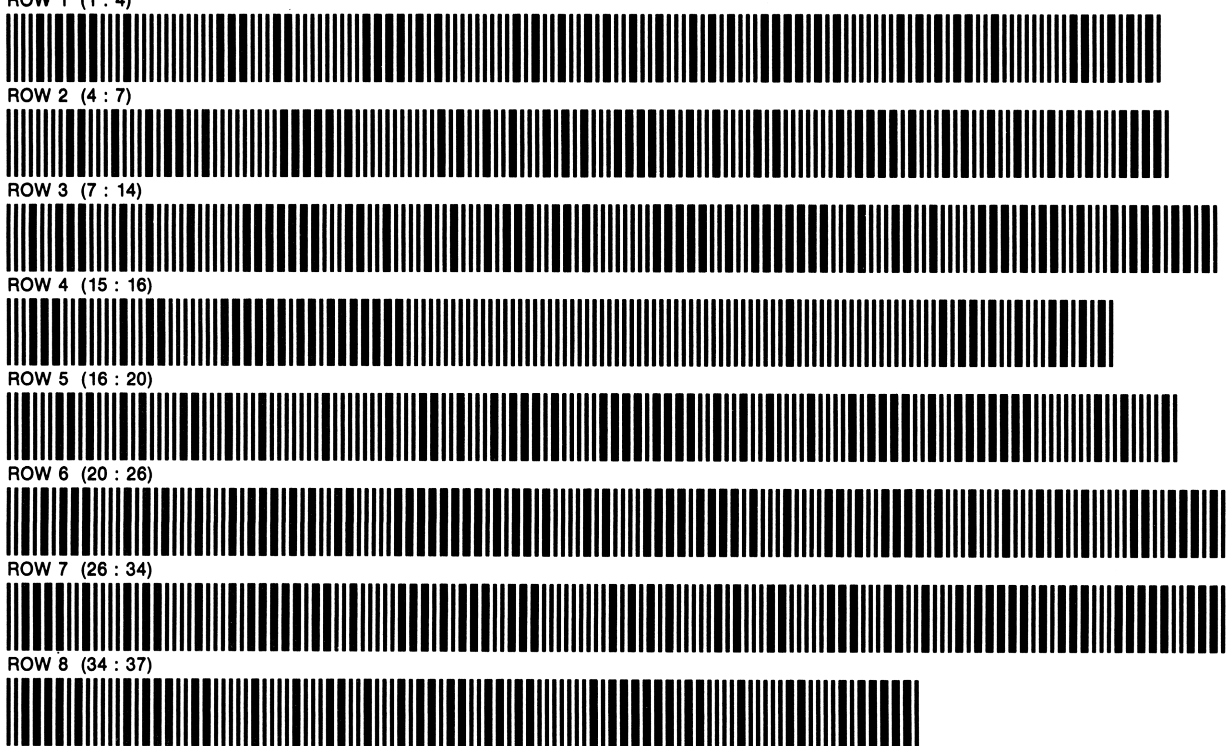
Notice that the values of single digit numbers are always unambiguous. It is only when two or more symbols are combined into a multi-digit number that we need a subscript to specify the number system in use.

