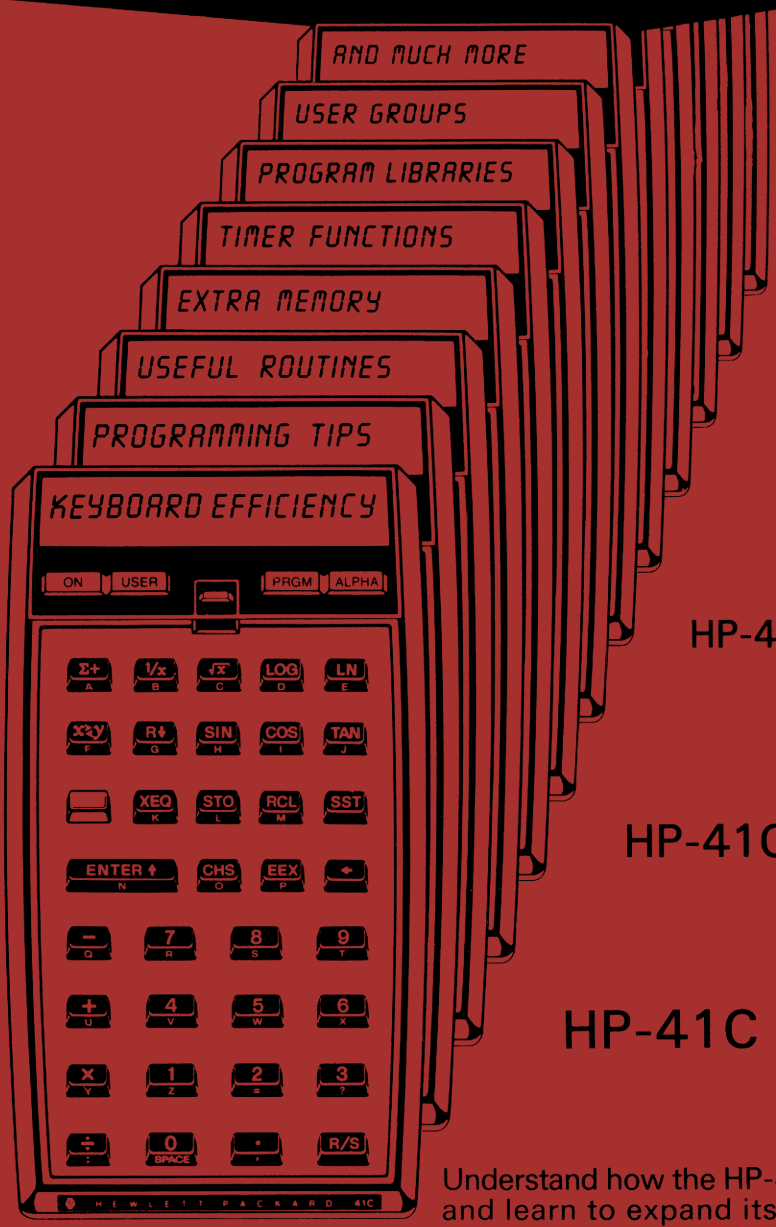
EXTEND YOUR HP-41



HP-41CX

HP-41CV

HP-41C

Understand how the HP-41 works and learn to expand its abilities

by W Mier-Jędrzejowicz

E X T E N D

your HP-41

by
W.A.C. Mier-Jędrzejowicz, Ph.D.

"Extend your HP-41" by W.A.C. Mier-Jędrzejowicz, Ph.D.

Second printing November 1985

ISBN 0 9510733 0 3

Library of Congress Card Catalogue Number : 85-062787

Published by:
W.A.C. Mier-Jędrzejowicz
40, Heathfield Road
London W3 8EJ
United Kingdom

United States publishers:
SYNTHETIX
P.O. Box 1080
Berkeley, CA 94701-1080
U.S.A.

Write to the above addresses for price information.
Please enclose an addressed envelope for reply.

U.K. edition printed by:
Rank Xerox Copy Bureaux
30/34 High Street
Slough
Berkshire
United Kingdom

Copyright 1985, W.A.C. Mier-Jędrzejowicz

FOREWORD

When is a calculator not a calculator? -- When it becomes a computer!

The Hewlett-Packard HP-41 works just like a pocket calculator and looks like one, yet it can also be used as a powerful pocket computer. This book takes you on a trip through the ins and outs of the HP-41, beginning with simple calculations, and ending with sophisticated programs that let you use your HP-41 as a computer with features far more powerful than those described in the HP-41 manuals.

Why this book? As pocket calculators and computers have become more powerful their users have come to be faced with a new dilemma. When out in the field or in a laboratory and with only a pocket computer to hand,they ask themselves "Should I try to work my problems out on the pocket computer here, or should I go back to the office and use a larger computer?" The answer is "I'll have to go back and use my office computer" all too often just because users do not realise how much can be done on a pocket computer. This is particularly true of the HP-41 which at first sight looks like an ordinary pocket calculator. Yet the HP-41 is extremely versatile, more powerful than some pocket-sized BASIC computers, and can be used with many accessories including interfaces to HP and IBM personal computers. The HP-41 can be carried around and can serve as a pocket calculator, yet its uses can be greatly extended whenever necessary, in the field or in the laboratory. The purpose of this book is to show how any HP-41, not just the HP-41C but also the HP-41CV or HP-41CX (including the latest versions, see Appendix E) can be extended by an understanding of its operations, by a knowledge of special techniques and by the use of various accessories.

The book is divided into five parts with each part subdivided into chapters and sections. Part I is a general introduction and also includes advice for the complete beginner. Part II is made up of additional advice for users with some experience of an HP-41. Part III contains suggestions for advanced programming, including advice on selecting and using plug-in modules, particularly the Time and Extended Functions modules. Part III

also covers other accessories, and ways to obtain specialized programs instead of writing them yourself. Part IV provides information on non-standard programming techniques, information that is not provided in HP manuals but allows you to greatly extend the use of your HP-41, particularly if you have an Extended Functions module or an HP-41CX. This part includes information about equipment that allows the user to program the HP-41 in its own internal "machine language". Part V comprises a collection of Appendices, among them a list of other books on the HP-41, a list of known faults (bugs), and a list of HP-41 system flags and barcodes.

Copyright: The material in this book is copyright and may not be reproduced in any form, either in whole or in part, without the written consent of the publisher and author, except that the programs contained herein may be reproduced for personal use, and short extracts may be quoted for review purposes.

Disclaimer: All material in this book is published without representation or warranty of any kind. Neither the publisher nor the author shall have any liability, consequential or otherwise, arising from the use of any material in this book.

AD MAIOREM DEI GLORIAM

Dedicated to my Father who introduced me to the joys of multiplication tables and of computing, and who is still suffering the consequences and to my Mother who made it possible.

Acknowledgements: Many people deserve to be thanked for their help or contributions, and I fear that I may have omitted some names, so first of all "Thank You" to all the calculator enthusiasts, named and unnamed, who have provided information that is used by others. I must especially thank the members of calculator user groups, and in particular Richard Nelson who founded the original user group PPC and is now running the user group CHHU, and David Burch who set up and is still running the UK club, HPCC. Many thanks too to the following who have given me help and advice: Rob Woodhouse, Dave Bundy, Brian Steel, George Ioannou, Jeremy Smith, Terry Stancliffe; also Frank Wales and Graeme Cawsey (and Bill Regussy) and Julian Perry. A special thank-you to Bruce Bailey who made many helpful suggestions and provided a very useful program. Grateful thanks to Joanna Tobiasiewicz and David Burch for their help in typing and checking the text, to Rabin Ezra who helped with some of the Tables, and to Richard Rijnbeek for proofreading. My thanks as well to those who have allowed me to use equipment for this book: Geoff Hall, Robert Lewis, Brian Steel, Metyclean Ltd. and of course Hewlett Packard Ltd. and their employees.

Keith and Catherine Jarett have worked hard to make the second printing possible. An extra thank you to them and to Bruce Bailey for suggesting corrections and changes for this printing.

CONTENTS

Page

Table of Figures xiii
Table of Tables xiv

PART I - Fundamentals

Chapter 1. Introduction 3
1.1 The purpose of this book 3
1.2 This book and other books 5
1.3 Sources of information 7
1.4 Notes for experienced users 8

Chapter 2. About the HP-41 11
2.1 Overview 11
2.2 Some HP-41 history 11
2.3 The layout of the HP-41 14
2.4 Batteries and power 16
2.5 ROM, RAM and Continuous Memory 18
2.6 HP-41C, HP-41CV, HP-41CX 19
2.7 Some accessories 20

Chapter 3. Definitions and problems 29
3.1 Using the HP-41 29
3.2 The display and audible signals 29
3.3 Toggle keys, Keyboards and modes 32
3.4 Parameters, arguments and RPN 35
3.5 Instructions and functions, routines and programs, catalogues 38
3.6 Some common problems 44

CONTENTS

Page

PART II - Calculating and programming from scratch

Chapter 4. Starting from the keyboard 63
4.1 Turning on, and what to do if you cannot 63
4.2 Look after your stack 69
4.3 Make use of Alpha 78
4.4 Set your status 85
4.5 LASTX; corrections and constants 89
4.6 Efficiency: keyboard operations vs programming 93

Chapter 5. Know your functions 97
5.1 Choose your weapons 97
5.2 General mathematical functions 97
5.3 Times and angles 102
5.4 Summations and statistics 104
5.5 Indirections 106

Chapter 6. Something about programming 111
6.1 A simple program 111
6.2 Using labels to identify programs and routines 112
6.3 Searching for labels with GTO and XEQ 114
6.4 Finding your place, compiled addresses, CAT 1 and Indirect Execution 121
6.5 Checking, correcting and changing a program 126
6.6 Watching program execution 132
6.7 Using tests to control program execution 134
6.8 ISG, DSE and NOPs 137
6.9 Asking questions and displaying results 142
6.10 Using subroutines and structuring programs 149

CONTENTS

Page

Chapter 7. Some example programs 157
7.1 Hyperbolics and inverse hyperbolics 157
7.2 Review registers 158
7.3 Integration with infinite limits 161
7.4 Random numbers 166
7.5 Complete arc Tangent - ATAN2 171

PART III - Extended Programming

Chapter 8. More about memory 177
8.1 Space, time and numbering 177
8.2 Contents of RAM memory 181
8.3 The layout of RAM and ROM 184
8.4 Peripherals, the display and the CPU 194
8.5 Program instructions in RAM 196
8.6 Key assignments of instructions 210
8.7 41C or 41CV ? 216
8.8 Space saving tips 217

Chapter 9. Time functions 227
9.1 A growing system 227
9.2 Times and dates 228
9.3 Using the stopwatch 232
9.4 Using the alarms 237
9.5 Additional HP-41CX time functions 246

Chapter 10. Extended Functions 249
10.1 Extending your control over the HP-41 249
10.2 Alpha string control 250
10.3 Moving data and flags 259
10.4 HP-41 status control 267
10.5 Additional HP-41CX functions and features 273
10.6 Indirect comparisons 278

CONTENTS

Page

Chapter 11. Extended Memory 289
- 11.1 What is Extended Memory ? 289
- 11.2 Creating and deleting files 290
- 11.3 File pointers 292
- 11.4 Using data files 295
- 11.5 Using text files 300
- 11.6 Using program files 308
- 11.7 Checking the contents of Extended Memory 310
- 11.8 Additional HP-41CX Extended Memory functions 311
- 11.9 The HP-41CX Text Editor 313
- 11.10 Generalised key assignment program - GASP 316

Chapter 12. Peripherals and plug-in modules 327
- 12.1 More programs, more equipment 327
- 12.2 Printers and display devices 327
- 12.3 Card Reader and Wand 335
- 12.4 HP-IL and other peripherals 344
- 12.5 Plug-in modules and XROM conflicts 349
- 12.6 Application program modules 354
- 12.7 Utility program modules 367
- 12.8 System Extension modules 378
- 12.9 Diagnostic and service modules 390

Chapter 13. Advanced programs and user groups 393
- 13.1 Advanced programs ? 393
- 13.2 Books and journals 393
- 13.3 Adapting programs from other calculators 395
- 13.4 Hewlett Packard Solutions books 397
- 13.5 User Libraries and user clubs 398
- 13.6 The benefits of belonging to a user club 400
- 13.7 Buying and ordering programs 405
- 13.8 Writing advanced programs yourself 406

CONTENTS

Page

PART IV - Synthetic Programming

Chapter 14. Introduction to Synthetic Programming 411
14.1 How many bytes make a million ? 411
14.2 Non normalised numbers, tumble dryers and cement mixers... 412
14.3 Your first synthetic tool 415
14.4 Using the Byte Grabber 424
14.5 The Byte Table and the status registers 431
14.6 Status registers M, N, O, P 438
14.7 Register Q 444
14.8 Register d, the flag register 446
14.9 Register c, a vital register 458
14.10 Registers b- and e; making synthetic key assignments 465
14.11 Registers a and b - the current address and RTN stack 473

Chapter 15. Using Synthetic Programming 481
15.1 When should Synthetic Programming be used ? 481
15.2 More key assignments 482
15.3 Byte Grabbers, Byte Jumpers and program analysis 492
15.4 Addresses and multi-byte instructions 498
15.5 Four examples 504
15.6 Synthetic text and Q-loaders 514
15.7 Other SP bits and pieces 518
15.8 Do's and don'ts of Synthetic Programming 527

Chapter 16. Synthetic Programming and Extended Functions 535
16.1 New tricks for old 535
16.2 Alpha register operations 536
16.3 Flags and numbers 538
16.4 Registers, keys and programs 543
16.5 Understanding Extended Memory 546
16.6 Manipulating Extended Memory 555

CONTENTS

Page

16.7 Understanding buffers and a programmable PACK 563
16.8 A programmable PRP 569
16.9 Non-normalising recalls and RAM editing 579

Chapter 17. Where next ? 589
17.1 A better machine ? 589
17.2 Personalised software and keyboards 589
17.3 Hardware modifications 591
17.4 Black boxes and M-code 594
17.5 Missing functions 598

PART V - Appendices

Appendix A: Books and journals for the HP-41 603

Appendix B: Sources of information and equipment 611

Appendix C: HP-41 system bugs, nasty surprises, and ROM revisions 615

Appendix D: HP-41 system flags 643

Appendix E: Recent changes to the HP-41 series and new products 653

Appendix F: Barcodes 655

Index 673

List of programs .....Inside back cover

TABLE OF FIGURES

Figure	Title	Page
2.1	The I/O Ports	14
3.1	The HP-41 Display	30
4.1	First Stack Analysis Form for (a+2)*a	71
4.2	Second Stack Analysis Form for (a+2)*a	72
4.3	Stack Analysis Form for (a*b + c/d)*sin(e)	74
8.1	Different ways of seeing a number	180
8.2	A register containing a number	182
8.3	A register containing a text string	183
8.4	HP-41C Random Access Memory (RAM) Layout	185
8.5	HP-41 with Additional RAM	188
8.6	HP-41 Read-Only Memory (ROM) Layout	190
8.7	Layout of a Key Assignment Register	212
14.1	The HP-41 Status Registers	439
14.2	Register c contents	458
14.3	Moving the curtain	462
14.4	The key assignment flags	465
14.5	Registers a and b	474
16.1	HP-41 RAM memory, including Extended Memory	547
16.2	Extended Memory link register contents	549
16.3	Extended Memory file status header	551
16.4	Buffer header structure	563

TABLE OF TABLES

Table	Title	Page
8.1	Byte Table of HP-41 Prefixes	197
8.2	Byte Table of HP-41 Postfixes	198
8.3	The HP-41 Display Characters	206
12.1	Flag Settings for HP-IL print mode	335
12.2	XROM numbers used for plug-ins	352
14.1	The Byte Table (first half -- rows 0 to 7)	432
14.2	The Byte Table (second half -- rows 8 to F)	433

PART I

Fundamentals

CHAPTER 1 - ABOUT THIS BOOK

1.1 The purpose of this book.

What do you get when you buy an HP-41? First of all you get the HP-41 itself with a pocket guide. You also get the manuals whose size alone suggests that the HP-41 is more than just a pocket calculator. What you most certainly do not get is a friend to help you understand the HP-41 and use it to full advantage. Most owners get along as best they can without such a friend. Some have friends or colleagues who can help them, others ring Hewlett-Packard to try to get the advice they need. A lucky few find a user group whose members understand their problems and are willing to help.

Certain questions and answers come up time and again at user groups. What should I do if it locks up and refuses to respond to the keyboard? How can I type in this program if there is no XROM key on the keyboard? Can you suggest a good program for my work? A single collection of answers to these questions would be valuable for new members of user groups and for people who cannot or will not join a user group.

This book contains just such a collection of answers, along with other information. It has grown out of the monthly meetings in London of the user group HPCC, formerly called PPC (UK), but the questions and answers should be of interest to HP-41 users everywhere. Even those owners who have long been members of a user group will find useful reference material, new tricks, and some interesting programs here. If you are thinking of buying an HP-41 then Part I of the book will tell you what you can get and will help you decide if that is what you really want.

Some questions can be answered with a straightforward "press this button" or "look it up in such-and-such a book". Others can only be answered through an explanation in detail of how the HP-41 works. Users who take the trouble to understand their HP-41 will be able to write better and faster programs, maybe even to write and sell specialist programs. This

book will therefore give detailed explanations of many HP-41 operations.

You may prefer to read the details after first looking quickly through the whole book. No book can hope to answer all questions so there will be places where you will be referred to another book or to a magazine article. At times you will be advised to go to a user group meeting, since some questions are impossible to answer in a book. Books which are referred to are included in the reference list in Appendix A. Maybe you should look through this Appendix right now and perhaps buy a different book! It is most important to read the HP manual, but some people find this hard to read or maybe they have a second-hand HP-41 without a manual. This book will do as an introduction if you do not have the manual but you should really read that too.

The book has five parts arranged to help the reader use it for reference as well as to read through it. Part I covers the fundamental information required by a reader, particularly if he or she is new to the HP-41. It also explains some of the computer jargon that is used to describe the HP-41, and it even covers some HP-41 history. Part II goes through normal HP-41 operations. It does not give all the information available in HP manuals but it includes further details, explanations and suggestions that are not provided in them. Part III goes on to extended programming; this means using the HP-41 with plug-in modules and with peripheral devices such as printers, it also means writing or obtaining programs more complicated than you would normally write yourself. Many users see advertisements for HP-41 accessories but do not know how much use these are. Some users even buy equipment that is completely unsuitable and this part of the book may save readers from the same fate. Few owners know that some extremely useful accessories for the HP-41 are made by other companies, some of these are covered in Part III.
Part IV covers the use of non-standard functions which means using instructions that are not described in the HP-41 manuals. These functions can be used on any HP-41 and do not require any extra equipment. Their use (referred to as Synthetic Programming) greatly extends the capabilities of the HP-41. This part ends with a short description of further ways to

stretch your HP-41, including the use of the HP-41 internal "machine language". Part V contains Appendices with information that is best collected in one place for easy reference. This includes an Appendix on the latest changes to the 40 series of calculators, and on how these changes relate to what is written in the rest of the book. An Index is provided but if you are looking for a particular piece of information you may find it more quickly by checking through the Contents. Each part of the book is divided into chapters with the chapters divided into sections. Section numbers are used when one item refers to another.

By reading this book (and getting to understand the HP-41 better in other ways) you will help yourself get more from the HP-41 by extending the range of things you can do with it. Extending your HP-41 can mean many things - using its memory better or adding to its memory, using the functions more efficiently, recognising which peripheral devices will help you most - all this and more will be covered. If you are a professional who uses the HP-41 for engineering or scientific work, you should find advice that will help extend the usefulness of your HP-41. It seems reasonable then to call the book "Extend your HP-41", for that is its overall purpose.

1.2 This book and other books.

As stated above this book contains information that will help you understand the HP-41. You may want to buy or borrow some of the books and journals referred to for further details, see Appendix A. A few deserve to be mentioned right away. First of all, you should make sure you get and read the HP manuals. They vary in style and quality but it is unwise to use any product without knowing what the maker has to say about it. The HP-41CX manuals are particularly good and you may want to have a look at them even if you do not have an HP-41CX.

Other books come in three kinds. There are specialist books on certain features of the HP-41, such as the Extended Functions or Synthetic Programming. Secondly there are specialist books on particular subjects, such as navigation or forestry. Thirdly, there are general information or

reference books such as the HP-41 and HP-IL System Dictionary or "Tips and Routines for the HP-41". The book you are reading now is definitely of the third kind. It contains some of the same information as other books but does not replace the lot. In particular "Tips and Routines" by John Dearing is well worth having, and the "Synthetic Programming Quick Reference Guide" by Jeremy Smith provides a very useful pocket-sized reference.

This is primarily a programming book. It gives advice on solving your problems by means of programs or keyboard calculations. The peripheral devices have been left until Part III for this reason, and even then they are treated mainly as aids to programming. In other words, this is more of a software (programming) book than a hardware (nuts and bolts; and chips) book. Some hardware advice, even on opening up the HP-41 and speeding it up has been provided, but this too is designed to help in programming. If you want detailed advice on rebuilding your HP-41, or using it as a doorstop, then you need a book that is more concerned with the HP-41 hardware.

Now a few words about the way some information will be presented here. A new word or idea will be printed in **bold** characters at the place where it is explained. A short set of steps to be done on the HP-41 will be printed with each step appearing as it is seen on the keyboard and separated from the next step by a comma. For example the steps taken to work out the logarithm of six would be printed as 6, LOG which means press the key marked 6 then press the key marked LOG. Longer sets of steps and programs will be shown as they appear on an HP-41 printer. You do not need a printer to use this book, but if you do not have one then you should check through Section 2.7 which shows what a program looks like when it is printed. The **up-arrow** symbol, ↑ will be used to mean "to the power of" in arithmetic expressions, so that X↑2 means "X to the power of 2", or "X squared". The HP-41 displays X↑2 and Y↑X in the same way.

1.3 Sources of information.

The information in this book has been gathered from various sources and these deserve to be mentioned here. First of all, some important information from HP manuals has been repeated here albeit in a different form. HP also used to publish a quarterly journal "Key Notes" which gave much useful information including corrections, and suggestions from users. Some information from "Key Notes" will be mentioned.

A great deal of information has come through user groups, specially the oldest group, PPC, through its journals, through books published by its members, and through discussions with individuals. The books and journals are mentioned in Appendix A, many members of user groups have provided information and I thank them all. Sometimes I have thanked particular individuals for providing ideas or programs, but many bits of information have come from more than one source or from a source I cannot identify.

Information that has been published is subject to copyright and it is neither proper nor safe to copy text from other people's books. It is legal to publish short extracts for review purposes and in a few places I do mention a particular book or article and include a short extract to show what can be found in that particular text. In general though, books and articles are mentioned as sources of further specialist information on a given subject.

Of course some of the information has come from the author too. I have used my HP-41 for serious work, particularly in Space Physics, and also for playing about with in my spare time. Some of this has resulted in useful routines, or in ideas for improved control of HP-41 operations; these have formed a considerable part of the book, including some new uses of the Extended functions for Synthetic Programming.

1.4 Notes for experienced users.

New HP-41 users should find a lot to interest them in this book, but what is there in it for more experienced users? Those experienced users who are not members of a user group or who have only recently joined one will find much information that is not available outside user groups, and some readers may even decide to join such a group. The generalised key-assignment program in Section 11.10 is an example of a topic that should interest these readers; it provides a convenient introduction to the non-standard HP-41 functions without itself using any such functions.

It has not been my purpose here to publish a list of programs for use in specific subjects; readers who are experienced HP-41 users will no doubt have their own programs already. I have included a few programs for mathematical or engineering calculations, but these are provided as examples and my main purpose is to give answers to questions, and to make suggestions as to how the HP-41 can be used more effectively. To this end the book provides a mixture of new ideas and of information gained from the experience of people who wish to use the HP-41 more efficiently and to greater advantage.

If you are an experienced user but your normal use of the HP-41 has been confined to the instructions in the manuals you may be surprised to find that additional powerful instructions have been discovered and exploited. Even the HP-41 internal machine language (M-code) can be used for programming. You can compare the use of the normal instruction set to programming in BASIC on a home computer. The use of the additional instructions called "Synthetic instructions" is like using BASIC with PEEK and POKE instructions to access the operating system. The use of M-code is equivalent to programming a home computer in its own machine language.

Those readers who are experienced members of user groups will know already about Synthetic Programming (SP) and about M-code. This is not a book about M-code, although that is mentioned, so what use is it to an old hand? It provides two things for these users; firstly a review of normal HP-41

programming including new tips, and secondly various information on SP, particularly new ideas on the use of SP with the Extended Functions, illustrated by examples. It also contains reference material such as a list of bugs. It is a collection of ideas and suggestions concerning the HP-41, all in one place, so that owners of this book may be able to avoid looking through a stack of old journals every time they need to check something. If you do not think you need a book like this then pass it on to a new member of your local user group who asks all those silly questions you can no longer be bothered to answer.

Exercises.

A textbook without exercises is like a calculator without an ENTER key; what was in your memory gets lost when you start on something new. For the sake of those people who want to treat this book as a textbook each chapter will end with a few exercises. They should help you remember what the chapter was about, but you can ignore them if you wish.

1.A One of the subjects on the cover of the book is "Keyboard efficiency". How efficiently do you use the functions on the keyboard? Test yourself by finding the smallest number of keys you need to press (without using key assignments) in order to work out the square root of (A↑2 + B↑2 + C↑2) and display the result. Count every time you press a key including SHIFT, but do not count the keys needed to put in A, B and C. Hint: you will find the answer in Section 5.3 under P-R, R-P.

1.B Look through Appendix A. Do you recognise any of the books? Have you read any of them? If you are serious about using your HP-41 you should think about buying some of the books most closely related to your interests. If no suitable book exists maybe you should write one yourself!

1.C Get a notebook and keep notes of interesting items as you go through the book. This is particularly important if you are an experienced user: otherwise you may quickly forget new ideas if they are mixed in with things you already know.

CHAPTER 2 - ABOUT THE HP-41.

2.1 Overview.

This whole book is about the HP-41 so why a chapter "About the HP-41"? The first chapter was an introduction to the book; this chapter is an introduction to the HP-41. Its main purpose is to explain the HP-41, to help new owners and to introduce words that are normally used as computer jargon but have also come to be used to describe the HP-41. If you already know your HP-41 and the jargon, and if you really do not care why the HP-41 system is the way it is, then you may as well skip this chapter.

2.2 Some HP-41 history.

The HP-41C was introduced by Hewlett-Packard on July 16 1979 as a portable battery-operated programmable calculator with many "advanced" features including a full alphabetic capability. Since that time two further HP-41 models have been produced along with a host of accessories, books and other items. Whereas previous calculators were replaced by entirely new models after two or three years the HP-41 has grown with these additions, some of them foreseen in its planning, others introduced later. Thus the HP-41 has not been replaced quickly but has been extended by these new features, particularly the HP-IL interface loop.

The capabilities of the HP-41 series have been expanded so much that Hewlett-Packard nowadays call the HP-41 a portable computer. Nevertheless the HP-41 on its own looks very much like Hewlett-Packard's original scientific pocket calculator, the HP-35. The dimensions, the vertical format, the position of the display and keyboard, even the number and layout of the keys (buttons) are the same on the HP-41 computer as on the HP-35 calculator.

The HP-41 is therefore a calculator-based computer, as opposed to other pocket computers which are scaled-down versions of microcomputers used in the home and office. This difference extends to the languages used; the

micro-based pocket computers generally use BASIC which is the language used by most micros. In this language a single instruction (statement) can include complicated arithmetic expressions. The HP-41 language is made up of the instructions available on the keyboard together with other similar ones, and a program is made up of a collection of these strung together one by one. The relative merits of the two types of pocket computer can be argued, but the fact is that an HP-41 can be used as a calculator which turns into a computer when necessary. This evidently has strong appeal to many scientists, engineers, navigators, astronauts and others; Hewlett-Packard have sold over a million HP-41s.

Certainly most owners will defend the HP-41 against attacks on its supposed shortcomings. Its high price can (perhaps) be justified by its high quality and the level of after-sales service provided by Hewlett-Packard. Its vertical format allows it to be held and used with only one hand. Its language is as much a computer language as any other and has been named **FOCAL** (Forty-One Calculator Language); not an ideal choice since a computer language called FOCAL already exists (an operating system language for CDC's). The limited memory and speed are limited only in comparison to bigger and less portable machines.

Hewlett-Packard have of course introduced other personal calculators and portable computers since the HP-41. The series 10 calculators are lovely pocket-sized machines, faster than the HP-41 at certain tasks but with neither its general-purpose capability nor its features for communication with other devices. The HP-75 is a true computer, faster than the HP-41, but of a size to be held in the lap not in one hand, and much more expensive. The HP-71 is also beautiful: it looks like a scaled-down version of the HP-75 although internally it is more similar to the HP-41. It is hand-sized (just) and has an amazing calculator mode but it is a computer with a horizontal format and therefore less likely to travel with people who like to do their calculations out in the field using just one hand. An "HP-41 translator module" for the HP-71 has recently been introduced; it allows the user to write and run HP-41 programs on an HP-71, it also provides new HP-41 type functions and allows FOCAL programs to be

mixed with BASIC or FORTH routines and programs. This module is a great help for someone who wants to transfer their work from an HP-41 to an HP-71 and take advantage of its greater speed, but it does not make the HP-71 smaller, change its shape, or make it any cheaper. In short, Hewlett-Packard have introduced new items, but nothing to replace the HP-41 calculator/computer. No other company makes a device that is directly comparable to the HP-41, the nearest is probably the Psion "Organiser". This is the same size and shape as the HP-41 and is described as a pocket computer but it is more an electronic pocket notebook, with better text editing facilities than any current HP-41 but with only two ports for plugging in memory modules or specialist software modules.

To avoid "calculator or computer" discussion all three models (HP-41C, HP-41CV, HP-41CX) will be referred to as "an HP-41" or a "machine" without specifying whether this may be a computer or a calculator. Perhaps the best title for the HP-41 is an **HHC** - for Hand-Held-Computer which makes it clear we are not talking about a computer such as a PDP-11 or a Cray-1. The programming language will be referred to as FOCAL or simply as the **"user language"** since it is the language employed by ordinary users. The limitations of memory size and speed will be treated as obstacles to be overcome, not as reasons for replacing the HP-41 with a bigger "machine".

The milestones in the history of the HP-41 since its introduction have been the introduction of the HP-41CV with five times as much memory as the HP-41C has, the introduction of the HP-IL and the Time module and the Extended Functions module, and the introduction of the HP-41CX. If you ask members of a user group they will probably add other milestones such as the development of Synthetic Programming or the use of M-code. The introduction of the HP-71B is often described as the final milestone, actually more like a tombstone, but in fact the HP-71 is sufficiently different that it will not entirely replace the HP-41. Those people who prefer vertical format, calculator-style computers are waiting for Hewlett-Packard to produce a true successor to the HP-41s, either an even more extended HP-41, or an entirely new product.

2.3 The layout of the HP-41

Let us now examine the HP-41 itself. New owners should make sure they recognise the various parts of an HP-41 and their names. Presumably most readers will know that the word **key** means one of the buttons on the front, but is it fair to assume everyone knows this? If you know about the layout and names then skip this section and come back to it later if you need to.

At the top on the front is the **display** which shows results and messages. Below this are four **toggle keys** or **rocker switches** which determine how the HP-41 will react to any other keys on the **keyboard** below them. The **SHIFT** key is the yellow (or gold) key. On the back of the HP-41 is a sticker showing the ALPHA keyboard (or the Text Editor keyboard on the HP-41CX although this is mostly the same as the ordinary ALPHA keyboard). Above this sticker is the **battery compartment**. At the top of the HP-41 are four **Input/Output ports** or **I/O ports** or simply **ports**. These let you plug in **modules** or **peripheral devices** (known jointly as accessories or **plug-ins**) that can be used with the HP-41. The ports are numbered as shown below.

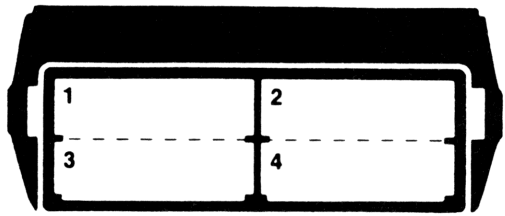


Figure 2.1 The I/O ports.

The same numbering is shown on the panel at the top left of the back. The serial number, including date of manufacture, is at the top right. You should keep a note of this in case your HP-41 is ever stolen. Finally there is an **AC adaptor socket** on the side to the right of the keyboard.

Since the HP-41 allows you to redefine the keys, it is designed to let you put an **overlay** on the keyboard. You get two general-purpose keyboard overlays with your HP-41 when you buy it. The overlays lie on top of the normal keyboard; they have holes cut out so that they slip over the keys and only cover up the printing above them on the flat part of the keyboard. The new functions you have chosen for each key can be written on the overlays so that you can check what each key does. Special overlays are also provided for use with some plug-in modules and for certain functions which redefine the keyboard, for example the Stopwatch. To hold down the overlays the HP-41 has a tab at the top of the keyboard and three slots at the bottom. You can also buy a super-overlay called a **membrane Touchpad.** This covers the whole keyboard, keys and all, and it contacts each key through a small pad. It can have the normal key functions printed on it or companies that use many HP-41s can order touchpads with special functions printed on them. The touchpad reduces the positive "click" that you normally feel when you press an HP-41 key. Some people dislike this, but the touchpad makes the keyboard quicker to use because it lets you slide your fingers over its surface from one key to another without lifting them off each key after it has been pushed.

The I/O ports, the slots for the overlays and the sliding tab all leave holes in the HP-41 case. There are also two holes to the left and right of the I/O ports,these are designed to hold the Card Reader firmly in place, you can see them in Figure 2.1. Whereas many previous HP calculators were encased in an impenetrable shell, the HP-41 will let in water and sand fairly easily because of these gaps. Yachtsmen generally put two polythene bags one inside the other around their HP-41s to keep out the water. In sandy deserts or very dusty conditions you should stick some heavy duty tape around the various gaps including the AC adaptor socket. You can use a touchpad to cover the keyboard and hide the gaps around the keys and the tab and slots. If you are the kind of person who often spills coffee (or sulphuric acid) on the keyboard you will find the touchpad very useful. (If your HP-41 dealers do not have the touchpad in stock tell them the part number, it is HP 82200A, otherwise they will probably try to sell you an

overlay kit!) The worst problem with coffee or other drinks is that the sugar in them dries out under the keys and jams them. If you drink coffee without sugar you may get away with spilling some on the keyboard. The touchpad is rather expensive, a much cheaper way to protect the keyboard is to cut out a piece of card large enough to cover the whole keyboard and hold this card down over the keyboard with sticky tape. Then glue one of the overlays over the card. You will find that you can press the HP-41 keys through the card; the overlay shows you where each key is, and you can write the key functions on the card, so that they can be read through the holes in the overlay.

2.4 Batteries and power.

If you have bought a new HP-41 or if your HP-41 has not been used for over a year then you should put in a new set of four 1.5 Volt alkaline size N (also called 2/3 size AA) batteries. In the UK ask for Duracell MN9100, VARTA 7245 or their equivalent. The battery replacement method is described in the manuals but I shall repeat it here for anyone who has still not obtained a manual. Remove the battery pack by pushing the inset lip away from the ALPHA keyboard sticker. The holder pops out and you can remove the old batteries and insert the new ones, making sure that each battery has its + end or its - end facing the bottom of the holder as marked on the underside of the holder. Replace the holder with the exposed battery ends facing the top and push forward and down until the holder snaps into place.

If you put in worn-out batteries, or if any of the batteries are the wrong way round then the HP-41 should not be harmed but it will refuse to turn on and the contents of its memory may be lost. Turn round any batteries that are upside down and do not put in or leave in any old batteries since they may leak. Instead of disposable alkaline batteries you could use a more expensive type such as lithium cells of the same size, or mercury cells; both types have a longer life. You could also use rechargeable Nickel-Cadmium 1.5 V cells (Ni-Cads for short). Whatever type of batteries you use, make sure the battery contacts are clean.

The HP-41 was originally designed to be run from an AC mains adaptor as an alternative to batteries and the AC socket on the side had two small gold-plated balls for contact with the adaptor plug. Instead of an AC adaptor HP eventually produced a rechargeable battery pack (part number 82120A if you want to order one) which replaces the whole battery holder. This battery pack can be charged from the same AC charger as is used for the HP-41 printer and the HP-IL printer and cassette drive. The charger plug can be fitted through the AC socket, but the gold contacts are no longer provided since an accidental electrical contact with them can damage the HP-41 circuits.

When you are using your HP-41 you will need to change the batteries occasionally, the BAT message will warn you of this, see Section 3.2. If you are replacing the batteries after using the HP-41, make sure it is turned off when you remove the battery pack. While the batteries are removed after the HP-41 has been turned off, internal capacitors provide enough current to maintain the contents of memory. You can be fairly slow in changing the batteries, the internal capacitors on most HP-41s will maintain memory for several days and some will last for up to a month! Do not try to turn on the HP-41 while the batteries are removed as this will speed up the discharge.

Once you have a set of good batteries correctly placed in the HP-41, you should be able to turn it on by pressing the ON toggle key. Normally the HP-41 comes on with its memory unchanged, but a new HP-41 or one whose batteries have not been replaced soon enough may display MEMORY LOST to show that the memory contents have been lost. If the display remains completely blank or if it shows MEMORY LOST, press the **backarrow key** and the display should show a zero value. If none of these things happen, see Section 4.1 for further advice. To protect against unnecessary battery use the HP-41 turns itself off automatically after 10 minutes if it is not being used; you can prevent this, see last paragraph of Section 3.3.

Following MEMORY LOST, the HP-41 sets its display and other features to

default values. A default value is one that is assumed if no other value has been specified. For example numbers are displayed with four digits after the decimal mark by default unless you specify a different display. (On previous HP calculators the default display was two digits after the decimal mark.)

2.5 ROM, RAM and Continuous Memory.

Should the HP-41 batteries be removed for too long a time, it will lose everything that the user has put into its memory (see previous section). On the other hand it will still remember that it is an HP-41! The display and the functions will work as before because their instructions are stored in a permanent memory which cannot be changed unless the HP-41 is damaged. This type of memory is called **ROM** (Read Only Memory). The HP-41 can only read instructions and information from this memory; it cannot change them. The ROM memory is written before the HP-41 is put together, it does not change afterwards (unless it is replaced when the HP-41 is being repaired), and it does not need batteries to be preserved.

The user must also be provided with memory that can be altered so that new numbers and programs can be stored. This memory in the HP-41 is called **Continuous Memory**, but it is only continuous so long as the user does not change it and there is an electric current to maintain it. It is actually called Continuous Memory because on earlier calculators all this user memory was lost every time the calculator was turned off. Newer circuitry takes a current of less than 0.1mA to maintain the memory and this is provided by a small "leakage current" from the batteries even when the machine is turned off, and from capacitors when the batteries are removed.

Memory that can be written to and read from used to be divided into two types, sequential access and random access, although this division into two types is rarely used now. Sequential access memory such as a magnetic tape needs to be read or written in sequence, from the beginning to the required place. Random access memory allows the user to go directly to any place without checking from the beginning every time. The Continuous Memory of

the HP-41 is of this type, called **RAM** (Random Access Memory). The layout of the HP-41 RAM and ROM memory will be described in Chapter 8.

HP-41 memory is divided up into **registers**. Each register can store a number with up to ten decimal digits, a decimal point and a sign followed by a signed two-digit **exponent** (power of 10). The part of the number that is not the exponent is called the **mantissa**, and each single figure (0 to 9) is called a **digit**. Each register can also contain a **text string** made up of zero to six **characters**. A character is any letter, numerical digit, punctuation mark or special symbol (like + or & or a blank space). Other things can also be stored in registers, and registers can be divided up into smaller elements or treated as groups. This will be discussed in Chapter 8. For now it is enough to say that the contents of a register will just be called a **value** if there is no need to specify what that value actually is. The contents of registers, and other items which are used during programs and calculations are known collectively as **data**.

2.6 HP-41C, HP-41CV, HP-41CX...

The HP-41C, the original (and cheapest) version of the HP-41, has 64 memory registers available for storage of programs, data and key reassignments. 64 registers may seem a lot for a pocket calculator but on an HP-41C they will be quickly used up unless it is dedicated to one task only and is never used for anything else. Up to 4 RAM memory modules, each with 64 memory registers, can be plugged in to extend the memory but they take up the ports which can therefore not be used for other plug-ins. In January 1981 HP introduced a Quad memory module equivalent to four ordinary memory modules. At the same time they introduced an HP-41 with a Quad module built into it, the HP-41CV; the V is a roman five to say the CV has five times as much memory as the C. The HP-41CV keyboard also had keys with a more slanted lower face, which made the blue letters easier to read than on the original keyboard. This improved keyboard is now used on all HP-41s.

In December 1981 HP announced the HP Interface Loop (**HP-IL**) and also the HP-41 Time module and Extended Functions/Extended Memory module. These two

modules were useful for control of devices attached to the the HP-IL loop, but they also extended the abilities of the HP-41 itself. It was soon rumoured that a new HP-41 model with these modules built-in would appear, as had been the case with the HP-41CV. In fact the HP-41CX (X for Extended) was cancelled and resurrected several times before it was officially announced in November 1983. The HP-41CX is just like an HP-41CV with a Time module and an Extended Functions/Extended Memory module built in, except that it has additional functions and features, including a Text Editor. The Time and Extended functions will be described in Chapters 9, 10 and 11.

As of December 1984 the HP-41CV was marginally more expensive (in the UK) than an HP-41C and a Quad memory module bought separately. An HP-41CX however costs little more than an HP-41CV, the difference is less than the cost of a Time module or an Extended Functions/Extended Memory module. Unless you are buying an HP-41 for a specific application which is fitted exactly by the HP-41C or the HP-41CV, then the HP-41CX is the best buy even if you do not use many of its additional functions. (Few people really do much text editing on a pocket computer, but it is nice to have a text editor on the HP-41CX just in case you want to make a few notes.) More details of the HP-41CV will be given in Section 8.7 and various features of the CX will be described in Sections 9.5, 10.5, 10.6, 11.8, and 11.9.

It is possible that Hewlett-Packard could introduce an even more extended version of the HP-41, perhaps with more Extended Memory and Extended functions than the HP-41CX. Most of the advice and programs in this book should work on such an additionally extended HP-41. (Any changes or additions to the 40 series that are made after the main part of the book has been written will be mentioned in Appendix E.)

2.7 Some accessories.

The plug-in modules and devices available for use with the HP-41 will be discussed in some detail in Chapter 12. As it will be necessary to mention them earlier here is an introduction to the plug-ins available.