CODE

PROGRAM REGISTERS NEEDED: 28



PROGRAM REGISTERS NEEDED: 15

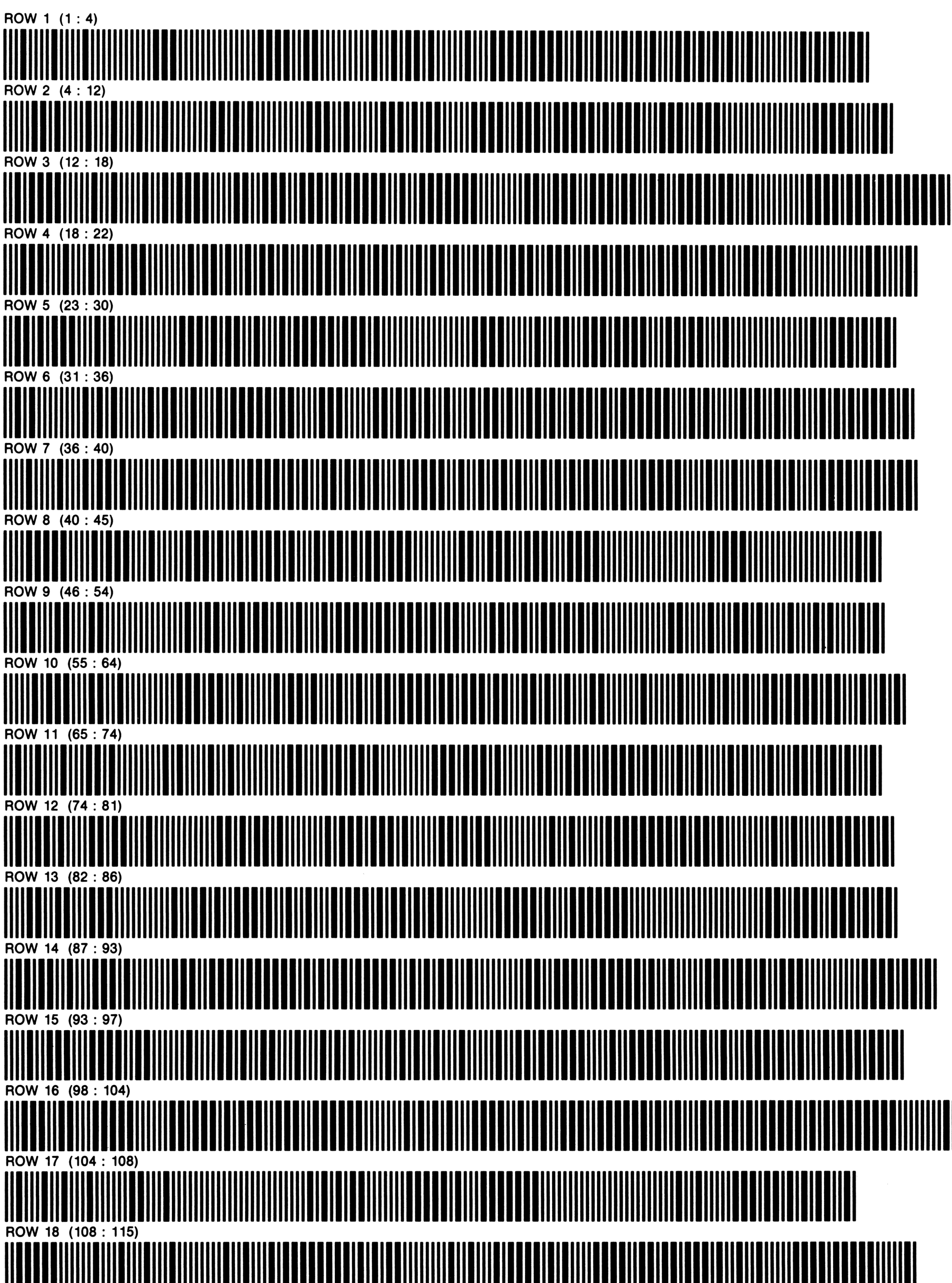


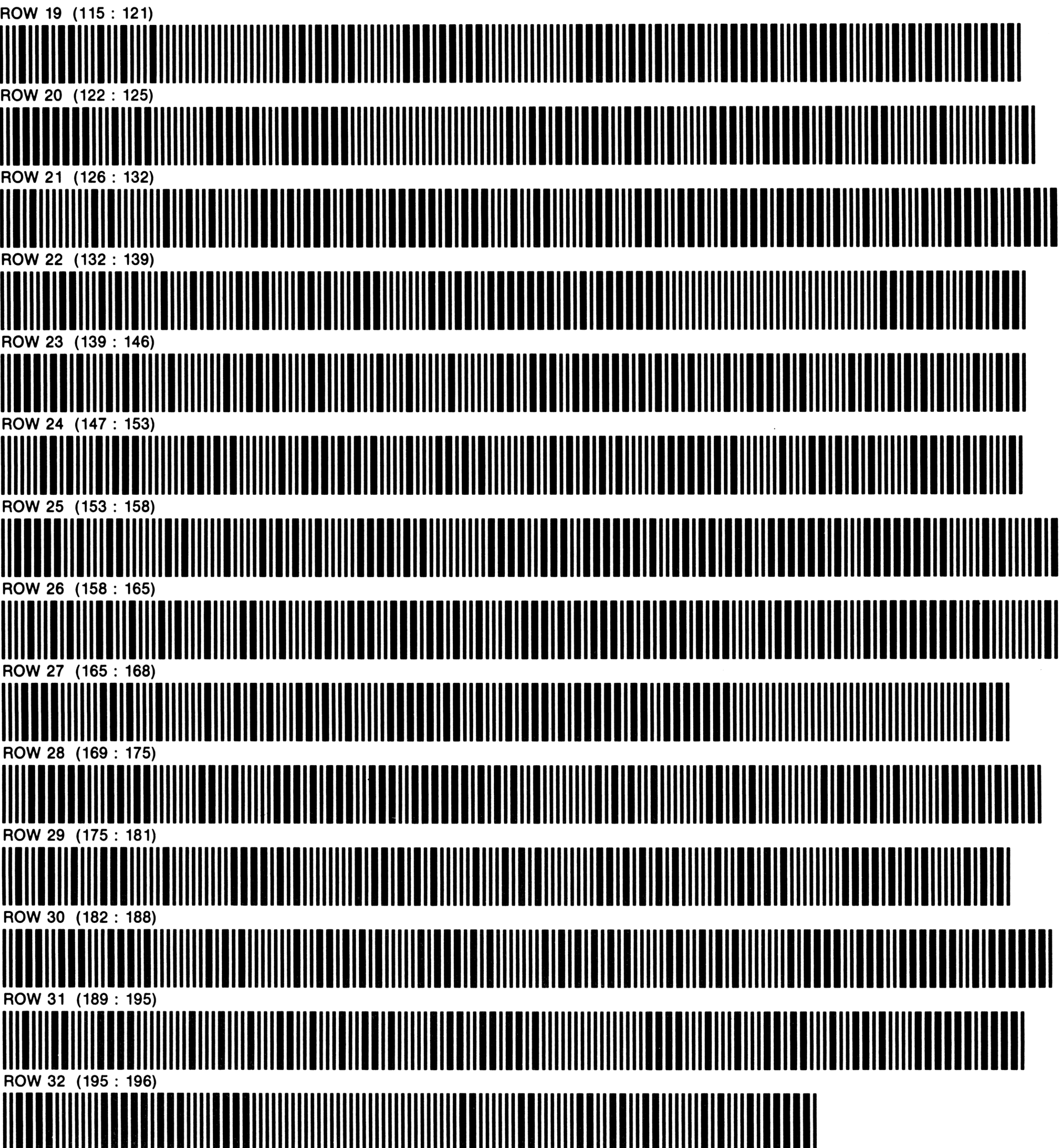
[Due to an oversight in the adaptation of the author's programs for barcoding, line 34 of "REG" is '34 "B2-"' rather than the '34 "REG"' described in the text. The "B2" refers to the PPC designation of the hardware bug ('Bug 2') in the HP-41C that led to synthetic programming. You may, of course, alter the line once the program is scanned into the calculator.]

KEY ASSIGNMENT PROGRAM

PROGRAM REGISTERS NEEDED: 59

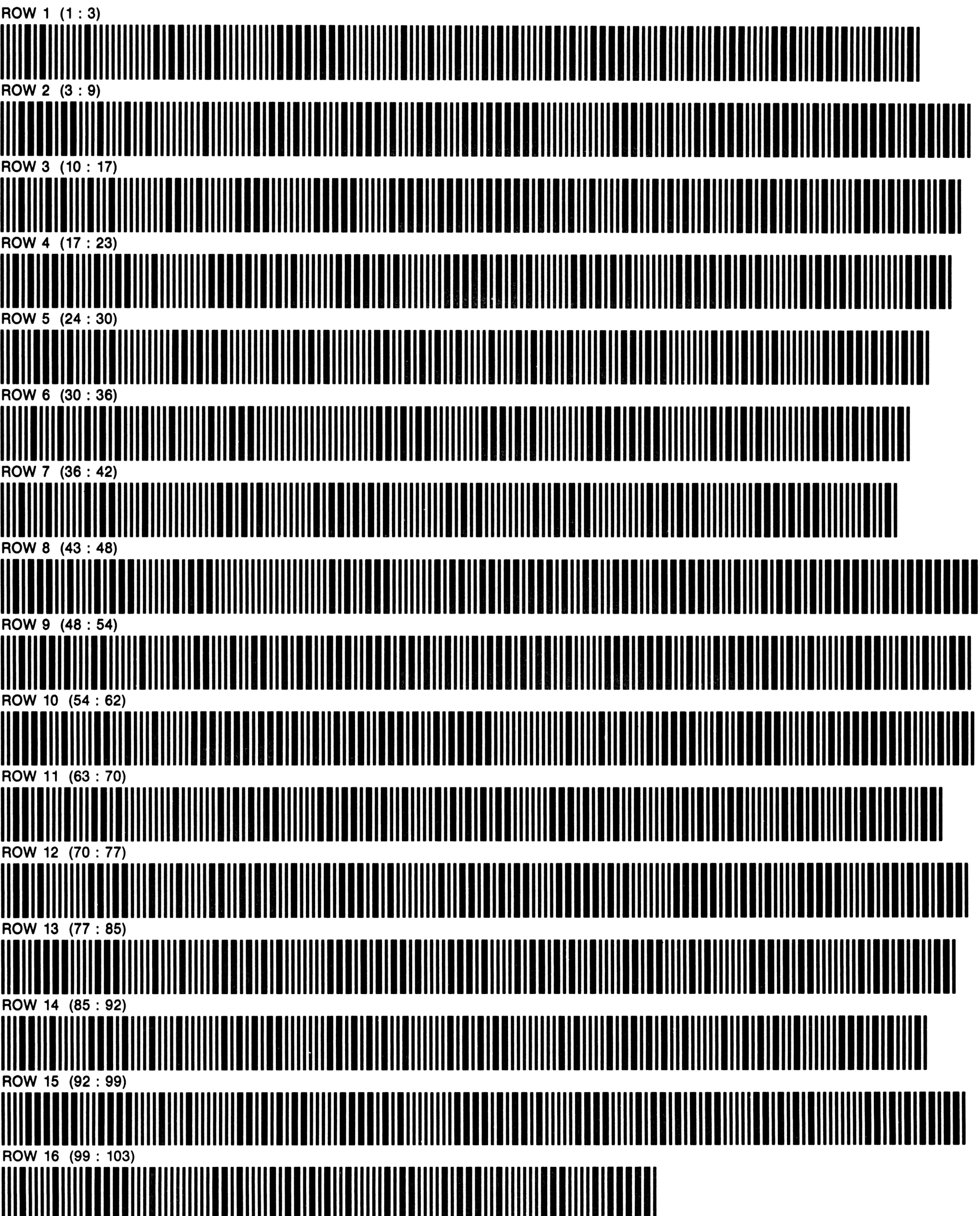
"KA" AND "EF"