As was explained in Section 2.5, the HP-41 uses ROM and RAM memory. RAM modules provide additional memory for the user in the form of memory modules, Quad memory modules and Extended memory modules. ROM modules, sometimes just called **ROMs**, contain pre-written programs for special applications, for example the Maths or the Petroleum Fluids Applications modules. Some special ROMs contain additional functions, not programs, and these modules may contain other parts such as a quartz oscillator in the Time module, or RAM memory in the Extended Functions/Extended Memory module. The plug-in modules are all the same size, just small enough to fit entirely into one of the HP-41 ports.

Plug-in devices each have a plug that fits into a port but the devices themselves lie outside the HP-41. The ports are called Input/Output ports but many of the accessories are used only for input or only for output. The **Card Reader** is used to store programs, data (information), and the status of the HP-41 on small magnetic cards. These cards can be treated as **mass storage**, an extension of the HP-41 memory without size limitations. The programs, data and status can be read back into the HP-41 at a later time and they can be transferred to other HP-41s. The Card Reader can also read HP-67 and HP-97 program and data cards and interpret them for use on the HP-41. This means that owners of these earlier calculators can carry on using their programs, and that HP-41 owners can use HP-67/97 programs if no available HP-41 programs meet their needs. The Card Reader provides 26 extra HP-41 functions designed to simulate HP-67/97 functions not available on the HP-41. (Some of these functions are very useful on their own and can be exploited in HP-41 programs.) It plugs into port 4 and looks like a part of the HP-41. If you have wondered why the soft carrying case has a piece of foam at the bottom, this is because the case is designed to hold an HP-41 with a Card Reader attached to it when the piece of foam is removed. The Card Reader is a true Input/Output device as information can be written or read with it; it should really be called a "Card Writer and Reader" but the name "Card Reader" has been borrowed from computers and will not change.

The **Optical Wand** or **Wand** can be used to read single instructions, data and whole programs from printed **barcode** which looks similar to the product codes on items in shops. Barcode is easier to mass-produce and copy than magnetic cards, it cannot be damaged by magnetic fields, and it can be easily stored and transported (for instance in books). Barcode for several programs has been included in this book whereas it would not have been easy to include magnetic cards. Some people prefer to use a barcode keyboard which provides all the functions they use; this can be easier than using the keyboard and spelling out or assigning instructions that are not available on the ordinary keyboard. Small amounts of barcode can be drawn by hand without too much trouble or it can be made up from the sheet of labels sold with the Wand. Barcode for the new fuctions available on the CX is not yet provided with the Wand so it has been included in Appendix F together with barcode for letters and some other functions. Large amounts of barcode take a lot of work to produce directly from the HP-41, and only printers and plotters available on HP-IL can print it. Even this is difficult unless you have a Plotter module. Barcode can also be ordered from companies which will produce it from magnetic cards. The Wand is clearly only an input device.

The HP-41 printer, the HP82143A, plugs directly into the HP-41 and is used to print data, programs and status information. It can also be used to print graphs, histograms and simple pictures, and to trace program execution. It increases the usefulness of the HP-41 enormously but costs half as much again as an HP-41CX.

All the above devices plug directly into one of the HP-41 ports and are designed for use with the HP-41 alone. (Devices like this, intended for one particular use only are called **dedicated** devices.) The Hewlett-Packard Interface Loop (HP-IL) works differently; it provides a means of attaching a variety of devices to each other on a single loop. The HP-41 can be one of these devices, and normally it expects to control the others. A different controller, for example an HP-75 computer can be used, and it is even possible to have more than one computer on a loop. Devices for use with HP-IL are designed to work with the HP-41 or any other controller that

can send them commands along the loop. This is more versatile than using items dedicated to the HP-41, but it is at the expense of having to use a special HP-IL module. This plugs into one HP-41 port and has two cables attaching the HP-41 to the rest of the loop.

When HP-IL was introduced, a printer very similar to the HP82143A was provided. This new printer, the HP82162A, has all the features of the HP82143A, plus the ability to format printing (space it out on a line as required), to justify text margins, and to print barcode. It costs more than the HP82143A and it can only work with an IL module which is itself expensive, but it can be used with any HP-IL computer, not just the HP-41. The older printer has one advantage; it can be used with an HP-IL Development module to let an HP-41 connected to a loop monitor what other devices attached to the same loop are doing. If an HP82143A printer is attached to the monitoring HP-41 it can print out a full analysis of the commands and data passed round the loop. An HP82162A printer cannot do this since it is attached to the loop in general, not to the monitoring HP-41. The HP82162A also prints some rarely used characters differently than does the HP82143A. Both these printers produce rows of only 24 characters and use thermal print paper which fades with age and exposure to heat or light. Important information from these printers should be stored carefully or photocopied to provide a more permanent copy if it is not to be lost.

The next few Chapters will introduce fundamental ideas and describe how the HP-41 is operated from the keyboard, but later on, particularly after Chapter 5, a great deal will be written about programming, and various programs will be printed out. HP-41 programs are shown in this book as they would be printed on an HP82143A or an HP82162A printer. Most program lines look exactly the same on such a printout as they would on an HP-41 display but there are a few differences, so a short example of a printed program will be useful. The printout of a program is often called a program **listing**. Here then is an example of a program listing:

```
 4.00PM 10.12
01◆LBL "LIGHT"
02 2.998 E8
03 "c="
04 ARCL X
05 AVIEW
06 END
```

The first line shows the time and date when the program was printed. This is provided by HP-IL printers but not by the HP82143A, and it is only printed if the HP-41 has a Time module attached or built-in. This helps you find the latest version of a program but is not much use in a book, so most listings in this book will be shown without it. The next line is line 01 of the program. Since it is a label it is printed with a lozenge on its left. This lozenge, or diamond, is printed next to all labels to help you find them in a long program listing. (In typed program listings, I shall use an asterisk in place of this diamond.) The label consists of the LBL function and its name "LIGHT" enclosed in quote marks. In the HP-41 display the name would not have quote marks around it but it would appear as 01 LBL"LIGHT. The " is a **text** symbol showing that what follows it is a string of text, not an instruction. The next line contains the number representing the speed of light in SI units. This consists of a mantissa (2.998) followed by a power of 10 (or exponent). In a program the exponent is separated from the mantissa by the letter E. In a normal display the separator is just a space. The next line contains no instruction, just the text "c=". In the display this would appear as "c= . It is important to realise that there is a difference between a piece of text on its own, which is used for displaying messages, and a piece of text that is part of a function such as the LBL above. The remaining lines of the program display a message showing the speed of light. This program can be used to get the speed of light for use in calculations and to display it clearly at the same time. Programs and programming methods will be described in detail in Chapter 6.

Larger printers are available for use with HP-IL, these generally have the advantage that they print on ordinary computer paper and also on single sheets but none of them have a set of characters that is fully compatible with the HP-41 which uses a few unusual characters. The new Thinkjet printer uses a thermal inkjet instead of a dot-matrix printer head (this works best with one kind of paper which particularly suits the Thinkjet printer) but otherwise it is fairly similar to the dot-matrix printers. The more advanced of these can be used for graphics since the individual dots can be addressed and used for printing any selected pattern. Video interfaces are also available for the HP-41, these really work like on-screen printers and do not provide any graphics. The printers are fundamentally output devices, although some limited talk-back to the HP-41 is available from their control buttons.

The HP-IL version of the HP7470A Plotter can be used for graphics or as a printer. It can also be used as an input device for digitising graphs and figures since it has an Enter button which sends the X and Y coordinates of the pen back to the HP-IL controller.

In addition to the Card Reader you can use magnetic cassettes and disks for mass storage. Unlike the Card Reader these can only be controlled through an HP-IL module. The printers and the cassette drive have a metal loop at the top left for use with a security cable. This feature is not generally advertised by HP.

Some care is needed in plugging-in and unplugging all modules and devices. The HP-41 and the devices can sometimes be damaged if they are connected or disconnected while either is turned on, but this very rarely occurs and you should not be unduly worried if you remove a module and then find that you forgot to turn the HP-41 off. The contacts in the HP-41 ports and on the plug-ins have a thin layer of gold plating which slowly wears away so you should not plug items in and out just for the fun of it. Hewlett-Packard will replace the HP-41 port connectors at a price, but they will not replace the contacts inside a module. (Some user groups have members who are willing to do this though.) The modules and the ports can collect

static electricity and when they are connected this can affect RAM memory, occasionally causing a "crash" or even a MEMORY LOST. This can usually be avoided by breathing gently into the port and the module opening before connecting them together. The moisture in your breath should be sufficient to discharge the static, but water is bad for the connectors, so do not breathe too heavily.

Early models of the Card Reader and the Time module may draw an excessive current from the HP-41 batteries when they are first plugged-in. Turn the HP-41 on and off after you plug them in to reduce the current drain to normal.

The I/O ports are one of the best features of the HP-41, although four are not enough. When the HP-41 was first introduced, cynics said that the ports were only there so that people would have to buy their HP-41 in bits and pay more for it. To some extent this may be true, but there are now so many bits that you may well be unable to fit all the ones you are using at once into the four ports. A partial solution is to buy an HP-41CX since this has so much already built in, but even then you may soon be playing "musical ports" as one HP-41 user (W. Kolb, PPCCJ V9N4p96) put it! A better solution is to use a **Port Extender**. This is a box which fits under the HP-41, plugs into one port, and provides more ports (the number can depend on the make, but is typically six extra ports). Only independent companies make port extenders, see Appendix B; Hewlett-Packard do not produce such a device since there are some complicated rules about what can go into each slot. Adventurous owners open up their modules and combine the contents of several into one, or even wire the module contents directly into the free spaces in their HP-41. Some companies provide this service too, they are best contacted via local user groups. Once again there are several rules about what can or cannot be done, one of these is that you cannot expect HP to help with this kind of thing, or to sympathise if the HP-41 has got damaged. Once again, ask for advice at your user group.

Exercises

2.A The HP-41CX Quick Reference Guide lists 222 functions on the CX. The C and the CV have 95 fewer functions and also less memory. The functions, data registers and the April 1984 prices of the 41s in the UK (excluding Value Added Tax) are given below.

HP-41C	64 data registers	127 functions	price: 152 Pounds.
HP-41CV	320 data registers	127 functions	price: 216 Pounds.
HP-41CX	446 data registers	222 functions	price: 242 Pounds.

Assuming the price to be A for the basic HP-41, B per data register, and C per function, work out A, B and C to the nearest hundredth of a Pound.

2.B When you have done Exercise 2.A work out how many extra functions would have to be added to the HP-41CX to create an HP-41CZ (Z for zero) which would cost nothing.

Which would you say is the more unrealistic, the cost scheme used in this problem or HP's pricing policy? Don't take this problem too seriously!

2.C Have you ever really examined your HP-41 closely? What colour is it? (It is not quite black.) Have a look at the keys; the number keys are slightly larger than the others. It could be argued that this makes it a little easier to identify them should you need to operate the HP-41 by touch alone, but in in any case it is a carry-over from the design of earlier HP calculators. If you have an earlier model (35,45,55,65,67), compare it with the HP-41. You may need to be able to operate by touch alone, particularly if you are reading numbers from an instrument and keying them directly into the HP-41. Try entering a few numbers without looking at the keyboard, obviously the large size of the ENTER key makes it easy to find so that a new number can be separated from the previous one. The R/S key is among the number keys as well so that you can run a short program by touch alone.

CHAPTER 3 - DEFINITIONS AND PROBLEMS

3.1 Using the HP-41.

The first two chapters have been rather theoretical since they are intended to provide information for people who are thinking about buying an HP-41 as well as for users. Now we can start using the HP-41! One of the first things that happen when someone uses a new device is that something goes wrong and the manual says nothing about the problem. At other times the manual does explain the problem, in great detail, but using words that mean nothing to the reader. This chapter therefore goes through some uses of the HP-41 but also contains a lot of definitions. These lead to explanations of the more common problems encountered by HP-41 users. Do not worry though, the main feature of the HP-41 is that it provides solutions, not problems. Problems rarely occur and are described here only to help the reader get solutions.

3.2 The display and audible signals.

When you turn on your HP-41 the display should show some letters or numbers. This **LCD** (Liquid Crystal Display) consists of twelve positions each able to display one character. Each character can be followed by a punctuation sign (. or , or :). Each display position is made up of 14 segments so 16,384 different characters are possible but at present the HP-41 only uses 79 of these (pity), and of course the three punctuation signs. (If the display circuit does not recognise a character it uses a default consisting of all fourteen segments turned on, ▣ sometimes called a **boxed star.**) These twelve positions normally show numbers, but you can use them to display messages, and the HP-41 also uses them to display warnings, messages and programs. You can select the number of digits to be used when a number is shown and you can also choose how commas and points are used in the display of numbers; this will be described in Section 4.4. Below the twelve characters there are twelve **display annunciators** that tell the user about special conditions, and these will be described below. Figure 3.1 shows a display with all the annunciators turned on.

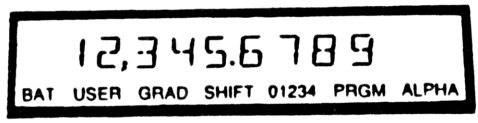


Figure 3.1 The HP-41 display.

BAT The batteries are running low. With alkaline batteries there are less than seven hours running time, or less than a month of memory protection left. With rechargeable batteries you probably have less than a minute of running time and only a few hours of Continuous Memory protection. If you continue running a program when the BAT annunciator is lit the display will eventually fade out while the program continues to run, then the program will stop but Continuous Memory will be maintained. After the program has stopped the HP-41 will stay on for another 10 minutes and will then turn itself off automatically as normal. You should replace the batteries as soon as possible if the HP-41 has reached this stage. BAT will also come on if the batteries cannot drive a plugged-in card reader or optical wand, but if you turn the HP-41 off and on again the BAT will vanish. In such cases you should use fresh batteries for the card reader or wand, then put back the old batteries and use them for normal HP-41 operations. If your batteries are flat but you have an HP82143A printer attached then you will be able to carry on because the printer's batteries will provide power to the HP-41.

USER See next section.

GRAD The HP-41 will perform trigonometric functions in gradians, not degrees (which is the default). 400 gradians = 360 degrees. The letter G is actually a separate annunciator, used only in conjunction with RAD to display GRAD.

RAD The HP-41 will perform trigonometric functions in radians.

Note: If neither GRAD nor RAD is displayed the HP-41 will work in degrees.

SHIFT The yellow SHIFT key has been pressed. See next section.

0,1,2,3,4 The flag corresponding to the displayed number has been set. For example if 0 and 2 are displayed but 1,3 and 4 are not then flags 0 and 2 are set but flags 1, 3 and 4 are clear. See Section 4.4.1 for details about flags.

PRGM Displayed when the HP-41 is in program mode (see next section). It is also displayed when the HP-41 is running a program.

ALPHA See next section.

Each display annunciator can be cleared by an appropriate action; putting in new batteries, setting the HP-41 to work in degrees mode, clearing flags 0 to 4, and so on.

In addition to displaying information or sending it to attached devices, the HP-41 can also use audible signals to communicate with the user. There are ten individual tones and one compound signal called BEEP. Press SHIFT, 4 to hear this. It is loud enough to attract your attention in a quiet room, say to tell you that a long program has finished, but it would be no use at all as a fire alarm. The tones are not intended to be particularly musical, though if you play them in turn from TONE 0 to TONE 9 you will get a scale of sorts (C minor). Combinations of tones have been used to let blind people use HP-41s, and of course in games. Time module alarms and the optical Wand produce additional tones.

Most HP-41s can also be persuaded to produce a high-pitched whistle. If you remove the batteries at the right moment during a tone and replace them then the HP-41 will make this whistle whenever you press any key or while it is executing any function. On any HP-41 made after the middle of 1983 (including all HP-41CXs and most HP-41s repaired after that time) you can get into this whistle mode by resetting during a tone. To do this press ON and ENTER at the same time as the HP-41 is executing a BEEP. You may have to try a few times to get this right, but it is worth it if you can: if you display the clock when the HP-41 is in this mode then you will get a short whistle once a second giving you a ticking clock. This mode is not cleared when you turn off the HP-41 but it is cleared whenever you sound an ordinary tone or BEEP.

3.3 Toggle keys, keyboards and modes.

The four keys below the display control how the keyboard will be interpreted when other keys are pressed. The SHIFT key belongs with these, called "toggle" keys because pressing them repeatedly toggles the HP-41 alternatively into and out of special states. The first time one of these is pressed the display shows that the special state is active, the next time the key is pressed the display shows that the state has been cancelled. Sometimes the result of pressing one of these keys depends on one of the other keys.

The ON key belongs with these since pressing it the first time turns the HP-41 on so that the whole display becomes active and the keyboard responds to other keys. The second time this key is pressed, all this is cancelled and the HP-41 is turned off. This is therefore often called the ON/OFF key. Pressing SHIFT, ON turns on the clock display if a Time module is plugged in or if you have an HP-41CX.

When the USER key is pressed the USER display annunciator comes on and the keyboard will respond according to the user's definition of the keys, unless ALPHA is also displayed. The USER setting is independent of the

other toggle functions, it changes only when USER is pressed again. When the PRGM key is pressed the PRGM annunciator comes on and subsequent keystrokes are recorded in a program instead of being obeyed at once. USER and ALPHA can also be active in PRGM mode. Pressing PRGM again cancels the program setting, and turning the HP-41 off also cancels it.

Pressing ALPHA makes the ALPHA display come on; the keys respond as shown on their lower faces and on the sticker at the back of the HP-41. Pressing the key again cancels this Alphabetic keyboard and so does pressing the PRGM key or the ON key.

SHIFT turns on the SHIFT annunciator and selects the functions assigned to the shifted keys (written above the keys, not on them). This is true in USER and ALPHA as well as in the normal keyboard state. SHIFT is automatically cancelled whenever any key except USER is pressed. Pressing SHIFT a second time cancels it without doing anything else.

Each of the above keys selects a **keyboard mode**. If none of these modes is set then the HP-41 keyboard behaves according to the normal functions printed in white on each key. This is often called the **Normal keyboard**. Pressing any of the ordinary keys in this state will make the HP-41 do what is printed on the key. If you keep the key down, the function corresponding to that key is displayed. When you release the key the function is executed unless you have kept it down for more than half a second in which case the function is cancelled and the display shows NULL. This **function preview** also works in PRGM, and in USER mode where it is particularly useful since it lets you check whether a key has been redefined. The function preview does not work in this way for keys that need parameters, see next section.

When USER, ALPHA, or PRGM is pressed, the keyboard reacts differently than with the normal keyboard. Some other functions, and many programs redefine the keyboard for their own use. This leads to additional keyboard modes such as the Stopwatch keyboard or the Text Editor keyboard. The word **mode** is also used on its own to describe a state of the HP-41. For example the

HP-41 can be in degrees mode, or in radians mode or in gradians mode. This will affect the results of trigonometric functions regardless of which keyboard is active. If you press PRGM you take the HP-41 into program mode so that instructions will be stored in a program, but any one of the three keyboards ALPHA, Normal or USER can be active. The opposite of PRGM mode is usually called **RUN** mode since programs are run more often than they are written. This name is inherited from earlier calculators which did not have toggle keys but instead had a switch that could be set to PRGM or RUN. When a program is running or a function is working, they are often said to be **executing**. This comes from the original meaning of the word execute - to do something - in this case the program or function is doing what it is told to do.

The word "mode" describes other things too. For example the HP-41 can do trigonometry in one of three modes; degrees, radians or gradians. The GRAD and RAD annunciators show the trigonometric mode. The "display mode" determines how numbers are shown; FIX 9 or ENG 3 for example.

It is rather unfortunate that Hewlett-Packard has also chosen to use the word "mode" to describe the level of power consumption, or electricity use, of the HP-41. In **deep sleep mode** the HP-41 is off and only enough power is used to maintain Continuous Memory and to run the Time module crystal oscillator if one is present. In **light sleep mode** the HP-41 is on but is not actively doing anything. It is however keeping the display on, monitoring the keyboard for any keystrokes, and counting through ten minutes at the end of which it will go into deep sleep if nothing has happened. This is also called **standby** or **idle** mode. There is an ON function which is separate from the ON switch, it sets flag 44 and is used to stop the HP-41 from turning off after ten minutes of inactivity. Finally **operating mode** means the HP-41 is executing a function, running a program, performing a lengthy operation (such as packing the memory), or displaying a running clock or stopwatch. The HP-41CX goes into this mode too when it is displaying CAT 4,5 or 6 or when it is running the Text Editor; it leaves this mode after 2 minutes of inactivity in any of these states. If BAT is displayed and the HP-41CX is stopped at CAT 4,5 or 6

then it exits from this state after one minute. The clock will not be displayed if BAT is on.

3.4 Parameters, arguments and RPN.

Most functions use the displayed value to work on, for example LOG normally calculates the logarithm of the displayed number. The value used by a function in this way is called its **argument**, and arguments must be provided before a function is used. This is the basis of the **RPN** - Reverse Polish Notation - system used by the HP-41. Arguments are always provided before the function as in the example of LOG just above; even in the case of addition you type in the first number, then ENTER, then the second number and finally + . That is why the system is called Reverse; numbers are put in before the functions that use them. It is called Polish because this way of writing down expressions (or notation) was invented by a Polish logician Jan Łukasiewicz, and not everyone can pronounce "Łukasiewicz" as well as I can. By the way, the largest single user of RPN calculators appears to be the petroleum industry, and one of the people who started the petroleum industry was another Łukasiewicz! Ignacy Łukasiewicz refined crude oil, sank one of the first European oil wells (in Eastern Poland) and invented an efficient oil lamp (well, he had to do something with all that oil). Just one Łukasiewicz after another!

If you are not familiar with RPN you should practice using it before reading much more of this book. The HP-41 manuals give a fairly good introduction or you may want to read one of the beginners' books suggested in Appendix A. Alternatively you could just practice with a few functions on your HP-41.

For mathematical functions that use one argument (**monadic** functions) you make sure the argument has been been made available, then press the function key. The argument can be entered from the keyboard, recalled from memory, or created as the result of a previous mathematical function. Following any of these operations the argument is in the **X-register** which is the register used for most mathematical functions and for displaying

results. After the function has done its job, the result is in the X-register and is also displayed. Many functions take more than one argument though, for example conversion from radial coordinates to polar coordinates involves two values. The first is put into the X-register, then copied into the **Y-register** when you press ENTER. The second argument can then be put into the X-register, and the result can be obtained by pressing SHIFT,R-P. The result consists of two values too, one in the X-register, the second in the Y-register which can be thought of as sitting above the X-register. R-P is a **dyadic** function which means that it starts with two arguments and finishes with two resulting values. The most common functions though are + - * (multiply) and / (divide). They start with two arguments and finish with one result. These are called **bifid** functions; they work very much like the dyadic functions, you put the first value in, press ENTER, put the second value in, and press the function key. This outline might help you understand RPN and more will be said about it in Chapter 4. If you are a complete beginner though, remember the advice - read the manual first.

The mathematical functions and most others expect an argument before the function is performed. Some functions however require additional information <u>after</u> the key has been pressed. This is because such information is part of the function. For example pressing STO produces the display **STO__** meaning store -- where? The answer to "where" is called a **parameter**; for STO it can be any number from 00 to 99 or one of several other values. Functions that require parameters are **control functions**, not mathematical functions, and they do not have a function preview since you can cancel them by pressing the backarrow key instead of supplying a parameter. You can use the backarrow key to cancel one of the numbers (digits) you have used to reply, or to cancel the whole instruction.

The __ that comes after STO is called a **prompt**, it prompts you to provide a two-digit parameter and this parameter replaces the __ prompt. Parameters for different functions can be one, two, three or four digits long. For example TONE can only take a one-digit parameter, so you can only create TONE 0 to TONE 9 (until you read Chapter 14). STO takes a two digit

parameter so you can use STO 00 to STO 99. DEL takes a three digit parameter, so you can use it to delete any number of program lines from 000 to 999. You may however have a program that is 2000 lines long and you may wish to delete the first 1100 lines. To change a three digit prompt DEL___ into four digits you can press EEX. This will change DEL___ into DEL 1___ so you can delete up to 1999 lines. All three-digit prompts can be changed in this way with EEX, but the largest four digit parameter that can be produced is 1999. Instead of pressing the number keys to fill in a prompt you can also press one of the keys in the top two rows. The first row produces the values 1 to 5 in reply to any prompt for one, two or three digits. The second row produces 6 to 10, except that the TAN key produces 0 instead of 10 if the prompt is only one digit long. These keys cannot be pressed after EEX to complete a 4-digit parameter, but if you own a ZENROM, see Chapter 12, they can be used after EEX to complete a 3-digit parameter.

Some instructions can take either a number or an ALPHA value for a parameter. For example GTO__ suggests that you should provide a two-digit number, but if you press ALPHA you will see GTO_. Then you can type in a parameter made up of 1 to 7 ALPHA characters. When you press ALPHA again the parameter is complete and the instruction can be obeyed. A few instructions such as ASN will only take ALPHA parameters; they provide a single prompt but will only take a parameter after ALPHA has been pressed. If you change your mind and decide to cancel an instruction with an ALPHA prompt you can press the backarrow key even in ALPHA mode; the instruction will be cancelled and ALPHA mode will also be cancelled. Most of the instructions that take numeric parameters will also take the parameters X, Y, Z, T or L. These refer to the stack registers which will be described in the next chapter. Many instructions take indirect parameters too and these will also be described in the next chapter. A stack parameter is obtained when you press the decimal point key and then one of the above letters. An indirect parameter is obtained if you press SHIFT and then a number or the decimal point and a letter. As you can see, there are many options for producing parameters, the HP-41CX manual covers them all by describing a **parameter keyboard.**

3.5 Instructions and functions, routines and programs, catalogues.

What should we call the operation of calculating a sine on the HP-41? What should we call the collection of operations that calculate a hyperbolic sine? At the lowest level of computer programming each elementary operation is called an **instruction**. To the average user of an HP-41 the ordinary user-language instructions such as X=Y? or CLX or SIN are the low-level instructions of the machine. Unfortunately the Owner's Manuals also give the reader "instructions" on how to use X=Y? and the others, so this is not a very good word to use.

To a mathematician, sine or logarithm are "functions", and the HP-41 uses the SIN and LOG instructions to work out approximate values of these functions. By analogy SIN, LOG, X=Y? and all the other instructions on the HP-41 are called **functions**. This may displease mathematicians but at least it is fairly clear. In this book the words "instruction" and "function" will both be used to describe things the HP-41 does.

High-level computer languages (ones that are more similar to written algebra than the machine-level languages) have more powerful and longer instructions called **statements**. A typical statement might be

```
X1 = (-B + SQRT( B**2 - 4*A*C ) )/2/A
```

The HP-41 does not have statements, the functions are the most complicated level of single instructions available. On the other hand a single HP-41 function can do many things. For example SIN is equivalent to the following two statements on a bigger computer

```
LASTX=X
X=SIN(X)
```

A collection of computer statements which perform a fairly simple task is called a **subroutine**. On the HP-41 a collection of functions starting at a label and ending at a RTN or an END makes up a subroutine. Such a

subroutine saves you the trouble of writing out the same group of instructions every time they are needed and it can be called on whenever the same task needs to be performed. A larger collection of statements, often including many subroutines, and able to perform some complete task, is called a **program**. A program may work out a shop's tax for a financial year, and a small subroutine in this program may be used to work out the tax on the sales of odd socks. On the HP-41, a complete collection of functions starting immediately after one END and going on to the next END is called a program. The functions in a program, recorded one after another, are given numbers, and each such numbered function is called a **program step**, a **step** or a **line**. The use of labels, RTNs and ENDs will be covered in detail in Chapter 6.

The trouble with HP-41 programs is that one person's program may be another's subroutine. A program written to calculate hyperbolic sines may do a complete job for one owner, but it may be used as a subroutine by someone else who is working out complex integrals. As a compromise, the hyperbolic sine routine can be called a **routine**, and so can other similar programs. In fact any subroutine or program could be called a routine in the right circumstances.

HP-41 functions always consist of just a name (sometimes followed by a parameter) and are recorded in routines as one step by that name alone. These functions are written in the machine-level language of the HP-41 and are recorded only on ROM, in the HP-41 or on plug-in modules. HP-41 routines, on the other hand, consist of collections of steps, and can be recorded either on ROM or in RAM in the HP-41 continuous memory. When a routine is executed as a step in a program, the step is stored either as XEQ "routine" if the "routine" is in RAM, or as XROM "routine" if it is in a plug-in ROM.

The following programs show examples of functions and routines. Each program asks the user to provide a value X, then calculates a mathematical function of X and displays the result. The XEQ and XROM instructions are obtained by pressing the XEQ key, then ALPHA, then spelling out the function name, then pressing ALPHA again. This is called **ALPHA execution**, it will be covered in detail in the next chapter.

```
01♦LBL "CALC1"
02 "WHAT IS X?"
03 PROMPT
04 SIN
05 "SIN(X)="
06 ARCL X
07 AVIEW
08 END
```

```
01♦LBL "CALC2"
02 "WHAT IS X?"
03 PROMPT
04 XEQ "SINC"
05 "SINC(X)="
06 ARCL X
07 AVIEW
08 END
```

CALC1 calculates the sine of X, using the HP-41 function SIN at line 04. CALC2 calculates the sinc function of X [SINC(X) = SIN(X)/X] using a user-written routine SINC. In the first case, the function name appears without quotes, since it is an HP-41 function. In the second case, the name appears as XEQ "SINC" since SINC refers to a routine name in RAM memory.

```
01♦LBL "CALC3"
02 "WHAT IS X?"
03 PROMPT
04 XROM "SINH"
05 "SINH(X)="
06 ARCL X
07 AVIEW
08 END
```

```
01♦LBL "CALC3"
02 "WHAT IS X?"
03 PROMPT
04 XROM 01,33
05 "SINH(X)="
06 ARCL X
07 AVIEW
08 END
```

CALC3 calculates the hyperbolic sine of X, using the SINH routine in the HP-41 plug-in MATH ROM. The program only stores the information that step 4 will execute function 33 in ROM module 01. The listing on the left was

printed with the Maths module plugged into the HP-41, so the HP-41 could check the routine name. The version on the right is the same program, but was printed with the Maths ROM removed from the HP-41. Since the module is unavailable, the name of the routine cannot be found, and a numerical identifier is printed on line 04 instead. In both cases, step 4 begins with XROM which means "execute a routine from a ROM". If a program tries to use a function or program from a module that has been removed then it will stop at the XROM line and will display the message NONEXISTENT.

```
01♦LBL "CALC4"
02 "WHAT IS X?"
03 PROMPT
04 SINH
05 "SINH(X)="
06 ARCL X
07 AVIEW
08 END
```

```
01♦LBL "CALC4"
02 "WHAT IS X?"
03 PROMPT
04 XROM 31,25
05 "SINH(X)="
06 ARCL X
07 AVIEW
08 END
```

CALC4 is the same as CALC3 except that it was written while a special Maths module was plugged into the HP-41. This module provides a SINH function, written in machine language, not a SINH routine. When the module is plugged in, the function appears like an ordinary HP-41 function as is seen in the listing on the left. When the module is removed, the numerical identifier shows up as an XROM but with a different identifier from that in CALC3.

The programs in a plug-in module are made up of collections of functions, just like programs you write yourself in RAM memory. You could therefore print out a program from a module, or copy it out onto a piece of paper, then write it line by line into RAM. This operation is made much easier by a special COPY function which copies a whole program directly from a module to RAM. You can then change the copy if you like, and use it instead of the original in the module, even if you take the module out of your HP-41. Functions in modules are different, they are written in HP-41 machine-level language and cannot be copied into RAM, so they can only be used when the

module that contains them is plugged in.

Whenever you write a program and refer in it to a routine or to a function by spelling out the name of the routine or function (using Alpha execution), the HP-41 immediately tries to find the name you have spelled out. This is done so that the name can be correctly stored in the program as a function name, or as a reference to a ROM program, or as a reference to a program in RAM memory. The HP-41 refers to the names of functions and routines by means of three separate lists called **catalogues**. The first catalogue (called CAT 1) gives the names of all routines in RAM memory, identified by global labels; these are programs and routines you have written yourself or read from some device such as a Card Reader. The second catalogue (CAT 2) contains the names of all ROM modules plugged into the I/O ports (plus the Time functions and Extended Functions built into the HP-41CX), and the names of all routines and functions contained in these modules. The third catalogue contains the names of those functions that are built into all HP-41s (not the additional functions in the HP-41CX). You can examine these catalogues yourself by pressing CAT (SHIFT,ENTER) followed by a 1, a 2, or a 3. Notice that CAT 1 contains global labels and the corresponding ENDs but no function names, CAT 2 can contain routines and functions, CAT 3 contains only function names.

The catalogues are searched in the order of their numbers, starting at 1 and ending at 3. This is sensible because it allows you to write your own alternatives to the HP-41 functions. For example you may want to write your own ATAN routine as you are not sure if the HP-41 ATAN function works the way you want it to. If you subsequently XEQ "ATAN" then your own version will be used as it will be found in CAT 1, which will be searched before CAT 3. (If you press SHIFT, TAN then you will still use the HP-41 function because this is tied to the SHIFT, TAN key.) Plug-in modules (CAT 2) are also searched before CAT 3. A function called FACT in a plugged-in ROM module will be found and stored (or performed) instead of the CAT 3 function called FACT. Should you write your own program and give it a particular name, forgetting that the same name is used in CAT 2 or in CAT 3, then your own program will be referred to later when you refer to

that name. The CAT 2 or CAT 3 item will be ignored, and you might not even be aware that you had used a duplicate name.

If you want to write a program which uses a function or routine from a plug-in accessory, you can record a step to execute the function even if the accessory is not attached. (Press XEQ and spell out the name of the function or routine.) The step will be recorded as XEQ"function" and will work rather slowly when the program is run (because the HP-41 will have to look for the name in Catalogues 1 and 2), but the function or routine will be found if the accessory has been plugged in. A whole lot of trouble can be caused if more than one program or function in the HP-41 or in plug-in accessories have the same name. Details will be given in Chapter 12, for now you should bear the following in mind.

When you spell out a name (Alpha execution - remember?) the HP-41 searches for a program or function with that name by going through different possible locations in the following order:

1. Your own programs in RAM memory (CAT 1). The search begins at the end of CAT 1 and goes backwards so that the most recently created programs are checked first. This means that you can write two programs with the same name and the more recent one will be used without your having to delete the older version.
2. Time module, printer and HP-IL if any of these are connected (these come at the beginning of CAT 2 whichever I/O port they are plugged into).
3. Any other plug-in modules or devices (the rest of CAT 2), going through ports 1,2,3 and 4 in that order. (Extended Functions are searched last if you have an HP-41CX.)
4. The functions built into the HP-41 itself (CAT 3).

From this you can see that a program you have written yourself and called SINH will be executed in preference to a program or function called SINH if that is in a plug-in module. A function called SIZE? on a module in port 1 will be executed in preference to a function with the same name on a module

in port 2, and so on. A function called SST (there is one in the HP-IL Development module) will get executed in preference to the HP-41 function SST if you execute it by spelling out its name (the SST key will still execute the normal HP-41 function). You should avoid writing programs with duplicate names, and you should unplug modules that contain duplicate names. If you are obliged to have several modules plugged in and some names are duplicated then plug the modules into the ports in such an order that the functions you want are on a module in a lower-numbered port than other modules which contain the same name. (Read Sections 8.3 and 12.3 for more details.)

If a particular program or function is named on a key (see Section 4.3.2 for details of assigning functions and programs to keys for use in USER mode), then that program or function will be used without a search through the catalogues for the name. Thus you can assign a function to a key, then write a program with the same name, and then refer to the function by pressing the key and refer to the program by spelling out its name. More details of how the catalogues are searched will be provided in Section 6.3; details of assigning names to keys will be given in Section 4.3.2.

3.6 Some common problems.

Certain problems are encountered often, particularly by new users of the HP-41. Now that the required words have been explained, the problems can be outlined, and you might even find a few solutions here. This list is given here for the sake of completeness, do not let it frighten you; the HP-41 is a reliable machine and most users never have serious difficulties.

i. Unrecognised instructions and confusing characters

You are copying a program from a book or article into your HP-41 when you come across a function that you do not recognise. For example you are ploughing through a long program and in the middle you find

100 OFF

What is that on line 100? If you press the ON key, you will turn the HP-41

off, but you will not record OFF in a program. There is no OFF function on the keyboard so what do you do? You must use Alpha execution; press the XEQ key, then ALPHA, and spell out the unrecognised function, then press ALPHA again. This was described above in Section 3.5. Whenever you come across a function you do not recognise, you should press XEQ,ALPHA, then spell it out and press ALPHA again. This will also work for functions on plug-in ROMs and even for functions on the Normal keyboard. If you see XEQ "name" or XROM "name", you should do the same, press XEQ and spell out the name in ALPHA. If the unrecognised name comes from a plug-in module or a peripheral device such as the printer or Card Reader then the fact should have been mentioned in the program description. Printer and Card Reader functions will be covered briefly in Chapter 12. By the way, OFF is a function used in programs to turn the HP-41 off since a program cannot press the ON key.

Remember always to press XEQ before you spell out the name, otherwise you will simply enter a line of alphabetic text, not a function. If a function has a space in the middle, spell out only the part before the space; the function will prompt for the part after the space. For example if you see **DSE Y** press XEQ, Alpha, D, S, E, Alpha. After a moment the HP-41 will display **DSE __** and you will be able to fill the prompt by pressing . and then Y. You should not try to spell out the whole function or the HP-41 will look for a function called "DSE Y", which is not the same thing at all.

When you see XROM "name" do not try to spell out the word XROM, just press XEQ and then spell out the "name". The difference between XROM and XEQ was explained above in Section 3.5.

Program listings can be printed in lower case mode and with double-wide characters. A long program line printed in double-width mode will overflow onto the next line of a listing; the few odd-looking letters at the beginning of a line are actually part of the previous line. Do not forget that numbers can be displayed (and therefore printed) with or without digit grouping marks at every third digit. The role of the dot and comma can be

reversed for use with European notation. As a result you might see a printout with a comma on its own on a line; this will be the same as a decimal point on its own; both have the same effect as a zero.

The up-arrow character ↑ is used for three separate kinds of function and this can be confusing. In the middle of an instruction ↑ means "to the power of", for example Y↑X means "Y to the power of X". At the end of an instruction ↑ means "move the RPN stack up", for example ENTER↑ means "complete the number in the X register, then copy X into Y, Y into Z, and Z into T" (this will be explained in Section 4.2.2). An up-arrow put in front of an instruction or program name in the ALPHA register means that the instruction is to be performed by an alarm, see Chapter 9.

You may have come across the instruction /// among the instructions for using programs in HP-41 Solutions Books. This is Hewlett-Packard's way of writing "press the SHIFT key" in these books. Sometimes they use a small black rectangle or the letter d followed by a full stop instead.

When you key in a program from a listing in a Solutions book, you may come across functions that you do not recognise. These might be functions that you have forgotten about, or they may be functions from some peripheral or plug-in module; this should be mentioned in the program description, but a few programs are provided without an explicit mention of the fact that they use a printer or a Card Reader. Check above for advice on unrecognised functions.

You may also have come across program steps which consist entirely of an exponent, something like

97 E67

97 is the line number and E67 is a number with only an exponent. You cannot enter a line like this from the HP-41 keyboard, but it is just the same as 1 E67 which you can enter by pressing EEX,6,7 . The exact reason for this kind of line will be explained in Section 14.1.

ii. Foggy functions.

This delightful name was used in HP Key Notes to describe functions whose behaviour the user is not sure of. A prime example is the MOD function - OK it gives a remainder, but is it of X/Y or Y/X ? What happens when X or Y are negative, or what if either is zero? The simple answer is to check the manual if you are even a little uncertain. Otherwise you may forget that you were unsure of the function's meaning. Then you may get the wrong answer if you are doing a keyboard calculation or you might write an incorrect program. You will be totally lost when you come back to the program later and begin to get wrong answers. If you do not have a manual to hand, check any other HP-41 book you may have nearby, even this one (the MOD function is described briefly in Section 5.2). Never put a function into a program thinking "I'll check it later" - you'll forget. If you have no helpful books or journals nearby, try out the function in Run mode. You cannot physically harm the HP-41 by wrong use of a function, at worst you will lose the contents of RAM memory and this is preferable to building a bridge that will collapse. Check several examples of the function though, this should prevent you from making incorrect assumptions based on incomplete information. With foggy functions the motto has to be: if in doubt, try it out!

iii. Sticky keys.

If a keyboard has been used a great deal, or if the HP-41 has been dropped, some of the keys may become "sticky". They might not respond at all, or they may need to be pushed very hard, or they could do their job twice over at a single push (this is called **bounce**). It is worth checking if a piece of dirt is stuck on one of the sides of a key, but there is little else you can do about the key except hope that it will get better: they sometimes do. If you do not want to send your HP-41 for repair then you may be able to reassign the key's functions to some other key. With Synthetic Programming (see Section 15.2) it is even possible to reassign the numeric digits; the keys 0, 1 and point are often the first to suffer faults

because they are used the most. If you want to use the sticky key for entering a letter then you can write a short program which appends that letter to the Alpha register, then assign that program to another key.

The toggle keys at the top of the keyboard are especially likely to be damaged if the HP-41 is dropped. The USER key can be replaced by SF 27 and CF 27. The ALPHA key can be replaced in certain cases by executing AON to get into ALPHA mode and by single-stepping an AOFF step in a program to get out of ALPHA mode. USER, PRGM and ALPHA can all be assigned to other keys by use of Synthetic Programming (Section 15.2). You can assign OFF to a key but turning the HP-41 on if the ON key is damaged can be very difficult. It is best to execute the ON function so the HP-41 does not turn off at all or to set a repeating Time module alarm which turns the HP-41 on every hour in case you need it.

If you have a Wand then you can do entirely without the keyboard. The Wand will even turn the HP-41 on if you scan some dark lines on a light surface, so you can turn the HP-41 on with the Wand without using barcode.

iv. Odd displays.

Sometimes the display does odd things or you do not know how to restore it to normal. If BAT comes on at the bottom left, you need to change your batteries soon. If RAD or GRAD appear at the bottom, you are in radians or gradians mode and you can get rid of these by executing DEG. All of the special annunciators have been described in Sections 3.2 and 3.3. Nothing unusual should happen to numeric displays, at worst they will go into exponential notation if the numbers are too big or too small to be displayed otherwise. You can control this with the FIX, SCI and ENG instructions followed by the number of digits you want to display. Read the first paragraph of Section 4.4.1 for details of when these displays round a number and when they truncate it. The printer function BLDSPEC, and also some Extended Functions and many Synthetic Functions can produce unusual display characters. There are also some display errors, described in Appendix C.