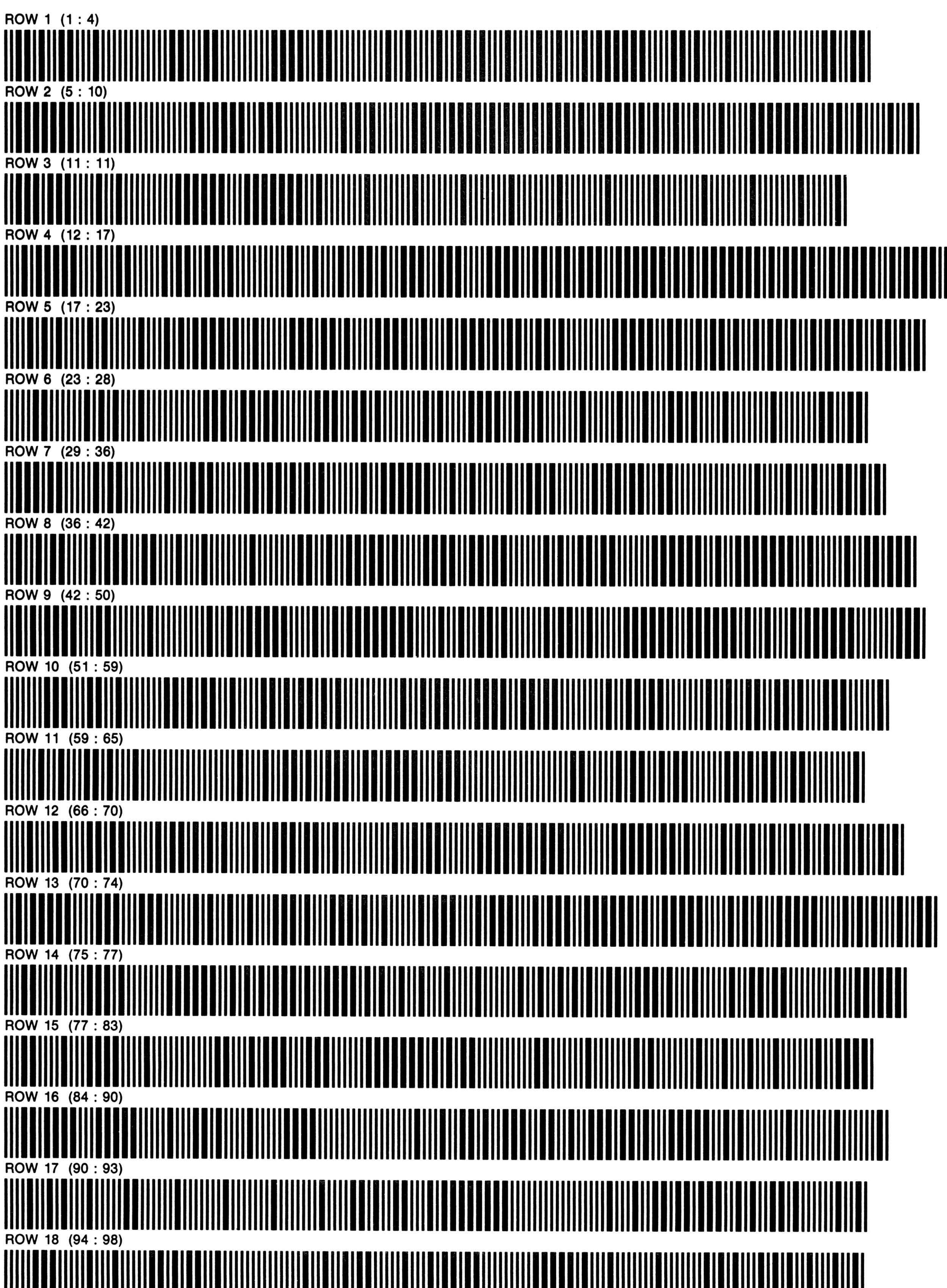

ROW 19 (115 : 121)
ROW 20 (122 : 125)
ROW 21 (126 : 132)
ROW 22 (132 : 139)
ROW 23 (139 : 146)
ROW 24 (147 : 153)
ROW 25 (153 : 158)
ROW 26 (158 : 165)
ROW 27 (165 : 168)
ROW 28 (169 : 175)
ROW 29 (175 : 181)
ROW 30 (182 : 188)
ROW 31 (189 : 195)
ROW 32 (195 : 196)