The HP-41 occasionally displays error or warning messages that you may not be familiar with, for example ROM means you have tried to edit a program in ROM (Read Only Memory, so you can not alter it). All the messages are described in the relevant manuals. You can clear messages created by the HP-41 or by yourself by using the backarrow key (see ix. below). When a program is running, the **execution indicator** (or **flying goose,** ⤜) normally appears in the display and moves across it. The goose takes one step to the right every time the program executes a LBL function, so it may stay still for a long time if the program has few labels.

Finally, the display may go completely crazy if the HP-41 is faulty. It will do this as well if it is disrupted by static electricity, or confused by the presence of incompatible plug-in modules. It will show unknown characters, blink on and off, or even fade in and out slowly. See vi. and xii. below, if the problem is not explained there then you will probably have to get your HP-41 repaired.

v. NONEXISTENT and DATA ERROR.

Most error messages cover a limited number of possibilities, but these two, particularly NONEXISTENT, cover many errors. When you see NONEXISTENT, and you are not sure why, you should check the manual but also remember the following. NONEXISTENT can arise when you use a function which could be alright, but is wrong in this particular case. For example statistics functions can be alright at one time, but when you change SIZE then some or all of your statistics registers become NONEXISTENT and you get this message. Or you may be using a function from a plug-in module which has been taken out. (In the case of HP-IL you may be using an HP-IL printer function with the printer switch set to Disable.) You may also be referring to a program or a label that you have deleted. A program written on an HP-41CX may give NONEXISTENT when you try to run it on an HP-41C or CV because the HP-41CX has extra functions. The flag functions (SF__ , FS?__ and so on) can all be recorded in a program with any parameter from 00 to 99, but only flags 00 to 55 actually

exist, and only flags 00 to 29 can be altered. You can therefore write a program with lines such as SF 99 or FS?C 32 but you will get NONEXISTENT when a program comes to any use of flags greater than 55 or to any attempt to change flags greater than 29.

The most difficult NONEXISTENT messages to trace are caused by Time module alarms. A control alarm can be set to run a program or execute a function at some future time. When the alarm time is reached, the alarm goes off and gets deleted. Only then does the HP-41 try to execute the function or program. The function or program name may have more than six letters, the function or program may be on a module that has been removed, or the program may have been deleted. In all these cases you will see NONEXISTENT, but you will not know what is NONEXISTENT since the alarm has already been deleted. If the alarm at fault goes off while a program is running, the HP-41 will stop and you may think something has gone wrong in the program itself. So be careful with alarms.

DATA ERROR also covers many errors though it is easier to check them. If you want to look at DATA ERROR more carefully then try Exercise 3.B.

vi. Incorrect behaviour, crashes and static electricity.

A **crash** is what happens when a computer stops working or **hangs up**, and does not respond to commands. On the HP-41 this can happen because you have done something the designers did not allow for (this will be covered in Chapter 14), because the HP-41 has been disrupted by static electricity, or because it is faulty. (Instead of failing to respond to the keys at all, the HP-41 may also react with the wrong functions when keys are pressed. In particular the toggle keys may behave as if something had been assigned to them and other keys may act as if adjacent columns had been exchanged on the keyboard.) The HP-41s are not very well protected against static electricity. The worst affected are those made before 1981. Static electricity can also cause the HP-41 to turn on with an unexpected (and most unwelcome) MEMORY LOST, or to get MEMORY LOST when you plug something into a port. (See Section 2.7 for advice on this.)

Alternatively it may come on but refuse to respond to the keyboard, and may show the unusual displays mentioned in iv. It may also sound unexpected tones, or chirrup softly as if it was trying to sound a tone but not managing. In all these cases refer to the advice on restarting the HP-41 given in Section 4.1.

The worst case is if a static charge disrupts just a few registers, or alters a few flags, or part of a program, and you do not notice what has happened, but you get a wrong answer. If you are running really important programs you should run them more than once, and if you are using data that are put into the HP-41 separately from the program, you should put the data in a second time (or at least double-check them). Always be aware that static problems can occur, particularly:

When you turn the HP-41 on or plug something in.
Under cold, dry conditions.
On nylon carpets or near other material that encourages static.
During thunderstorms!

If none of the above seems to fit your problem then check the rest of this chapter, and check Section 4.1 but be aware that the trouble could well be due to a real fault in your HP-41. Sometimes a part of the RAM memory can be faulty and data or program steps will not be recorded as you enetered them.

vii. System flags.

Many of the flags are used to control HP-41 system features. For example clearing flag 26 disables all tones and beeps until the HP-41 is turned off next. You should know what the various flags do; see the list in Appendix D and the details given below and in Section 4.4.1.

Flags 12 to 19 are used to control devices such as the printer and card reader; these flags can be reset if necessary. If you turn the HP-41 off, they all get cleared and you must remember to set them again as necessary

when you turn on again. Flags 31 to 35 also control external devices but you cannot alter these flags in any normal way unless the external device itself lets you alter them. Sometimes the state of these flags will be altered by static electricity (see above). If flag 33 gets set then you will not be able to use any HP-IL devices and should flag 35 get set then the Auto-start module will not work. Flag 33 can be cleared if you have an HP-IL Development module, but otherwise the only normal way to clear these flags is to do a MEMORY LOST. Two special ROMs are however available with functions that will toggle any flag (see Chapter 12) and Synthetic Programming can be used to do this too (see program SCF in Chapter 14).

Flags 21 and 55 control printing and the display according to whether a printer is attached or not. Flag 21 can stop program execution during VIEW or AVIEW if it is set but flag 55 is clear, so be careful if you are using it. Check the manual or Appendix D if you are not sure about flag 21.

Setting flag 25 allows the HP-41 to ignore an error. The function causing the problem (for example LOG of a negative number or RCL of a nonexistent register) is ignored, the stack remains unchanged, and flag 25 is cleared (so the next error will not be ignored unless flag 25 is set again). Ignoring an error can be useful (for example if you are taking square roots and know that you will sometimes come across numbers that should be zero but are actually very small and negative), but it can cause havoc if you ignore a problem that you should not have ignored. You should therefore set flag 25 only if you know an error might occur and that you can deal with it, then you should clear flag 25 as soon as possible. If something very strange is happening in a long program, check that flag 25 is clear, so that errors will not be ignored. Flag 24 is similar to flag 25; it allows the HP-41 to ignore overflow errors (numbers too large for the HP-41 to store are turned into 9.999999999E99, or -9.999999999E99 and the HP-41 carries on calculating). Flag 24 is not cleared after an overflow, and it too can cause you to miss an error that should not be ignored. Furthermore, flag 25 is not cleared if flag 24 is set and an overflow error occurs. Be careful with flags 24 and 25, and check if they could be to blame for strange results in programs or during keyboard calculations.

viii. Loose connections.

When the HP-41 is attached to something through a cable, the connection can come loose. Even worse is the fact that cable connections can break inside the cable, showing no external fault. The main culprits here are the AC chargers which can develop a loose connection or a complete break at the point where the cable enters the connector plug. When this happens you get a BAT warning or the HP-41 stops working even though the AC adaptor is connected. This has been a fault on previous HP calculators which ran off the mains and could not recharge their batteries if the AC adaptor was plugged in but the wire had broken at the plug. It is often possible to get a plug to work again by twisting the wire that comes into it. You can relieve the strain on the joints by strengthening them with acrylic cement or some other substance that will harden around them.

Another item that can suffer the same fate is the pair of HP-IL loop wires which can break at the point where they come out of the HP-IL module if folded too hard. The Wand cable should not be wound too tightly either.

ix. Using the backarrow key.

The **backarrow** key, also called the **correction key** or the **delete key**, can do many things, and it may not always be clear what is happening while you are using it. During the entry of a number or an ALPHA text string this key deletes digits or characters from the end of the string one at a time. (An exception to this is that the minus sign in front of a number is deleted at the same time as the last digit, other than zeroes or decimal points, is deleted.) When digit entry or ALPHA entry has been completed, the same key deletes the whole number or text string, and it also does this during entry if it is pressed after SHIFT. ALPHA entry can be restarted by pressing APPEND (SHIFT, K) and the backarrow key can then be used again for corrections. ALPHA entry can also be restarted by the ARCL function, but both these methods only work during keyboard operations, not in a program. Digit entry cannot be restarted by any normal HP-41 function, which is a

nuisance if you want to change the last digit of a long number. A short program to re-enable digit entry is provided in Section 16.3.

When a message is displayed by an HP-41 operation, or by AVIEW, VIEW or PROMPT, the backarrow key deletes the message and restores the normal display of X or ALPHA or the current program step. A second press of the key is then needed to clear the value in X or Alpha. Some care is needed if you are in ALPHA mode: in this case keyboard execution of AVIEW does not display a message, so the first push on the backarrow key clears the Alpha register. On most HP-41s execution of PROMPT in Alpha mode does not display a message either; the contents of the Alpha register are displayed anyway, so the first push of the backarrow key deletes the contents of the Alpha register. Only the oldest HP-41s (those with bug 3, see Appendix C), display a message and retain the contents of the Alpha register if backarrow is pressed once following a PROMPT in ALPHA mode.

In PRGM mode the key deletes characters or digits one at a time during character or digit entry, otherwise it deletes one whole program line every time it is pressed, and then displays the previous program line. This means that repeated use of the key in PRGM mode deletes program lines going backwards. If you are entering a function that prompts for a parameter, you can use the key to delete characters or digits from the prompt so long as there are any left. If there are no characters or digits in the prompt, the key cancels the function itself, and also clears ALPHA mode if an Alpha parameter was being supplied.

Other things that are cancelled by this key are Catalogue displays (but only if they have been halted), Stopwatch execution (you can also exit from the Stopwatch by SHIFT,backarrow) and Card Reader prompts.

x. Misunderstanding special HP-41 features.

A number of HP-41 features that differ from other calculators or from what the user expects may also cause trouble. (This is not quite the same as the "Foggy functions", which are just misunderstood.)

a. The subroutine return stack allows for a maximum of six subroutine calls awaiting a RTN or END. If you use more than six XEQ or XROM calls, only the first six returns will be honoured, and your program will stop at the seventh. In cases like this it is better to use GTO and save return values as indirect addresses in storage registers. Alternatively, use Synthetic Programming to lengthen the return stack; see Section 14.11. A control alarm that goes off while a program is running and executes another program also uses the subroutine return stack since it acts as if a subroutine had been called by the running program. This too can take you past limit of six returns.

b. Unlike certain other calculators and some computers, the HP-41 does not automatically clear a flag when it tests it with the FS? or FC? functions. Use the FS?C and FC?C functions instead.

c. Certain I/O features of the printers and card reader are also unusual; these are explained in Chapter 12.

d. The HP-41 works to 10 digit accuracy, so it can obtain results more accurately than many computers, but it cannot be perfectly accurate. You should not be surprised if results involving very small or very large numbers, and particularly small differences between larger numbers are not very accurate. Results smaller than 1E-99 or -1E-99 are too small for the HP-41 to deal with (they are called underflows) so they are always replaced by zero. Results over 9.999999999E99 are replaced by this number (negative results are replaced by minus this value) if flag 24 is set and during statistics operations.

e. An unusual feature which can confuse but is very helpful concerns the filling of numeric prompts. The top two rows of keys can be used to fill in numeric prompts with numbers from 01 to 10. See the third paragraph of Section 3.4 for details if you are confused by this. One trick using these keys is to assign TONE to all of them. Then in USER mode you can press any one of them twice in succession to get a

TONE from 0 to 9 (in the case of a prompt for one digit the TAN key produces a 0 instead of a 10).

xi. Pressing keys too quickly or too slowly.

A few users seem to do everything correctly, yet they still get wrong results or no results. This can happen if you press keys too quickly one after another. The HP-41 reacts to one key and it can keep a record that another key has been pressed, but it cannot react to a third key until it has finished with the first one. If the first key is released and the second one is still being held down then the HP-41 will immediately start to process the second key without going into light sleep mode in between. This is called a **two-key rollover** because the HP-41 rolls over to the second key from the first one without a break. If you press a third key at this stage then it will be obeyed next, but if you have already pressed and released a third key, you will be in trouble. To see if you are pressing keys too fast, go into ALPHA mode, type a row or a column of letters and see if they are all recorded.

If you press a function key, and keep it down for too long, the function will be previewed and then cancelled. The display will show NULL instead to show the function has been cancelled, but you might not notice this if you are copying instructions from a printout. This has already been described in Section 3.3. Functions that need parameters do not behave in this way, so you do not need to worry that the function will be nulled, except that it will be lost if you do not give a parameter within ten minutes and the HP-41 automatically turns itself off. However when you have filled in all the prompts of a function you can null it by keeping down the last key used to fill the prompt. (Try pressing STO 10 and keeping the 0 down; the STO 10 will be replaced by NULL. This will happen too if you press STO, TAN and keep the TAN key down.) Neither numerical entry nor ALPHA entry can get nulled just because you press the key for too long.

xii. Plug-in modules and conflicting names or numbers.

Plug-in modules and devices can sometimes give rise to problems. The most common one is that you may try to use a program or a function from something that is not plugged in. This has already been mentioned in v.

You can also get into trouble if the same name is used by functions or programs in more than one module. For example five different modules contain functions or programs called "SIZE?". If more than one of these modules is plugged into your HP-41 at the same time then you should check which particular version of SIZE? will be used. See the last part of Section 3.5 for details.

A third problem is that modules and plug-in devices use numbers from 1 to 31 to identify themselves to the HP-41. (These are the XROM numbers mentioned in Section 3.5) In some cases the same number is used by more than one item (because there are only 31 numbers but more than 31 different plug-in items). For example both the Games module and the Auto-Start module have been given the number 10. Only one plug-in module or device with a given number should be attached to the HP-41 at a given time, otherwise functions from one module may be confused with those from another, or functions may not be found at all. (Example 4 in Section 15.5 provides a Synthetic way of running routines from two modules with the same XROM number plugged in at the same time.) Warnings are printed in the manuals of most modules concerned and a list of conflicting modules is given in Section 12.5. The most common example of this problem is having both an 82143A printer and an HP-IL module plugged in at the same time. Each one acts as though it were the only printer attached to the HP-41 and this usually stops the HP-41 from working; you should disconnect one of the two or set the HP-IL module switch to "Disable" so as to avoid conflicts.

Finally, you can get strange displays, odd behaviour or crashes (see iv. and vi.) if you plug something in and it has a static charge on it, or if it is not plugged in properly. In cold, dry weather breathe gently into the opening of a module before plugging it in, always push modules into the

ports as far as they will go, and check that your HP-41 is turned off when you plug in or unplug a module. In case of trouble, check that the HP-41 is turned off, pull out the module, then plug the module in again.

xiii. Bugs

If your HP-41 system is doing something highly unusual and you cannot find an explanation in the manuals or here then you may have been unlucky enough to come across a **bug**! This word describes behaviour that is contrary to what the manuals say and to what common sense leads you to expect. Some bugs can bite painfully, others are beautiful. Many computer bugs are removed by the manufacturer sooner or later, so they may not appear on all devices of a given kind. The HP-41 system bugs have therefore been relegated to Appendix C. Check through Appendix C, maybe your bug is there, maybe you have found a new one, or maybe your HP-41 is broken.

Exercises

3.A Reverse Polish Notation was being used on calculators long before electronic pocket programmable calculators were introduced. It is not even obvious whether the first designers had heard of Łukasiewicz, although his notation was certainly in existence before the first electric calculators were made. What is the oldest RPN calculating machine that you can find? Some of these machines had excellent manuals which are still worth reading because they explain RPN well and provide good examples.

For example the Friden 130 and 132 were very popular RPN electronic desktop calculators used up to the early 1970s. They had instruction books with examples of geometry, financial calculations, statistics, algebra, matrices and complex arithmetic using RPN. Many Universities and offices still have these Fridens and their instruction books hidden in cupboards; the examples are worth reading if you can find such a manual.

3.B The DATA ERROR message occurs sufficiently often that it is worth knowing what can cause it. Go through the HP-41 manual and check what mistakes you can make to get this message. You will also find that some functions give DATA ERROR in certain cases and NONEXISTENT in others. One such function is CAT. Store 10, then 100, then 1000 in register 01 and try CAT IND 01 each time. Can you find any other functions that give both types of error?

PART II

Calculating and programming from scratch

CHAPTER 4 - STARTING FROM THE KEYBOARD

4.1 Turning ON, and what to do if you cannot.

You have gone through the fundamentals, or skipped over them, and want to get on with using the HP-41. First of all you will approach the HP-41 through its keyboard and most of the operations described in this chapter will be done from the keyboard, not from programs. Start by pressing ON.

Easy enough? It should be unless the batteries are flat; go back and check Section 2.4. Sometimes though, the HP-41 refuses to turn on or comes on but later refuses to respond to the keyboard. Crashes and "Locking-up" were mentioned in Section 3.6vi, it seems sensible to cover them in more detail in this section about starting. Of course if you have got someone else's HP-41 they may have arranged to prevent unauthorised people from using it. In that case, give it back at once and get your own HP-41!

If it is your own HP-41 that is misbehaving then try doing the following before you decide to send it for repair. (Repairs are not cheap, they cost at least 50 Pounds in the UK, since HP overhaul any HP-41 sent in for even the most trivial repair. Even if the proposed repair costs you nothing because it is done under warranty you will have to do without your HP-41 for about a week.) The suggestions below should be tried in the order in which they are given, since the last few are rather desperate measures. If your HP-41 has started up without any difficulty you may prefer to go on to the next section and come back here only if you get into trouble.

i. First of all, if the display stays completely blank when you turn on, it may simply be displaying an empty text string or one made up of spaces. Press the backarrow key and see if the display becomes a zero or press SHIFT and see if the SHIFT annunciator comes up, showing the HP-41 is on.

ii. Try pressing ON and backarrow in turn a few times, waiting for about ten seconds between key-pushes. After a few times the display should show zero. Do not press backarrow and ON at the same time.

iii. Reread the instructions on battery replacement and check that everything is as described in those instructions. The battery pack sometimes comes loose if the HP-41 has been travelling, or the batteries may have been inserted the wrong way round. The battery contacts may even have got dirty or corroded. In that case clean them. If that does no good then replace the batteries even if they have not been used much.

You can check if the batteries are still supplying power to your HP-41 by holding it close to a radio which is set to receive AM signals. Press the ON key twice, if the batteries are still working then there should be a burst of noise from the radio each time and when the HP-41 is on then the radio should continue to make some noise.

If you have a printer you can check whether the batteries are flat by plugging it in and pressing the PRINT button. If nothing happens press the HP-41 ON key and push PRINT again. The printer will print something even if the HP-41 batteries are too flat to drive the display, so long as the HP-41 is working.

After you have checked the batteries remove any plug-ins from the ports. Some plug-ins, particularly the printer and the HP-IL module can cause a hang-up (especially if both are plugged in and the HP-IL switch is not set to "Disable"). Removing them and pressing ON once or twice sometimes solves the problem.

iv. If you have an HP-41 that can be reset by pressing backarrow and ENTER, pressing ON while keeping the other two down, then releasing all three keys, then try this procedure. This reset only works on HP-41s made since the second half of 1982 (including all HP-41CXs) and on HP-41s that have been repaired since that date. If this does not work

the first time, try it again once or twice. You could also try pressing ENTER alone, keeping it down, then pressing ON and releasing both. Do not press backarrow followed by ON as this normally results in MEMORY LOST.

If none of this works, or if you have an HP-41 that cannot be reset, then press the R/S key and keep it pressed down. Press and release the ON key, then release the R/S key. If nothing happens, try this again. It should clear many problems, in particular it will stop a running program.

v. If you have a Wand, try plugging it in and reading some barcode. The ON key of the barcode keyboard is a good choice, but even a blank piece of paper should do. The Wand is designed so that it will turn on the HP-41 if it is off; this has already been mentioned as a way to turn on an HP-41 whose ON switch is faulty. You can use a Wand with the paper keyboard to do everything if your HP-41 has a faulty keyboard. Functions that are not available on the paper keyboard can be spelled out (see Section 4.3.2) or you can use the barcodes provided in Appendix F.

vi. The HP-41 should have "woken up" by now. If the display is still blank, press R/S again, and if nothing happens, go on to step vii. If the display is showing something and the USER annunciator is on then press the USER key once and check that the annunciator has switched off. If the annunciator is off then press USER twice and check that the USER annunciator turns on and off again. If this has worked then the HP-41 is awake, otherwise carry on with step vii.

vii. One further trick to try before you have to clear memory is to attach a Card Reader and read a card through it. This sometimes jolts the HP-41 out of its locked-up state. The method was suggested by Clifford Stern and is described in Keith Jarett's book "HP-41 Extended Functions Made Easy"; see Appendix A. If the card goes part way through and stops, take off the Card Reader, replace the batteries

with a fresh set, and put the Card Reader back onto the HP-41.

viii. Now try to reset your misbehaving machine by taking the battery pack out and putting it in again. This is one of the methods suggested by HP and it can be done even with the HP-41 turned on, but that clears the time and date settings if you have a Time module, so it has been left till now. After replacing the batteries press ON a few times until the 41 comes on.

A refinement of this method was suggested by C.Close in an article on HP-41 crash recovery in PPCCJ V9N2p21. The whole article is worth reading as it provides interesting information on HP-41 crashes, but the method itself, called the **ON procedure**, is as follows. Press ON and release it. Wait about a minute, then remove the battery pack and replace it. Press the backarrow key for about five seconds; you should see CLX and the HP-41 should be awake. If this fails, repeat the whole process.

This is a good example of an idea provided by one member of a user group to other members in the interest of sharing information and helping each other. This particular idea has been included in the book as it is well-known now, but other ideas remain within user groups only, and readers should seriously consider joining a user group to take part in (and advantage of) this exchange of information.

ix. In some cases, particularly if you have been using Synthetic Programming, the above methods will unlock an HP-41 but only until you turn it off or do a GTO.. or PACK. (So if you have had a crash following some Synthetic Programming you should not turn your HP-41 off and on unnecessarily nor should you press the ON key repeatedly as has been described for some of the methods above.) The "unlocking" procedures will work again, but it may be simpler to use Synthetic Programming techniques to clear the section of memory that is at fault (usually the Buffer Area, see Chapter 8). The alternative is to clear the whole of HP-41 Continuous Memory. If you do not want to do this

but do not feel safe with Synthetic Programming, then try going to a user group meeting and asking for help (see Chapter 13).

x. If none of the above has worked, you will have to clear your HP-41 memory. This will solve all problems except those involving hardware faults. If the fault is intermittent, you can try to save the present status of the HP-41 onto magnetic cards or onto a mass storage medium. If your HP-41 is off, or if you can turn it off, then check it is really off. Now press the backarrow key and keep it down, then press and release ON, then release the backarrow key. You should see MEMORY LOST and the HP-41 should be on the go again. (If you change your mind at the last moment, release the ON key and press and release it again, keeping your finger on the backarrow key all the time; then release the backarrow key.) If the HP-41 cannot be turned off then you might be able to do a MEMORY LOST by finding or creating a synthetic instruction such as STO c and executing CLX, STO c. If you foresee doing this often and you have a card reader, then create a magnetic card with flag 11 set and with a program containing CLX, STO c on it. Then you can read this card through the Card Reader to clear memory whenever necessary.

xi. If the HP-41 is on and cannot be turned off, and you do not have access to a Card Reader or Synthetic instructions, you will have to discharge the HP-41 by taking out its batteries. This can take a very long time, it can be speeded up by putting all the batteries upside-down into the battery pack, and putting the battery pack into the HP-41 for at least 15 minutes. This should drain any residual charge, and the internal protective diodes <u>should</u> prevent any harm to the circuitry. Unfortunately, a few HP-41s have diodes that can not cope with a set of four completely fresh batteries, and the reverse current will cause severe damage. You should therefore use old batteries, and even so consider this as one of the last resorts before sending the HP-41 to HP. (This method was suggested by Bill Hermanson in Volume 9, Number 1, Page 32 of the PPC Calculator Journal, abbreviated as PPCCJ V9N1p32. See Appendix A for a description of this journal.)

Alternatively, you can short together the battery terminals at the two ends inside the battery compartment with a length of wire or even with two fingers of one hand. You may need to keep the wire in place for quite a while to remove the residual charge. Then put back the batteries, the right way round, and press ON one or more times until MEMORY LOST shows up in the display.

xii. If you have got this far then you are in serious trouble. You might possibly be suffering from a bad battery pack or from a loose connection. If you have a rechargeable battery pack, try replacing it with batteries or with a new, fully charged pack. Old battery packs can cause havoc, particularly on an old HP-41, as they may develop a loose connection or be unable to hold a charge. If this has happened to you, do not use the old pack again, though it may be safe to use it on a newer HP-41. The methods given below will probably be necessary to wake up an old HP-41 that has been locked-up by such a faulty battery pack.

To check for a loose connection in the HP-41 or in an old battery pack, tap the HP-41 gently. Set it down on a firm surface, lift up one end about an inch and set it down fairly sharply. The rubber feet will prevent any excessive shock. Never test the HP-41 for a loose connection by banging it hard, this will cause loose connections or worse, not cure them.

Your HP-41 may also have a massive charge of static electricity, or if it is an early model the two display drivers may have failed to synchronize because the batteries are flat. (If you do not understand this, wait till you read Chapter 8.) These problems do not always respond to the solutions described so far. They can be dealt with by taking the HP-41 apart and putting all the parts down separately for a few hours, then putting the HP-41 together again. This can also solve some loose connection problems if you put the HP-41 together again and tighten all the screws (but do not overtighten them). You

must be pretty brave or sure of yourself to do this, since the HP-41 contains CMOS circuits which are very easily damaged by stray charges that you may introduce while taking the HP-41 apart. You also invalidate any HP guarantees if you take your HP-41 apart. Nevertheless, I have saved a couple of trips to HP in this way.

xiii. Unlucky thirteen! If none of the above work or you are unwilling to attempt the more drastic techniques, you will have to get your HP-41 repaired. Some user groups have members with access to service facilities but in general you should pack the HP-41 carefully and send it to your local Hewlett-Packard repair centre. (The addresses should be available from your dealer, some are provided in the back of the Owner's Guides.) Remove the batteries, but send a battery pack or any other items that you think may be at fault. Include a Service Card, and a description of the problem, if you can. Most repair centres post the repaired unit back within five working days of receiving it, but even so this means you will be without your HP-41 for about a week. (If you go to a repair centre yourself they may be willing to do a repair while you wait, but ask before you go.)

4.2 Look after your stack.

4.2.1 Using the whole stack.

Your HP-41, once it is ready to be used, provides a powerful set of mathematical functions for execution directly from the keyboard. These functions are based on the RPN system (explained briefly in Section 3.4) which uses an operating **stack** of 4 registers: X, Y, Z and T. An L register (Last X) is also available, this will be described in Section 4.5. The stack lies at the heart of the HP-41 operations and some forethought in its use can pay considerable dividends.

Many people tend to use only the X and Y registers although the Z and T registers would simplify their calculations. As a very simple example let us take the calculation of **a↑2 + 2a**. The most obvious method is to

calculate this as **a*(2 + a)** as follows. (Each line shows one step and is followed by some comments. The comments are separated from the step by a semi-colon.)

```
Type in a          ; a is now in the register you see, the X register.
Press ENTER        ; a is now copied into the next register, Y.
Type in 2          ; following ENTER the value a is still in X, but 2
                   ; replaces it so the stack now contains 2 in register X
                   ; and a in register Y.
Press +            ; this adds the contents of Y to the contents of X, and
                   ; leaves the result in X.
Type in a again    ; this puts a into X and pushes the previous contents
                   ; of X up into Y again.
Press *            ; this multiplies together the contents of X and Y, and
                   ; puts the result into X so that it is displayed.
```

This calculation only uses the X and Y registers, but it involves typing in **a** twice. Registers Z and T can be used to store extra values and with a little planning one can rewrite the calculation to use them. This can be done by thinking of the equation as **(2 + a)*a** .

```
Type in a          ; a in register X
Press ENTER        ; a is in X and Y, but the contents of X will be
                   ; overwritten if a number is typed in now.
Press ENTER        ; pressing ENTER again copies X into Y and Y into Z so
                   ; that a is in registers X, Y and Z.
Type in 2          ; this puts 2 into register X, replacing a but leaving
                   ; the value a in registers Y and Z.
Press +            ; this adds registers X and Y, putting the result (2+a)
                   ; into X and copying register Z into register Y so that
                   ; a is available in Y for the next step.
Press *            ; this multiplies together the values in X and Y,
                   ; giving the result (2 + a)*a.
```

In this case we use the Z register to store a second copy of **a** and thus

replace the second entry of **a** (which required a press of the ENTER key). It may have been rather difficult to follow what was happening to the contents of registers X, Y and Z during the above example. It is often helpful to use a **stack analysis form** such as the one in Figure 4.1. This shows each step performed and the contents of X, Y, Z and T after that step. Figure 4.1 shows how the stack contents move during the first calculation above, and Figure 4.2 shows the stack movements during the second calculation. Some functions cause **stack lift** which means that the contents of Z are copied into T, then Y is copied into Z, then X is copied into Y. Other functions cause **stack drop** which means that Z is copied into Y, then T is copied into Z. What happens to X depends on what the function does. The stack analysis form helps you see exactly what happens in each case. Some functions like ENTER and CLX are said to **disable stack lift.** This means that a new value put into X will replace the old value instead of pushing it into Y. The figures have an extra column next to the X column and a cross in this means that stack lift has been disabled.

Step	Stack Lift Disabled	X	Y	Z	T	Comments
a		a				Puts a in X
ENTER	x	a	a			Copies a into Y
2		2	a			Puts 2 in X
+		2+a				Adds and puts in X
a		a	2+a			Puts a in X again
*		a(2+a)				Final result in X

Figure 4.1 : First Stack Analysis Form for (a+2)*a.

Step	Stack Lift Disabled	X	Y	Z	T	Comments
a		a				Puts a into X
ENTER	x	a	a			Copies X into Y
ENTER	x	a	a	a		And again
2		2	a	a		Puts in the 2
+		2+a	a			Adds,Z drops to Y
*		(2+a)a				Multiplies

Figure 4.2 : Second Stack Analysis Form for (a+2)*a.

There are some alternative ways of checking stack contents. The HP-41 Standard Applications Module (and also the Standard Applications Handbook which came with each new HP-41C and CV) has a simple RPN primer program to display the entire stack after each arithmetic operation and you can use that instead of a stack analysis form. Printers and video interfaces on HP-IL have a stack trace option (see Section 12.4) which lets them display the contents of the stack after each operation, but this option does not work on the HP82162A printer. On the HP82162A and HP82143A printers use the PRSTK command to print the contents of the stack registers. If you have a Card Reader, use its 7PRSTK function, this prints the stack contents if you have a printer attached, otherwise it shows the contents of the four stack registers one by one in the display. After some practice with any one of these methods you will be able to visualise the whole stack and its contents.

Let us try another example of stack use, with the stack analysis form to help see what is happening. To save space, instructions such as "type in **a**" or "press 2" will be shown simply as "**a**" or "2". Say we want to calculate **(a*b + c/d)*sin(e)** with one value of **e** but various values of **a**, **b**, **c**, and **d**. First we work out SIN(e) and store it in a register, say register 05, by doing STO 05. Now we do:

a, ENTER, **b**, *, **c**, ENTER, **d**, /, +, RCL 05, *

Check this on Figure 4.3 and see that SIN(e) is still in register Y. We could have skipped the RCL 05 step and still got the same result. The stack drop has actually left SIN(e) in registers Y, Z and T because the contents of T are always copied into Z when the stack "drops". To calculate the expression **(a*b + c/d)*sin(e)** again with different values for **a**, **b**, **c** and **d** it is enough to press CLX (or RDN) and then repeat the operation above, without the RCL 05 step.

This example shows that use of a stack analysis form can remind you when a useful value is still in the stack so that a numbered data register need not be used. The message is simple: be aware of all the registers in your stack and of what they contain. When calculations are to be performed manually from the keyboard, this awareness of the stack can save keystrokes, but in programs it has a further advantage. This is that long programs use many data registers, and calculations which use the stack as efficiently as possible will avoid using additional data registers where this is unnecessary. If you are using two programs each of which needs 50 data registers, then you will be very glad to use the stack registers for storing some of your intermediate values.

Step	Stack Lift Disabled	X	Y	Z	T	Comments
a		a	sin(e)			Puts in a
ENTER	x	a	a	sin(e)		Makes room for b
b		b	a	sin(e)		Puts in b
*		ba	sin(e)			Multiplies
c		c	ba	sin(e)		Puts in c
ENTER	x	c	c	ba	sin(e)	Makes room for d
d		d	c	bə	sin(e)	Puts in d
/		c/d	ba	sin(e)	sin(e)	Divide, stack drops
+		c/d + ba	sin(e)	sin(e)	sin(e)	Add,stack drops
RCL 05		sin(e)	c/d + ba	sin(e)	sin(e)	Recall sin(e)
*		result	sin(e)	sin(e)	sin(e)	Get result in X

Figure 4.3 : Stack Analysis Form for **(a*b + c/d)*sin(e)**

4.2.2 Stack manipulation functions.

Many readers will already be familiar with the ideas outlined above but they, as well as readers to whom the ideas are new, may want to be reminded of the stack manipulation functions available on the HP-41.

0-9
EEX
CHS

These **number entry** keys build up a number in register X. The number consists of a signed mantissa followed by a signed exponent (power of 10). The mantissa can have no more than ten digits. The exponent is optional but if used with a mantissa that has nine or ten digits it requires that a decimal point should come before the ninth digit. CHS changes the sign of the mantissa if pressed before EEX, otherwise it changes the exponent sign. The decimal point (or comma if you are using European radix mode, see Appendix D: flag 28) on its own acts as a zero (and it is faster than 0 when used in a program).

PI Enters a ten digit approximation for pi into the X register.

ENTER↑ Separates one number from another, copies the contents of Z into T, Y into Z and X into Y. Disables stack lift so that a new value, put into X immediately after ENTER↑, replaces the value in X. In this book the ↑ after ENTER will usually be omitted. ENTER really does two separate jobs (terminating number entry and lifting the stack) which could be done by two separate keys.

RDN Rolls down the stack contents. If the stack is visualised as four registers with T at the top, followed by Z, Y and X, then RDN pushes the contents of Y, Z and T down by one register each, and copies the contents of X into T.

R↑ Rolls up the stack contents. Exactly the opposite of RDN. It can also be imagined to be like ENTER except that the contents of T are copied into X and stack lift is not disabled.

X<>Y X exchange with Y. This is useful if X and Y are in the wrong order, for example if you want to divide the value in X by the contents of Y. Also useful for looking at Y, then replacing it.

X<>__ In addition to X<>Y, the HP-41 has this general function to exchange the contents of X with those of any other register, even the other stack registers, for example X<>Z.

RCL__ RCL recalls the contents of any register, even stack registers, into the X register. The stack is lifted unless stack lift has been disabled by the previous instruction.

STO__ STO stores the contents of X into any register, including stack registers. The previous contents of the register are lost, but the contents of X are unchanged. STO into a stack register does not lift the stack.

LASTX Recalls the previous value of X from register L; see Section 4.5.

CLX Clear X. Clears the contents of the X register, replacing them with zero. This operation is often used to correct a mistake, so it disables stack lift and the value put in directly after CLX replaces the previous value.

Note: Stack lift is disabled by CLX and ENTER, but the next operation enables stack lift again unless it too disables stack lift or is neutral. (Only CLX, ENTER and the statistics plus and minus functions disable stack lift. SHIFT, ALPHA, PRGM, USER and On/Off are neutral: they neither enable nor disable stack lift. CHS and backarrow are also neutral during number entry.) If you want to put a zero onto the stack, CLX alone will not do since a number entry or RCL following it will write over the zero; you should CLX,ENTER or press . or 0.

CLST Clears the whole stack, replacing X, Y, Z and T with zeroes. It does not clear L, so you can clear the whole stack except X by doing +, CLST, LASTX. (Use SIGN instead of + if you know that X might contain Alpha data; see the discussion under SIGN in Section 5.2.)

4.2.3 Stack calculations, rearrangements and operations on X.

Most HP-41 mathematical operations use or alter the contents of register X. The mathematical functions will be covered in the next chapter except for + , - , * and /. These are well understood, though new users should remember that the operations - and / produce the results Y-X and Y/X. If you want X-Y or X/Y you must do X<>Y first. All these four functions save the previous contents of register X in register L (Last X, see Section 4.5) and drop the stack.

The functions ENTER, RDN, R↑, X<>Y, X<>__, RCL__, STO__ can be used in various combinations to produce any required rearrangement of the stack. On previous HP calculators which did not have R↑ it was sometimes much quicker to use - (subtraction) in combination with the other operations to get some arrangements, particularly ones involving zeroes. John Dearing's book "Tips and Routines" gives 4 pages of stack rearrangement operations for the HP-41, and John Ball's "Algorithms for RPN Calculators" gives an even more complete list including all combinations that involve zeroes in the stack.

The operations **ST+__**, **ST-__**, **ST*__** and **ST/__** are obtained by pressing the STO key and then one of the four arithmetic keys. They work like the four ordinary arithmetic operations, but instead of combining X with Y they combine X with any numbered or stack register. For example ST- 23 subtracts X from register 23 instead of subtracting it from register Y. The value in X is not changed and so X is not saved in L. These four operations are called **storage arithmetic**. Some calculators also have corresponding **recall arithmetic**, but not the HP-41. Fortunately, recall arithmetic can be simulated very easily. For example RCL*nn (multiply X by the contents of register nn) can be performed by doing X<>nn, ST*nn, X<>nn. The use of two X<>nn operations will let you perform all four recall arithmetic operations in the same way. The PANAME module (Section 12.8) provides some recall arithmetic functions as well.

Storage arithmetic is particularly helpful when it is used on the stack. For example you can add X to Y and to Z by doing ST+Z, +. The steps ST+Z, +, / let you work out (X+Z)/(X+Y) more quickly than is possible on ordinary RPN calculators. The operations ST+X, ST-X, ST*X and ST/X are especially interesting. ST+X replaces X with 2X without affecting registers Y, Z, T or L. ST-X replaces X with 0 but unlike CLX it leaves stack lift enabled. ST*X replaces X with X squared, again without affecting L. ST/X replaces X with 1, this can be a useful alternative to SIGN which replaces a number in X with 1 or -1 and changes L (see Section 5.2).

These operations on X can be combined to provide additional useful results without altering Y or Z or T. For example ST+X, ST*X produces 4X↑2 and ST*X, ST+X, SQRT produces the absolute value of X multiplied by the square root of 2. An interesting use is 1/X, ST+X, 1/X which produces X/2 without changing Y, Z or T. (2, ST/Y, RDN produces X/2 without changing L.)

RCL X pushes the stack up like ENTER but leaves stack lift enabled. X<>X and STO X do nothing at all except that they enable stack lift if it was disabled. Functions such as these two which do nothing are called **NOPs** (Null Operations), and there are some uses for them in programs, especially in combination with ISG and DSE as described in Section 6.8.

4.3 Make use of ALPHA.

4.3.1 Using the ALPHA register to display messages.

The alphabetic capabilities of the HP-41 allow meaningful messages, instructions and results to be displayed or printed. This can be done from the keyboard, under program control, or from alarms operated by the Time module. All this is done through the **ALPHA register** which can hold up to 24 characters (described in Section 3.2). The information in the ALPHA register is **text** or **alphanumeric** data, which means it contains characters that can be letters, numbers, punctuation marks, or special characters such as %. The ALPHA register can be imagined to be rather like the X register which holds and displays up to 10 decimal digits and two signs, with points

and commas between them. Up to 12 characters in the ALPHA register can be displayed at one time; punctuation marks between other characters do not count among these 12 characters although they do take up one of the 24 places in the ALPHA register.

If the ALPHA register contains more than 12 characters (not counting embedded punctuation marks), then it will initially show the first 12 characters, and then it will roll characters off to the left one at a time, and add new characters from the right, until it is displaying the last 12. This is called **scrolling** and it has the rather disconcerting effect of making the first part of a long message disappear off the display as soon as it has appeared. If a message is over 12 characters long then it is a good idea to start it with a space, so that the first character which scrolls off the display is this space, and the message can be taken in more easily. This tip, along with several others, was suggested by R.Nelson (founder of the original user group, PPC) in the journal, PPCCJ V6N5p32.

You put characters into the ALPHA register (often just called ALPHA or Alpha) by pressing the ALPHA toggle key, then using the keys to type in characters. Keys have letters marked in blue on the lower face, additional letters and symbols are obtained by pressing SHIFT first. The shifted letters are not marked on the keyboard, but they are shown on the ALPHA keyboard sticker at the back of the HP-41, and on the ALPHA keyboard layouts printed in the quick guide and the manuals. So long as characters are being added to ALPHA one after another, they simply add on at the right-hand end. Once you switch out of ALPHA mode though (by pressing ALPHA), text entry is terminated. If you press ALPHA again and start typing in a new text message then the old contents of ALPHA will be deleted and the new message will replace them. CLA (SHIFT, backarrow in ALPHA mode) will also delete the contents of the ALPHA register. Corrections to the contents of ALPHA can, however, be made using the APPEND operation, as will be explained later in this section.

Before going on to more mundane uses of ALPHA, try it out by entering a message as follows. Turn on the HP-41 and press ALPHA, SPACE, T, H, E,

SPACE, L, O, R, D, SPACE, I, S, SPACE, M, Y, SPACE, S, H, E, P, H, E, R, D. When you press the last D the HP-41 sounds a tone, just to let you know that it has no more room for additional letters and will have to delete letters from the left-hand end of ALPHA to make room for any more that you put in on the right. Now press ALPHA, ALPHA and watch the message scroll across the display. Observe that the space at the start of the message makes it easier to read the first letter before it vanishes. While still in ALPHA mode, press AVIEW (SHIFT, R/S); this displays the whole message again, AVIEW is particularly useful for displaying information during running programs. If you had a printer plugged in, AVIEW should have also printed the message (unless the printer was OFF
Scrolling takes up time; you can stop it and get to the last 12 characters at once by pressing any key (even ON or backarrow) while the message is being scrolled. Be careful; if you press a key after scrolling is finished then that key performs its normal action.

The use of the backarrow key to correct or delete the contents of ALPHA has already been described in Section 3.6ix, and some short programs that used ALPHA to display a result were shown in Section 3.5. These programs used ALPHA to create messages such as "SIN(X)=", and then they used ARCL X to recall the value from X into ALPHA and add it to the message. The format of the X value would be the same as that seen in X because the display mode controls how a value is brought into ALPHA, including the rounding. Displays which lose their ninth or tenth digit because it is hidden under an exponent will show all digits in ALPHA, so SCI 9, ARCL X, AVIEW can be used to display all ten digits and the exponent of X. (See Section 4.4.1 for a description of display modes.)