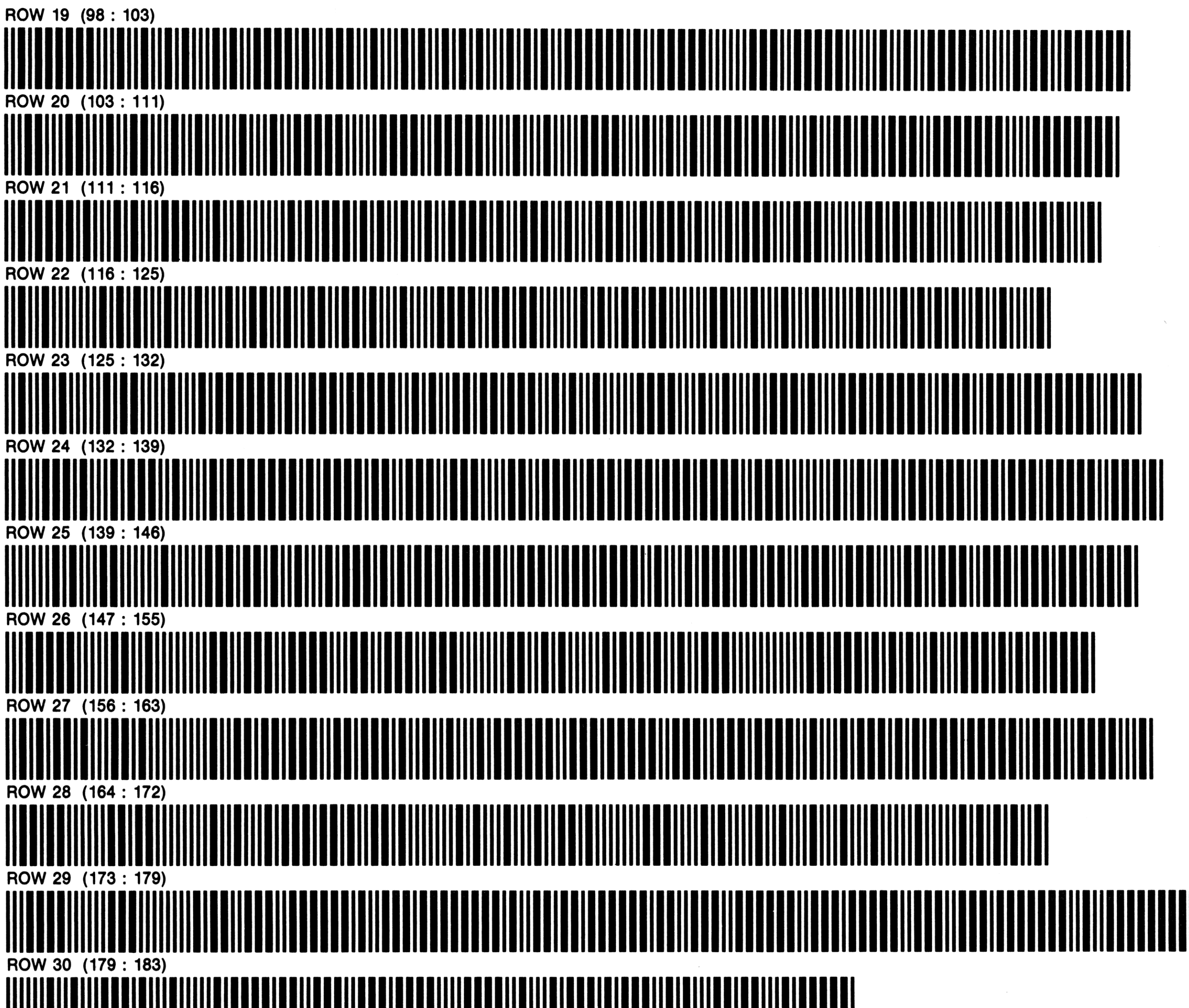
DECODE

PROGRAM REGISTERS NEEDED: 29



PROGRAM REGISTERS NEEDED: 56




ROW 19 (98 : 103)
ROW 20 (103 : 111)
ROW 21 (111 : 116)
ROW 22 (116 : 125)
ROW 23 (125 : 132)
ROW 24 (132 : 139)
ROW 25 (139 : 146)
ROW 26 (147 : 155)
ROW 27 (156 : 163)
ROW 28 (164 : 172)
ROW 29 (173 : 179)
ROW 30 (179 : 183)

APPENDIX 3

THE BARCODE CHARACTER TABLE

This book represents the results of a year of experimentation with synthetic programming on the HP-41C. At this time, it is clear that the Wand will be a powerful