ARCL adds a value to the end of what is already in the ALPHA register, but what if you want to add more than one value, perhaps with some extra text mixed in? Three things can be done to add more text to the right of ALPHA. ARCL can be used more than once; each time it adds to the right. ARCL also enables further addition from the keyboard. Thus ALPHA, Y, =, ARCL Y, F, E, E, T produces the message "Y=1.23FEET" if register Y contains 1.23 and the display has been set to FIX 2 (Section 4.4.1). Text that you have

typed in immediately following ARCL is added to the contents of ALPHA; it does not replace the contents of ALPHA. The third way to add to the right of ALPHA is to use the APPEND function. This function can only be performed by going into ALPHA mode and pressing SHIFT, K. Text that is typed after SHIFT, K will be added to the right of the ALPHA register instead of replacing the previous contents. (When you are doing this from the keyboard, APPEND simply displays the prompt character _ at the right of the text in ALPHA, but in PRGM mode APPEND is stored as a special control character ⊢ at the beginning of a text string.) You can use APPEND, and then delete characters from ALPHA as well as adding characters to it. If ALPHA already contains 23 characters then putting in more characters, or using APPEND to add characters later will make the warning tone sound, but ARCL will not produce the warning tone even if some characters are lost.

Quite often you will find that a message will not all fit into the 12 characters visible in the display at one time. Messages such as

X1=1.234579201E-32

will not fit all at the same time into the 12-character display and can be positively dangerous if the user sees the last 12 characters only; the user may think the answer was 34579201.E-32 since the true decimal point is not visible. Here are some hints on making a message fit into the 12 visible characters.

i. Use a display that shows as few digits as necessary. FIX 2 is better than FIX 8 unless 8 digits after the decimal point are absolutely necessary. SCI n is often a better display setting since it always produces between n+4 and n+6 characters (depending on the signs of the mantissa and exponent) whereas FIX n can produce various numbers of characters.

   This is also good discipline for students who would otherwise write down answers to 10 significant digits just because their calculators give answers to that many digits.

ii. Abbreviate words in messages. Use FT instead of FEET, or SN= instead of SIN=. You may think that SN is not easy to recognise as an abbreviation for SIN, but the context of the problem should make abbreviations clear.

iii. Use the punctuation marks . and , and : as much as you can. They fit between other characters and do not take up one of the 12 character positions unless two punctuation marks are entered one directly after the other. The colon is particularly helpful since it can be used instead of a space to separate words. A display such as X=12.34:Y=56.78 takes up 12 positions and is just as clear as X=12.34 Y=56.78 which takes up 13.

A few additional points now about using the ALPHA register. The single instruction PROMPT can be used in programs instead of AVIEW, STOP. Both methods display the contents of the ALPHA register and stop a running program. PROMPT is usually used to display a message that prompts the user for a value during a running program. It is easy to forget that PROMPT can also replace AVIEW, STOP to display a result and save one program step. Even some HP Solutions books use the latter in places where the former would do. The instruction ARCL can be followed by any stack register or numbered data register, or it can be followed by an indirect address (see Section 5.5) to allow the viewing of a set of registers one by one. The value appended to ALPHA by ARCL is a representation in characters of a numeric value, and this character representation cannot be used in calculations. (The Extended function ANUM converts characters back to numbers; see Section 10.2.) The function ASTO does the opposite of ARCL; it stores a text string, taken from ALPHA, in a data register. Remember that ASTO overwrites the previous contents of the register, and it does not lift the stack, so that ASTO X replaces the previous contents of register X without changing Y, Z or T. ASTO only stores the first 6 characters of the ALPHA register, counting from the left, into a single data register. If there are fewer than 6 characters in ALPHA then these are stored with null spaces to their left, making a total of 6 characters. If there are

more than 6 characters in ALPHA then the first 6 have to be removed before the next 6 can be stored in another register. The function ASHF is used for this purpose; it removes the leftmost 6 characters from the ALPHA register. The HP manuals say that these characters are shifted out of the ALPHA register but it may be better to say that they are deleted. ASHF can be used repeatedly to store up to 24 characters in four data registers. The contents of these registers can be returned to the ALPHA register at a later stage by means of the ARCL function. A text string in a data register can be displayed as text by the VIEW function and text can be ASTOred in X so that a message is displayed instead of a number when X is seen.

4.3.2 More uses of ALPHA.

The second use of the ALPHA keyboard is to provide parameters, particularly for ALPHA execution, already described in Section 3.5. To execute a function that is not on the keyboard, or a program, you press XEQ, ALPHA, then spell out the function or program name, then press ALPHA again. The name that you have spelled out is not stored in the ALPHA register, it is used to find the function or program and then execute it (or store it in a program). If you want to go to a program, but not to run it, you can press GTO and spell out the program name. (You cannot GTO a function.)

To identify a program so that you can XEQ it or GTO it you must have a label to identify the program. Once again you use the ALPHA keyboard: press LBL, ALPHA, spell out a name of one to seven characters including numbers and spaces but not punctuation marks, then press ALPHA again. If you do this in RUN mode, the LBL instruction will stay in the display for a while, then it will vanish, but if you do it in PRGM mode you will create a label.

XEQ and labels will be discussed yet again in Section 6.3, for now it is enough to note that this is the most common way used to execute programs and functions that are not on the keyboard. The other way to execute functions and programs is to assign them to a key (or a shifted key) and to

press that key in USER mode. (Programs can also be executed by pressing R/S if the program pointer is already set to them. CAT 1 can be used to move to a required program.) To assign a key for use with a chosen function or program in USER mode you use the ASN (assign) function. Press ASN, ALPHA, spell out a function name or the name on a program label, then press ALPHA again. Unlike other functions that use parameters, ASN needs two parameters not one, so it now displays ASN aaaa _ asking you to press a key, or SHIFT and a key, to define which key is to be assigned. (aaaa is used here to represent the name you have spelled out. If you press ALPHA twice in succession then no name is displayed, and the HP-41 cancels any assignment that was previously made to the key.) When you press a key, the display shows ASN aaaa rc, where r is the keyboard row in which the key lies, and c is the column. (To be exact, c is the key number in row r, counting from the left.) If you press SHIFT before the key then rc is preceded by a minus sign. Once you have (successfully) assigned a function or a program to a key, you need only go into USER mode and press the key to execute the function or program. This is obviously much simpler than Alpha execution but assignments usually take up space in memory (see Section 8.6).

4.3.3 Extended uses of the ALPHA register.

Although the ALPHA register was originally intended as a place where messages could be created, its uses have expanded a great deal. The Timer module uses it for alarm messages, the Extended Functions and HP-IL system use it to build up, or analyze, file names, commands or text strings. These can be transferred to and from Text files in Extended Memory or HP-IL devices. Further editing features are provided by the Extended I/O module, the HP-IL Development ROM, and the PANAME module.

With Synthetic Programming you can use the ALPHA register for storage of numbers, almost like a set of additional stack registers. You can even create and run short subroutines in ALPHA. The extension of the uses of the ALPHA register is a prime example of the way the HP-41 has grown from a calculator into a pocket computer; these uses are part of the subject of

this book, to be covered in the later chapters.

4.4 Set your status.

4.4.1 Flags

A great deal of control over the status of the HP-41 is available to you. Most of the status information is recorded by flags 00 to 55. Each flag can be "set" or "clear". The flags can be thought of as answers to yes/no questions, or as numbers which can only be 0 or 1. Understanding and using the flags well can significantly extend your control of the HP-41. A simple example of the HP-41 status is in the choice of angular mode. This can be "degrees" mode or "radians" mode or "gradians" mode as described in Section 3.3. The trigonometric mode is controlled by the settings of flags 42 and 43. Each flag answers one question; flag 43 answers the question "is radians mode set?" and if flag 43 is clear then flag 42 answers the question "is gradians mode set?" If the answer to both questions is "no" then both flags are clear, and the HP-41 is in degrees mode.

Similarly the status of the display can be set by means of the FIX, SCI and ENG functions which use combinations of flags 36 to 41. FIX 0 to FIX 9 make the display show numbers without an exponent if possible, with 0 to 9 digits after the decimal point (again if possible). If the number is too large or too small to be displayed in FIX mode then it is shown in SCI mode. SCI n instructs the HP-41 to display a mantissa with one digit before the decimal point, n digits after the point, and an exponent. ENG n is like SCI n but always displays an exponent that is a power of 3, so there are one, two or three digits before the point. The last digit shown is rounded up if the undisplayed part of the number is equal to or greater than 5. However an exponent hides the last two digits of the mantissa if the mantissa display is nine or ten digits long. In this case the rounded digits are hidden, and the rounding is not seen. If you want to see all the digits of a number, set SCI 9 status, then ARCL X and look at the display in the ALPHA register. (If you are not clear on this, do Exercise 4.C.)

Other status information that is controlled by flags includes the role of the point and comma in the display, and switching off of audible signals. The ALPHA, PRGM, USER and SHIFT status are also recorded by flags. Check Appendix D for a list of all the flags and what they do.

Any flag can be tested using FS? to see if it is set and FC? to see if it is clear. For example SHIFT, FS?25 tests if flag 25 is set. From the keyboard, you would get a message "YES" or "NO" to show the answer. If you look in Appendix D you will see that YES means that the HP-41 is set to ignore the first error that occurs, and that no error has occurred since this flag was set. The use of flags in programs will be described in chapter 6. Only flags 00 to 29 can be directly set or cleared; the commands SF (set flag) and CF (clear flag) are available for this purpose. If you set any one of flags 00 to 10 you are saving information for your own future use but not affecting any HP-41 operation. If you set or clear any of the flags 11 to 29 then you are controlling the status of the HP-41 or of a device attached to it. Flags 30 to 55 are used by the HP-41 internal system to keep a record of what it is doing, so they are not normally available for you to change even though you can test them. (You can set some of these flags by using a function that sets the corresponding status; for example RAD sets flag 43.) Flags 00 to 29 can also be cleared by the operations FS?Cnn and FC?Cnn. These two operations test whether flag nn is set or clear, and clear it before going on to do anything else.

What can you use flags for? You can record the answer to a simple yes/no or up/down question, and then you can check this answer at a later stage. You can also define your own modes for the HP-41. For example you may want to calculate distances in either miles or kilometres. Set flag 05 (or any flag you choose) when you are working in kilometres, clear flag 05 when you are working in miles. Flag 05 becomes a mode flag for use in programs; you can check it to see if you are working in miles mode or kilometres mode. Setting flags 00 to 04 also turns on the corresponding annunciators 0 to 4 (see Section 3.2), so these flags can be used to display information concerning a program.

Some subroutines that are used a lot may need to use a flag temporarily while they are doing something. It is best to reserve a few flags for these routines, say flags 08, 09, 10 and not to use them in any other programs that call these subroutines. If you do a lot of matrix mathematics, and your matrix addition routine needs a flag, then let it use flag 10 and make sure that none of the routines which call this subroutine use flag 10.

The message so far is: know your flags and use them correctly. Read Appendix D or the section on flags in the HP-41 manual. (Appendix D has more information though.) Always set any special status that you need, it is disastrous for example to run a program that does trigonometry in degrees and then find that someone or something (like another program) has set your HP-41 to radians mode. Be particularly careful to clear flag 25, the "error ignore" flag as soon as you have finished using it, otherwise your HP-41 may ignore a serious error later on. And remember to reset any flags that may be cleared because the HP-41 was turned off. One extra warning; the HP-41 turns off for a moment when you display the time using the Time module functions CLOCK or SHIFT, ON. Check this in Section 9.2.

4.4.2 SIZE and Σ REG.

Two other functions affect the status of the HP-41. The first one to consider is **SIZE**. The HP-41C has 64 data registers (in addition to the stack and ALPHA). These can be used for storing data (any kind of numeric or text information) and for storing programs, key assignments and additional specialised information. The SIZE function tells the HP-41 how many of these 64 registers are to be used for storing data. If you type XEQ, ALPHA, S, I, Z, E, ALPHA you will see SIZE ___ in the display. You can enter any number between 000 and 063 in reply to the three prompts. For example SIZE 017 means that 17 registers are to be used for data. The other 47 will be used for storing programs, key assignments, alarms and buffers. SIZE 017 is the default when the HP-41C is reset by a MEMORY LOST. If you plug in extra memory modules or a quad memory module on an

HP-41C, all the extra memory is initially added to the data registers, so you can do SIZE 017, then plug in two memory modules (64 registers each) and have a SIZE of 145. On an HP-41CV, 47 registers are again kept for programs, assignments and buffers after a reset, but the only way to change the number of data registers is by means of the SIZE instruction because you cannot plug in or remove any more ordinary memory modules. This is not much good because the extra memory of an HP-41CV should be more fairly divided between programs and data registers. On the HP-41CX the initial SIZE is set to 100. Remember that the lowest numbered data register is register 00, so a SIZE of 50 means that you can use registers 00 to 49. If you try to use register 50, you will get NONEXISTENT.

Since you can repeatedly change the SIZE by executing the SIZE command or by plugging in extra memory modules, you may need to check what the present SIZE is. There is no easy way to do this on an HP-41C or CV, so a great many SIZE-finding routines have been written. The simple-minded ones just count from 0 upwards, recalling each register till a NONEXISTENT error occurs. Cleverer versions use some search method, for example a binary search. This starts at 319 (the largest possible, assuming there are no programs and the search is done from a ROM or from the keyboard), and repeatedly halves the difference between the last successful recall and the last unsuccessful recall. The cleverest methods use Synthetic Programming to recall the SIZE information directly from the place where the HP-41 itself stores it. The Extended Functions module provides a built-in SIZE? function to solve this problem, but several Application modules also have SIZE? functions which check what size you want, not what the SIZE actually is, so you need to take some care with this function name, see Section 12.5. Alternatively, use the PPC ROM function S? (see Section 12.8).

The remaining status-setting function to consider here is Σ REG. This is used to put the six statistics registers where you want them, by specifying the location of the first one. After a reset, this is register 11, see the Owners' Manual for details. If, for example, you are using registers 11 and 12 for some data then you can set Σ REG 021 so that the six statistics registers are in registers 21 to 26. To avoid accidentally destroying

important data when you do some statistical calculations, you can make the statistics registers NONEXISTENT. To do this, set a large SIZE, say SIZE 090. Then put the statistics registers at the top of memory, say REG 080, and then decrease the SIZE again, say to SIZE 050. Any use of the statistical functions will now result in NONEXISTENT and there is no danger of destroying important data. Just as with SIZE, there is no easy way of checking where these registers are. The PPC ROM contains a function Σ? and the HP-41CX has a Σ REG? function to tell you where the first of these six registers is.

If you have a printer available then you can print all the status information discussed in this section. Simply execute PRFLAGS and you will get the SIZE, the number of the first statistics block register, the angle mode, the display mode and a list of all the flag settings.

4.5 LASTX; corrections and constants.

As none of the worthy readers of this book ever make mistakes, there is little point in explaining that LASTX recovers X so that corrections can be made. But once a number is copied from X to LASTX (during a calculation), it can be used repeatedly, not only for corrections, but also for arithmetic with a constant. The normal use of LASTX would be as follows:

You intend to multiply 7.5 by 3.9

```
7.5          ; you put in 7.5
ENTER        ; put 7.5 in Y so you can put in the 3.9
3.9          ; put in 3.9
/            ; whoops, you divided instead of multiplying
LASTX        ; get 3.9 back and ...
*            ; multiply to get back to the original value
LASTX        ; get 3.9 back again and ...
*            ; multiply again to get the required answer
```

Note that LASTX,* was used twice, to multiply by 3.9 two times. Since you never make mistakes, you will not need this for corrections, but you can

use the same process with 3.9 or any other number for repeated multiplication by a constant. First you put 3.9 (or your chosen constant) into the LASTX register by putting the number into register X and dividing by it (or doing some other arithmetic operation) as above. Then you could repeatedly put numbers into X and press LASTX, *. The same process can be used for adding, or subtracting or dividing by a constant.

The function LASTX recalls a value from **register L.** Register L is rather like an extra stack register (in addition to registers X, Y, Z and T). It is used to store the previous value of X after most arithmetic operations so that the function LASTX can be used to bring the value back. Register L can be used just like X, Y, Z and T for operations such as X<>L or ST/L. In fact LASTX is equivalent to RCL L. However register L is not exactly a stack register because it does not move when the stack is changed by roll up, roll down or ENTER. Register L is also not cleared by CLST. You can therefore clear registers Y, Z, and T but keep X unchanged by doing something like +, CLST, LASTX.

Any mathematical function that explicitly changes the value in X is preceded by an automatic STO L. (Except that L is not changed if the function fails for any reason.) HP-41 non-mathematical functions that use X or change it, such as ASTO X, do not save X in L. A few Extended Functions and Time module functions do save X in L though, because they alter X, see Chapters 9 and 10. Mathematical functions that do not normally change X, such as ST+, do not store X in L, even in the case of ST+X. CHS really performs two different operations; during number entry it negates the mantissa or exponent (and it is sometimes called the NEG function under these circumstances). At other times CHS changes the sign of the mantissa of X; this is really a mathematical operation on X, but it is treated like the first case and does not save X in L (even though it does enable stack lift).

Many new users of RPN calculators complain about the absence of a K (constant) feature without, realising that the L register and LASTX provide this same facility. If you need to do a lot of constant arithmetic, you

can write five very short routines that will allow ENTER to act as a constant store key and + , - , * , / to act as constant operations. Write the following five subroutines and assign them to the keys suggested. If you are unsure about assigning keys, check Section 4.3.2, and if you are unsure about writing subroutines or programs then come back here after you have read Chapter 6. The subroutines are:

```
LBL"K       ; Assign this to the ENTER key
STO L
RTN

LBL"+       ; Assign this to the + key
LASTX
+
RTN

LBL"-       ; Assign this to the - key
LASTX
-
RTN

LBL"*       ; Assign this to the * key
LASTX
*
RTN

LBL"/       ; Assign this to the / key
LASTX
/
END         ; This can be an END or a RTN
```

To use these routines set USER mode, type in your constant, press ENTER to store it, then enter any number and press + or - or * or / repeatedly to use the constant. Of course you must not do any additional mathematics that would change the value in register L. If you prefer not to use the

keys suggested above then you can choose to assign the functions to different keys.

An alternative way to do constant arithmetic is to use the stack drop feature that repeatedly copies T into Z. This involves placing the required constant into X, pressing ENTER three times, then repeatedly typing in a number, and pressing + or - or * or / . After each calculation you must press CLX then type in the next number for the calculation with a constant. This seems more cumbersome than using LASTX, but it allows you to subtract from, or divide into, a constant (because the constant is in Y, not in X).

User-written mathematical routines should, if possible, use only the stack and save X in L just as the HP-41 functions do. Using only the stack prevents any conflicts between different subroutines that might want to use the same numbered data registers for temporary storage. Storing the original X in L makes it much easier to treat user-written routines exactly like HP-41 functions. A good example is a version of X↑3 suggested by Frank Wales, published in HP Key Notes, and subsequently included in the HP-41CX manual. The value that was in register X at the start is called x (little x) here to distinguish it from the X register itself.

```
X↑2      ; Obtains x squared and puts the original x into L
X<>L     ; Puts the original x into register X and x↑2 into L
ST*L     ; Multiplies the contents of L by x giving x↑3
X<>L     ; Puts x back into L and puts x↑3 into X
```

At the end of this subroutine, registers Y, Z and T are unchanged. The X register contains x cubed, and register L contains x which was the original value in X. If you compare this with the HP-41 function to square X, you will find that it behaves in exactly the same way.

Register L can also be exploited by programs that need to use a single register, without affecting the stack or any numbered registers. An example of this is the program REGS given in Section 7.2. Experienced

programmers often use functions such as ABS or SIGN to copy X into L. These functions are faster than STO L and use less memory. SIGN is particularly good, since it works for text strings as well as for numbers.

4.6 Efficiency: keyboard operations vs. programming.

If you only wanted a calculator to do keyboard calculations, you are unlikely to have bought an HP-41. (Or did you buy it to impress your friends?) Presumably you will want to use programs to solve your more complicated problems. In the following chapters, there will be more emphasis on programming than on keyboard calculations. Do not forget though that keyboard calculations also have their uses. Working through a problem on the keyboard shows you all the intermediate answers. This is one of the advantages of RPN: it lets you follow the problem and think of better ways to approach it. Nor is it efficient to write a special program for solving quadratic equations if you only have one equation to solve. If you are very short of memory, you can do some preliminary calculations from the keyboard so that the program you are writing will take up less space in memory. One example: if a longish program starts by calculating SIN(1/X↑2) then you can save three steps by doing X↑2, 1/X, SIN from the keyboard and then running a program to do the rest of the long calculation.

This book is meant to help any reader get more from their HP-41, so it provides suggestions that will simplify keyboard operations as well as programming. Many of the programming tips are equally suitable for keyboard calculations, and this should not be forgotten. The next chapter will describe the mathematical functions of the HP-41 and will give tips for their use. Many of these tips too will be suitable for keyboard use. A point to note though is that programs should be as short and fast as possible, whereas keyboard calculations do not have to be as fast, and they need to be clear and to use few keystrokes. Thus multiplying a number by two is best accomplished in a program by the instruction ST+X (add X to itself, so doubling it). From the keyboard this is not very clear and takes a lot of keystrokes. ENTER, + or 2, * is clearer.

2, * is better than ENTER, + if you want to use the keyboard for this simple operation, because it involves the least movement of your fingers. Similarly if you are doing a lot of rapid calculations and entering a lot of numbers then ENTER, * demands less extra movement of your fingers than SHIFT, X↑2. Another tip is that -, CHS is easier than X<>Y, -. In all these three cases you get the same result in X using either alternative, but the first one is more efficient because it demands less motion of your fingers.

Exercises

4.A Try to find a copy of John Ball's book "Algorithms for RPN Calculators", published by Wiley. You may find it in a local university library. He gives a list of stack rearrangements for use on calculators without a R↑ key. (See Section 4.2.3) Try to rewrite some of these rearrangements for the HP-41, using the R↑ key and the other extra functions such as RCL Y, STO Z, X<>T. You will see how much more powerful the extended HP-41 RPN functions are, and why the HP-41 language deserves a special name since it goes further than the RPN on earlier calculators.

4.B Given the three values:

A=5.37 ; B=214E-2 ; C=1.18

find a way to put all this information (names and numbers) in the display simultaneously by making a suitable choice of display characters.

4.C Put **PI** times 10↑10 into register X. Set FIX 7 mode, then FIX 8 mode. Observe that the last displayed mantissa digit is rounded up to 7 in FIX 7 mode, but that the rounding is not seen in FIX 8 mode. This is because the next mantissa digit has been rounded, but it is not seen as it is covered up by the exponent. Use the ALPHA register to check the last mantissa digit in the two cases.

4.D A subroutine that works exactly like X↑2 but calculates X↑3 was shown in Section 4.5. Try to write a similar subroutine to calculate X↑4. It should save the original X in L, put X↑4 into register X and leave registers Y, Z, and T unchanged. If you found this easy, try to write a cube root subroutine that works in the same way. (I have been unable to do this without using Synthetic Programming.)

CHAPTER 5 - KNOW YOUR FUNCTIONS

5.1 Choose your weapons.

There is a sufficient choice of functions on the HP-41 that most users select a favourite few and rarely use the others. This is quite satisfactory until a user spends hours creating a program to provide a function that already exists. There was a maths professor who once wrote a whole program to extract the "invisible" digits of a number instead of using FIX 9. The best way to avoid forgetting about functions that are not used much is to reread the manual once a year, but who has the time to plough through 450 pages of HP-41CX manual every year?

This book is not a replacement HP-41 manual, so there is no point in simply repeating all the function descriptions here. Many functions are mentioned more than once in the book, some may not be mentioned at all. To help the reader choose the most suitable weapons for dealing with mathematical problems, the mathematical and statistical functions are described all together in this chapter. The descriptions are not intended to be simply a rewording of the manual, they concentrate on tips for the use of the functions. The functions used specifically for writing programs will be described in the next chapter.

5.2 General mathematical functions.

The functions will be described in groups of similar functions. Some functions appear differently on the keyboard than in the display, the version used in the display has been used here.

+ - * / CHS
ST+ ST-
ST* ST/

Need no special comment, they have been described in Section 4.2. Remember that CHS, ST+, ST-, ST* and ST/ do not save the previous value of X in L.

ABS SIGN
FRC INT

Functions in this group alter X in a manner that seems obvious but can be difficult to describe mathematically. They are therefore more useful in programming operations than in keyboard calculations. SIGN returns zero if X contains a text value, and 1 if X contains a number that is positive or zero. This provides a useful test for ALPHA values, but is not a true mathematical signum function. The latter can be provided by STO L, X≠0?, SIGN. This replaces X with +1, 0 or -1 if X is a number, and stops with an error if X is a text value. SIGN is very useful for storing the contents of X in register L; unlike + it works for text values, and it is shorter than STO L.

RND

Do not make the mistake of thinking that RND is a "random number" function. The HP-41 does not have a built-in random number generator, although these are available on plug-in modules, and some are described in Chapter 7. RND is a rounding function, it replaces the exact value in the X register with the displayed value. This is very useful when a number is obtained as a result of a set of calculations and may be inaccurate. If such a number is to be compared to an exact value, it should be rounded before the test X=Y? is performed. The longer the calculation, the less accurate the result, so choose a suitable number of digits for rounding, something like SCI 4. FIX 4 would only round the fractional part of a number, so a number greater than 10↑5 would not be rounded at all in FIX 4 mode, but SCI 4 will round to five significant digits in all cases. (If rounding would make the number greater than the maximum that the HP-41 can store then it rounds to the largest value it can store.) If the number in the display contains an exponent then the last digits of the mantissa may be hidden (see Section 4.4.1) so the result of RND might not be exactly the same as the result in the display.

MOD

This is also called the remainder function. (It can be considered to be an extension of FRC which finds the remainder of X divided by 1.) It is sometimes difficult to remember exactly what the result of this function is when X or Y are negative. Best remember the formula

MOD = Y - MINT(Y/X)*X

MINT(Y/X) is the maximum integer not larger than Y/X. For example Y=-4 , X=3 gives

-4 - MINT(-4/3)*3 = 2

because the largest integer not greater than -4/3 is -2. The special cases are: X=0 returns Y as the result, and Y=0 returns 0. A typical use of MOD is to bring a value in degrees into the range 0 to 360 degrees. Put the value (say -790) into Y, put 360 into X, and execute MOD to see that -790 degrees is equivalent to 290 degrees. This is simpler than using a collection of X>0? and X<=Y? tests.

An additional amusement here. How can you compute the sum of the digits of any integer N ? (For example the sum of the digits of 1985 is 5 because 1+9+8+5=23 and then 2+3=5.) Simply use N, ENTER, 9, MOD to get this number.

X↑2 SQRT Y↑X

The limitations of these function should be remembered. SQRT requires that X be positive. Y↑X requires that X must be integral for negative Y, and X must be positive for zero Y. The sequence SQRT, X↑2 rarely returns exactly the original value and should be followed by RND, as described above, before any test is made. Y↑(1/X) - called involution - is calculated by doing Y, ENTER, X, 1/X, Y↑X.

% %CH Remember that these functions use X and Y, but leave Y unchanged. The sequence 1, % is shorter and faster than 100, / but it leaves Y unchanged whereas 100, / drops the stack. Similar sequences can be used in cases such as .03, * which can be replaced by 3, % saving two keystrokes and some program space. It is useful to remember that %CH provides the value

((X-Y)/Y) * 100

Volume 2 of the HP-41CX manual gives a good description of this function together with some extra tips.

LOG 10↑X
LN E↑X
LN1+X E↑X-1 The two functions E↑X-1 and LN1+X should really have been called (E↑X)-1 and LN(1+X). They provide improved precision for financial calculations and for some special mathematical functions such as hyperbolics. In financial calculations, compounding can be done more accurately, and in hyperbolic function calculations, TANH and ATANH can be obtained more accurately. See Chapter 7 for a hyperbolic function program and Chapter 12 for a description of the PPC ROM which uses these functions in an accurate financial program.

DEC OCT The HP-41 allows for arithmetic on hours-minutes-seconds numbers (see below) but otherwise it works in decimal. This stick-in-the-mud attitude is somewhat alleviated by DEC and OCT. DEC allows for the conversion of octal numbers to decimal so that arithmetic can be done on them; OCT allows for the conversion of the results back into octal so that they can be checked. Both DEC and OCT work only on integers, so numbers with fractional parts have to be scaled by the appropriate power of 10 or 8. DEC and OCT can of course also be used just for conversions, say of octal numbers in a computer dump. This is fine if you are using a computer that displays results in octal, such as a CDC. It is less help if your computer displays results in hexadecimal, such as IBM or

Hewlett-Packard (!) computers. (Even the HP-41's internal microprocessor has a hexadecimal mode.) DEC and OCT are also useful for shuffling digits in a number for random number generation or encoding messages. See Chapter 12 for a description of the HP-IL Development ROM which provides a far more complete set of functions for working with binary, octal and hexadecimal numbers. The ultimate tool for such work is the HP-16C computer science calculator.

FACT The factorial function (X!) is useful for work with combinations and permutations, but only works for integer values of X up to 69. Setting flag 24 (to allow arithmetic overflow) is dangerous when used with FACT since it can lead to answers that are incorrect even though this is not immediately obvious. For example 70!/69! will come to 58.4 if you have flag 24 set. This is incorrect, but is close enough to 70 that the error may not be noticed till later if at all.

Judging by the number of programs in the User Libraries, the extension of the Factorial function to non-integer values and values over 69 is important to many users. Any HP-41 successor should certainly have these two features, perhaps in the form of a log Gamma function. In the meantime, rearrangement of functions, logarithms, and use of Stirling's approximation and approximate Gamma functions (one is given in the High-Level Maths Solutions Book) will have to do.

ISG DSE Although these are really loop control functions, they can also be used to repeatedly add or subtract a constant. See Section 6.8 for details, and also for a note on the Card Reader functions 7ISZ, 7DSZ, 7ISZI, and 7DSZI which can be used instead.

5.3 Times and angles.

HR HMS Very useful for converting to decimal hours (or decimal degrees), and back to hours, minutes and seconds (or degrees, minutes and seconds). Remember that fractions of a second are decimal numbers, not parts of 60. See the warning (about display formats) after HMS+, HMS- below.

HMS+ HMS- Save the need for converting minutes and seconds to fractions before doing arithmetic with them. (Very useful with the Time module.) HMS- seems an unnecessary luxury since it is equivalent to CHS, HMS+. The first HP programmable pocket calculator, the HP-65, had the same function, called DMS-. The successors to the HP-65, the HP-67 and the HP-97, did not have such a function, and many users complained. This may be why the HP-41 has HMS-, even though it is not strictly necessary.

Be warned: the display will round up values such as 1 hour, 59 minutes, 59.6 seconds to 1 hour, 59 minutes, 60 seconds, or even to 1 hour, 60 minutes. This is because the rounding function knows nothing about rounding up Babylonian (base 60) numbers; only the HMS functions know about that. It would probably have been better to leave out the HMS- function and provide a time rounding function on the HP-41. The simplest way to avoid potentially embarassing time displays such as 1 hour, 60 minutes is to execute RND (round the value in X to the displayed value), then HR (convert the value in X to decimal hours), then HMS (convert the value in X back to HMS format, but HMS will calculate and display times correctly).

DEG RAD GRAD Use these to make sure the correct mode is set before calculations with angles. Remember that many operations such as integration require the natural mode for angles, namely radians not degrees. See Sections 11.10 and 15.2 for discussion of a single key assignment that will let you toggle among all three angular modes.

R-D D-R Convert radians to degrees or degrees to radians. Faster than using pi and 180, they do not lift the stack. See Section 7.4 for a way to use R-D as part of a random number generator.

SIN ASIN COS ACOS TAN ATAN SIN, ASIN becomes somewhat inaccurate for angles very close to 90 degrees, and COS, ACOS has the same problem near zero. ASIN and ATAN provide answers in the range -90 to +90 degrees, but ACOS provides answers in the range 0° to 180°. SIN has a bug on early HP-41s when using very small angles, see Appendix C.

Many computers provide an ATAN2 function to provide an answer in any of the four quadrants (from -180 to +180 degrees). On the HP-41 R-P can be used to provide the same function; see below. If it is necessary to calculate both the sine and the cosine of an angle, P-R can be used to do this quickly; see below again.

P-R R-P These are extremely useful functions, not only for converting between polar and radial coordinates in two dimensions but also for several other operations.

i. To obtain the sine of an angle (in Y) and the cosine of the angle (in X) at the same time. Put the angle in the X register, then do: ENTER, 1, P-R. This is much faster in a program than SIN, LAST X, COS. It does not save the angle in L, but you can save the angle in Z by doing ENTER two times.

ii. To obtain SQRT(a↑2 + b↑2 + c↑2 + ...). Type in a, press ENTER, type in b, press R-P, type in c, press R-P and continue for all the terms in the square root. At the end, the result is ready in X.

iii. To obtain the angle between -180° and 180° whose tangent is given by X/Y. This is a function provided on many computers under the name ATAN2(X,Y). Type in X, ENTER, type in Y, press R-P, X<>Y. The angle is now in register X. A program to simulate ATAN2 exactly is given in the next chapter. See Exercises 5.B and 5.C for additional uses of R-P and P-R.

5.4 Summations and Statistics.

These differ from the other mathematical functions in several respects and should therefore be used with care. The most important difference is that these functions use, and alter, a block of six numbered data registers without making it obvious which registers are affected. All other mathematical functions leave the numbered data registers alone, except for the storage arithmetic functions which have a parameter to show which register they use. Overflow errors caused by Σ+ and Σ- are ignored, even when flag 24 is clear, but they may subsequently cause errors in the calculation of MEAN or SDEV. Σ+ and Σ- can give error messages even if X and Y contain valid data, since the statistics registers themselves may contain unexpected values such as ALPHA DATA. It is always nececcary to execute CLΣ before starting a new set of statistical operations. Once again, this can affect unexpected registers in memory, so CLΣ should be preceded by ΣREG nn. ΣREG allows you to define the block of six consecutive registers that will be used for statistics. You have to give the register number, nn, of the first register in this block, and you have to ensure that the SIZE is set so that all six registers exist (the default setting of nn after MEMORY LOST is 11). The six registers contain Σx, Σx↑2, Σy, Σy↑2, Σxy and n, where n is the number of data values accumulated. (This information is given in the Owner's Manuals and in the

HP-41C Quick Reference Guide, but it is left out of the HP-41CX Quick Reference Guide. Best put it into one of the large and wasteful blank spaces of the HP-41CX Guide. Page 28 would be a good place.) The statistics functions suffer from two (fairly trivial) bugs; see Appendix C.

Σ+ Σ- CL Σ+ and Σ- leave Y unchanged, save X in L, put n into X and disable stack lift. This means that LAST X, Σ- immediately after a Σ+ will cancel the effect of the Σ+ if you decide you made a mistake. It is for this reason that stack lift is disabled after Σ+ (and Σ- in case you change your mind again).

MEAN SDEV MEAN and SDEV replace the contents of registers X and Y without lifting the stack. X is saved in L but Y is irretrievably lost. Σ+ and Σ- allow numeric overflow and simply store 9.999999999E99, but MEAN and SDEV may give overflow errors as a result. To correct this, you will have to alter the register whose contents are too large, or store a large number in n, then rescale the result. MEAN and SDEV calculate the mean and standard deviation of both X and Y; since an overflow on Σy will prevent you from getting any results for ΣX it is a good idea to clear the Y register before doing any statistical accumulations that use X only.
SDEV gives the sample standard deviation; to obtain the population standard deviation execute MEAN,Σ+, SDEV. This operation will not change the mean, but it will turn the result in terms of the original value of n into a population standard deviation. See Exercise 5.D for details.

ΣREG ΣREG?(41CX) Once you have executed ΣREG nn, you cannot easily find out what nn is. The HP-41CX provides a ΣREG? function for this purpose. On the HP-67 and 97 there was a RCL function which overcame this problem at least partly, by returning Σx to X and Σy to Y. The Card Reader provides the same function, 7RCLΣ, but if you do not have a Card Reader you will have to

write the function yourself; see Exercise 5.E.

On the HP-67 and HP-97, the statistics registers always started at register 14. The Card Reader therefore translates HP-67/97 programs so that every statistics function except RCL Σ is preceded by ΣREG 14. This can waste a lot of space. Any HP-67/97 program that contains a statistical function immediately after a test (such as X=0? or 7ISZ) will also go wrong because it will skip the ΣREG 14 line, not the statistical function itself. For these two reasons it is best to delete all the ΣREG 14 lines in a translated program and just put ΣREG 14 at the start of the program. 7RCL Σ assumes that the statistics registers begin at register 14 and recalls register 14 to X and register 16 to Y, overwriting the previous contents of X and Y. It does not save X in L as it should, this too can cause errors on translated HP-67/97 programs.

5.5 Indirections.

This chapter so far has contained lots of directions for using functions. Now let us have some indirections! (What other name would you suggest for "indirect directions" ?)

A simple example first; you want a program to execute all the TONEs from 0 to 9. One way to do this is to store the 10 instructions TONE 0 to TONE 9 in the program. A different way is to put the numbers 0 to 9 into the X register one after another and to execute TONE IND X each time. The TONE to be executed each time is given by the number in X; it is provided "indirectly", not attached directly to the TONE instruction itself. You need not put the numbers in X; you could put them in any other stack register or data register numbered below 100. For example you can put various numbers into register 12 and execute TONE IND 12 each time. Instead of giving a TONE number, you give an address where the TONE number is to be found - an **Indirect** parameter.

Here is another way of looking at it: In a moment of indiscretion I loaned my HP-41 to Dave, now I want to go to Dave and get it back. But I do not know where Dave lives. Instead I have to go to Ian first and ask for Dave's address, then go to Dave. This can be called **Indirect Addressing.** Instead of going directly to Dave's address, I have to go indirectly via Ian.

The HP-41 allows similar indirect addressing. You can directly recall something from register 10 by using RCL 10. You can also put 10 in register 00 (or any other register up to register 99), then use RCL IND 00. Instead of using address 10 directly, the HP-41 looks for an indirect address in register 00, finds 10, and then recalls the contents of register 10. This means that you can write a program even if you do not know which registers that program will use. The program can use indirect instructions to work out for itself which registers to use.

To execute RCL IND 00 as above, you press RCL, then the SHIFT key, then 00. The SHIFT key has an extra use, working as an "indirection" key when it is pressed after a parameter function. Any numbered register from 00 to 99 can be used after IND. The stack registers X, Y, Z, T, and L can also be used. (Press the point key after SHIFT and then press one of the keys marked with the letters X, Y, Z, T, L. This works in just the same way as using the stack registers for direct operations such as RCL Z.)

Let us take a more realistic example than RCL 10 used above. Say you need to store a value in one of the numbered registers. Normally you would do something like STO 20, but in some cases you may not know whether 20 is free or if something important is already stored there. (For example if you have a list of telephone numbers, you may want to add another telephone number to the end of the list. You do not know how many telephone numbers there are when you write the program; it must be written to let you find a free register and put the next number in it, and also increase a counter that tells you how many telephone numbers there are in the list.) So you decide to use register 25 to tell you how many registers are already in use. If registers 00 to 49 are all being used, then register 25 will

contain the number 49. This will tell you that register 49 is in use and that register 50 is the first unused register. Your program (any program) could then add one to the number in register 25, then do STO IND 25. This will store the contents of register X in register 50, and you will also have changed register 25 so that you know that registers 00 to 50 are in use.

STO, RCL, and X<> can only directly address registers 00 to 99 (unless you use Synthetic Programming) so all registers over 99 have to be addressed indirectly. For this reason registers 00 to 99 are sometimes called **primary storage** registers and registers 100 and above are called **extended storage** registers. This naming is not really necessary, and is best avoided to prevent confusion of extended storage registers with Extended Memory.

Most functions that take a direct parameter can also take an indirect one. This includes storage arithmetic and flag operations. If you want a program to set the display to show a chosen number of digits then the program can ask the user for the number, store it in register nn, and later use FIX IND nn to set the display when required. Indirect addressing is particularly useful with the program control functions GTO and XEQ, this will be discussed in Chapter 6. Of the non-programmable functions only CAT takes indirect parameters. This is not really much use since it is easier to press CAT n than to press n, then store it, then press CAT IND nn. Indirect catalogues have a couple of rather trivial bugs (see Appendix C) and the HP-41CX manuals deal with this by simply not mentioning the indirect use of CAT.

During keyboard calculations the user normally remembers things such as display settings and register numbers, so there is rarely much need for indirect operations. The indirect functions are of most use in programs. A particularly effective use is that of STO IND or RCL IND together with ISG or DSE; this will be discussed in Section 6.8. Although the indirect instructions can use stack registers to hold an indirect address, the indirect address itself must be a number. Thus STO IND Z is allowed, but

the indirect address in Z must be a number, not a letter such as Y or L. (The only indirect operations that allow stack registers as addresses are the HP-41CX indirect test functions described in Section 10.6.)

Exercises

5.A CHS can be used both as a mathematical function and as part of a numeric entry. How about EEX? What happens when you enter a number without a mantissa by just pressing EEX? Does the same thing happen in PRGM mode? Is this sensible? See Section 14.1.

5.B Just as R-P can be used to speed up the calculation of SQRT(a↑2 + b↑2) so the SIN, COS, TAN, ASIN, ACOS, and ATAN functions can be used to calculate results like SQRT(1-x↑2) in fewer keystrokes. Try to write out combinations that will let you work out SQRT(1-x↑2) and SQRT(1+x↑2). Use only registers X, Y, Z, T, and L.

5.C The functions R-P and P-R can be used for rotations in three dimensions as well as for two dimensions. Work out the steps required to go from a Cartesian X, Y, Z coordinate to a Spherical Polar R, Theta, Phi coordinate and back again. Use only X, Y, Z, T, L.

5.D Prove that the operations MEAN, Σ+, SDEV provide a population standard deviation instead of the sample standard deviation given by SDEV. The formula for a sample standard deviation is:

$$\text{SQRT}\{ [n\Sigma(x\uparrow 2) - (\Sigma x)\uparrow 2]/[n(n-1)] \}$$

and the formula for a population standard deviation in terms of the same values is:

$$\text{SQRT}\{ [n\Sigma(x\uparrow 2) - (\Sigma x)\uparrow 2]/(n\uparrow 2) \}$$

5.E Write a subroutine to recall Σx to register X and Σy to register Y without altering the statistical values themselves. See if you can write the routine so that the previous value in register X is saved in one of the registers Z, T, or L. Hint: remember that Σ+ and Σ- recall n to register X.