tool for synthetic programming, largely due to the work of Jacob Schwartz (a PPC member who is also responsible for the eminently sensible layout of the Wand Paper Keyboard). Most of the wand techniques for synthetic programming are still in their infancy, but a few examples will convince you of the promise of this device.

Always use a protective sheet over the barcodes while you are scanning with the wand!

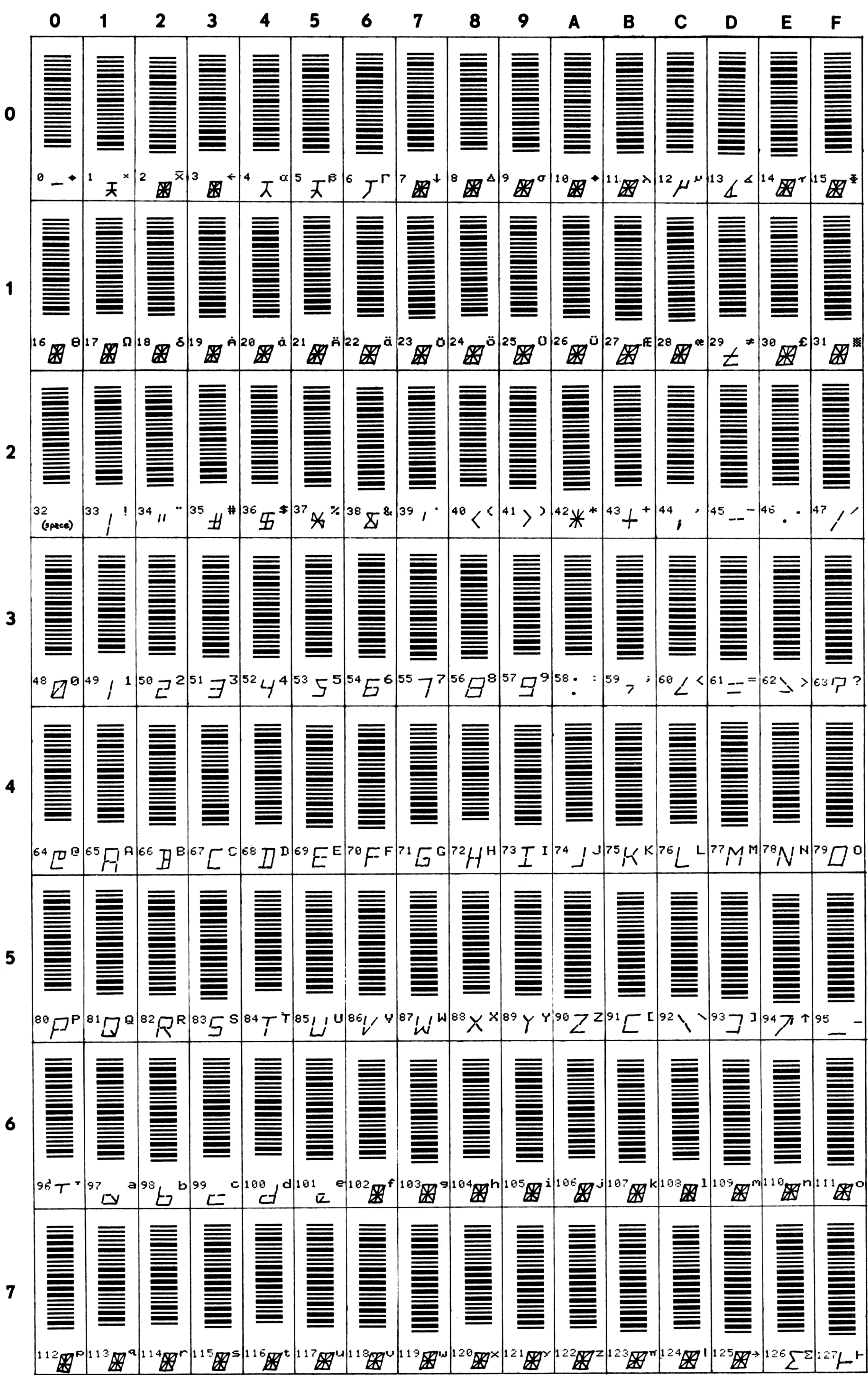
First, you can add to your 'Paper Keyboard' barcodes for the byte jumper and the Q-loader:

BYTE JUMPER:

Q-LOADER:

The 'Bar Code Character Table' on the following page was supplied by Jacob Schwartz. Any non-standard alpha character from the upper half of the Byte Table can be added directly to an alpha string, either in the alpha register, or in a program text line, by simply scanning the barcode in the appropriate location in the chart.

The Barcode Character Table



ERRATA: Synthetic Programming on the HP-41C

1. Page 15, par. 6, 4th line: '9' means packed
2. Page 68: the 40-byte 'normal' steps omitted from par. 1 of sect. 6E are:

01 CF 13	07 CF 13	13 CF 13
02 "B"	08 "D"	14 35
03 ACA	09 ACA	15 ACCHR
04 SF 13	10 SF 13	16 "7"
05 "IG"	11 "EAL"	17 ACA
06 ACA	12 ACA	18 ADV

3. Page 48: Line 125 of "KA" is 'F3 0C 0C 0C'
4. The following synthetic codes were omitted:

Page	Program	Line #	Code
67	"AL"	18	F3 01 01 01
67	"AL"	25	F2 01 01
77	"TONE"	13	F2 7F 9F
77	"TONE"	24	F2 9F 7F
80	"DI"	04	F6 80 3A 80 3A 80 3A

5. In all programs, the symbol "♦" in program listings indicates byte '00'

KEYBOARDLOCKY

'Twas octal, and the synthetic codes
Were scanned without a loss.
In and out of PRGM mode,
Byte-jumpers nybbled the CMOS.

"Beware Ø STO c, my son,
The MEMORY LOST, the keyboard lock.
Beware the NNN, and shun
The curious phase 1 clock."

He took his black box codes in hand,
Long time the backwards goose he sought;
The secret beast from Aitchpee land--
All searches came to nought.

In demented thought he stood, and then:
The goose, with LCD's alight,
A leap for every LBL 10,
Came honking left-to-right!

STO b! STO d!, and RCL P!
His keyboard went clickety-clack.
With the proper code in number mode
The goose came flapping back.

"And hast thou found the phantom fowl?
Come to my arms, my binary boy.
Let Corvallis hear us howl
As we chortle in our joy!"

'Twas octal, and the synthetic codes
Were scanned without a loss.
In and out of PRGM mode,
Byte-jumpers nybbled the CMOS.

--Apologies to Lewis Carroll