5.F If you want to clear every second register from the keyboard so as to leave only the even registers unchanged, you will have to press CLX, STO 01, STO 03, STO 05, STO 07 and so on. After a while it is very easy to slip up and press an even number by mistake. If you need to do something of this sort, see if you can do the same thing more safely by putting 1.00002 in register Y, clearing X, and then alternatively pressing STO IND Y and ISG Y until all the registers have been cleared and you get NONEXISTENT. If you know about Synthetic Programming, or if you use the GASN program in Chapter 11, you will even be able to assign STO IND Y and ISG Y to keys so that each of these functions can be performed with a single keystroke. The safest thing of course is to write a program to do the whole job, and this leads us to the next chapter.

CHAPTER 6 - SOMETHING ABOUT PROGRAMMING

6.1 A Simple Program.

To calculate the area of a circle whose radius is in X, you need to press X↑2, PI, *. For every new circle you need to press these three keys again. It leads to fewer errors, and is much easier, if you store this sequence as a program and execute it by pressing a single key. In effect, this key represents a new HP-41 function which calculates the area of a circle.

When you put the HP-41 into PRGM mode, then press a set of keys, the HP-41 memorises the functions instead of executing them. The memorised sequence of functions makes up a single program. If you want to try writing a circle program, follow the next paragraph, otherwise skip over it.

Press GTO.. first. This packs program memory, removing unused space from other programs and makes an area in memory ready for the new program. Next press PRGM to enter program mode and press X↑2, PI, * to write the program. Each step is stored and displayed with its step number. Press SST twice to go past the END of the program and back to its first step (the step X↑2 and its line number appear in the display). Press PRGM again to go back to RUN mode. Now you can run the program repeatedly by putting different radii in X and just pressing R/S for each value. Try it and see. Put 5.642 in X and press R/S - the answer should be just over 100.

If all programs were as simple, there would be no need for the rest of this chapter. However most programs need to do more than go straight through working out an equation. Special functions are used to let programs make tests, ask for information, and display results. Further instructions are used to alter programs, and to identify different programs. These and other functions will be covered in the rest of Chapter 6, and some example programs will be given in Chapter 7.

6.2 Using Labels To Identify Programs And Routines.

The example program given above works, but what happens when you write another program, or ten more programs? Each program must be separately identified, and in big programs each section may need to be identified. On many calculators (and in some versions of BASIC) the only way to identify a piece of a program is by the line number. You can do this on the HP-41 too, for example go to line 03 of the circle program by pressing GTO then . (the . means go to a line number), then 003. Press PRGM and you will see that the HP-41 displays line 3 of the circle program. The HP-41 with full memory can however store up to 2238 lines of program, and looking for a particular one of these could take a long time. To make life easier you can split the program memory up into separate programs, by using GTO.. as in Section 6.1 and each program can be identified by one or more labels. Try putting a label at the front of the circle program. Press GTO.000, this gets you to line 000 which is the front of the program. As there is no step number 000, the HP-41 displays the number of free registers still available for writing programs. Now press LBL, ALPHA, A, R, E, A, ALPHA. This produces a new line 1: 01 LBL"AREA" and the other lines have all moved down by one. The circle program can in future be identified by this label.

Now that the program has an identification, you can write a new program, say to calculate the volume of a sphere. First press GTO.. to separate the old program from the new one. You could go to the last line of the program and press XEQ "END" instead (remember to press ALPHA before and after pressing the alphabetic keys to spell the function or program name). Both methods put an END at the end of the previous program, but GTO.. also packs memory. Make sure you are in PRGM mode, then write the new program as follows:

LBL "VOLUME", X↑2, LASTX, *, 4, *, 3, /, PI, *, PRGM

To run the program, you can press SST twice to get to the front, then press R/S, or you can press RTN (on the SHIFT EEX key) to get to the front, then press R/S. You can also press XEQ "VOLUME". Since R/S runs a program

from the step at which the HP-41 is positioned, you can no longer run the circle program by pressing R/S. The step pointer is now positioned inside the volume program. A program, in this meaning, is the set of instructions from one END to the next END, including the second END, but not the first. The steps in a program are numbered consecutively starting at 01, and the step pointer points to the **current line**. The current line is the line that would show up if you were in PRGM mode and the **current program** is the one which contains this line.

You can run the circle program again by pressing XEQ "AREA". You can also assign the program to a key by using ASN. Press ASN, "AREA" and then the particular key that you wish to use, or shift and the shifted key. For more details of ASN, see Section 4.3.2. You can now run the area program whenever you wish just by pressing the selected key on the USER keyboard.

The labels AREA and VOLUME are called **global labels.** This means that they can be recognised at any time and from any place in HP-41 memory. There are also local labels which are only recognised inside their own program. Local labels are numeric labels LBL 00 to LBL 99 and the special alphabetic labels LBL A to LBL J and LBL a to LBL e. A typical example of using local labels would be in a program that solves quadratic equations. One label identifies a section that solves for real roots, with another label for the section that solves for imaginary roots. If these two sections are identified by LBL 00 (created by pressing LBL, 0, 0) and LBL 01 respectively, then these two labels would be found by a GTO only if the HP-41 was positioned at that particular program. Other programs could also have their own LBL 00 and LBL 01 without causing confusion.

The same local label can be repeated several times in a program. Each time the HP-41 needs to find a local label it starts searching from the current line in the program. It searches down to the END, then restarts at the top of the program until it comes back to where it started or finds the label. The first time it finds a label, it finishes the search, so another label with the same number could be put further down in the same program. This label would not be found by a GTO which has already found another

label with the same number. However another GTO further down in the program could use the second label as its target.

The local labels identified by single letters A to J and a to e behave like numeric labels, but have an additional feature. You can create these labels by pressing LBL, ALPHA, letter, ALPHA. If the letter is in the range A to J or a to e (obtained by pressing SHIFT followed by one of the top row keys), then you get a local label. If you press any other letter, or more than one letter, then you get a global label. Global labels appear in the display with a text marker ($^{\intercal}$) in front of them while local labels do not have this marker.

The special feature of the local alphabetic labels is that they are automatically assigned to the top two rows, and the shifted top row, of the keys. Thus in USER mode, you can press the [E] key (function LN), and the HP-41 will look through the current program for a local LBL E and will run the program after that label. If LBL E is not found, the function LN will be performed. If a different program or instruction has been assigned to that key, then that assignment will be used, and LBL E will not be searched for. More than one LBL with the same letter can be put in a program, the search always starts at the current line, and stops at the first corresponding label. It is important to realise that a LBL E in another program will not be found; only the current program is searched. It is also important to realise that global labels A to J and a to e cannot be created by normal means. Special methods to synthesize them will be described in Chapter 14. It is possible to create global numeric labels such as LBL "12" by pressing LBL, ALPHA, SHIFT, 1, SHIFT, 2, ALPHA. You must be careful not to confuse a label like this with a local LBL 12.

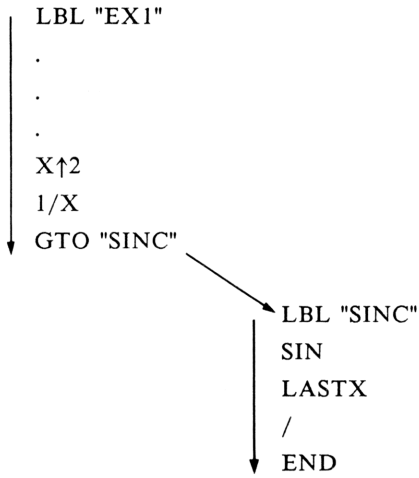
6.3. Searching For Labels With GTO and XEQ.

A label is no use unless it can be found. One way of finding labels is to assign them to keys, or use the default assignment of local alphabetic labels to the top two rows of keys. This will only work if your **target** label (the label you are aiming to find) is assigned to a key or is one of

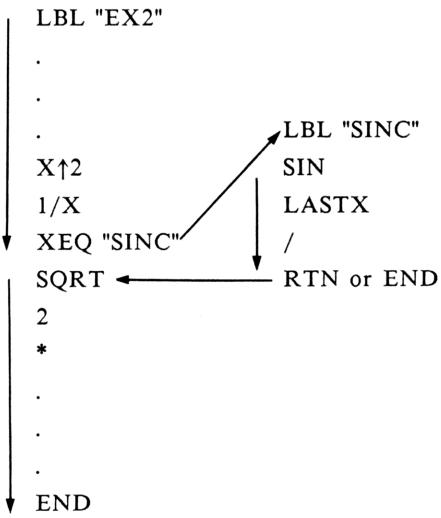
the local labels (A to J and a to e). A more general method is to use GTO followed by either a two-digit label number, or by an alphabetic local or global label. In PRGM mode, this will be recorded as an instruction whether or not the label can be found. In keyboard execution, the label becomes the current program line, or if the label cannot be found, the HP-41 displays NONEXISTENT. The GTO. instruction (created by pressing GTO and the decimal point), cannot be programmed and therefore can be used in PRGM mode. As described in the previous section, it can be used to go to a specific line in the current program. GTO. can also be used in PRGM mode to go to a global alphabetic label. If you are looking at the program VOLUME in PRGM mode, you can press GTO, ., ALPHA, A, R, E, A, ALPHA to go to LBL "AREA" without leaving PRGM mode. GTO. can also be used in run mode to go to line numbers, which can be useful, and to global labels, which is not particularly useful as GTO will have the same effect.

GTO will simply go to a label. If the GTO is a step in a running program then the program will carry on running from the label, but if GTO is executed from the keyboard then it will just cause the label to become the current program line. If you want to go to a label and run a program starting at that label then you should use XEQ instead of GTO.

The XEQ instruction lets you perform or "execute" a function or a subroutine starting at a local label, or run a program starting at a global label. All these concepts, together with the idea of "Alpha execution", were introduced in Section 3.5. (You may want to reread Section 3.5 to remind yourself of the basic ideas that will be used here.) When performed as a keyboard operation, XEQ causes the immediate execution of a function or routine. When performed as a step in a running program, XEQ executes the named function or routine as if it was a single line in the program; then the running program goes on to its next line. Let us take three examples to make this difference clear. The first one shows what happens when you use GTO to go to a routine in the HP-41. Imagine you go to the beginning of the program "EX1" and press R/S. The arrows show the order in which the steps will be carried out.

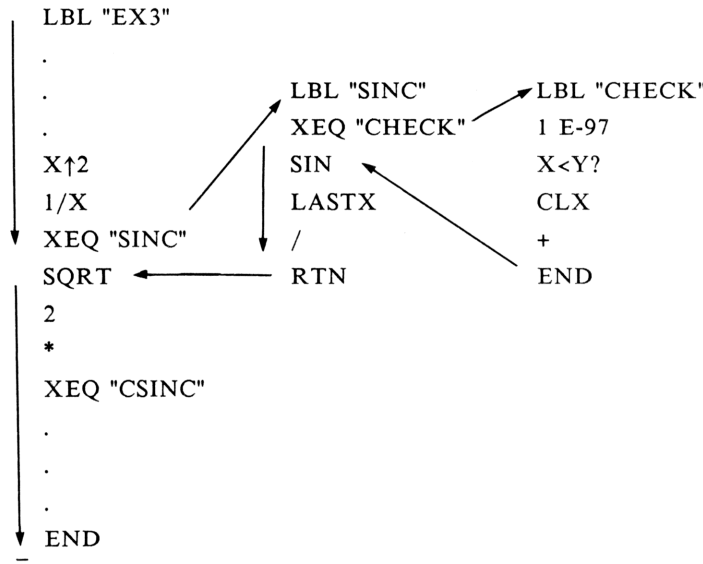


You can see how the HP-41 goes from the routine EX1 to the label "SINC", carries on from that label, and finishes at the END. (It would also have finished at a RTN. Indeed RTN and END are supposed to do the same job here except that END also marks the end of a complete program in HP-41 memory as was described in Section 3.5.) Now see what happens in the second example, which contains XEQ "SINC" instead of GTO "SINC".



This time the routine SINC is executed as if it were a single step in the routine on the left. When SINC is finished, the HP-41 goes back to the routine from which it was executed, and this carries on from the next line after the XEQ.

The routine SINC itself could execute another subroutine which would return to SINC, since RTN or END send you back to the next line after the latest XEQ. There will be no confusion between the return to SINC and the return from SINC to EX3, because the HP-41 saves each return address separately. The return address is stored until there has been a corresponding return to that address. Then it becomes the current address again and is removed from the list of return addresses.



(The routine CHECK checks if the value in X is very small, and if so then it replaces that value with a slightly larger one to avoid a possible division by zero in SINC.) The routine called from SINC could contain yet another XEQ, in fact up to six XEQ instructions can be saved, and the HP-41 will always return to the step after the latest XEQ. Until the return is actually carried out, the HP-41 is waiting for it, and the return is said

to be **pending** (because it is suspended, or "hanging on"). The HP-41 stores pending return addresses in a **subroutine return stack**, rather like the RPN stack of X, Y, Z, T. This stack has room for six return addresses, so if more than six XEQs are carried out without a corresponding return, then the earliest return address is lost and the HP-41 stops instead of returning at the corresponding RTN or END. This is rather like losing the top value off the X, Y, Z, T stack if more than four values are ENTERed. After the SINC routine has returned to the main program, there are no pending returns, so later on XEQ "CSINC" counts as the first return in the stack, not as the fourth.

The behaviour of XEQ requires some more explanations. When you press the XEQ key you see XEQ followed by two prompt signs. You can fill these in with two numbers representing a numeric local label, or you can press ALPHA followed by a letter representing a local Alpha label and ALPHA again. Instead of a local Alpha label, you can use one to seven characters which represent the name of a routine or a function. The effect of this differs in PRGM mode and in run mode, let us consider PRGM mode first. The HP-41 will search for the name through all of the programs and functions in memory. The search goes through CATalogues 1,2 and 3 in that order, as was described in Section 3.5. If the name is found in CAT 1 it is recorded as XEQ "name". If the name is found in CAT 2 then it is recorded as XROM "name" if it is the name of a routine, or as the name itself (without quote marks) if it is a function. If the name has still not been found then CAT 3 is searched, and the function name is recorded if found. (This is why searching for a CAT 3 function can be rather slow on an HP-41CX; it is not found until CAT 1 and CAT 2 have been searched, and CAT 2 contains a lot of functions on an HP-41CX.) Finally, if the name has still not been found, it is recorded as XEQ "name", on the assumption that it is the name of a routine which you have not yet written, or that it refers to a name in a module that is not plugged in. Local label XEQs are immediately recorded without a search.

If you use XEQ in run mode, then the search is performed in the same order, but the function or routine is executed at once if it is found. (If the

name is found in CAT 2 then the wrong instruction might be carried out if two modules have the same number; see Section 12.5.) Local labels are searched for in the current program only and executed at once if they are found. The NONEXISTENT message is displayed if the name (or local label number) is not found.

The "non-programmable" functions such as GTO., PACK and SIZE cannot be stored in a program. These functions are executed at once, even if you XEQ them in PRGM mode. Some of these are available in programmable versions, others can be made programmable by Synthetic Programming; this will be covered in Chapter 16.

Once an XEQ instruction has been used to write a line in a program, the corresponding label or function will have to be found when the program is run. There is no problem with CAT 3 functions since these are always available on every HP-41 (unless it is faulty), and are recorded as normal program steps. A CAT 2 function or program is actually stored as a module number and the number of the program or function in the module. If the module is removed then the step will be displayed as XROM followed by these two numbers (see Section 3.5 again) and the program will stop at the XROM instruction displaying NONEXISTENT. If a different module with the same module number is inserted then the HP-41 will try to find the program or function with the specified number in that module, and will execute it if found. This can produce entirely unexpected results; see Section 12.5.

The most complicated situation arises when a running program finds an XEQ "name" instruction. The HP-41 first looks for the name in CAT 1, then in CAT 2, in the order described in Section 3.5. In this case, confusing results will be obtained if more than one module contains a program or routine with the same name (Section 12.5 covers this as well). If the name is not found in CAT 1 nor in CAT 2, it is most likely a name that you have deleted from CAT 1 or forgotten to put into CAT 1 (such as a program which you were going to read from a magnetic card, but had forgotten about). It could also refer to a module or peripheral device that you have forgotten to plug in. There is, however, a third possibility; it could be the name

of a CAT 3 substitute which you wrote and then deleted (for example an alternative version of ATAN which you were checking out). If you have deleted the substitute, you may have done so by accident, or you may have done it on purpose so as to use the ordinary CAT 3 function. In the first case the HP-41 should stop with NONEXISTENT, but in the second case you would want the HP-41 to execute the CAT 3 function. The HP-41 designers could not know in advance which of these cases to expect, so they chose the safest action; CAT 3 is not searched, so for example XEQ "ATAN" in a running program will not execute the ATAN function. This is particularly important if the name of your routine was the same as the name of a CAT 3 function but your routine did something completely different. If you wrote a random number generator called RND and later deleted it then you would not want the HP-41 to execute the CAT 3 function RND (round) instead. This behaviour of the HP-41 is consistent with HP policy which is to protect users from their own possible stupidity, but it should have been explained clearly in the HP-41 manuals.

Going into USER mode and pressing a key to which a function or routine has been assigned works just like Alpha execution. In run mode the routine or function is executed at once. In PRGM mode the routine or function is recorded as a program step, as a function name or as XROM or as XEQ. This too shows how XEQ is designed to be as similar as possible to executing an HP-41 internal function.

To summarise: the purpose of XEQ is to let users treat their own subroutines in exactly the same way as HP-41 functions. XEQ "MOD" means the HP-41 should execute the HP-41's MOD function, then go on to the next step. XEQ 01 means the HP-41 should execute the user's subroutine at LBL 01 till it comes to a RTN or END then go back to the step after XEQ 01. The subroutine at LBL 01 can call other subroutines to a maximum depth of 6. XEQ "FACT" means the HP-41 should execute or store the CAT 3 function FACT. XEQ "FRED" means the HP-41 should execute (either immediately in run mode or when the program being written is run) the user's own subroutine called FRED, exactly as if it were one of the HP-41 functions.

6.4. Finding Your Place, Compiled Addresses, CAT 1 and Indirect Execution.

Both GTO and XEQ have to find their target labels by looking through program memory. In a long program, this search may take some time and may be repeated often. Local GTO and XEQ instructions therefore store the distance to the target label when they find it the first time. Every time the GTO or XEQ is repeated it can then jump directly to the label, without looking for it, and this saves a lot of time.

This process of storing the distance to a label is called **compilation**, and the GTO or XEQ is said to have been **compiled**. When a program is altered so that instructions are moved or deleted, the jump distance may change, so the entire program is **decompiled** at the end of the editing. This means that each jump distance is changed to zero, which tells the HP-41 that the distance is unknown. As the program executes again, each GTO or XEQ is compiled again the first time it is encountered.

It would be unreasonable to decompile every program in memory just because one program has been altered. Decompilation therefore affects only local labels within the program that has been edited. (The editing of this program will not affect jump distances within other programs anyway.) A different scheme is used to speed up a search for global labels. The HP-41 always keeps a record of the position of the last END in memory. To show this is special, it is displayed as .END. (followed by xxx which is the number of free registers left for writing programs). The distance to the global label that comes before it is stored inside the .END. itself. Each global LBL or END in turn stores the distance to the preceding END or global LBL. In this way there is a chain of ENDs and global LBLs which can be followed without a need to look through every program step. The last END or global LBL in this chain is the first one in memory, since the chain runs backwards. This END or global LBL has nothing to connect to, so it stores a zero as the distance to the previous item. A running program searches up this chain, starting at the .END. whenever it comes across a GTO or XEQ that refers to a global label. If the LBL is found, execution

continues from it, otherwise CAT 2 is searched.

When you do a CAT 1, the HP-41 starts at the .END. and runs up the global chain until it finds the LBL or END with a zero distance to the previous item. It displays this as the first item in CAT 1. Then it runs up the whole chain again, displays the previous item and repeats this again and again. You may have noticed that CAT 1 actually speeds up as it progresses since it has a shorter distance to go each time until it eventually reaches the .END. and finishes. You can stop CAT 1 by pressing R/S. The END or global LBL displayed becomes the current instruction. You can then SST or BST to any other global LBL or END (do not press backarrow at this time or you will come out of CAT 1 and SST or BST will only move you through the current program). This is the only simple way of getting to a program that does not contain a global LBL. Use CAT 1 and stop at an END that does not have a global LBL before it, or stop the catalogue somewhere else and SST or BST to that END. When you press the backarrow key you exit from CAT 1, and you are at that program. Best put a global LBL in it.

To help save program memory, the local labels LBL 00 to LBL 14 and their corresponding GTOs take less space than the others. This means that the GTOs have less room for the distance to the labels, and they can only compile a distance of 112 bytes or less. Details of byte distances and GTOs will be given in Chapter 8. For now, it is best to remember that these **short-form** labels 00 to 14 and their GTOs save space but should only be used if they are less than about 40 lines apart.

Another point to note is that a subroutine can call itself. If a subroutine at LBL 50 needs to do a job that the subroutine does, it can contain XEQ 50. This is called **recursion** and is impossible even in computer languages like FORTRAN, but the HP-41 language, FOCAL, lets you do it. Of course, the maximum number of pending subroutines can still only be six, as mentioned at the end of Section 6.3.

Both GTO and XEQ can also take indirect parameters. You can calculate a number depending on a set of conditions, store it in register nn, and XEQ

or GTO IND nn. This will let you skip over various pieces of the program that are necessary only in certain cases. Take the following example.

A program to work out personal tax has to allow for people who pay no income tax, pay income tax only at standard rates, or pay income tax at a standard plus a higher rate. An individual may have worked in more than one kind of job during a year, so the tax has to be worked out in several places in the program. Do the following:

i) Work out a number that is 0 for no income tax
1 for standard tax rate
2 for high tax rate

ii) Store this number, in a register, say STO 10

iii) First work out tax for self-employed occupations, using the following piece of program

```
LBL A
 Set self-employed tax to zero
XEQ IND 10          (if 10 does not contain 0,1 or 2,
GTO B               you will get a NONEXISTENT error)

LBL 02
 Work out super-tax here and add it to self-employed tax
LBL 01
 Work out standard tax here and add to self-employed tax
LBL 00
 Work out any tax that applies regardless of income,
 such as health insurance and add it
RTN
 Go back to the line after the XEQ
```

iv) You are now at the piece of the program to work out tax for employment by a company:

```
LBL B
 Set employee tax to zero
XEQ IND 10
GTO C

LBL 02
 Work out super-tax and add it to employee tax
LBL 01
 Work out standard rate tax and add it to employee tax
LBL 00
 Work out any tax that applies regardless of income,
 such as health insurance and add it
RTN
 Go back to the line after the XEQ
```

v) Repeat the process of iii) for other cases such as periods of unemployment or work abroad.

```
LBL C
 Do the extra calculations here
GTO D
```

vi) Add up the different kinds of tax to provide the total.

This of course is a simplified version, but it shows two important points: first, XEQ IND can save repetitious calculation, and second, a local label can appear more than once. The rules of local label searching are described in Section 6.3. These same rules apply to indirect label searches. Since the indirect value can change, indirect GTOs and XEQs are not compiled.

GTO IND and XEQ IND can use the stack registers X, Y, Z, T, and L for the indirect address. The register, whether numbered or in the stack can

contain a number for a local label, or an alphabetic value for a global label. If the content is a number, its sign and fractional part are ignored. If the indirect address is alphabetic, it can contain a maximum of six letters. The single letters A to J and a to e will not be found, since they are local labels, not global labels. It is possible to create the corresponding global labels, this is described in Chapter 14. Alternatively, you can create a label containing the letter twice over, for example LBL "AA". If you have the letter "A" in the Alpha register, you cannot do:

ASTO 10, GTO IND 10

But you can do:

ASTO 10, ARCL 10, ASTO 10, GTO IND 10

This doubles the letter in the Alpha register, so that you can go to "AA".

Many plug-in modules have programs and functions whose names are seven letters long, but GTO IND allows a maximum of six. To call "PINBALL" in the HP GAMES PAC, you would like to put "PINBALL" in Alpha, ASTO X, then XEQ IND X, but this would only give NONEXISTENT. You need to use the following two lines in your program:

LBL "PINBAL", GTO "PINBALL"

The new label is six letters long; it will therefore be found by the XEQ IND, then the GTO will transfer execution to the ROM program, and that program's END will send you back to the original program that wanted to run "PINBALL". This trick saves a RTN, but does not work with functions. The printer function PRFLAGS, for example, must be executed as follows:

LBL "PRFLAG", PRFLAGS, RTN

HP-41s with bug 7 (see Appendix C) will not let you ASTO the first six letters in Alpha, they store six and a half letters. However, this only affects the comparison operations X=Y? and X≠Y?. The GTO and XEQ IND instructions will still work properly. For example, the following sequence

will execute the "PINBAL" program:

"PINBALL", ASTO L, XEQ IND L

Here is one more tip, courtesy of Joseph Horn, concerning the local labels A to J and a to e. In a long program that uses these labels it can take some time when you press a key such as A for the HP-41 to find the label. This is because the HP-41 has to search every step from the RTN or END where it has stopped, until it finds the label.

To speed the search up, put another LBL A immediately after a RTN, and follow it with GTO A. When you press A, the first LBL A will be found at once, and the GTO A will go to the real LBL A. This GTO will be compiled the first time you use it. From then on you can press A and the GTO A will execute a compiled jump to the real LBL A at once. This saves the time that would otherwise be lost while the HP-41 searched for the LBL A. You can sprinkle LBL A, GTO A all over the program to make sure the real LBL A is always found quickly, and of course you can do the same for other local Alpha labels.

The fact that a pair of instructions such as LBL A, GTO A is useful and saves time can only be realised by someone who understands how compilation and labels work. This tip shows how studying the HP-41 can have clear uses and is not just a matter of idle curiosity.

6.5 Checking, Correcting and Changing a Program.

It is difficult to write a program much more than ten lines long without making some mistakes. This section will cover the HP-41 functions that help you check a program and make changes. All these functions are explained clearly in the HP manuals, so the descriptions here will concentrate on features not mentioned in the manuals.

SST and BST

In PRGM mode, these functions simply move the program pointer forwards or backwards by one line. The program pointer is a number that tells the HP-41 what the current line number is. At the end of a program, SST takes you straight back to line 1 of the program, and BST from this line returns you to the END. SST and BST are executed immediately and cannot be cancelled by holding down the key. If you are going to use BST a lot, it is a good idea to assign it to an unshifted key, so you can save yourself pressing the SHIFT key for every BST. If you want to move a long way in a program, it is quicker to use GTO.nnn to go to a line somewhere near the one you want, then SST or BST just a few steps.

SST works quickly, because it is easy to move forwards one step in a program. BST is much slower. It goes back to the last global label before the program pointer, then counts lines forwards and stops one line earlier than before. It is therefore worth including several extra global labels in a long program while it is being written. This makes for shorter distances between global labels so BST has to go back a shorter distance and works faster. After the program is written and checked, these extra labels can be deleted. If you SST or BST to a long text string you will be delayed while the whole string scrolls across the display. Remember that you can terminate the scrolling by pressing any key.

In run mode, SST executes the current program line and moves the program pointer one step forward. If SST is held down a short time, the current line is displayed, then executed. If SST is held down longer, the step is NULLed, and the program pointer does not advance. Pressing SST quickly (so the current line is not displayed) after the END line moves the program pointer forwards to line 00, not line 1. Functions, but not programs, can be executed from the keyboard in between SST execution of steps without disturbing the SST process. A BST moves the program pointer back one step, but does not execute that step, and will not be cancelled by holding down for a longer time.

GTO. GTO.. RTN R/S

GTO followed by a point and a line number takes you to that line, whether you are in PRGM mode or not (unless the program is PRIVATE, in which case GTO. does nothing). In a very long program you can press EEX after GTO. to give a line number of over 1000. GTO. a line number greater than that of the END takes you to the END. Pressing a line number of 000 takes you to line 000 at the start of the program. In run mode, you can press GTO. and a local or a global label to get to that label. GTO. followed by a global label also works in PRGM mode.

GTO followed by two points packs program memory, puts an ordinary END after the last program in memory and puts the program pointer to line 0 of a new program at the bottom of program memory (the .END. is line 1 of this program).

RTN, in PRGM mode, is recorded as an instruction that behaves like END, but does not separate one program from the next. That is to say, RTN in a subroutine causes the subroutine to finish and return to the line after the last XEQ or XROM that had been executed. In a program that has not done an XEQ, RTN causes the program to stop. Pressing RTN in run mode puts the program pointer to line 00 of the current program and clears the subroutine return stack.

R/S, in run mode, alternatively starts and stops a program running without changing the subroutine return stack. Holding R/S down for a short time when a program is not running displays the current program step, which will be the first step to be executed when the program starts running. Holding R/S down for a longer time cancels this display, shows NULL, and prevents the program from running. This provides a simple way of checking whether the program you are about to run is the one you want.

GTO "program" followed by R/S starts the program running as does XEQ "program", but it does not clear the return stack. This means that

you should not interrupt a running program then restart it nor run another program with GTO "program", R/S. Otherwise the END or RTN that you expected the second program to stop at may simply act as a return to some address in the program that you had previously interrupted. Similar problems can arise whenever the return stack has not been cleared, for instance when a program is interrupted by PROMPT or STOP. Use RTN in run mode, as described above, to clear the return stack.

In connection with the above, note that when a program stops at an END the HP-41 clears the return stack and the next step to be executed after another R/S will be the first step of the program. When a program stops at a RTN, the return stack is not altered, and pressing R/S will restart the program from the next step after the RTN.

Inserting and Deleting Program Lines

To insert a new line in a program you just GTO or SST or BST to a line and put the new line after it. You can GTO.000 to get to the start of a program and insert a new line there. It is not possible to insert a new line after the END, although the HP-41 may sometimes look as if it were trying to do this. To see this, go to an END, see its line number, then press the STO key. You will see STO__ with a line number one greater than the END. When you put in the parameter nn, the HP-41 moves the END past the STO nn and renumbers the line. Whenever this happens, or if you put an extra line into a program, the HP-41 has to move down program instructions to make room for this new line. This moving will not happen if there is some spare room at the location of the new instruction (left over because an old one has been deleted), so it is best to delete lines before inserting new ones in the same place.

To delete a single line, you go to it and press the backarrow key. The line is replaced with invisible null spaces, and the previous line is displayed, ready to be deleted too if necessary. You cannot delete line 00 of a program, since it is only a place marker and does not

really exist, nor can you delete the .END. since its existence is required by the global chain. You can delete an ordinary END, and this allows you to merge two programs in memory.

To delete a block of lines, go to the first line of the block, and XEQ "DEL". DEL prompts for the number of lines to be deleted. If this is over 999, you can press EEX as with GTO. to obtain a number up to 1999. DEL will delete this many lines, or if fewer lines exist, it will delete all the lines from the program pointer to the END. DEL will not delete the END itself, otherwise you may accidentally delete part of the next program in memory or merge two programs. You can also execute DEL in run mode, but it will do nothing except display DEL nnn.

To delete a whole program, use CLP. XEQ "CLP", then enter the Alpha name of any global label in the program to be deleted. You can also go to the program, and just XEQ "CLP", ALPHA, ALPHA which clears the current program. CLP is automatically followed by packing; DEL is not.

Packing and Resizing

The HP-41's memory is large for a calculator, but small for a computer. You may therefore run out of memory while writing a program. The first thing to do in this case is to pack memory, by executing PACK. This will remove all the null spaces that have been left in a program during editing, deleting lines and inserting numbers. Indeed the HP-41 will automatically pack memory and display PACKING, then TRY AGAIN when you run out of program memory. Program instructions will be moved up to fill the empty spaces, so that all the empty space will be left free at the bottom of the program memory. PACK and GTO.. let you pack memory whenever you choose. If you execute PACK, the program pointer will stay at the same line of your program as you were at before PACKing. Should you wish to do a lot of program editing, you could assign PACK to a key. You should not PACK if you are in the middle of program execution because this loses any

pending subroutine returns. Packing also removes any completely unused key assignment registers (details are given in Section 8.6) and occurs automatically if you try to make a key assignment when there is not enough room for the assignment. As this clears the subroutine stack, you should not try to make key assignments while a running program has been interrupted.

If you are still short of room for your program, you should execute SIZE and make the size as small as possible. Remember that SIZE defines the number of registers available for storing data. If you execute SIZE 000 then no registers will be reserved for data storage until you execute SIZE again.

After you are finished editing your program, you can increase the SIZE again. The editing operation generally requires one register (7 bytes) for each insertion of a new instruction. The unused bytes are recovered by packing.

Copying Programs From Application Modules

Programs in Application Modules can be run as they are, but some users may want to copy them to RAM so that a module can be removed or a program can be altered. You might want to remove a module because you do not have enough ports to hold all the modules you are using, or you may want to change something that a program in an Application Module is doing. One reason for this is that some Application Modules have program errors; see Chapter 12. You cannot alter a program in ROM, you have to copy it to RAM and edit it there. Use the COPY function to make the RAM copy. XEQ "COPY", then press ALPHA and spell out the name of a a label in the program you wish to copy. The entire program will be copied to the bottom of RAM program memory if there is room for it. If there is not then the HP-41 will display PACKING and TRY AGAIN. Do try again; if the same thing happens again try SIZE 000, and COPY again, then delete the parts of the program that you do not need in RAM and you will then be able to increase SIZE again. COPY always puts an END before the program it copies, so you

may finish up with some unwanted ENDs in program memory. (Actually, the .END. is converted to an END, with a new .END. added.) Use CAT 1 to find them, then delete them.

After you have COPYed a program to RAM, the program pointer will be at its first line so that you can edit the program. GTO and XEQ will use the RAM copy because it is in CAT 1 and is found before the ROM version in CAT 2. You cannot make another copy of the ROM program though, because the RAM version will be found first, and you cannot COPY programs from RAM. The way around this is to delete a global label from the program copy in RAM, then execute COPY and specify that global label. Now the label will be found in ROM (in CAT 2) and you will be able to make a second copy of the program. If the program pointer is already in the ROM program, then you can give COPY no name (just press ALPHA twice) and the current program will be copied; this provides an alternative way of making a second copy of the program, and also works for any unnamed program in a ROM. Of course you have to get to these programs first. GTO will work for named programs, but Synthetic methods are required for copying unnamed programs from a ROM. You will not be able to copy PRIVATE programs from a ROM.

Some ROM programs XEQ other ROM programs, so if you want to make a copy and remove a ROM, be sure to copy all the ROM programs you need. You will also need to alter the RAM program, replacing XROM "label" with XEQ "label" for every label that occurs in the copied program.

COPY is one of the few non-programmable functions that have not been made programmable by Synthetic Programming (see Chapter 16) or by Extended Functions (see Chapters 10 and 11). A programmable COPY would have to be written in HP-41 machine language (see Chapter 17).

6.6 Watching Program Execution

Now for an amusing and useful trick before we carry on with serious matters. When you test a new program, you need to check what happens

step by step. There are two methods approved by HP. Firstly SST through the program in run mode (see above). Secondly run the program with a printer attached and set to TRACE mode. The printer will print every step, and each value put into X. This uses up a lot of paper. The first method takes a lot of key-pushing. Why not program the HP-41 itself to SST through a program and display every step? This means running a program in PRGM mode.

Aha, you might say, a contradiction in terms. Run mode is for running a program and PRGM mode is for editing and viewing a program. But actually, run mode is only needed to start a program (you can even get round that if you know enough about Synthetic Programming). Once a program is running, it can set PRGM mode, and then PSE. PSE shows the default display which can be X, ALPHA or the current program step in PRGM mode. If you were able to run a program in PRGM mode, the default display would be the current step, which is the step that will be executed next. This is possible because PRGM mode can be set by setting flag 52, see the flag table in Appendix D.

Flag 52 cannot be set by SF 52 (only flags 00 to 29 can be set and cleared by SF and CF), but two of the bugs (see Appendix C) let us do it. On early HP-41Cs (made in mid 1979), bug 3 lets you set flag 52 as follows:

52, STO nn (where nn = any numbered register), SF IND nn

Try this - if your HP-41C has bug 3, it will go into PRGM mode. On all later HP-41Cs this results in NONEXISTENT, but bug 10 lets you do the same. Try entering the program below, the comments next to the program listing explain what each line does. This program shows how you can do things that the manufacturers never expected of the HP-41.

```
01♦LBL "WATCH"
02 ΣREG 11
03 CLΣ
04 1 E60
05 STO 13
06 1/X
07 STO 16
08 SF 25
09 MEAN
10 AOFF
11 CF 11
12 PSE
13 DEG
14 PSE
15 CLD
16 PSE
17 CF 00
18 PSE
19 CF 03
20 PSE
21 CF 04
22 PSE
23 BEEP
24 END
```

Lines 2 to 9 cause bug 10, which sets a lot of flags including flag 52. Lines 10 and 11 clear two flags that are best cleared as soon as possible. Lines 12 to 21 alternatively PSE to display a function, then execute it so that you see the indicators turning off one by one. You can run this program to amuse yourself or your friends.

Lines 2 to 11 can be included in any program to show what it is doing. A PSE has to be put before each step that is to be displayed. Then sit back and watch the program run. When flag 52 is set, the HP-41 treats any numeric entry in a program as a programming instruction. Therefore, any number entry instruction 0 to 9 or . or EEX will be copied repeatedly into program memory. Instead of using numeric entry steps, put each number into a data register before setting flag 52, and RCL the number from that register when you need it.

Bug 10 also sets flag 44, the continuous ON flag, so remember to turn your HP-41 off when you finish with this trick.

6.7 Using Tests To Control Program Execution

Back to serious matters now. One of the most powerful programming features of the HP-41 is its collection of "do if true" tests. They all follow the same rule:

Make a test,
Perform the next step if the answer is yes,
Otherwise skip the next step.

Tests, something like this, are used quite frequently:

X>0?
SQRT

If X is positive, take its square root. The square root of zero is zero anyway, and the square root of a negative number gives the HP-41 indigestion. The HP-41 has five tests for comparing X with zero, five for comparing X with Y, and four flag tests (see Section 4.4). The HP-41CX has additional indirect tests, and some plug-in ROMs, particularly the HP-IL Development Module, have additional do-if-true tests.

GTO is the most useful instruction to follow a do-if-true test. For example, if X is negative, go to a routine to deal with negative values. A program can also test something at one place, and set or clear a flag. The flag can then be tested whenever necessary - FS? 19, GTO IND 05, which means "Go to the address specified by the register whose address is in 05, but do this only if flag 19 is set."

Avoid wasteful use of GTOs. Both the following programs do one thing if flag 19 is set and another thing if flag 19 is clear:

```
FS? 19                  FC? 19
GTO 01                  GTO 02
GTO 02                  .
LBL 01                  flag 19
.                       set
flag 19                 operations
set                     .
operations              GTO 03
.                       LBL 02
GTO 03                  .
LBL 02                  flag 19
.                       clear
flag 19                 operations
clear                   .
operations              LBL 03
.
LBL 03
next part of program
```

By inverting the flag test you have saved using the GTO 01 and the LBL 01 - clearly more efficient.

If two instructions are dependent on the same test, it is best to make the same test twice; the four lines on the left calculate SIN(X↑2) if flag 05 is clear more economically than the five lines on the right.

FC? 05	FS? 05
X↑2	GTO 10
FC? 05	X↑2
SIN	SIN
	LBL 10

It is sometimes useful to put one test directly after another as in the example below:

```
FS? 06
FS? 07
CHS
```

This will change the sign of X if flag 6 is clear, or if both flag 6 and flag 7 are set, but not otherwise. To perform an instruction if either one of two tests is true, or if both are true do:

step 1	opposite of test A (NOT A)
step 2	test B
step 3	instruction

An example of this is to simulate the missing test X>=Y? by using X≠Y? followed by X>Y?

To perform an instruction only if both test A and test B are true do:

step 1	Test A
step 2	opposite of test B (NOT B)
step 3	FS? 53 (or some other test which is never true)
step 4	instruction

These two suggestions are given in "ENTER" by Jean-Daniel Dodin (see Appendix A). His book, and "Calculator Tips and Routines" by John Dearing, are both full of excellent ideas like this.

If a test generates an error, program execution will stop at the test. If flag 25 is set, then the error will be ignored, and the next instruction after the test will be performed. For example X=0? with a text value in X will generate an error, but if flag 25 is set then the error will be ignored. As a result the next step will be performed even though X is clearly not zero. It is therefore important to design tests with some thought of possible errors. In the case of checking values that could be alphabetic or numeric, it is better to use X≠0?. With X≠0?, if an ALPHA DATA error occurs, flag 25 will be cleared and the following step will not be skipped. Thus nothing unexpected or incorrect will happen.

6.8 ISG , DSE and NOPs

Apart from the do-if-true tests, the HP-41 has the loop control functions ISG (increment and skip if greater) and DSE (decrement and skip if equal or less). A loop is a set of instructions that are to be performed several times. Your program may need to repeat a calculation seven times for the seven days in a week, or ten times for a matrix with ten rows, or an unknown number of times for a set with an unknown number of elements. A typical loop has the structure:

i) LBL pp
ii) Instructions for element number n of set
iii) Increase n
iv) Is n greater than number of elements?
v) If not, GTO pp
vi) Otherwise carry on with the next program step

n is a control number which can be stored in the stack or any directly or indirectly addressed data register. The general form of n is a 5-digit integer followed by a 5-digit fractional part:

i i i i i . f f f c c

where: - iiiii is the initial value, and later the present value of the control number n. It is assumed that only the integer part of the control number will be used for controlling loops, and that the fractional part will be used for other purposes as below.

- fff is the final value for n. This is taken to be three digits long. A fractional part of .3 will be interpreted as fff = 300. A fractional part of .003 will mean 3. When iiiii is greater than fff (for ISG) or less than or equal to fff (for DSE) the loop has completed and the step after ISG or DSE will be skipped.

- cc is the loop increment or decrement counter. Every time ISG or DSE is executed, the number iiiii is incremented or decremented by an integer cc. In most cases, the increment of a loop is 1, so if cc is 00 or not given, then it is assumed to be 1.

A few examples should help. What will happen to a program that contains the following ?

```
377.02577
DSE X
GTO A
```

The number 377.02577 is put in X. DSE X will decrement 377 by 77 leaving 300.02577 in X. 300 will then be compared with 025. Since 300 is greater than 025, the next step will <u>not</u> be skipped, and the program pointer will go to LBL A. Further operations would decrement the counter until it reached -8.02577 at which stage, the step following DSE X would be skipped.

Now look at this:

```
99976.30029
STO 01
ISG 01
BEEP
```

The ISG will change the value in 01 to 100005.3002. The number is truncated to ten digits, and the cc part changes from 29 to 20. 100001 is greater than 300, so there will be no BEEP. If register 01 is used again for ISG or DSE, the increment or decrement will be by 20, not 29. The fact that the last digit is truncated, not rounded, is surprising. (If 29 were added to 99976.30029, the result would be 100005.3003.) A cc value of 09 will be truncated to 00, turning the increment to the default value of 01. In most cases this would be a trivial error, but if the fraction is to be used as a control number, this is serious, and is mentioned in Appendix C.

A third example:

```
1
ENTER
FIX 0
-54321.
STO IND Y
LBL 55
VIEW IND Y
PSE
ISG IND Y
GTO 55
TONE 0
TONE 2
TONE 4
TONE 0
TONE 8
```

The value cc has not been given, so it defaults to 01. -54321 is incremented by 1 and becomes -54320. The program displays a count down timer. It may seem that the three-digit limitation of fff would limit useful values of iiiii to five digits, but the use of a negative value

means you can use ISG with up to ten digits. If you try to use a number less than -1 E10, then you will get into an infinite loop, since the result of ISG will always be rounded back to the number you started with. Nevertheless, cc and fff can have as few digits as are necessary, and iiiii can have correspondingly more digits. If there are any digits after cc, they are simply ignored, but remember they will be truncated, not rounded, if the integer part gets too large.

Use of a value other than 1 for the increment gives considerable flexibility to ISG and DSE, but everything is done in integers. This is acceptable if the control number is to be used as an indirect address or a flag number when only the integer part is used. If a fractional part is required, it is best to recall the counter to X, then use INT, 10, / .

A good use of ISG is to store values in a block of registers or to set/clear a block of flags. The following loop will store 0 in every jth register starting at bbb and ending at eee.

```
bbb.eeejj
ENTER
CLX
LBL 02
STO IND Y
ISG Y
GTO 02
```

The HP-41CX function CLRGX will do exactly the same job of clearing the registers specified by bbb.eeejj.

At times you will want to use ISG or DSE just as an arithmetic function, to add or subtract a number cc, or to add or subtract 1. Under these conditions you will not want to skip the next step. Take a case where you want to calculate

SIN(X) + 1

without changing Y, Z, or T. Instead of SIN, 1, + use ISG as follows to get the result to 3 decimal places:

```
SIN
FIX 3
RND
ISG X
ADV
```

The FIX 3 and RND steps make sure that cc is zero, so the value in X will be increased by 1. Unfortunately the ISG X may or may not cause the next step to be skipped. The way to avoid trouble is to put a do-nothing step after the ISG instruction. Then it will not matter whether the step is executed or skipped. ADV is a fairly good choice for this purpose as it does nothing unless there is a printer attached. A step like this is called a NOP (no-operation or null operation, instructions called NOP were provided on several HP calculators, but not on the HP-41). ADV is not a good NOP if you use a printer though, and you may want to use some other NOP. The instructions X<>X and STO X were mentioned at the end of Section 4.2.3 as possible NOPs and you may like to use them. Unfortunately, they take up twice as much memory space as ADV, and they take more time too. I prefer to use LBL 00 as a NOP, it is as fast as ADV, takes no more space, and I recognise it as a NOP since I only use Labels 01 and higher for GTO and XEQ. Many people who use Synthetic Programming prefer the "spare" byte 240 (hex. F0) which is known to be a NOP, and makes it clear that they know about Synthetic Programming! F0 is not quite as fast as LBL 00 though, and it could be confused with a line of Synthetic text in a program printout. Whatever you use as a NOP you should make a note somewhere to avoid confusion when you come back to the program later or if someone else tries to understand the program.

None of the NOPs described above are true null operations, because they all enable stack lift, but who needs a NOP after a stack disabling or neutral operation? Still, if you do ever need it, the ZENROM module provides a NOP function which can be used as a true NOP, although if used that way it takes up twice as much memory space as LBL 00 (see Section 12.8 and

Appendix B for details of ZENROM).

You can avoid trouble from fractions entirely by using the Card Reader functions 7ISZ and 7DSZ which ignore fractional parts and always add or subtract 1. These two functions always use register 25 but two more functions, 7ISZI and 7DSZI, use register 25 to provide an indirect address. All four functions skip the next step if the integer part of register 25, or of the indirectly addressed register, is zero. Thus a NOP is still required after them, and of course you need a Card Reader.

If you want to avoid using a NOP at all, arrange for the ISG or DSE to be the last step before an END instruction. A program cannot skip over an END as this would drop it into the next program or into the waste land beyond the .END. and therefore any of the increment or decrement functions will have to execute the END even if they were going to skip the next step. You can write a subroutine which is called to do a particular arithmetic operation, and you should put this subroutine right at the end of a program with the increment step as the last step before the END.

6.9 Asking Questions and displaying results

While a program is executing, it normally displays the **flying goose** or **program execution indicator** ⊦ . Alternatively, you can use VIEW nn where nn is a register number, a stack register, or an indirect address. This shows the contents of that register in the display until you use VIEW or AVIEW again, get an error message, stop the program by pressing R/S, or use CLD to clear the display. In this way, one result can be displayed while another result is being calculated. VIEW will display a number or a text string up to six characters long.

To show a longer result or message you need to put it in the ALPHA register and display it with AVIEW. This can display up to 24 characters, 12 of which can be seen in the display at any one time. AVIEW has already been described in Section 4.3.1, and the example program in Section 2.7 also showed how the ALPHA register can be used to display a result.

Apart from displaying results in ALPHA, programs can also use ALPHA to ask for information. It is not always convenient to have the user put all the information for a program into the stack at the start. A program may need to ask additional questions, such as "this date is after February 28, please tell me if it is a leap year". The question only needs to be asked if the date is after a leap year and the year is not given, so it would be a waste of time to ask the question every time the program was used.

A collection of ways to ask questions will be given here. You may not need all of them, but some of the ideas may help in your own applications.

i) To ask for a single value to put in X.

```
"WHAT IS X ?"      The User should put in the number,
PROMPT             then press R/S.
```

ii) In i), the user could press R/S without entering anything. If you want to check whether anything has been put in, use FS? 22 to check the numeric entry flag. To force the user to put a number into X, you can use:

```
CF 22              Now the question will be repeated until
LBL 01             something is put into X. Unfortunately,
"WHAT IS P ?"      flag 22 will be set even if something is
PROMPT             put into X, then deleted with the back-
FC?C 22            arrow key.
GTO 01
```

iii) Sometimes, you need to remind the user of the units used in a program. Rewriting i) to prompt for X in feet per second gives:

```
"X <FT/S> ?"
PROMPT
```

iv) To ask for an alphanumeric value, for instance in a game of hangman, use:

```
"GUESS A CHAR."
AON
STOP
AOFF
```

A better, but longer, alternative is:

```
"LETTER ?"      CLA prevents the user from pressing
AVIEW           SHIFT, K, and appending the letter to
CLA             the contents of ALPHA.  The use of
AON             PSE also limits the time available
PSE             for a guess, and saves the need for
AOFF            pressing R/S.  PSE waits about one
                second, but a new one-second wait
                starts every time a key is pressed.
```

v) If you want to use an alphanumeric prompt without altering ALPHA, store a six-letter message in a register and view that register as necessary:

```
"NEW X?"        This stores the prompt in a register
ASTO 99         unlikely to be accidentally altered.
                Whenever the prompt is needed, do:
CF 22
VIEW 99
STOP            The user can either put in a new
FS? 22          value of X and press R/S or just
GTO 10          press R/S to stop the program.
END
```

vi) When asking for a large number of inputs, make sure the user will not get lost. For instance, request a matrix one element at a time and specify which element is wanted next. To ask for a 4*4 matrix and to store the values in registers 1 to 16 you could use an ISG counter in register 00.

```
01♦LBL "MATRIX?"
02 1.016
03 STO 00
04 FIX 0
05 CF 29
06♦LBL 12
07 "M("
08 RCL 00
09 DSE X
10♦LBL 00
11 4
12 /
13 INT
14 ISG X
15♦LBL 00
16 ARCL X
17 "┣,"
18 LASTX
19 FRC
20 4
21 *
22 INT
23 ISG X
24♦LBL 00
25 ARCL X
26 "┣)"
27 CLX
28 PROMPT
29 STO IND 00
30 ISG 00
31 GTO 12
32 END
```

Lines 2 & 3 set the counter.
Lines 4 & 5 set a useful display mode.
LBL 12 starts the loop.
Line 7 starts the display.
Line 8 gets the current number.
Lines 9 & 10 decrease this by one, so the first element counts as 0. The use of DSE followed by a NOP was described in Section 6.8.
Lines 11 to 15 obtain the row number, using ISG followed by a NOP.
Lines 16 & 17 put the row number into the display.
Lines 18 to 24 obtain the matrix column number using LASTX which was saved at line 13. The use of ISG X instead of 1, + kept LASTX unchanged.
Lines 25 & 26 complete the prompt.
Line 27 puts a zero in X, so that the user can press R/S without any numeric entry to store 0 in a matrix element.
Line 28 is the actual prompt.
Line 29 stores the value in a matrix element.
Lines 30 & 31 increment the counter and repeat the loop 16 times. After that, the program can use the matrix, or ask the user to check it.

This is not a particularly quick way to set up a matrix, but it shows the uses of ISG and DSE.

vii) The examples so far have assumed that the stack can be used as

required, and contains nothing important. If you want to save everything that is in the stack, including the value in X, you can use the following method to prompt for a value and save it in register nn without altering any of the stack registers (not even LASTX):

```
CF 22              This routine stores X in register nn
X<> nn             while the new value is being
RDN                obtained. I am grateful to Jeffrey
"WHAT IS N ?"      Smith who suggested the use of flag
PROMPT             22 to check if a number has been put
FC?C 22            into X. If the user just presses
R↑                 R/S then the stack is rolled up and
X<> nn             the original values are put back in
                   X and in register nn.)
```

viii) One common type of question demands a YES or NO answer. A simple way of dealing with this is to put a Y into register Y, put the reply in X, and compare the two:

```
"Y"
ASTO Y             This is fairly short, but it only
"TOO BIG? Y/N"     checks for a "Y" answer. Any other
AON                reply, or no reply at all, is taken
STOP               to mean NO. Sometimes it is better
AOFF               to check the letter N, or to check
ASTO X             for both these letters.
X=Y?
GTO "REPEAT"
RTN
```

ix) One way to record the answer to a question, is to set or clear a flag. If you need to ask several questions, you can save space by having a subroutine to ask the question, check the answer, and set or clear a flag. To use the following subroutine, put the flag number in register X, put the text of the question into ALPHA and XEQ "YN". The

characters :Y/N will be added to the question, which can be up to nine characters long without scrolling. (If you are unsure about ALPHA operations, reread Section 4.3). The flag whose number is in X will be set if the reply is Y or cleared if the reply is N. The X and Y registers will be unchanged.

```
01♦LBL "YN"
02 CF 25
03 SF IND X
04 RDN
05 RDN
06 AON
07♦LBL 01
08 "┣:Y/N"
09 STOP
10 ASTO X
11 "Y"
12 ASTO Y
13 X=Y?
14 GTO 02
15 "N"
16 ASTO Y
17 CLA
18 X≠Y?
19 GTO 01
20 CF IND Z
21♦LBL 02
22 AOFF
23 R↑
24 R↑
25 END
```

Line 03 sets the flag given in X so that it need only be cleared if the answer is N. If the flag number in X was not 01-29, then an error message stops the program here before any harm is done (flag 25 was cleared at 02 to make sure this would happen).
Lines 04 & 05 save X & Y in registers T & Z.
Lines 8 to 10 add :Y/N to ALPHA, display the question and store the answer in register X.
Lines 11 to 14 check if the answer was Y and if so go to tidy up.
Lines 15 to 19 check if the answer was N, and if not, go back to LBL 01 to ask the question again. Line 17 makes sure only the question is seen, not the incorrect answer.
Line 20 clears the flag if the answer was N.
Lines 23 & 24 replace X & Y.
Lines 06 & 22 toggle ALPHA mode to display the questions and answers.

Delete lines 04 and 24, and change line 20 to CF IND T if you only need to save X. The idea of a YES or NO subroutine was given on page 14 of HP KEY NOTES Vol 5 Number 1.

x) Now for an original idea. Suppose you want to ask a yes or no question without altering anything in the stack, ALPHA, or any numbered data registers. All you need is to set a flag if the answer is YES. You can use flag 47, the shift flag to do this. SHIFT can be pressed during a PSE without stopping the program and without altering anything except flag 47. Then you can check if the flag is set by FS? 47. You can even clear flag 47 with CLD.

Here is a short program that stores a question in register 19, and views that register to ask the question so that ALPHA is unaffected. The question asks the user to press SHIFT if the answer is yes. If SHIFT is pressed, the program sets flag 00 to provide a permanent record of the reply, because any message, including error messages, will clear flag 47.

```
01♦LBL "F47"
02 "↑ IF Y"
03 ASTO 19
04 "   "   "
05 "   "   "
06 "   "   "
07 VIEW 19
08 PSE
09 PSE
10 CF 00
11 FS? 47
12 SF 00
13 CLD
14 "   "   "
15 "   "   "
16 "   "   "
17 FS? 00
18 BEEP
19 END
```

Lines 02 & 03 store the question.
Lines 04 to 06 could be anything you need.
Line 07 displays the question.
Lines 08 & 09 let the user see the question and press SHIFT to answer YES. There are two pauses, not one, to give the user more time to reply.
Lines 10 to 12 clear flag 00, set it if SHIFT is pushed, and clear SHIFT; you could use flag 47 itself if the display is not altered first.
Lines 14 to 16 could be anything.
Lines 17 & 18 provide an audible feedback - after all you normally press SHIFT to get a BEEP.

6.10 Using subroutines and structuring programs

To round off the chapter this section will provide some suggestions about the structuring of programs and the use of subroutines. Two important things to do when you are writing a program that will be used more than just once are to plan out the program before you write it and to **document** it afterwards. "Documenting" a program means writing down what it does and how so that you can use it later or rewrite it without having to work out what it does. Planning a program before writing it is just as important, and indeed the original plans can be a useful part of the documentation. When you are planning a long program you should split it up into several parts each of which can be designed and tested on its own. Each part could be written as a subroutine, with a main routine to call (i.e. execute) each subroutine. You can check each part, then put them all together and remove unnecessary or duplicated pieces to make the program shorter.

A subroutine is a set of instructions which can be written as a separate part of a program. The most common reason for writing a set of instructions as a subroutine is that this set is to be used more than once, in more than one place. Instead of repeating the whole set you can write it as a subroutine, then call that subroutine (by using XEQ) whenever necessary. A set of instructions that is to be used several times in the same place is best made into a DSE or ISG loop, not a subroutine. A very short set of steps is not worth turning into a subroutine: more space can be wasted on LBLs, RTNs and XEQs than is saved. Exact formulae for space savings will be given in Chapter 8 but it is rarely worth turning less than four steps into a subroutine. Since a maximum of six subroutines can be pending (see Section 6.3) you should not use subroutines if it is not necessary. (Synthetic Programming allows you to extend the subroutine return stack; see Chapter 14.)

In some computer languages each subroutine can have its own set of data values, different from variables that have the same names in the main program. This set of local variables constitutes an **environment** which belongs only to one particular call of a subroutine and to returns to that

subroutine. On the HP-41, FOCAL does not provide separate environments so every subroutine uses the same set of flags and the same numbered data registers, the stack and the ALPHA register. If a subroutine changes any of these and returns to the main program, then the main program will have to deal with the changed values too. Subroutines which do one specific job should make as few alterations as possible so that the main program does not have to make allowances for changing values or flags. Mathematical subroutines in particular should be designed to do all their work in the stack so that the program or routine which calls them can use all the numbered data registers freely. If a subroutine is designed to preserve the stack, or has to use a large amount of data, then it should be very clear which data registers it uses, so as to avoid conflicts with other routines. One way of avoiding conflicts is to use "position-independent" routines which only use indirect addressing. The main program can then decide which subroutine uses which addresses (by putting into the stack the numbers of the registers that the subroutine should use so that the subroutine can pick up these register addresses).

Position-independent subroutines and those that use only the stack can be transported from one program to another without trouble, but this demands that they be properly documented so that you can use the same subroutine later without puzzling about what it is doing. If you are going to write a lot of subroutines like this, you will eventually be able to write entire programs just by putting together a short main routine and a set of pre-written subroutines that it calls. Under these conditions you may decide to use register 00 to tell each subroutine which registers it should use. If you are going to write subroutines that handle a lot of information you may find it easier to use data files; the Extended Memory and the functions to use it are designed for this purpose, see Chapter 11. You can also use the Extended Functions REGMOVE and REGSWAP to copy or exchange the contents of a block of registers, moving them to a different place in memory. This can be done to save the register contents while the registers are used for another purpose, or to put the values into a set of registers where a subroutine expects to find them. Alternatively you can use Synthetic Programming to renumber data registers so that every subroutine

seems to be using registers 0 to 10 (or any other selected set of registers) but in fact different registers are used each time.

Apart from using XEQ to call a subroutine you can GTO to a routine. Any pending returns will be unaffected, so you can XEQ one subroutine, then use GTO to go from that subroutine to another one, then use RTN or END to return to the point from which the first routine was called. The use of GTO IND can however replace XEQ and RTN. This can be particularly useful if a subroutine calls itself, since such a subroutine might well call itself more than six times. (A subroutine which calls itself is "recursive", see Section 6.4; for example a program which calculates the third derivative of a function may require the derivative subroutine to call itself twice over.) The following example shows how GTO IND can be used instead of RTN. A program uses a subroutine to calculate SQRT(1 + X↑2) but six subroutine levels have been used already. Instead of using RTN at the end of the subroutine you can finish it with GTO IND 00 :

```
1                   This part of the program
STO 00              stores 1 in register 00, so
GTO 22              that the routine at LBL 22
LBL 01              will come back here.
.
.
2                   Here 2 is stored in register
STO 00              00 so that the routine will
GTO 22              come back to LBL 02.
LBL 02
.
.
LBL 22              This routine calculates the
RDN                 value SQRT( 1 + X↑2 ) after
X↑2                 removing an unwanted number
1                   from register X. Then it uses
+                   the number in register 00 to
SQRT                go back to the place where it
GTO IND 00          was called from.
```

If you are finding it difficult to trace an error due to an indirect GTO then remember that you can check the values involved by using VIEW nn and VIEW IND nn. You may find it easier to recall an indirect address to register X before going to that address. This puts the address in X where you can check it in case of problems. You can also use double indirection by doing RCL IND nn, GTO IND X, which is like saying GTO IND IND nn.

A separate use for subroutines is to allow a conditional test to execute a group of instructions instead of just one. Consider the set of instructions:

```
X>0?
LN1+X
STO 12
```

This calculates the logarithm of (1+X) only if X is greater than zero. What if you want to calculate the logarithm of (2+X) under the same conditions?

```
X>0?                    LBL "LN2+X"
XEQ "LN2+X"             1
STO 12                  +
                        LN1+X
                        RTN
```

In general you would use a local label for a subroutine like this because a local label and the corresponding XEQ take less space, and can be executed more quickly since there is no need to search CAT 1.

The use of subroutines to help in designing a program was mentioned at the start of this section. There are two separate ideas involved here, **structured programming** and **program structure**. Structured programming means adopting a particular method for writing programs. The purpose of the program should be made clear, then the different tasks that the program must carry out should be identified. A typical program will need the following sections:

1. Set up program requirements, such as the angle mode (initialisation)
2. Get any information the program requires (data input)
3. Deal with the data and do the required calculations (processing)
4. Present the results (data output)
5. Tidy up (clear flags, delete files and so on)

If you undertake a structured programming approach, you will plan out how to deal with each of these sections, perhaps by splitting some of them into smaller parts. You will keep notes on what each part does. You will check exactly what information you will be given, and what results are expected. Perhaps you will even read a book about structured programming. Then you will write and check each part of the program. Only then will you try to put all the parts together and test the whole program. If something is not working, or if the program is too big to fit in the memory available, you will be able to go back to the separate parts and your notes about them. You will be able to use your notes to find errors or remove duplicate pieces of the program to make it smaller and faster. At the end you will have a clear and well documented program. You will have done a lot of paperwork, but you will have made fewer errors than otherwise.

The above may seem like taking a sledgehammer to open a peanut if you are writing a short program for the HP-41 but it does provide a good plan for writing large programs. If you are writing an ordinary HP-41 program then you should still think about program structure, which means the structure of the program itself. The position of various subroutines deserves some thought. If a program or routine finishes by executing a subroutine more than once then it is useful to put that subroutine at the end. For example to execute the subroutine at LBL 90 twice in succession you could finish your routine with:

```
XEQ 90
LBL 90
.
.
END
```

This runs the subroutine twice, or more times if there are two or more XEQs before the LBL. (Be sure not to put XEQ 90 after LBL 90 since this produces an unending loop.)

Another thing that deserves consideration is where the program stops. If a program stops at a RTN or at a STOP then unexpected and nasty things might happen if you press R/S afterwards. One way to avoid this is to follow every RTN or STOP with a GTO that points back to some safe place such as the beginning of the program or a LBL just before a STOP. You can safely finish a program at the following loop:

```
.
.
.
LBL 90
STOP
GTO 90
```

Pushing R/S after the program has stopped here will just make it stop again. (Both the above suggestions also come from John Dearing's book, described in Appendix A. It really is a very good book.) Another way to stop a program is to make it finish at the END; the program is then automatically ready to run from the beginning again when you press R/S. A good trick is to put LBL 99 as the last step of a program and to put GTO 99 at any place where the program might finish.

Exercises

6.A Do you prefer to use global or local labels? A global label provides an easy way to remember what the subroutine does, since it can give it a meaningful name, but it takes up more space than a local label and slows down a running program. Go through some programs (your own or somebody else's) and check if any global labels could be replaced by local labels, or if some local labels could be usefully replaced by global labels.

6.B Try rewriting the matrix input routine (example vi in Section 6.9) to make it more efficient by using some of the ideas suggested in this chapter.

CHAPTER 7 - SOME EXAMPLE PROGRAMS

The chapters following this one will contain many items concerning HP-41CVs, CXs and additional equipment. It is worth pausing and looking at a few example programs that can be run even on the humblest HP-41C with no additions. Some of the programs may be exactly what you need, or you may find them a total waste of time. They are provided as examples of using ideas from the first six chapters. If you wish, you can skip the whole chapter; I won't mind (much).

7.1 Hyperbolics And Inverse Hyperbolics

We are told in the HP-41 manual that the functions E↑X-1 and LN1+X are particularly useful in working out hyperbolic functions and their inverses. In fact only TANH and ATANH gain in accuracy from the use of these extended precision functions. Yet in the Math ROM neither TANH nor ATANH use them.

Here is an alternative set of hyperbolic functions, a little shorter than those in the HP Math ROM. They use only the stack and all preserve the Y register unchanged. They do not call each other, so any selection of them can be kept in the HP-41 without needing the others.

Despite the wide range of functions on the HP-41, there is often only one way to do a job most efficiently; the COSH routine given here turns out to be identical to that in the Math ROM. Moreover all of the routines SINH, COSH, ASINH, ACOSH here are identical to ones written by John Kennedy and given in John Dearing's book "Calculator Tips and Routines". There seem to be no obviously better ways to calculate these four, but TANH and ATANH here are a bit shorter, more accurate and work over a slightly wider range.

Instead of giving the routines in the order SINH, COSH, TANH, ASINH, ACOSH, ATANH, the listings show them in the order SINH, ASINH, COSH, ACOSH, TANH, ATANH. This order makes it possible to put a value in X, then press R/S twice to check the accuracy of SINH and ASINH. Another two

presses of R/S check the accuracy of COSH and ACOSH. A final two presses check TANH and ATANH. This is a worthwhile trick whenever you are writing routines to calculate a mathematical function and its inverse. Just to be doubly sure, you can try executing each pair in reverse order too.

```
01♦LBL "SINH"
02 E↑X
03 ENTER↑
04 1/X
05 -
06 2
07 /
08 RTN
09♦LBL "ASINH"
10 ENTER↑
11 X↑2
12 1
13 +
14 SQRT
15 +
16 LN
17 RTN
18♦LBL "COSH"
19 E↑X
20 ENTER↑
21 1/X
22 +
23 2
24 /
25 RTN
26♦LBL "ACOSH"
27 ENTER↑
28 X↑2
29 1
30 -
31 SQRT
32 +
33 LN
34 RTN
35♦LBL "TANH"
36 ST+ X
37 E↑X-1
38 RCL X
39 2
40 +
41 /
42 RTN
43♦LBL "ATANH"
44 LN1+X
45 1
46 LASTX
47 -
48 LN
49 -
50 2
51 /
52 END
```

7.2 Review Registers

Not everybody uses hyperbolics, so here is a program of potential use to anyone with an HP-41. During keyboard calculations one tends to store values haphazardly in any data register that happens to be free. The HP-41 Continuous Memory makes sure those values stay there until they are needed again, but with so many registers available, there is a lot of work in finding a value.

For example: you have four numbers in the stack registers and you want to save them all for future use. You also want to find the register where you put the value of the speed of light earlier on. How to do this without disturbing the stack or losing any data values? You could spend 20 minutes

doing X<>nn for every nn from 00 to the last memory register, or you could use VIEW nn for every register. (But what SIZE have you got anyway ?)

The program here displays the contents of every register that contains anything other than zero. If you have a printer, the program can be used instead of PRREG, saving paper because it will not print zero values. As registers containing zero are not displayed, the process is speeded up; those registers not displayed are free for use. The L register is used to hold an ISG counter, so that the stack and numbered registers are unchanged. Alphanumeric values are displayed with :A after them so that numbers will not be confused with text displays of numbers.

Everyone has his or her favourite SIZE finder, but this program shows the SIZE as a fringe benefit if you let it run until it stops. Alternatively, you can stop the program whenever it displays a register. Do not stop the program while the flying goose is displayed though, or you may find X exchanged with a data register. In that case, just do X<> IND L. This program will work even if the SIZE is zero.

```
01♦LBL "REGS"
02 SF 25
03 X<> L
04 CLX
05 X<> L
06♦LBL 01
07 X<> IND L
08 FC? 25
09 GTO 09
10 X≠0?
11 GTO 03
12 X<> IND L
13 GTO 02
14♦LBL 03
15 FIX 0
16 CF 29
17 "R"
18 ARCL L
19 "⊢="
20 FIX 4
21 SF 29
22 ARCL X
23 FC?C 25
24 "⊢:A"
25 X<> IND L
26 AVIEW
27 PSE
28 CLD
29 SF 25
30♦LBL 02
31 ISG L
32♦LBL 00
33 GTO 01
34♦LBL 09
35 "SIZE="
36 ARCL L
37 AVIEW
38 END
```

Lines 2 to 5 set the loop counter in L. LBL 01 starts the loop, trying to get the next data register, and then going to the size display section if the register cannot be found - thereby giving an error to clear flag 25. Line 10 checks if X is non-zero and goes to the display section at LBL 03 if X is not zero. A text value will give an error, but because flag 25 is set, the program will go on to the next step and will again get to LBL 03. (See the remarks about errors in conditional tests, in Section 6.7.) Lines 12 and 13 restore X and go on to the section that prepares the next loop execution. At LBL 03 the register number and its contents are displayed. FIX 4 is used as it is the default display setting and prevents scrolling in most cases - you can replace this with a display setting of your own choice. Lines 23 and 24 put :A after alphanumeric strings. An alphanumeric (text) string in X will have caused an error at line 10, so flag 25 (the error flag) is clear if X contains a text string. Line 25 puts back the value being displayed. Line 26 displays the register number and contents, then line 27 pauses. The user can interrupt the program safely here because everything is in its proper place, and flag 25 is clear so that any subsequent error by the user will not be ignored. Line 28 clears the display and only then is flag 25 set again to check for errors at steps 07 or 10. Lines 30 to 33 prepare the next loop execution. Lines 34 to 38 display the SIZE at the end of the program. At the end flag 25 is clear (cleared at step 07) and the display mode is as set by lines 20 and 21.

This program follows general pieces of advice given earlier. It leaves flag 25 clear, and stops at the END so that it can be re-run with R/S. Register L is used so that the rest of the stack is unchanged. X≠0? is used (with flag 25 set) in preference to X=0? so that the alphanumeric values will not cause trouble. Finally, a point is provided at which the user can safely interrupt the program.

7.3 Integration With Infinite Limits

Most numerical integration programs for the HP-41 integrate between finite limits. The High-Level Maths Solution Book provides for an infinite upper limit:

$$I = \int_{a}^{\infty} f(x)\,dx$$

This also lets you integrate between minus infinity and infinity:

$$I = \int_{-\infty}^{\infty} f(x)\,dx = \int_{-\infty}^{0} f(x)\,dx + \int_{0}^{\infty} f(x)\,dx$$

$$= \int_{0}^{\infty} f(-x)\,dx + \int_{0}^{\infty} f(x)\,dx$$

In general $$\int_{-\infty}^{a} f(x)\,dx = \int_{-a}^{\infty} f(-x)\,dx$$

and so the formula in the Math Solutions book can be used for integration from minus infinity to plus infinity. However the program in the book uses sixteen-point Gaussian quadratic integration so if you want to use more steps to obtain greater accuracy, or to use a different formula, you have to try something else. Replacing infinity with the largest number the HP-

41 can handle can lead to nonsensical results since most of the values of the integrand will be calculated at very large values of x. Looking for a reasonable upper limit by trial and error can take a very long time. An example to show that it does not work is given at the end of this section.

An alternative method is to use one of the general-purpose numerical integration programs such as INTG in the Math ROM, or IN in the PPC ROM and to change the variable that has infinite limits to one that has finite limits. The programs use a user-supplied routine to calculate the function f(x) at a value x and they make a choice of x values which lets them estimate the integral. If the variable is changed from one that has infinite limits to one that has finite limits then these general-purpose programs can be used provided that the routine to calculate f(x) is supplemented by a second routine that makes the change of variable. In the following, I shall call these routines FX and TX respectively. A reasonable choice for the change of variable is:

$$x = \mathrm{TAN}(\theta)$$

or

$$\theta = \mathrm{ATAN}(x)$$

because TAN(infinity) is pi/2 radians, so infinity can be replaced by pi/2, and the integral can then be rewritten:

$$I = \int_a^b f(x)\,dx = \int_{\arctan a}^{\arctan b} f[\tan(\theta)]\,\frac{d\theta}{\cos^2\theta}$$

When you make this change of variable the following things happen:-

Where the lower limit is $-\infty$ it becomes - pi/2
Where the upper limit is $+\infty$ it becomes + pi/2

Where the routine to evaluate f(x) was FX, it now becomes a new routine TX which uses FX as follows:

```
01 *LBL "TX"
02 RAD
03 COS
04 ST* X
05 X=0?
06 RTN
07 STO nn
08 X<> L
09 TAN
10 XEQ "FX"
11 RCL nn
12 /
13 RTN
```

02 Put the HP-41 into RADians mode for correct integration.

03-04 Get $\cos^2\theta$

05-06 If $\cos^2\theta$ is zero, then return with zero since the integrand has to be zero at pi/2 or -pi/2 if it is a closed integral. $\cos^2\theta$ is compared to zero in preference to Cos θ as it reaches zero sooner. This is not an exact test, but it is sufficient in nearly all cases.

07 Store $\cos^2\theta$ in a register

08-09 Put tan θ into X. It would be ∞ if cos θ were zero, but the RTN at line 06 avoids this.

10 Now execute the original function FX, or whatever its name is. Note that the angular mode is RAD; FX may need to change it.

11-12 Divide the result by $\cos^2\theta$

13 Return to the main integration routine.

To calculate a numerical integral with an infinite limit using one of the standard HP-41 integration programs (Math ROM, PPC ROM, etc.) with TX do:

1) Enter the function f(x) as a program in the HP-41. The program is assumed to start with x in the X-register and to return with f(x) in X. It can have any global label up to six characters long.

2) Enter the function TX given above. At lines 07 and 11 use a numbered data register that is not needed by the integration or the f(x) program. At line 10, use the name of the f(x) program, or use an indirect register containing the name.

3) Run the integration program as usual except:
   i/ Give the lower limit as ATAN(a) in radians instead of a, or as -pi/2 instead of - infinity.
   ii/ Give the upper limit as ATAN(b) in radians instead of b, or as pi/2 instead of + infinity.
   iii/ Give the function name "TX" instead of the original function name (which I have called FX in these notes).

For example you can use the Math ROM program INTG to calculate:

$$I = \int_1^{\infty} 1/x^2 \, dx$$

by doing the following:

1) Enter into your HP-41 the two functions given below

```
01 *LBL "TX"        14 *LBL "FUNC"
02  RAD             15  1/X
03  COS             16  X↑2
04  ST*X            17  END
05  X=0?
06  RTN
07  STO 08
08  X<> L
09  TAN
10  XEQ "FUNC"
11  RCL 08
12  /
13  RTN
```

2) Make sure the Math module is plugged into your HP-41.

3) XEQ "INTG" to initialise the integration program.

4) Execute RAD, 1, ATAN, PI, 2, /, SHIFT, A to give the upper and lower limits to the integration program.

5) Key in 16, SHIFT, B to compute the integral using 16 sub-intervals.

6) Key in the function name TX, then press R/S to run the program.

7) Wait for the answer: 1.0000 accurate to four decimal places.

For comparison, an attempt to estimate a 16-point integral of $1/x^2$ from 1 to infinity by using INTG with the limits shown, gives the following results:

upper limit	result
10	.9071
100	2.2988
1000	20.8415
1 E90	2.0833 E88 (!)

None of these is reasonably close to the true answer.

Naturally enough this whole section assumes that the functions to be integrated do not have any singularities in the range of integration and that the integrals are closed (finite).

7.4 Random Numbers

Random numbers may be required in games programs, in statistical analysis programs, or in the design and simulation of experiments. Obviously, it is sensible to choose a Random Number Generator (RNG) to fit the job at hand. A Monte-Carlo integration program demands a more sophisticated RNG than does a Sub-Hunt game. Nevertheless, the best RNG used by HP on the HP-41 is in the Games ROM. I shall give a (relatively) short description of this RNG and of other RNGs, then suggest a few RNGs suitable for various uses on the HP-41. You can skip the theory if you prefer and go directly to the list of RNGs at the end. A lot of work on RNGs for calculators has been done by members of HP user groups and most of the suggestions here come from user group journals. The references are given at the end of this section; this is another good example of what you can learn and take part in by joining a user group.

The purpose of a good RNG is to provide a sequence of randomly distributed numbers X where:

$$0 \leq X < 1$$

Evidently, it is required of a RNG that its behaviour should be well known and understood. Computer RNGs rely on mathematical algorithms, so the same results are obtained if a RNG is used a second time. This means that they

are not truly random, (to underline this they are called Pseudo Random Number Generators) but also that they can be studied. The best studied type of computer RNG algorithm uses the Linear Congruential Method. This algorithm takes three numbers c, a, m to generate a new random number X_{n+1} from the previous number X_n .

$$X_{n+1} = (a*X_n + c) \text{ MOD } m$$

On calculators, the modulus, m, is nearly always taken as a power of 10, usually 1. The formula then becomes:

$$X_{n+1} = \text{FRC } (a*X_n + c)$$

This is called a Mixed Congruential Method. To speed things up, c is often set to zero giving

$$X_{n+1} = \text{FRC } (a*X_n)$$

This is often called a Multiplicative Congruential Method. The choice of a and c has been studied extensively. The initial choice of X, called the **seed**, X_0, also deserves some consideration.

An excellent book on this subject is "The Art of Computer Programming, Volume 2" by Donald E.Knuth. Details of this book are given in Appendix A. Knuth's book tells us what to understand by a random sequence; in brief, the numbers should have a mean of 0.5 and a standard deviation of 1/SQRT(12), they should not cluster around certain values, nor should they show any marked relation between pairs, sets of three or any larger sets. Ideally, a sequence should never repeat itself (cycle), but on a machine capable of representing only a finite set of possible numbers cycling is bound to occur eventually. The RNG used in the Games ROM, the PPC ROM and the CCD module was originally developed by Don Malm for the HP-65 using advice from Knuth's book. It uses the algorithm:

$$X_{n+1} = \text{FRC}(9821*X_n + 0.211327)$$

This satisfies Knuth's most stringent test, the spectral test, and has a long cycle; it repeats the same sequence of a million numbers regardless of the starting value (seed) used. For most purposes, this is eminently satisfactory, though it is not a perfect RNG, as can be shown by additional tests involving the sums and differences of consecutive pairs. This only

shows up in long runs; it has been demonstrated by Brian Steel on an HP-110, and may therefore not really be a fair test for an HP-41 program.

For anyone sufficiently dedicated (or crazy enough) to want more than a million random numbers, a suggestion by R. Moore (see references below) is:

$$X_{n+1} = \text{FRC}\ [\ 21*(\ X_n + \text{PI}\ )\ ]$$

which gives a cycle period of 100 million, and uses less memory on the calculator because PI takes one keystroke, and 21 uses two. If you replace the 21 with 61, 101, 121 or 161 you get a cycle of 10 million. These numbers are too small for an ideal RNG (because two or three digits cannot guarantee sufficient randomness), but the results are pretty good for a handheld device.

There are just a couple of minor reservations about these wonderful RNGs. The first is that they compensate for their long full cycle periods by cycling rapidly from the back. The least significant digit goes through a cycle of ten values, the least significant pair cycle every hundred values and so on. If your use of RNGs is significantly affected by the low order digits, you should take care. In particular, all digits but the first repeat after one tenth of the cycle. The other reservation concerns the use of the stack. An ideal RNG would behave like any other monadic function. It would replace X_n in the X register with X_{n+1} without altering registers Y, Z, T and X_n would also be saved in register L. Since all RNGs given here finish with FRC, register L cannot contain X_n unless numbered data registers are used. Some RNGs can, however, preserve Y, Z, and T.

One such RNG, suggested by V. Albillo, R-D,FRC, gives:

$$X_{n+1} = \text{FRC}\ (\ 180/\text{PI}*X_n\ )$$

This is simply a linear congruential RNG; given a negative seed it returns a negative value.

Another short RNG that changes only X and L, suggested by J.W.Mills is: ATAN, FRC. This does not work in RAD mode, but is surprisingly good in DEG

mode and not too bad in GRAD mode. Both these RNGs though are sensitive to the seed used. Vic Heyman suggests PI, SIN, TAN (in DEG mode) as a good seed for the second case (this is equivalent to just PI, SIN here, but it can be used as a seed for other RNGs too).

If you do not choose a good seed, both these RNGs will soon hit cycles which repeat, in the worst case, every 209 values for R-D and every 424 for ATAN in DEG mode. (ATAN in GRAD mode has one cycle that repeats every 358 values.) Worse still, both get stuck at zero. I prefer adapting R-D to provide a much longer worst cycle of 9432 with a reasonable distribution:

CHS, 10↑X, R-D, FRC

This does not get stuck at zero and will give positive values from a negative seed. It also avoids the problem that R-D, FRC can produce pairs of small numbers if you start at a value near zero (any number less than 0.0017 is bound to be followed by a second number smaller than 0.1). (Mind you, any value close to 1.0 will be followed by a value close to 0.7309. It is in the nature of pseudo random number generators that a given value is followed by another one which is predetermined by the PRNG formula.) The last three RNGs are shorter than the mixed congruential ones and preserve registers Y, Z, and T. The first of them is the fastest RNG for the HP-41. The first two are ideal for games programs, except that the second one must not be used in RAD mode. The third one takes about 10000 steps from most starting values before it hits the cycle, and almost another 10000 steps before the cycle first begins to repeat; a total of 20000 random numbers with a reasonable distribution, which is quite enough for most purposes. With a seed of PI, SIN, TAN it takes more than 20000 to reach a cycle and 16000 before the cycle repeats. Nevertheless, the mixed congruential RNGs beat the others for most serious purposes. (The three just given are only safe for one-dimensional distributions.) Another linear congruential RNG, suggested by John Baker, uses 3579 as the factor a and .031019 as a seed. This has an effective period of 25000 and passes the spectral test, so it can be used for 2, 3, 4, and 5 dimensional distributions.

Here then is a list of RNGs you can try for various applications:

1) 9821, *, .211327, +, FRC	Provides 10↑6 values and comes out well in the spectral test. Uses a lot of memory though, unless you call it from the Games, CCD, or PPC module.
2) PI, +, 21, *, FRC	Nearly as good as 1) and provides 10↑8 values. You can also use 61, 101, 121 or 161 instead of 21 -- use these if you want the good properties of 2), but do not want to use 2) itself. Note that 1) and 2) cycle in their lower digits.
3) 3579, *, FRC	For a good linear congruential RNG use a seed of .031019. If you want to avoid disturbing the stack, store 3579 in a numbered data register nn. Then generate X_{n+1} from X_n as follows. Put X_n into register X. X<>nn, ST*nn, X<>nn, FRC. The effective period is 25000, and after this a new pattern emerges, differing from the first one by only a constant.
4) R-D, FRC	For a very fast RNG in games or quick calculations.
5) ATAN, FRC	(Not in RAD mode; preferably with a seed of PI, SIN in DEG mode). Like 4), but it is slower, has a slightly better distribution, and shares the property of not disturbing Y, Z or T.

6) CHS, 10↑X, R-D, FRC	Another alternative to 4) and 5). Slightly slower than 4), much longer cycle and less influenced by the seed. Will not produce an excess of pairs of small values near zero which 4) can do. Does not get stuck at zero, does not disturb Y, Z, or T and gives positive values from a negative seed.

One final word of advice. To paraphrase Knuth: Run each program that depends on random numbers at least twice using different sources of random numbers. This will give an indication of the stability of the results. Better still, read Knuth or some other serious work on RNGs for yourself before you embark on any important Monte-Carlo calculations.

All the references in this section except Knuth were taken from the PPC Calculator Journal and its predecessor 65 NOTES. Volume, issue and page are given below. Details of user clubs and journals are in Chapter 13.

Random Number Generators by Vic Heyman V4N8 p1-6
RN Generators by Rick Moore V6N2 p20
Fastest Random Number Generator by Valentin Albillo V7N6 p35
A simple RNG test program was given by Wm. Kolb in V7N4 p17a
On Calculator RNGs by John L. Baker V8N3 p23-24
See also HP11C RNG simulator program V9N4 p17d (barcode V9N8p29d)

7.5 Complete Arc Tangent - ATAN2

Finally a short example: the ATAN function can only provide an angle between -90 and +90 degrees, but you sometimes need an answer between -180 and +180 degrees. This can be obtained if the arc tangent is found in terms of the ratio between two numbers x and y. The function ATAN2(X,Y) gives an answer:

$$\text{ATAN2}(x,y) = \text{arc tangent}\ (\ x/y\ )$$

A routine to simulate this function exactly like an ordinary bifid HP-41 function (such as division) should use the values in registers X and Y, return the result in X, drop the stack so that Z was copied into Y and T was copied into Z, and would save the previous X in L. Writing a routine like this instead of just any routine to calculate the function demands a little extra work, but the result is a routine that you can use without worrying about its effects on the stack or other registers. Two ways of doing the job assuming registers X and Y contain the numbers x and y are:

```
LBL "ATAN2"        LBL "ATAN2"
P-R                P-R
CLX                RDN
RCL Z              RCL Z
RDN                RDN
RTN                RTN
```

You can check that both simulate an HP-41 bifid function exactly. They work in any of the angle modes. Both RDN and CLX followed by RCL Z copy the original value of T into X, and then RDN puts it back into T. CLX followed immediately by RCL Z does not lift the stack, because CLX disables stack lift, and it is about 6 milliseconds faster than RDN (which is important only in very long programs). If you are unlucky enough to press R/S just after CLX though, you will enable stack lift again and the stack will be wrongly arranged when you restart the program. It is therefore better to use RDN, as on the right, not CLX whenever you are rearranging the stack.

Routines that you will use often or that you will give to other users are best written to simulate the native (CAT 3) HP-41 functions if at all possible, saving the stack values and placing X in LASTX. Some subroutines to save the stack contents will be described in Section 11.4.

Exercises

7.A If you do not have a Math ROM or a PPC ROM then try to write a numerical integration program yourself. It should request the upper and lower limits of the integration interval, and the name of a CAT 1 routine, FX, which will evaluate the function f(x) at the point x. The integration program should then select a number of points between the lower and upper limits, call the routine FX to evaluate f(x) at each of these points, and then calculate an approximation to the integral.

7.B Try combining the hyperbolic functions and the infinite limits integration routine to integrate 1 - TANH(x) from 1 to infinity.

7.C One use of RNGs is for numerical integration. The RNG is used to select a large number of points between 0 and 1, the range is scaled up to cover the integration range from the lower to the upper limit, and f(x) is calculated at each point. The results are added up, divided by the number of points and multiplied by the integration range to provide an estimate of the integral. This method is used for functions which are difficult to integrate using normal numerical methods, and also for multi-dimensional integration; you generate two or more random numbers to produce each random x,y or x,y,z point at which you then calculate f(x,y) or f(x,y,z). A use like this obviously demands that the RNG should not produce related groups of numbers. Try writing a random-number based integration program (they are called Monte-Carlo integration programs) for the HP-41, and see how many random points are needed to obtain an integral as accurate as that produced by an ordinary integrator on a module or written for Exercise 7.A above.

PART III

Extended Programming

CHAPTER 8 - MORE ABOUT MEMORY

8.1 Space, Time and Numbering

By now you should know enough about the normal functions available on every HP-41 that you can do calculations from the keyboard efficiently and can write good programs. (Or else you knew it all anyway and skipped the earlier chapters; in that case you missed some useful tips: too bad.) You have seen that the HP-41 can do much more than you would learn just by reading its manuals, which are there to tell you how an HP-41 works, but not necessarily what you can do with it. We shall now go on to more ambitious uses of the HP-41 and to the devices available for use with it.

The kind of question that this chapter will help to answer is -- how can I make my programs shorter and faster? Program speed and length are closely related; a shorter program is usually faster just because it has fewer instructions to perform. The answers to such questions can be found by studying how the HP-41 memory is used to store programs and data. The chapter will also help you understand how the HP-41 sends data to plug-in accessories, and this information will be useful later on. The layout of program instructions in memory may be of little interest to people who do most of their calculations from the keyboard, but related topics such as the storage of data or key assignments in memory are of interest to them and will also be covered.

Before turning to details of data or program structure in the HP-41 memory let us have a quick look at information storage in general. A popular type of quiz game consists of a panel asking questions to which the answers can only be "yes" or "no". Any piece of information can be obtained in this way if the right questions are asked. Electronic computing devices work on the same principle, they store a mass of yes/no answers which are meaningful only if the corresponding questions are known. For example each HP-41 flag is stored as an answer to some question like "is USER mode set?" or more generally "is the flag set?"

The answer to a single yes/no question is the smallest item of information that can be stored. On electronic devices this answer is treated as a number, 1 for "yes" and 0 for "no", and it is called a **bit**; this is an abbreviation for "binary digit" which means it is a number that can have only two values. By combining bits a computer can do arithmetic with **binary, octal, decimal** or **hexadecimal** numbers. The operations that a computer carries out on bits are often called **logical operations**, not arithmetic. Those readers who know about computer number systems can skip the next two paragraphs if they wish.

If you do not know how a string of bits can be used to represent a number consider the following set of yes/no questions:

i. Is the number odd or even? Write a 0 for even, or a 1 for odd.

ii. Is the result of dividing it by two odd? Write a 0 or 1 to the left of the first 0 or 1.

iii. Is the result, if you divide by two another time, odd or even? Write another 0 for even or 1 for odd to the left of the previous answers.

iv. If the result of dividing by two, once again, odd or even? Write another 0 or 1.

The answers to these four questions, written down as four bits, will let you uniquely describe any number from 0 to 15. Try it and see; if you take the number 10, the answer to the first question is 0 (because 10 is even). Divide 10 by two and you get 5 which is odd, so the answer to ii is 1. Divide by two again, ignore the remainder, and the result is 2 so the answer to question iii is 0. Divide by two again and the result is one, which is odd, so the answer to iv is 1. Write these four answers down right to left and 1010 is the result that you get. Repeat the process for the number zero and the same four questions will give the result 0000 (because zero is an even number). Fifteen will be written as 1111. The four questions and their answers let you describe any number from zero to

fifteen by writing down a binary string of four bits. The number sixteen will give the same result as zero, so another question will have to be asked to distinguish between sixteen and zero. All numbers bigger than fifteen will need more than four questions, and more than four bits. With another four bits any number between zero and 255 can be represented.

A row of eight bits is fine for a computer but difficult for a human being to recognise or memorise. To make a long binary string more legible we can split it up into groups of three or four bits, and write each group as a single symbol. In octal arithmetic, groups of three bits each are replaced by the symbols 0 to 7. In hexadecimal, groups of four bits are written down as a number between zero and fifteen, using the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. You can see that F stands for fifteen. Any number that would be written in the ordinary numbering system (decimal) as a value from zero to 255 can be represented as two groups of four bits (see previous paragraph) or as two hexadecimal digits, from 00 to FF. For example the decimal number 139 can be written in binary as 1000 1011. If you write this as two decimal numbers you get 8 followed by 11 and if you use the hexadecimal symbols this becomes 8B. Converting numbers from decimal to hexadecimal is a nuisance if you have to do it yourself, but you can use a conversion table (there will be some of these later in the chapter), and computers do it automatically anyway. Arithmetic and other operations can be done on binary or octal or hexadecimal numbers, and although this may be difficult for people who are accustomed to the ordinary decimal numbers used in everyday life, it is easy for computers. Hexadecimal numbers will be used in several places in this book because they are closer to the symbols used inside the HP-41, but if you feel uncomfortable about them then use one of the tables (Table 8.1 will do) to convert them back to decimal. (The words "octal" and "hexadecimal" are recent inventions; "hexadecimal" is a Latin-Greek mixture and should really be written as "hexagesimal" or "sexadecimal". Somehow it never is.)

The HP-41 stores your program instructions as groups of eight bits or two hexadecimal digits. These groups are called **bytes**. Since a byte can have one of the values from 0 to 255, there could be 256 program instructions

one byte long each. In fact many program instructions are more than one byte long, but their basic unit is always a byte. Text characters are also represented by one byte each. The numeric digits 0 to 9 can be represented by half a byte (four bits). Half a byte is called a **nybble**; a small bite is a nibble!

How are the bytes put together? The fundamental unit of memory available to the HP-41 programmer is the **register**. We have met registers already, one register is used to store one numeric data value; the SIZE is expressed in terms of registers. A register contains seven bytes, so it can hold a text string identifier and six letters, making seven bytes in all. This is equal to fourteen nybbles. A register can hold a ten-digit number with a sign, and a signed two-digit exponent, making fourteen elements. A register is also equal to fifty six bits, so it can hold fifty-six flags, and indeed the HP-41 flags are numbered from 00 to 55. Figure 8.1 shows the different ways in which a register can be split up into its components. Registers are often called **words** on computers, and the same is done on the HP-41. What with bytes and nybbles inside their words, it seems that HP-41 users are talking with their mouths a bit full !

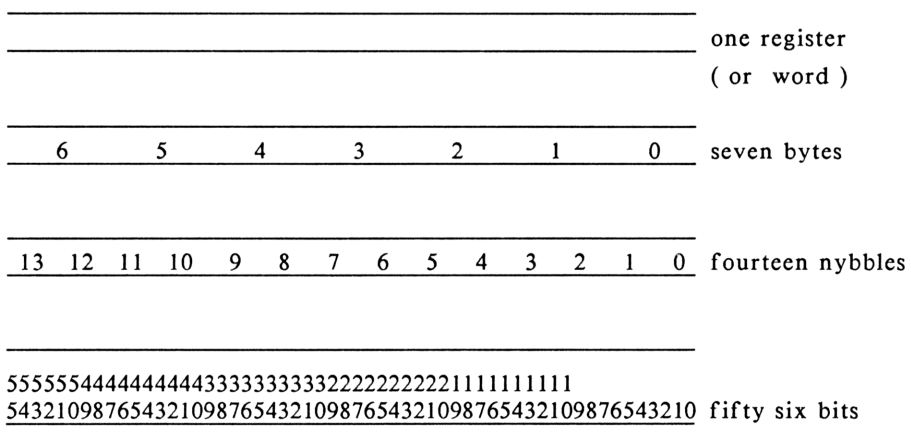


Figure 8.1 Different ways of seeing a register

Most HP-41 operations use bits, nybbles, bytes or registers, but information can be handled in other units. We have already studied programs; a program contains a whole number of bytes, it stretches from immediately after one END (or from the start of program memory) to the next END. Programs are identified by the global label or labels they contain, and they are treated as units by CLP (which clears a whole program from memory) and other functions. Extended Memory and Mass Storage (cassettes or mini disks) are arranged to contain **files**. A file is a complete collection of information, such as a program, a set of key assignments or a personal telephone directory. Each file contains a whole number of registers and is identified by a name. Files can be split up into smaller units called "records", all this will be explained in Chapter 11 which will describe Extended Memory. You can invent your own collections of information too, for example you could decide to treat 9 registers as a 3*3 matrix. Information that is being handled by a computer or by the HP-41 is often called **data**; this word is used to describe any quantity of information that is being used in some operation. Anything that demands the movement or use of a serious amount of data is called **data processing**. Calculating a wages bill for ten employees is data processing, squaring the sine of a number is not, because it uses little data although it does a lot with it.

8.2 Contents of RAM memory

Now you know the possible ways of dividing up one register of the user's Random Access Memory. How are various kinds of information stored in such a register? The HP-41 most commonly stores numbers; Figure 8.2 shows how a number is stored in the fourteen nybbles of one register.

13	12	11	10	9	8	7	6	5	4	3	2	1	0	Nybble
Mant-issa sign	-------		M	a	n	t	i	s	s	a	-------	Expo-nent sign	Exponent	Contents

Figure 8.2 A register containing a number

Nybble 13 contains the sign; a zero is used for positive numbers and a 9 for negative numbers. Nybbles 12 down to 3 contain ten numeric digits, each digit being a number from 0 to 9. The most significant digit is in nybble 12, this should only be zero if the whole number is zero. The decimal point (also called a **radix mark**) lies between nybbles 12 and 11, but does not take up any room. The exponent (power of ten) is stored in nybbles 2 to 0. If it is positive then nybble 2 contains zero while nybbles 1 and 0 contain the exponent value as two digits from 00 to 99. A negative exponent is added to 1000 and the result is stored in nybbles 2 to 0. An exponent of -1 is therefore stored as 999, an exponent of -57 as 943. The HP-41 expects to use only exponents between -99 and +99 so nybble 3 should contain a zero or a nine. (So it is sensible to use 0 and 9 for the mantissa sign too.) It can be seen though that the HP-41 could have dealt with exponents from -500 to +499. It is even possible to create numbers with exponents in this range (using Synthetics of course), and some functions like LOG deal correctly with such numbers. Numbers with unusual exponents like this, or with digits other than 0 to 9 (remember a nybble could contain any value from 0 to 15), or with a zero in nybble 12 are called **Non Normalised Numbers** and can do strange and exciting things (yet another plug for Synthetic Programming).

An example of a number stored in a register is given below:

0 2 5 4 0 0 0 0 0 0 0 9 9 8

This is +2.54E-2 which is the number of metres in an inch. The unused digits after 254 are all filled with zeroes and the exponent is 1000 - 2.

A number stored in this way, with only byte values from 0 to 9 used to hold the digits is called a **Binary Coded Decimal** number. This is usually abbreviated to **BCD**. (An extended version of this code, called EBCDIC is used as an alternative to ASCII, see below.)

Apart from numbers, the registers can hold text strings (also called Alphanumeric values). A register containing text has a 1 in nybble 13 (the sign nybble). Nybble 12 usually contains a zero, but does not need to, since it is ignored if nybble 13 is a one. This accounts for byte 6 of the register, bytes 5 to 0 contain one text character each. The bytes are filled from left to right, but if the string is less than six characters long it is filled out with null bytes at the left (remember a null byte is one whose value is zero). Figure 8.3 shows a register containing the text string "HP-41". Each byte is represented as a two-digit hexadecimal number. (Remember hexadecimal numbers? They were explained a couple of pages ago.)

6	5	4	3	2	1	0	Byte number
1 0	0 0	4 8	5 0	2 D	3 4	3 1	Byte contents
Text		H	P	-	4	1	Meaning

Figure 8.3 A register containing a text string

Null bytes are not displayed if they are stored in a register like this, so you could put this value into register X and would see just the characters HP-41. The numbers used to represent the characters are in the ASCII code (American Standard Code for Information Interchange) which is used by most computers and printers, so that text can be easily exchanged between an HP-41 and some other computer or sent to a printer. The ALPHA register is treated differently from ordinary 7-byte registers on the HP-41; it can contain up to 24 characters (bytes). Null bytes always fill it from the left up to the first character of any text. These nulls at the left are

not displayed, but any nulls that come in between other characters do show up (as a bar at the top of the display). Since the ALPHA register is expected to contain text data it does not need to start with a 10 byte. When a text string is ARCLed to ALPHA, its byte 6 (which should be 10) is only checked to see if the register contains a text string, but is not copied to ALPHA.

If the HP-41 needs to store non-numeric data in one of the user's registers it turns that data into a number or text string. For example the printer function BLDSPEC and the Extended function RCLFLAG store special data as a text string in register X. These text strings contain 1 in nybble 13 and information in nybble 12, so this nybble is not always a zero. Such text strings should not be copied to the ALPHA register as byte 6 will be lost. Another example of storing non-numeric data is the Extended function X<>F which turns the values of flags 0 to 7 into a number and puts this into X.

Any value that is stored in one of the user's registers should either be a number, with a sign nybble of 0 or 9 and obeying the rules for a number, or it should be a text string, with a sign nybble of 1. The HP-41 checks this whenever it recalls a value from a numbered data register and when it uses the contents of a register as an argument for a function. If the value does not obey the rules, it is first turned into one that does obey them, by a process called **normalisation** (nothing to do with East-West relations). This is why a value that does not conform is called a Non Normalised Number (NNN for short). Many values used internally by the HP-41 are NNNs and their use will be an important subject in Chapter 14.

Other kinds of information commonly stored in any HP-41 are programs and key assignments. These will be dealt with in Sections 8.5 and 8.6.

8.3 The Layout of RAM and ROM

Now that we have studied the design of a single data register in RAM, let us see how the RAM registers are put together in groups within the HP-41. Then we can look at the layout of ROM (Read Only Memory which cannot be

altered, the memory that contains the HP-41 built-in system and instructions in plug-in devices). Figure 8.4 shows the layout of the Random Access Memory in an HP-41C without extra memory modules, assuming that the SIZE has not been changed and the statistics registers have not been moved.

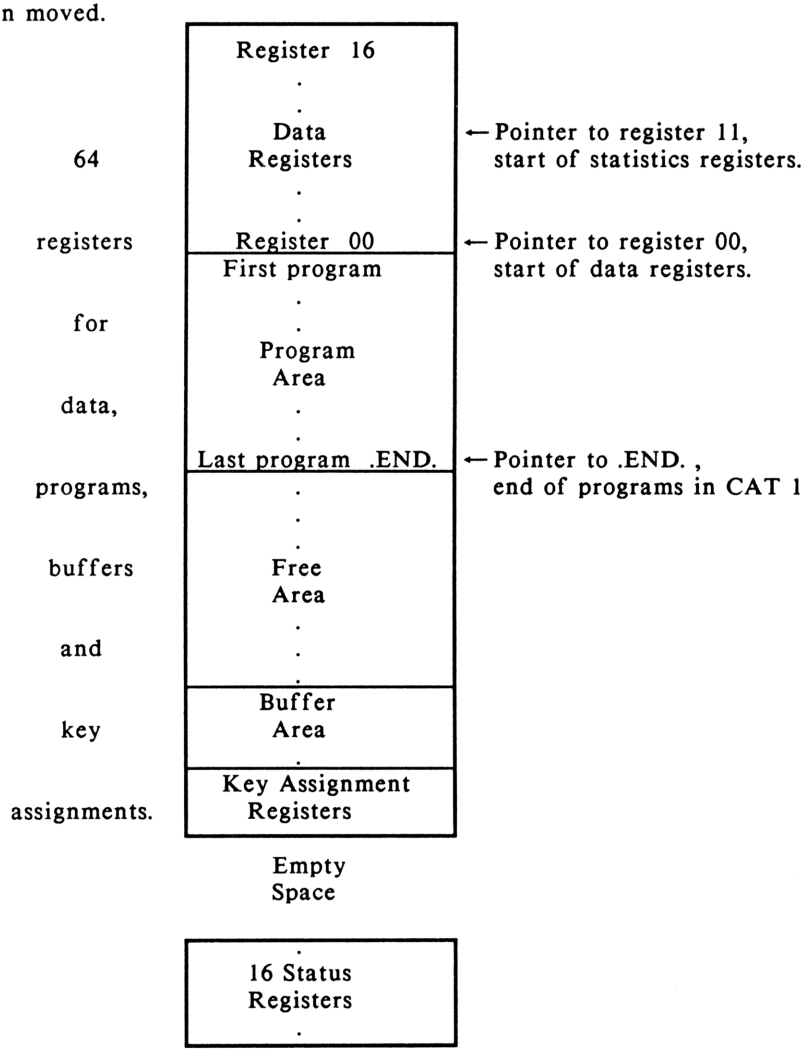


Figure 8.4 HP-41C Random Access Memory (RAM) Layout

The memory consists of one block of 16 registers and a second block of 64 registers. (On early HP-41s all memory came on chips of 16 registers, so the memory was made up of multiples of 16.) The block of 16 contains the X, Y, Z, T, and L registers, the ALPHA register (which is actually made up of 4 ordinary registers) and the flag register. It also contains the pointers to the .END. and to register 00 and the statistics registers, the subroutine return stack and other intriguing things. All this information is stored by the Card Reader on the first track of a "status" card, so these 16 are called the **Status Registers**; much more will be said about them in Chapter 14. Above the status registers lies an empty space, then the main memory.

Going up through main memory (follow Figure 8.4) you first find the registers which store key assignments, unless no CAT 2 or CAT 3 functions have yet been been assigned. Next comes a **Buffer area**. **Buffers** are pieces of memory used temporarily by some plug-in modules. For example the Plotter module can create a buffer to hold plot information, this buffer is no longer used when plotting is finished and the Plotter module has been removed, so it is deleted. The Time Module creates a buffer to hold alarms; this too is deleted if the Time Module is removed or all the alarms have gone off. An HP-41C without plug-in ROMs does not contain any buffers, but a module that wants to create a buffer will put it here, above the key assignments. After the buffers is a free area. This can be swallowed up by key assignments or buffers as they are created and expand upwards. As new programs are written they move down into this area too. If assignments, buffers or programs are deleted then this area gets bigger again. Similarly if the SIZE is increased all the programs are pushed down into this area to make room for more data registers, and if the SIZE is cut down then the program area moves up again and the free area expands. This is why you should make the SIZE as small as possible before writing a long new program or reading one in; you make more room to let the program expand into the free area. After the program has been written or read, you can PACK to make the program area as small as possible, and then increase the SIZE again.

Next comes the program area. The bottom register of the last program contains the .END. in its last three bytes. The .END. points to the END or global label immediately above it and this repeats up to the first link in the global chain (CAT 1), with the programs themselves lying between the ENDs. Immediately above the first program lies data register 00, and the numbered data registers go upwards from it unless SIZE has been set to 000, in which case there are no data registers. Register 00 is in effect a curtain between data and programs, its address is sometimes called the **curtain address** or just the **curtain.** If the SIZE is 000 then the non-existent address above the top of memory is pointed to as the curtain. The address of the curtain has to be stored in a special **pointer** which tells the HP-41 where the data registers begin and where to look for the start of the first program in memory (this is necessary as the program might not have a global label at its beginning). Two other pointers keep track of the position of the .END. and of the first register in the block of statistics registers. If you change the SIZE then the statistics register pointer has to move as well (so that the first statistics register is always the same distance above register 00), and this can mean that the pointer moves above the top data register. This does not give any warning, but an attempt to use the statistics functions will result in NONEXISTENT if any of the statistics registers are located above the top of memory.

The three pointers shown in Figure 8.4 must give addresses in terms of a scheme that does not change when things move in the HP-41 or when memory modules are added or taken away. An **absolute addressing** scheme is used; this means that the addresses are absolutely fixed to the HP-41 hardware. In this scheme, the bottom of memory is called address 000 (at the bottom of the status registers) and all other addresses are numbered in sequence upwards from address 000. Figure 8.5 shows the RAM memory of an HP-41C with four memory modules (or of an HP-41CV), giving the addresses at the top and bottom of each block of memory (in decimal and in hexadecimal numbers). The extra memory provided by the memory modules behaves exactly like part of the main memory shown in Figure 8.4; it merely increases the total number of registers.

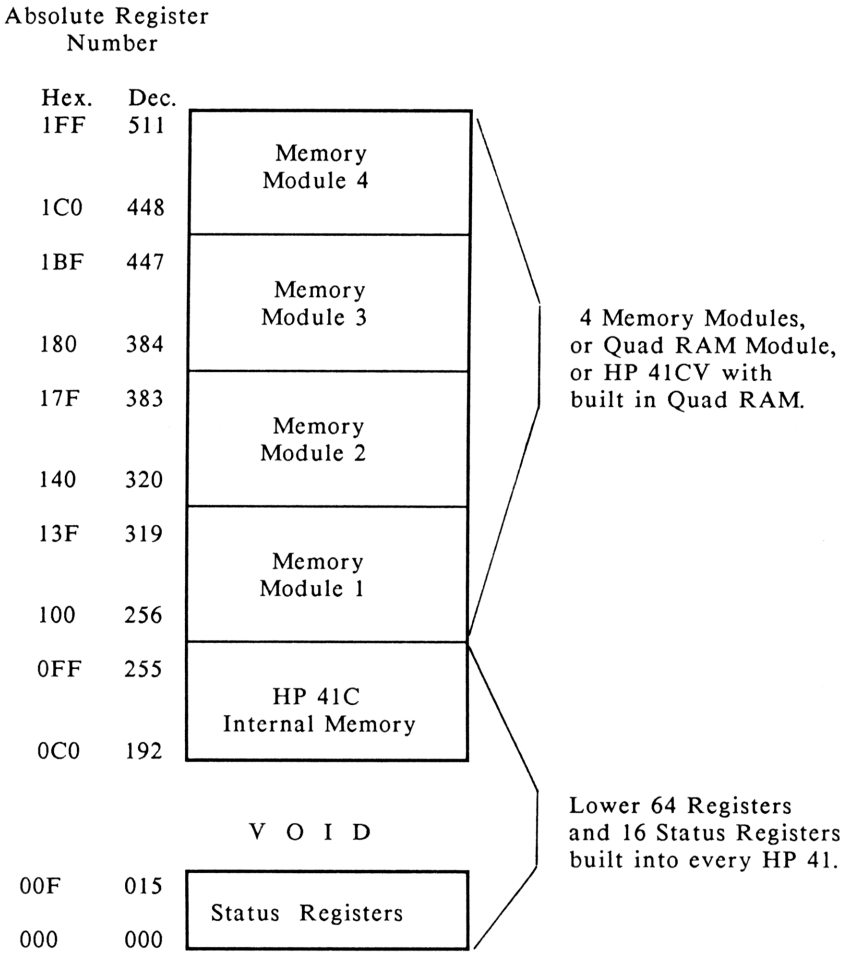


Figure 8.5 HP 41 with additional RAM

Each of the three pointers of Figure 8.4 is stored as a three-digit long hexadecimal number which is the absolute address of the .END., the curtain, or the first statistics register. Once you know how the RAM memory is laid out and how the three pointers are used, you will be able to understand how data and programs are addressed. The way in which the Extended Memory fits into the same RAM addressing scheme will be described in Chapter 15.

It has already been stated in Chapter 2 that the HP-41 internal instructions, and instructions on plug-in application modules, are in ROM (Read-Only-Memory which cannot be altered so you cannot change the HP-41 operating system or the instructions in plug-in modules). This ROM memory is entirely separate from the HP-41 RAM and has its own addressing scheme. Instruction bytes in ROM are 10 bits long, not 8. (Some instructions are 20 bits long.) A single 10-bit byte in ROM is sometimes called a **word**. These words are grouped in **blocks** of 1024. (The symbol **K** is used to mean 1024 in computer jargon, so 1024 words make a 1K block.) The blocks are grouped in **pages** of four blocks each. (A block is therefore a quarter of a page and is sometimes called a **quad**. Just to add confusion, a page is sometimes called a **chip**.) The whole of ROM memory is laid out in units of 4K pages; Figure 8.6 shows this layout. The starting address of each page is given on the left and a brief description of each page is at the right.

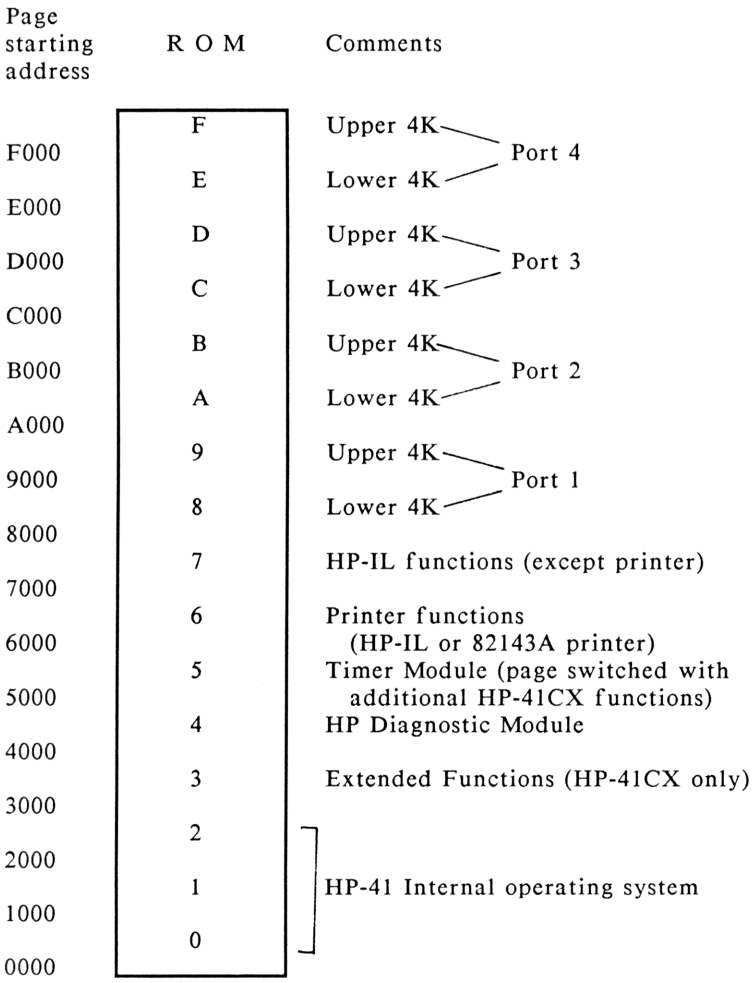


Figure 8.6 HP-41 Read Only Memory layout

ROM chips 0, 1 and 2 are built into every HP-41 and contain the instructions which make an HP-41 behave like an HP-41. Thirteen other blocks can be addressed but they are not always used. Modules and devices that are plugged into an HP-41 fit into these blocks. Originally block 3 was intended to be an additional part of the HP-41 system, containing a special "language ROM", but this was never produced. On the HP-41CX, there is a built-in chip 3 which contains the Extended Functions.

Each of the four HP-41 I/O ports is of a size to hold one RAM module (like a memory module), or one ROM module (like an Application Pack or the control module on a printer cable). If a ROM module is plugged into a port, the ROM addressing scheme lets the HP-41 read up to 8K of instructions from that module, split into two 4K pages. The 2 pages are addressed separately, and most ROM modules only use one of these pages. This one page can occupy the lower half or the upper half of the 8k space available. Pages 8 to F are therefore split up into four pairs of pages, as shown in Figure 8.6. Most of the time only a few of these pages will be in use; the rest will be empty. ROM modules can be moved from one port to another, except for the Card Reader whose shape will only let it fit into port 4 so it uses page E.

Four special ROMs are not normally built into an HP-41 but are treated as part of the HP-41 operating system if plugged in. Regardless of the port they are plugged into, these modules are identified as system modules and are addressed as pages 4 to 7. Page 7 holds all the functions of the HP-IL module except for the printer functions. Page 6 carries the printer functions. These can come from an HP82143A printer or from an HP-IL module. If an HP82143A printer and an HP-IL module are both plugged in, the HP-IL printer functions must be disabled by a switch. Otherwise two sets of instructions are found on the same page, which causes the HP-41 to do strange things and eventually stop working until one set is removed. Page 5 contains the Time module functions. These can come from a plugged-in module, but on the HP-41CX they are built-in. Plugging a Time module into an HP-41CX can lead to the same sort of trouble as two printer modules. On the HP-41CX, however, page 5 does serve two ROMs. One is the

Time module, the other is an additional module containing the extra HP-41CX Extended functions and Time functions. These functions were provided for the HP-41CX in addition to the Extended and Time functions from the Extended and Time modules. When any of these extra functions are needed, a special page-switching instruction is used by the HP-41 to disable the Time module and replace it with the alternative page (or **bank**). The special CX function is executed, then the ordinary Time functions are enabled again. This unusual operation was used by the HP-41CX designers because they did not have any spare pages available for the extra functions. In principle, bank-switching could be used for other modules, so that two 4K banks could be connected to each page, and 12K or 16K modules could be designed.

What about page 4? If a ROM module is plugged in and identifies itself as page 4 when the HP-41 is turned on, then the HP-41 stops obeying pages 0, 1 and 2. Instead the HP-41 tries to follow the instructions it finds in page 4. This is so that a special Diagnostic or Service module can be used to check the HP-41. The Diagnostic ROM, used by HP Repair Centres, can be plugged into an HP-41 to let the repair people look for faults. Even if one of the pages 0, 1 or 2 is faulty, the Diagnostic ROM will usually work since it is not part of the HP-41. Many HP calculators have built-in self test programs, but these can fail if the whole calculator is damaged. With the HP-67 and HP-97, a self-test program can be read from a magnetic card provided in their Standard Applications Pack. Hewlett Packard have decided not to make the HP-41 Diagnostic module available to most customers, though some user groups or large customers have obtained such modules. It would be possible to design other modules to take charge of the HP-41 in the same way, for example a module that would let the HP-41 act as a pocket computer using an alternative language. When the HP-IL printer functions are disabled, they are redirected to page 4 but they jump back to page 0 instead of taking over the HP-41. A disabled HP-IL module can be used in this way to stop a Diagnostic module from working, if the HP-IL module is plugged into a port in addition to a Diagnostic module.

The layout of programs in ROM is too complicated to deal with in detail now; it really deserves a separate book (or several). An outline is given

here for the interested reader and a little more will be said in Chapter 17, the best places to look for are the book "HP-41 MCODE for Beginners" or the ZENROM (see Appendix A). Most ROM modules contain programs in the HP-41 User language (FOCAL), the same language as you use to write programs in RAM. This consists of the familiar functions such as + or SIN or GTO which are made up of 8-bit bytes. Instructions in ROM are 10 bits long, and the two extra bits are not used in FOCAL programs, except to identify the start of a program and the first byte of every instruction (as some instructions can consist of several bytes). The ROM modules can also contain instructions written in the HP-41 internal machine language, which will be referred to here as **M-code**. M-code instructions use all 10 bits of a ROM word; they are similar to instructions in other machine languages. Instructions such as **LDI** (Load Immediate) or **GOC** (Go to if carry bit is set) are found in M-code. These instructions are related to operations carried out by specific circuits in the **CPU** (Central Processor Unit - the circuitry that does the hard work, see next section) of the HP-41, whereas each User language instruction actually tells your HP-41 to execute a whole subroutine written in M-code. The HP-41 system ROM chips 0, 1 and 2 are written entirely in M-code, but plug-in ROMs can carry a combination of User code and M-code instructions. Programs in ROM always run from lower addresses to higher addresses. This is the same direction as data addressing in RAM, but the opposite direction to program addressing in RAM.

All this leads to several important results. The HP-41 has to know if a User code program is in RAM or ROM since the instructions are read in the opposite directions in the two cases. COPY has to reverse the instructions when copying a program from ROM to RAM. M-code programs cannot be written or stored in normal HP-41 RAM memory since this has only eight bits for each instruction, not ten. M-code is however so much more powerful and faster than User code that some HP-41 owners have developed ways of using it; this will be mentioned again in Chapter 17.

The ROM layout is such that a single port could support several modules, if only they could all fit in it. With a Port Extender (see Chapter 12), several modules can be connected to one port. This port could then hold a

lower 4K ROM (for example a Maths module) and an upper 4K ROM (for example an Auto Start/Duplication ROM), or a single 8K ROM. It could at the same time hold a system extension ROM, such as an HP-IL module (since this is actually addressed through blocks 6 and 7), and a Memory module, and an Extended Memory module (since these are addressed separately via the RAM addressing scheme). A Port Extender could therefore let you connect as many as five modules to the HP-41 through just one port.

When you do a CAT 2, the HP-41 looks at each page in turn, starting at page 5, and lists all the functions it finds there if a module is connected to that page. Pages 5, 6 and 7 are looked for first, so that the printer functions will be displayed before the functions in a program module even if the printer is plugged into port 4 and the program module is in port 1. Page 3 was originally intended for a ROM that would work as a language extension ROM together with the system ROMs 0, 1 and 2. Since this page has been used on the HP-41CX to hold the Extended Functions, it was just added to the end of CAT 2. The names and addresses of the additional CX Extended functions are stored on page 3 together with those of the ordinary Extended functions, so the names of all these functions come at the very end of CAT 2. The additional CX Time function names are stored in page 5 with the other Time functions. All Time functions appear at the beginning of CAT 2.

8.4 Peripherals, the Display, and the CPU

Figure 8.6 treats all plug-in ROMs as if they were the same. In fact some plug-in ROMs are attached to peripheral devices such as a Wand, a printer or a Card Reader. The scheme used to address these ROMs is the same as that for other ROMs, but some plug-in devices also have their own RAM memory. The printer has an area to store data it will print, the Wand has two internal buffers to store the barcode it is reading. The Time module has RAM memory to store the current date and time, the time display format, accuracy factor and so on. Each of these pieces of RAM is controlled by its owner and does not fit into the RAM layout of Figure 8.5. Some plug-in ROMs also create temporary buffers inside the HP-41 main RAM memory, in the

Buffer Area shown in Figure 8.4. This RAM area is not owned by the plug-in, it is only borrowed, and is returned to the HP-41 when it is no longer needed or when the plug-in is removed.

The HP-41 display is treated as a peripheral which is always plugged in. Information can be sent to the display and read from it. The display is driven by two chips, the right-hand chip drives the left-hand half of the display and vice-versa. These chips decipher instructions and turn them into display characters. Only 83 of the possible 2↑14 characters are produced by the drivers built into the HP-41; different drivers could be designed to display more characters. In fact, HP-41s manufactured since late 1985 have an updated display that supports a full set of lower case characters. The display drivers are also used to count the 10 minutes before the HP-41 turns off when it is not used. The two drivers must be synchronized, on early HP-41s flat batteries can cause loss of synchronisation which can hang up the HP-41; fresh batteries should solve the problem.

The CPU is the piece of the HP-41 that really does all the work. Each M-code instruction activates part of the electronic circuitry in the CPU, making it move some bits, change some bits, or send some bits to an address. When enough bits have been shunted around, a meaningful result is obtained, for example 1 is added to a byte. Then the CPU picks up the next M-code instruction and does another lot of bit-shuffling (and so on...). The CPU has its own block of registers in which it does most of its work. When two numbers are being added, they are copied into two of the working registers belonging to the CPU (CPU registers are often called **accumulators**) and added. Then the result is rounded (the CPU can do arithmetic with a 13-digit mantissa, but has to round it to 10 digits), normalised (to produce a mantissa whose decimal point lies between the first and second digits on the left, while the power of 10 is contained in the exponent), and finally copied into the HP-41 main RAM memory.

The internal registers of the CPU store additional information such as the M-code program pointer, the M-code subroutine return stack (M-code

functions also call subroutines), and internal flags which are not available to User code programs (such as the PRIVATE mode flag). The CPU also looks after the keyboard. When a key is pressed, the CPU is alerted, and a keycode is stored in a register so that the CPU can deal with it at the right time. The CPU makes the TONEs as well, by setting and clearing a register whose status controls the **bender** (a piece of metal that vibrates to make sounds).

8.5 Program Instructions in RAM

M-code programming, analysis of peripherals, and other tricks require special equipment. Let us come back now to ordinary programming on an HP-41 which is after all what most users do. The layout of individual instructions in a program is worth studying since it gives you the knowledge needed to write shorter programs.

A program is stored as a string of bytes in RAM. Each byte value from 0 to 255 is used in a program as an instruction or part of an instruction. Let us begin by studying the meaning of each byte if it is stored as the first part of an instruction, using a table to organise the bytes. The 256 bytes can be laid out in 16 rows and 16 columns, as in Table 8.1. Each byte is written as two hexadecimal digits, the first digit down the side of the table, the second along the top. Each box in the table represents one byte, and the meaning of that byte in a program is written inside the box, along with the decimal number equivalent to the hexadecimal value. As an example, take byte 6E (row 6, column E); you can see that this is equal to a decimal value of 110 and represents the function RND. As another example, the byte 5A represents the function COS and is equal to the decimal number 90.

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
B	DEG 128	RAD 129	GRAD 130	ENTER↑ 131	STOP 132	RTN 133	BEEP 134	CLA 135	ASHF 136	PSE 137	CLRG 138	AOFF 139	AON 140	OFF 141	PROMPT 142	ADV 143
9	RCL __ 144	STO __ 145	ST+ __ 146	ST- __ 147	ST* __ 148	ST/ __ 149	ISG __ 150	DSE __ 151	VIEW __ 152	BEEP __ 153	ASTO __ 154	ARCL __ 155	FIX _ 156	SCI _ 157	ENG _ 158	TONE _ 159
A	<- - - - - Catalog 2 Functions - - - - ->[1] 160															
B	Spare 176	GTO 00 177	GTO 01 178	GTO 02 179	GTO 03 180	GTO 04 181	GTO 05 182	GTO 06 183	GTO 07 184	GTO 08 185	GTO 09 186	GTO 10 187	GTO 11 188	GTO 12 189	GTO 13 190	GTO 14 191
C	<- - - - - - - - - - Global labels or END - - - - - - - - - ->[2] 192														X<> __ 206	LBL __ 207
D	<- - - - - - - - - - - - - - - GTO__ - - - - - - - - - - - - - - ->[3] 208															
E	<- - - - - - - - - - - - - - - XEQ__ - - - - - - - - - - - - - - ->[4] 224															
F	TEXT 0 240	TEXT 1 241	TEXT 2 242	TEXT 3 243	TEXT 4 244	TEXT 5 245	TEXT 6 246	TEXT 7 247	TEXT 8 248	TEXT 9 249	TEXT 10 250	TEXT 11 251	TEXT 12 252	TEXT 13 253	TEXT 14 254	TEXT 15 255

Table 8.1 Byte Table of HP-41 Prefixes

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00 0	01 1	02 2	03 3	04 4	05 5	06 6	07 7	08 8	09 9	10 10	11 11	12 12	13 13	14 14	15 15
1	16 16	17 17	18 18	19 19	20 20	21 21	22 22	23 23	24 24	25 25	26 26	27 27	28 28	29 29	30 30	31 31
2	32 32	33 33	34 34	35 35	36 36	37 37	38 38	39 39	40 40	41 41	42 42	43 43	44 44	45 45	46 46	47 47
3	48 48	49 49	50 50	51 51	52 52	53 53	54 54	55 55	56 56	57 57	58 58	59 59	60 60	61 61	62 62	63 63
4	64 64	65 65	66 66	67 67	68 68	69 69	70 70	71 71	72 72	73 73	74 74	75 75	76 76	77 77	78 78	79 79
5	80 80	81 81	82 82	83 83	84 84	85 85	86 86	87 87	88 88	89 89	90 90	91 91	92 92	93 93	94 94	95 95
6	96 96	97 97	98 98	99 99	(100) 100	(101) 101	A 102	B 103	C 104	D 105	E 106	F 107	G 108	H 109	I 110	J 111
7	T 112	Z 113	Y 114	X 115	L 116	M [117	N \ 118	O] 119	P ↑ 120	Q _ 121	← ˇ 122	a 123	b 124	c 125	d 126	e 127

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

Table 8.2 Byte Table of HP-41 Postfixes

The table makes most sense if it is studied row by row. Functions that are similar to each other are usually grouped in one row. For example the trigonometric functions are all in row 5, and so are all the evolution (something to the power something) functions. Rows 4, 5, 6 and 7 contain the arithmetical, mathematical and conditional functions. Row 8 contains the functions that implicitly set flags (RAD, AON and so on). It also has the functions that might stop or interrupt program execution (STOP, RTN, PSE, OFF, PROMPT). Thirdly it contains a few functions left over from rows 4, 5, 6 and 7 (ENTER, CLRG, ADV).

Functions in rows 4 to 8 are the simplest kind because they act on their own, without needing anything after them (a register or flag number or other parameter). Each of these functions is represented by one byte in a program, and that's all. The remaining functions can have several parts.

Most are two-byte functions like STO nn or LBL nn or FS? nn. In these, the first byte or **prefix** defines the function. The second byte defines the parameter nn and is called a **postfix**. (Suffix would be better but it is too late to change the name.) XEQ nn and GTO nn are three bytes long because an extra byte is used to hold the compiled distance to the LBL nn. Numbers and text strings can vary in length. Global labels and GTO or XEQ followed by a global label can also have a variable length, depending on the number of letters in the global label. The structure of most multi-byte instructions will be explained in the rest of this section, but some of the more complicated ones will be left until Chapter 15.

A byte which comes second in a two-byte function is interpreted as a postfix. The way bytes from 0 to 255 are used as postfixes can be seen in Table 8.2 which again contains 16 rows and 16 columns, but this time each box holds a parameter value, not a function name. You can see at once that the first half of the table contains the direct postfixes while the second half provides the indirect ones. You may not recognise some of the parameters in this table, they will be explained in Chapter 14. Combining Tables 8.1 and 8.2 lets you interpret any two-byte function. For example

the pair of bytes 90,CF is RCL (row 9, column 0 of Table 8.1 is the prefix RCL) followed by IND 79 (row C, column F of table 8.2 is the postfix IND 79). In other words 90,CF in a program stands for RCL IND 79. You may wonder how the HP-41 knows if a byte is a prefix or a postfix. The operating system has to start at the nearest global label before a byte and count down to the byte, checking line numbers and byte layouts in each step until it comes to this byte. That is why it can be some time after you press the PRGM key before you see the current step displayed.

In order to help programmers save space, some two-byte functions have been given special one-byte forms. RCL nn normally takes two bytes, but RCL 00 to RCL 15 are represented by just one byte each, using the bytes in row 2. STO 00 to STO 15 are represented in the same way, using row 3. These shortened functions are called **short-form** instructions. LBL 00 to LBL 14 are represented by the bytes in row 0. This row can be used for only 15 instructions because byte 00 is the special null byte which replaces any byte that has been deleted. Null bytes in a program do not show up as steps, but they do use up space until they are replaced with a program step or are removed by packing. In parallel with the short-form labels 00 to 14 there are short-form GTOs in row B. They are two bytes long instead of three, which means that they save a byte of space but have less room to store a compiled GTO distance. Byte B0 is one of four **spare bytes** on the HP-41; it is not used as a GTO because byte 00 is not used as a LBL.

The short-form GTO instructions can only store a jump distance of 112 bytes or less. When you understand the whole of the byte table you can work out whether a GTO distance is 112 bytes or less. The short-form GTO distance includes all the bytes from the end of the GTO up to the byte before the target LBL. Consider the two examples below:

GTO 05		LBL 05
SIN	- one byte -	SIN
RCL 22	- two bytes -	RCL 22
+	- one byte -	+
LBL 05		GTO 05

The program on the left jumps over 4 bytes from the end of the GTO 05 to the front of the LBL to reach that LBL. The program on the right requires a jump distance of 7 bytes, since the backwards jump takes in the GTO 05 (2 bytes) and the LBL 05 (1 byte) as well. The details of how jump distances are stored in the short-form GTOs, in the ordinary **long-form** GTOs, and in XEQs (which are all long-form) will be left until Chapters 14 and 15. A short-form GTO which jumps more than 112 bytes remains uncompiled and is therefore slower than a long-form GTO, but it saves two bytes (one on the LBL and one on the GTO). This is typical of the trade-off between speed and space which is sometimes necessary. An understanding of how short-form GTOs work, and how long other instructions are, lets you make the best use of them. A knowledge of synthetic programming lets you use long-form GTOs to jump to short-form LBLs, giving you the best of both.

Back to the tables: byte AF is another spare byte like B0 (it makes a two-byte instruction that is ignored in a running program), the rest of rows 9 and A in Table 8.1 are two-byte functions. Bytes 9C to 9F are the functions that take a one-digit postfix. Bytes 90 to 9B are functions that take a two-digit postfix (00 to 99) or a stack register postfix (X,Y,Z,T or L). All the functions in row 9, and the flag functions in row A (A8 to AD) can take indirect parameters too, using bytes from the second half of Table 8.2. The function X<> nn really belongs in row 9 or A but has been fitted in at the end of row C. LBL nn is also in row C with the global labels, it is really a two-byte instruction but is not quite the same as the others because it takes a different set of postfixes. These can normally be 00 to 99, or the local alpha label postfixes **A** to **J** and **a** to **e**.

Byte CF is the local label instruction whereas bytes C0 to CD are used for global labels and ENDs. The first nybble, C, identifies an instruction that belongs in the global CAT 1 chain. The next three nybbles give the distance back to the previous global instruction. Following these four nybbles (two bytes) there is either one more byte in an END, or there are two more bytes plus the characters that make up the name in a global label.

Once again, details will be given in Chapter 15.

Rows D and E are GTO and XEQ instructions corresponding to the local labels. These instructions can take 00 to 99 and **A** to **J** or **a** to **e** as postfixes and are 3 bytes long. This length lets them store any jump distance up to 513 registers (3591 bytes, more than the largest distance you can ever jump unless you use synthetic methods to jump to a program in Extended Memory). GTO and XEQ of a global label lie in row 1; we shall soon see why. GTO IND nn and XEQ IND nn cannot store a compiled distance, since the jump length depends on the target label, and this can change. GTO IND and XEQ IND are therefore both stored as a two-byte instruction. The first byte is AE, and the second byte gives the indirect address. If this second byte comes from rows 8 to F of Table 8.2, the instruction is interpreted as XEQ IND nn, which is the usual rule for Table 8.2. The two bytes AE,F3 therefore mean XEQ IND X. AE followed by a byte from the first half of the table is interpreted as GTO IND. AE,73 thus stands for GTO IND X. This breaks the rule that only rows 8 to F provide indirect postfixes, but it apparently saves some work inside the HP-41 operating system, although the spare byte AF could equally well have been used for GTO IND.

What exactly does row 1 contain, and why are the global versions of GTO and XEQ included in it? The rest of row 1 contains the number entry functions 0 to 9, radix mark, EEX and NEG. The NEG instruction means "negate the following mantissa or exponent". It is stored as part of a row of digits used to enter a number from a program, and is different from the CHS function (byte 54) which stands on its own and means "change the sign of X", even though both are obtained by pressing the CHS key. If you assign CHS to a key, this key will perform only the CHS operation, not the NEG function, so it cannot be used to change the sign of an exponent. The special feature of bytes 10 to 1C is that they can be combined to create functions without a fixed length (e.g. **1** which is just one byte long, or **-12345678.90E-12** which is 16 bytes long). After you have put one of these bytes into a program you can put in additional ones which become part of the same numeric string. There are a few limitations, of course. The

decimal point will be ignored if it is pressed a second time, or after nine mantissa digits have been entered, or after EEX has been pressed. EEX is ignored if it is pressed a second time, or after nine mantissa digits without a decimal point. (If EEX is pressed on its own then a mantissa of 1 is automatically put in; this wastes a byte and can be avoided as will be described in Chapter 14.) No more than ten mantissa digits and two exponent digits can be entered in one numeric string. Deleting all the digits of a mantissa except for a decimal point or **leading zeroes** (zeroes at the front) causes the HP-41 to cancel a minus sign too, which can give trouble. (Go into PRGM mode and press 0.52 then CHS. Now press backarrow twice. You will see -0.52 then -0.5 and then 0., which means you have lost the minus sign. If you had been trying to put in -0.62 then you must put in the sign again, or you will finish up with 0.62 instead.) The other rules are obvious and can be checked by trying out combinations of keys.

To put two numbers into a program one after another, you must **terminate** numeric entry, then restart it, otherwise the digits all make up one numeric string. If you press 1, then 2, then 3, and then 4 you will get the number 1234. If you press 1, then 2, then press ALPHA twice, then press 3 and 4 you will get the two numbers 12 and 34. Pressing ALPHA terminated numeric entry but how does the HP-41 know later that these are two numbers, not the single number 1234? Normally you would separate the two by pressing ENTER, and this byte would tell the HP-41 that they are separate, but if there is no byte between the numbers then the HP-41 leaves a null between them as a separator. Indeed the HP-41 always puts a null byte in front of a numeric entry, in case the previous step was a number too. This makes sure that two numbers will never join up, but it wastes space. When you pack memory, unnecessary nulls are removed, but if a null is the only byte that separates two numbers then it can not removed, so it is better to put a useful instruction whenever possible between two numbers in a program.

You can see that row 1 contains bytes which are used to build up strings of variable length. The global GTO and XEQ instructions can make up variable length instructions too, so row 1 is a reasonable place to put them. You

press GTO or XEQ, then ALPHA, then any number of text characters from 1 to 7, then ALPHA again. The text characters are stored temporarily in a special register until you press ALPHA a second time, and the same register is used when a number is being entered into a program, so this too makes it reasonable to keep these functions in the same row. When you press GTO or XEQ, the HP-41 stores the bytes D0 or E0, but if you press ALPHA next then these become 1D or 1E and produce the row 1 versions. The label name is stored after these bytes in the form of a text string; these will be described in the next paragraph. Byte 1F, the last in row 1, does not represent any real function. The other spare bytes; AF, B0 and F0 do nothing if they are found in a running program, but byte 1F tries to act like bytes 1D and 1E. In a program it displays as the letter W followed by a text string and it usually does something when it is executed from the keyboard (this requires a synthetic key assignment) or in a program. On the HP-41C and HP-41CV the results depend on the module plugged into port 2 and can be exciting, but generally cause the HP-41 to crash (stop working until one of the methods described in Chapter 4 is used to wake it up again), particularly if port 2 does not contain a ROM module. On the HP-41CX, this function also tries to execute instructions from port 2, but often executes the Extended Function GETP, particularly if port 2 contains no ROM.

The last row of Table 8.1 contains bytes that come at the beginning of a text string. When you put a text string into a program you are putting a string of bytes into a program, not into the Alpha register. The text string may be on its own, or it may be the name of a global label in a LBL or GTO or XEQ instruction. In all these cases the HP-41 must be warned that the bytes are to be treated as text, not as instructions. This warning comes in the form of a byte in front of the text string. The first nybble of the byte is an F, the second nybble gives the number of characters in the string, from 1 to F (a maximum of fifteen characters). For example, the bytes FB, 53, 4F, 4C, 49, 44, 41, 52, 4E, 4F, 53, 43 will be treated as a text string of 12 characters, because FB means that a text string of B (twelve) characters follows. The individual bytes are interpreted and printed as characters according to the ASCII code (see

Section 8.2), but a few are displayed differently (Table 8.3 below shows how characters are displayed, Table 14.1 will show how they are printed). If the first byte were changed to F5, the same line of bytes would be interpreted as:

"SOLID" , - , SQRT , P-R , R-P , Y↑X , /

Since no byte can be larger than FF, no program text line can be more than 15 characters long. The Alpha register can hold a maximum of 24 characters so any extra ones that are needed then have to be added to the first fifteen. A text string with byte 7F as its first character appends the following characters to the contents of the Alpha register (added to the right-hand end). Any other text string replaces the contents of Alpha (clears the present contents before putting in the new text). Byte F0 is interesting; it marks a text string of length zero. As its length is zero it does not put anything into the Alpha register and therefore does not clear it either. So far as HP were concerned this made F0 a spare byte (like AF and B0), but it can be used in programs as an instruction that is one byte long and does nothing, which makes it very useful as a NOP (null operation; see Section 6.8).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Table 8.3 The HP-41 Display Characters

Table 8.3 shows how each byte from 00 to 7F is displayed as a text character. Most characters follow the code known as ASCII, used by many computers. The boxes that are left empty in Table 8.3 correspond to bytes which the display does not recognise. These bytes are all displayed as a boxed star, a character with all fourteen display segments turned on, like the character on the right in the box for byte 3A. All the bytes greater than 7F are also displayed as boxed stars, so they have not been included. The characters 2C, 2E and 3A are special because they turn on the punctuation marks instead of the normal fourteen segment positions. Only three punctuation characters are available, because the display circuits use just two bits to control punctuation. This gives four possible combinations: no punctuation, dot, comma or colon. Under special conditions, bytes 2C, 2E and 3A produce a left-facing goose, a right-facing goose and a boxed star, so they have two characters in each box. The right-facing "goose" is seen during a running program, the boxed star is used for other non-standard characters anyway, but the left-facing goose is extraordinarily difficult to get into the display. One time when the geese would appear is during the creation of a global label. Pressing the dot or comma during entry of a global label name would display the geese although these would turn into a dot or comma once the label was completed. These embarassing displays are avoided because it is impossible to put dots or commas into global labels. Chapter 15 will show a way of getting the geese into the display. All this may seem to be just idle amusement but it helps one to get a better understanding of the HP-41.

We had not yet finished with row A. Bytes A8 to AD are the six flag instructions. The postfix can be any byte from 00 to 99, even though there are only 56 flags, and only 30 of these can be altered by the flag instructions. Steps with larger postfixes can be recorded in a program, but they will give NONEXISTENT when the program tries to execute them. Bytes A0 to A7 are used as the first byte of a two-byte instruction to execute a program or function from a plug-in ROM module or peripheral. For example the Card Reader function RSUB is stored in a program as the two bytes A7,84 and the Wand program WNDTST is stored as A6,C6. When the devices are plugged in, these steps display in a program as the function

name RSUB and as the routine name XROM "WNDTST". When the devices are removed, they show up as XROM 30,04 and XROM 27,06. To convert from the pair of bytes to the XROM number in each case, first write out the hexadecimal byte values and divide them into the nybble A followed by two numbers of 6 bits each. Then write the two six-bit numbers in decimal. (The nybble A followed by a zero bit identifies the instruction as an XROM - eXecute from ROM.)

In this way, bytes	A7 , 84	and	A6 , C6
become	A,0111,1000,0100	and	A,0110,1100,0110
then	A 011110 000100	and	A 011011 000110
and then	XROM 30 , 04	and	XROM 27 , 06

The first number is the ROM number and the second one is the function or program number in the ROM. Therefore WNDTST is function/program number 6 in module 27. (As far as numbering goes, it does not matter whether an entry is a function or a program.) The HP-41 Optical Wand is identified as ROM number 27 regardless of which port it is plugged into.

The ROM number can be anything from 01 to 31 but ROM number 00 does not work as an identifier (a ROM identifier value of 0 indicates to the HP-41 that the port is empty). XROM instructions cannot be followed by a parameter, because they are already two bytes long and do not allow for an extra postfix. It would be possible to follow an ordinary XROM instruction with an XROM 00,nn instruction and use nn as a postfix but no-one has done this yet. ROM numbers larger than 31 would not have a zero at the start of the second nybble, so they would not be recognised as XROM instructions. The function/program number can be anything from 0 to 63, but 0 is usually used as the module name, not as a function or program.

When a program is running and finds a byte A0 to A7, it reads the next byte too, works out the module number, and looks for a module with that number in the 4K ROM blocks (see Section 8.3), beginning at block 5. (On the HP-41CX block 3 is examined separately after all the rest if the XROM has not been found anywhere else.) If the module is found, the HP-41 then looks

for the function/program number within the ROM to find if it exists and to get its address. Once the port and the address have been found, the function is executed using the ROM addressing scheme shown in Section 8.3.

We have almost finished with Table 8.1, but there are still the HP-41 non-programmable functions such as CAT or SIZE. These cannot be put into a program, but they still need a byte number to identify them, for example if they are assigned to a key. (We shall come to key assignments in the next section.) The bytes in row 0 are used for the non-programmable functions as well as for the short-form local labels. Bytes from row 0 are treated as short-form labels or as the null byte when they are found in a program, but they are treated as non-programmable functions when they are assigned to a key and executed from the keyboard. Row 0 in Table 8.1 therefore has two halves. The top half tells you what the HP-41 will do if it finds that byte assigned to a key, and the bottom half tells you how the HP-41 will interpret the same byte if it encounters that byte in a program. Eleven of the bytes are used for recognisable functions, but the other five were meant to be "spare" bytes, which were not supposed to be used by ordinary HP-41 programmers. With Synthetic Programming, or the generalised key-assignment program in Section 11.10, these five bytes can be assigned to keys, and the table shows what they do. Byte 0E lets you assign the SHIFT function to keys other than the yellow key. Byte 0B acts like the PRGM mode backarrow key, deleting the current program line, but it does this whenever it is pushed, even outside PRGM mode. Byte 0C acts like one of the toggle keys USER, PRGM or ALPHA; which one depends on the keyboard row used. These three bytes can sometimes be useful if assigned to a key (particularly 0C which you can assign to some other key if one of the toggle keys stops working), but the others are of less use. Byte 0D acts just like the W function (byte 1F), but is not programmable. Finally, byte 01 is displayed as @c and sometimes acts like GTO.. while at other times it does nothing.

By now you should have a good idea of how a program is arranged in the HP-41 memory. In order that the current byte be interpreted correctly as a prefix or a postfix the HP-41 needs to go to the beginning of the program,

or to the nearest previous global label, and count off bytes as one-byte instructions or as instructions made up of several bytes. When you press R/S to start a program running, the HP-41 assumes that the program pointer is pointing correctly to the first byte of an instruction and it then executes instructions one by one, interpreting each byte as an instruction on its own or as part of a multi-byte instruction.

You should be able to make some of your programs shorter with the help of Table 8.1. For example it will remind you which registers (00 to 15) can be accessed with one-byte instructions, saving a byte every time one of them is used in preference to registers numbered above 15. If you want to divide a number by 100, you may think that 100, / and 1, % are both two lines long, but Table 8.1 will show you that the first takes four bytes while the second uses only two bytes. The second version is therefore shorter (but it leaves the Y register unchanged unlike the first version). Whenever possible you should use the short-form labels, GTO, RCL, and STO instructions to save bytes. Section 8.8 at the end of this chapter will give some tips for saving space.

8.6 Key Assignments of Instructions

When an HP-41 instruction is put into a program it is stored in the RAM memory. An instruction that is assigned to a key has to be stored as well but in a different way. Key assignment information is stored in three different places in the HP-41. First of all, there are two status registers which contain key assignment flags. One register contains a flag for each unshifted key. A flag is set if the corresponding key is assigned, and clear otherwise. The second register contains flags for each shifted key. Just like the 56 system flags in the ordinary flag register, each key assignment flag is stored as one bit.

If you press a key while the HP-41 is in USER mode, the operating system first checks the corresponding key assignment flag. If this flag is clear then there is no need to look further. The HP-41 executes the normal function (unless the key is in one of the top two rows when the HP-41 must

first check for a corresponding local Alpha label). If the flag is set then the HP-41 looks for the corresponding key assignment in a second place; the Key Assignment Registers (shown in Figure 8.4). Any assignment of a function or program from CAT 2 or CAT 3 will be recorded there. If no CAT 2 or CAT 3 assignment is found, the HP-41 operating system knows that the key must have been assigned to a global label in CAT 1. The information that a global label has been assigned to a particular key is stored as part of that label itself. The HP-41 therefore goes through the global chain until it finds a label assigned to the corresponding key. Once the assigned instruction has been found it is displayed, or executed, or stored in a program.

If you assign a global label to a key, the assignment information becomes part of the program. When you make a copy of that program on a card, or in Extended Memory, or on an HP-IL device, the assignment is saved together with the rest of the program. You can read the program back later into the same HP-41, or a different one, and the key assignments will be re-established automatically if USER mode is set when you read the program back. This gives you a choice; you can make a new set of key assignments and keep them when you read the program back, or you can read back the old assignments used by that particular program. Should you want to recreate the old assignments, you will clearly lose any new assignments you have made to the same keys. CAT 2 and CAT 3 assignments are not stored as part of a program, so they are not recorded with a program, and they are not affected when a program is read back, except that they are replaced if the program has a global label assigned to the same key. When a program is read back with USER mode set, the key assignment flags have to be reset, and the HP-41 takes advantage of this to recheck all key assignments and reset all the key flags. This means that you can accidentally (or intentionally) clear all the key assignment flags, then restore them by reading a program with USER mode set. Furthermore, it turns out that this restoration of the key assignment flags occurs even if USER mode is not set. This peculiar fact was discovered by Clifford Stern.

While the key assignment flags are clear, the HP-41 will not use any key

assignments but will immediately execute the default key function (or the corresponding local Alpha label). Chapter 14 will show how you can clear the key assignment flags so as to use the default key functions, then recover the assignments by reading in a program. Your CAT 2 and CAT 3 key assignments can also be saved on magnetic cards or on HP-IL media by means of the WSTS and WRTK functions. The CCD Module (see Chapter 12) lets you save your key assignments in Extended Memory too, or you can use a Synthetic program to do the same thing.

Every assignment from CAT 2 or CAT 3 has to be stored together with a code for the assigned key. The CAT 1 assignments are stored within the assigned global labels, but the CAT 2 and CAT 3 assignments are stored in a separate area; the Key Assignment Area shown in Figure 8.4. The CAT 2 instructions are stored in exactly the same way as in program memory; two bytes each (for example A7,84 to store RSUB as described in the previous section). The key code is stored as one byte following the instruction. This means that each key assignment takes up a maximum of three bytes. Since a register has seven bytes, the HP-41 can store two key assignments in a register, with one byte left over. This byte is used to identify the whole register as a **Key Assignment Register (KAR)**. CAT 3 instructions are all stored as a prefix (or a single byte) only, using just one byte. This is the same byte as was shown in Table 8.1. A "filler" byte is put before the instruction byte, this is byte 04 used to fill the other of the two bytes to record the function. The keycode is again stored after the function. Figure 8.7 shows a KAR containing the assignments of INT and RSUB to the "-" key and the shifted "-" key.

Byte number	6	5	4	3	2	1	0
Byte contents	F0	04	68	05	A7	84	0D

Figure 8.7 Layout of a Key Assignment Register

The contents of this register are shown in hexadecimal and are interpreted as follows. Byte 6 contains F0 to identify the register as a KAR. Every KAR has F0 in this byte. F0 was considered a spare byte and HP did not

intend it to be used in programs, but it does serve the purpose of identifying KARs. Bytes 5 to 3 contain one key assignment, and bytes 2 to 0 contain the other assignment. Byte 4 is the INT function (see Table 8.1). INT is a CAT 3 function and only uses one byte, so byte 5 contains 04 as a filler. Byte 3 contains 05 which identifies the - key (see below). Bytes 2 and 1 contain the two-byte identification of the ROM function RSUB, already mentioned in the previous section. Byte 0 contains 0D which identifies the shifted - key. If the assignment in bytes 5 to 3 had been of a two-byte function from CAT 3, byte 4 would have contained the prefix, and byte 5 would still have contained the filler 04. For example the assignment of STO would be achieved by putting 91 in byte 4. If bytes 2 to 0 were to contain a CAT 3 assignment then byte 2 would also contain the filler 04.

When you first make a key assignment using ASN, the assignment shows up in the display as for example ASN INT 51. INT is the name of the function, and 51 is a keycode in which the first digit is the row and the second digit is the column of the key on the keyboard (all of which is explained in the HP manuals). The hexadecimal code used in the KAR to identify the key is derived from this keycode. The first hexadecimal digit is the column number minus one. The second hexadecimal digit is the row number. If the assignment is of a shifted key then 8 is added to the second digit. Looking back at Figure 8.7 you can see that the - key is in the first column so the first digit is 0, and it is in the fifth row so its second digit is 5. The shifted key is identified by adding 8 to the second digit, giving 0D in hexadecimal.

Three additional explanations are needed for this scheme. First of all the bottom row is row 8, and adding another 8 for the shifted keys means that the second digit becomes 0, and 1 is carried and added to the first digit (this is hexadecimal arithmetic). The four shifted keys in the bottom row are therefore identified as 10, 20, 30 and 40. Secondly the three top rows of the HP-41 keyboard have five keys each, whereas the lower four rows contain four keys each. In both cases the keys are numbered sequentially, starting from 0 for the leftmost key. The last key in the top row is thus

41 and the last key in the bottom row is 38. Thirdly, the fourth row is treated as if there were a hidden key under the right-hand half of the ENTER key, so the hexadecimal keycodes for this row are 04, 24, 34, 44.

The first CAT 2 or CAT 3 key assignment that you make after MEMORY LOST is put in bytes 2 to 0 of absolute register 0C0. (This is the first KAR; see Figures 8.4 and 8.5.) At the same time, byte 6 of register 0C0 is set to F0 and a bit is set in one of the two key flag registers. The next key assignment goes into bytes 5 to 3. When you make a third key assignment, register 0C0 is copied into register 0C1, then register 0C0 is cleared, the assignment is put in bytes 2 to 0, and F0 is put in byte 6. This is repeated for further assignments, with all the KARs being moved up one place at every second assignment. If you assign the same CAT 2 or CAT 3 instruction to two different keys, two different assignments are stored. (You cannot assign a CAT 1 label to two different keys because only a single byte is available in the global label to store the assigned key code. You could create one global label and assign it to a key, then create a second global label with the same name lower down in CAT 1, and assign that label to a different key.) If you have the same name in CAT 1, CAT 2 and CAT 3, the CAT 1 label will be assigned, since the catalogues are searched in the order 1,2,3 at the time the assignment is made. You can assign several different functions with the same name to different keys by first assigning the CAT 3 function, then plugging in a ROM and assigning the CAT 2 function with the same name to a different key, and finally creating a global label with the same name in CAT 1 and assigning it to yet another key.

When you cancel a CAT 2 or CAT 3 assignment (by assigning a blank Alpha value to the key), the keycode (byte 3 or byte 0) is set to zero, but the rest of the KAR is unchanged: the unused space is not returned to you. Only if you delete both the assignments in one KAR, then PACK or GTO.. will you recover the unused register. Should you want to get a register back by cancelling assignments, you must always make key assignments in pairs, and keep a record of the pairings so that you can cancel the same pairs and recover the registers. The space left in a KAR will be reused however when

another assignment is made, so it will not be wasted if you cancel one assignment then make another one.

It is obviously better to cancel any unnecessary assignments before you make some new ones. If you make a new assignment from CAT 2 or CAT 3 to a key that already carries a CAT 2 or CAT 3 assignment then two things happen. First the old assignment is cancelled. Then the new assignment is put into the first unused KAR space available, starting from register 0C0. This may or may not be the same space as was freed by the cancelling of the previous assignment, but in any case the new assignment will not take up any additional space.

The inability of the HP-41 to recover unused KAR spaces, except to use them for new assignments, can be a considerable nuisance to people who are short of HP-41 memory space. The PPC ROM (described in Chapter 12) contains a special routine which packs the KARs and recovers this space.

If you press an unassigned shifted or unshifted key from the top row, or an unshifted key from the second row, and you are in USER mode, you may have to wait a while before that key's function is executed. The HP-41 first checks the assignment flags, and if it finds none it then looks for a corresponding local label. (This can take over a second if you are in a long program.) Only then does it carry out the normal key function. If you have the space available you can assign the key's own function to itself. This avoids the search for a local label and so speeds up execution of the function. Especially helpful is assignment of X<>Y and RDN to their standard keys.

A few more words are needed to explain how the CAT 3 functions are stored in a KAR. Going through Table 8.1, you may recall that the row 0 bytes represent both the non-programmable functions and the short-form local labels. Obviously the non-programmable functions are the ones which will be used for the purpose of key assignments. Rows 1 to 3 can only be assigned by Synthetic methods. Rows 4 to 8 are the one-byte functions, and are assigned as a single byte with a 04 byte before it. Row 9 and bytes A8

to AD are assigned as a prefix only. The first half of row A makes up the XROM functions already described. Rows B to F can only be assigned by Synthetic means, except that END is stored as byte C0, and bytes CE and CF are stored as one-byte prefixes.

This section has shown how an understanding of the HP-41 can help to improve its use. You have seen how the user can avoid wasting space in Key Assignment Registers, and how execution of functions on the top two rows of keys can be speeded up. The last paragraph will show another use for your understanding of the KARs. All this should show that studying how the HP-41 works is not just a matter of idle curiosity but also has practical uses.

Figure 8.7 showed that a KAR can hold a two-byte ROM instruction as well as a one-byte CAT 3 instruction. Soon after the introduction of the HP-41, Richard Nelson (founder of the original user group PPC) suggested that two-byte CAT 3 instructions could perhaps also be stored in a KAR. Instead of having a filler (04) and a prefix, why not put a prefix and a postfix into the two available bytes? It took a few months, but it worked! Not only can you assign RCL to a key, you can also assign RCL 55 to a key if you like. This may not seem very useful, but it does save a couple of keystrokes. Assigning GTO IND X to an unshifted key saves four keystrokes (SHIFT, GTO, SHIFT, . , 6 is replaced by one keystroke), it is achieved by storing the bytes AE, F3 in a KAR. Byte combinations which cannot normally be put in a program can also be put in a KAR, for example FIX 20 can be assigned, and turns out to be very useful. The ASN function is not designed to do this, but a general-purpose key assignment program to make such assignments will be presented in Chapter 11. You can also use Synthetic Programming techniques to make such assignments, but you may prefer the program in Chapter 11 because it does not use any synthetic instructions.

8.7 41C or 41CV ?

The answer to the above question is still likely to be trite, but the information in this chapter will help to quantify even a trite answer. If

you already have a 41 of one kind or another, the question is philosophical anyway, and the next section will be of more interest.

The 64 data registers of the 41C on its own are enough to let you store 20 data values, 10 key assignments, two 20-line programs, and a longer 60-line program. If you are sufficiently ruthless to delete programs and data as soon as you have finished with them then this will be enough. Unfortunately the Continuous Memory makes you want to hang on to old programs "just in case", and you will soon be buying additional memory modules, then a quad memory module. There is a large supply of spare memory modules among HP-41C owners, and you will find it very difficult to get much money back for them. If you are at all likely to expand your use of an HP-41, better buy an HP-41CV directly, save money in the long run, and leave yourself with four free I/O ports. Better still read the next three chapters and decide whether or not an HP-41CX might be the best buy.

A few cases will still arise where an HP-41C is clearly the best choice. If the HP-41 is to be used strictly for one task only, it is better to use an HP-41C and avoid the temptation to put in additional programs or data. If you want to indulge in page-switching of memory modules to increase the HP-41 memory (see the PPC ROM Manual) you must use an HP-41C unless you are willing to pull apart a CV or a CX. If you really are short of money, you could buy a second-hand 41C from someone who is getting a more expensive model, then buy a second-hand quad memory module. Take this second-hand 41 to a user group where someone will be able to tell you if it is in working order before you finally pay for it. You could even go to a user group meeting and ask if someone has a second-hand HP-41 for sale. Some HP-41's have had quad memory modules wired inside them; perhaps you could get one of these or have your own module wired in, to get the equivalent of a CV.

8.8 Space Saving Tips

Advice to buy an HP-41CV will be of little help to anyone who already has an HP-41C. More to the point is some advice on how to fit more data and

programs into an HP-41C with restricted memory, and this advice will be of help as well to HP-41CV and HP-41CX users who are short of memory space. Here then to finish this chapter are some memory-saving tips. You will find more tips in later chapters, but most of the ones given here need no extra knowledge or equipment.

A. In programs

1. Make text strings as short as possible. Use abbreviations if you can.

2. Make global labels short. This saves space in the labels, but also in GTO or XEQ instructions which refer to these labels. You can also access the programs from the keyboard more quickly.

3. Use subroutines whenever a sequence of several instructions is used in more than one place. If a repeated sequence contains **S** bytes and is used in **N** different places then the use of a subroutine will save bytes as follows:

   Bytes saved using a subroutine called by XEQ nn = S*N - ((S+2) + 3*N)

   Bytes saved using a subroutine with XEQ IND nn = S*N - ((S+2)+2*N+7)

   The 7 in the second equation is the seven bytes of the register which holds the indirect value. If you use a free stack register then these bytes do not count as a separate register wasted, and you can remove this 7 from the equation. These formulae were first given, together with tables of bytes saved by direct and indirect execution, in an article by Richard Nelson in PPCCJ V6N6p31. Tables like this are also provided in the "HP-41 Synthetic Quick Reference Guide"; see Appendix A. The tables are not given here as you can work out the numbers from the formulae.

4. Use the short-form labels, GTOs, RCL and STO instructions whenever possible. Table 8.1 shows these instructions, each one saves a byte over the normal versions.

5. This is a particular example of 4: if you want to clear register nn without disturbing the stack then X<>nn, CLX, X<>nn will do it in just five bytes, but if the register is number 15 or less and if X contains a number (not a text string) then STO nn, ST- nn will do the same thing in 3 bytes.

6. Look through Table 8.1 to see where else you can change instructions to make a program shorter. Some functions have very useful side-effects. For example ABS or SIGN can be used to save a value from register X to register L and both are one byte shorter than STO L. (SIGN works for text strings, so it is best if X can contain a number or a text string. ABS will produce an error message if X contains a text string, so it can be used to save numbers only and to reject text strings. Both are fast and do not affect Y, Z or T.) Σ+ can be used to add 1 to a register (the last register in the statistics block), and Σ- can be used to subtract 1 from the same register.

7. A good example of using shorter instructions lies in the use of flags. SF nn and CF nn use two bytes each, but AON and AOFF set and clear flag 48 and use only one byte each. You can therefore use combinations of AON, AOFF, FS? 48 and FC? 48 to save several bytes in a program that sets, clears and tests one flag. AON and AOFF are also faster than SF nn and CF nn, but you will not be able to use FS?C 48 or FC?C 48. Obviously you have to be careful if you also want to use AON and AOFF for their real purpose of setting or clearing ALPHA mode; in any case you should include a note in the program documentation to remind yourself why you are using AON and AOFF.

   Furthermore you can set or clear flag 27 by pressing the USER key during a PSE, and you can even set or clear flag 47 (the SHIFT flag) by pressing SHIFT during a PSE. You can test both these flags, and

you can clear flag 47 with CLD (one byte long, shorter than CF 27). Executing CLOCK will clear all of the flags 11 to 20 in just two bytes if you have a Time module, but not on an HP-41CX. The Extended Function X<>F also lets you change several flags at once.

If you want to save some bytes while using a pair of flags, you can use the angle mode flags unless you are actually using the trigonometry functions. GRAD sets flag 42 and clears flag 43. RAD sets flag 43 and clears flag 42. DEG clears both flag 42 and flag 43. All three are one-byte functions. Even more exciting, you will be able to use synthetic functions like FIX 20 to set flag 43, then 42, then both 43 and 42, then clear both, in a cycle, using only 2 bytes. If you want to go further with this sort of thing to use combinations of flags as binary counters you can use FIX, SCI or ENG with direct or indirect values to set or clear combinations of flags 36-41. See Appendix D for more details of flags.

8. Despite the remarks above it is often quicker to avoid flags altogether and to use the one-byte comparison functions X--Y? or X--0?. The tests comparing X with zero can be used in combination with the one-byte functions ABS, CHS, CLX to set a value in X at one place and check it somewhere else in a program. If you want to select one of three possible choices, you can store 1 in X, use this for the first choice, then use CHS for the second choice, and use CLX for the third choice. Later on you can find which choice was made; the first choice will be identified with X>0? or the second choice with X<0? or the third choice with X=0?. You can even identify or reject combinations of choices by using X<=Y? and X≠0?

9. You can use ISG or DSE to increment or decrement a number without making a test. Simply follow ISG or DSE with a NOP (see Section 6.8). Save space by using a one-byte NOP such as LBL 00, or by putting the ISG or DSE as the last instruction before an END (since an END cannot be skipped so a NOP is unnecessary).

10. Some functions are so rarely used that you forget about them. For example CLST is not much used, but it will clear X and Y in one byte whereas 0, ENTER or CLX, ENTER take two bytes. Check if a function like this can make your program shorter.

11. Avoid unnecessary instructions. Do not use CLA if it is followed by a text string which will clear the Alpha register anyway. Do not add ten values together, then divide by 10, if you can use Σ+ followed by MEAN to do the same thing and save space. Remember to do CLΣ first. If you are taking the mean of the X values only, then use CLST to make sure Y is empty while you are using Σ+.

    Two numbers, one after the other have to be separated by some instruction, otherwise a null is put between them and wastes a byte. Wherever possible, put a useful instruction between two numbers. Use 72, DEG, 12 instead of 72, 12, DEG to save a byte for instance.

12. Keep numeric entries as short as possible. Never put zeroes at the front or end of a number. Remember that exponents can be one digit long, they do not need a leading zero. 60E03 can be replaced by 6E4 saving two bytes (and 66 milliseconds). Avoid decimal points too. 6.0E4 can obviously be replaced by 6E4, but 2.997E8 can also be replaced by 2997E5. Even better, 5.74E11 can be replaced by 574E9, saving two bytes.

13. In some cases you can use functions to make number entry even shorter. 1,000,000 takes seven bytes, 1E6 takes three bytes, but 6,10↑X only takes two bytes. Again 180 takes three bytes but PI, R-D takes only two bytes.

14. If you have checked through all the above and you are still short of space, try rearranging parts of your program. You might be able to move a subroutine so that a program falls straight into it instead of calling it. You might be able to change the order of some operations so that numbers you want are still in the stack and do not need to be

re-entered or recalled. If you must use the same long number in several places, try to store it in a register and recall it. This uses 7 bytes for the register and two bytes for the STO and for each RCL instruction (unless you use a register number below 16). If the number entry is over 11 bytes long then using STO and RCL saves space even if the number is used in only two places.

You might also be able to rewrite a program that included various tests while you were writing it and checking it. Once the program works you should be able to remove lines that were only used to check that the program was OK, and you may be able to rearrange the program too.

15. See if a piece of program that you are using is duplicated in another program or in a plug-in module. You may be able to call that duplicate piece of program as a subroutine and delete the same piece from the program that is too long.

16. If all else fails, and you have not yet learned about the tricks of Synthetic Programming, sleep on the problem. If even that fails, look for someone who uses an HP-25 and ask for advice. HP-25 programmers have only 49 program steps at their disposal so they have long practice in writing short RPN programs. They will offer you their sympathy if nothing else.

B. Key Assignments

17. Remember that every second key assignment from CAT 2 or CAT 3 takes up another register. Do not waste space by making unnecessary assignments.

18. Cancel unwanted key assignments before you make any new assignments. The space freed by the cancelled assignments will be used for the new ones. If you make the new assignments first, and later cancel the old ones, then the space from the cancelled assignments will be wasted.

19. If you plan to make several assignments in a group, and then delete them in a group, try to make all the assignments in pairs. If two assignments are made together and fit into the same register, then they can both be deleted later at the same time, and the register can be recovered by PACKing. To see if two assignments are being put into the same register, go to the .END. and go into PRGM mode. You will see .END. REG nnn. Make the first assignment; you can stay in PRGM mode since ASN is not programmable. If the number nn goes down by one after the first assignment then the new assignment has taken up half a new register, and the second assignment will fit in the same register. If in doubt, reread Section 8.6.

C. Use of data registers

20. Make SIZE as small as possible. This means you should rewrite your programs to use as few data registers as possible. Sometimes a program uses one data register to store one value which is not needed later, and then the same program uses a different register to store a different value. See if you can avoid this; try to make your program use the same register several times for different purposes. This will free other registers. If you do this, you must make sure that you have a proper description of what the program does. Otherwise you might come back to the same program later and make some changes without realising that certain registers are used for more than one purpose.

21. Use the stack registers, including register L as much as possible. This will reduce your need to store values in numbered data registers. (Using the stack registers is also faster than using numbered registers.) If you want to use the value in X more than once, use ENTER to store it in Y instead of using STO nn. You can often keep a value in L unchanged by using arithmetic operations which change X but not L. For example ST*X does the same as X↑2 but does not change L. The first instruction is one byte longer but you save seven bytes that

would otherwise have to be used for a data register. In the same way ST+X has the same effect as 2, * but it leaves registers T and L unchanged. Remember as well that CHS does not alter L.

22. You can often store two separate numbers in the same register as the integer and the fraction part of a single number (the fraction part has to be positive unless you take special precautions). You can even store a number between -99 and +99 in the exponent part of a register if you know that the number in that register is larger than 1 but smaller than 10. This can let you store three numbers in a register, one signed value with N digits, one positive number with 10-N digits and a signed number with two digits.

This scheme can be extended much further by **data packing** techniques. You can for example store five positive numbers of two digits each in a single data register. Multiply the first number by 100, add the second number, multiply by 100 again, add the third number and so on. To get the numbers back use 100,MOD or divide by 100 repeatedly; take the fractional part each time, then take the integer part and divide by 100 again (save some more space by putting 1 in the Y register and using % repeatedly).

Data packing need not be restricted to numbers which are smaller than a selected power of 10. You can use any maximum value and pack several values by repeated multiplication. For example you can store three byte values each of which is in the range 0 to 255. Multiply the first by 256, add the second, multiply by 256 and add the third. You will not be able to fit in a fourth byte unless you use the exponent too. The rule is that you must always multiply by a number that is one bigger than the largest number you wish to store. Obviously this whole approach is only worth taking if more space is saved by packing than is used by the data packing and unpacking routines. If you have the packing and unpacking routines in a plug-in program module then they will not take up any space in your program memory. This is why the PPC ROM (see Chapter 12) contains packing and unpacking routines.

D. Other tricks will be mentioned later in the book. Synthetic Programming is a prime example of space-saving. It lets you use the Alpha register as four spare data registers. Multi-purpose key assignments can be made so that one key can be used instead of several. New short instructions can be used instead of longer ones. GTOs can be used to jump anywhere into a program, without the need for labels, and you can jump to any line of a program in a plug-in module. Let's wait till you reach Chapter 14 though.

The Time Module and Extended Functions Module also provide various places where data can be stored. These will be discussed in the next three chapters and at the end of Section 16.6.

Exercises

8.A How would an HP-41 interpret each of the following byte strings stored in the X register?

93,14,15,92,65,40,00	01,20,00,00,00,00,34	00,01,20,00,00,00,35
10,00,00,45,41,53,59	20,00,00,48,41,52,44	00,00,00,00,00,00,00

Which of these are Non Normalised Numbers?

8.B What would the following bytes mean if they were in a program?

00,19,10,41,67,B2,00

8.C How would the HP-41 understand the following bytes if they were stored at absolute address 0C0?

F0,00,00,00,04,69,32

CHAPTER 9 - TIME FUNCTIONS

9.1 A Growing System.

On their own the HP-41 and HP-41CV are advanced calculators which can do some types of work that are usually done on a computer. Plug-in memory and program modules let them store more information and run longer programs, and we shall see that Synthetic Programming allows a user to have greater control over their computer-like features. A real computer though is a whole **system** which not only does calculations but can obtain data from various sources, including instruments, and store it. After doing calculations on this data, a computer can present its results on a display, print the results, store them, or send them somewhere, for example to another computer. Some of these operations have to be carried out at a particular time, when an instrument has its results ready, or when another computer is waiting to receive a set of results. In the next four chapters you will see how you can extend the HP-41 by using it as part of a system with a clock and extensive data handling features.

The HP-41 was always designed to be used as part of a system, not just a calculator. At first the **HP-41 system** was made up of the HP-41 itself, a printer, a card reader and then an optical barcode reader (wand). The introduction of the HP-IL module, the Time module, and the Extended Functions and Extended Memory modules was a major advance. The HP-IL module allows the HP-41 to interact with many devices including computers. The Time module lets the HP-41 turn on at chosen times and execute programmed operations such as taking a set of measurements. The Extended Functions/Extended Memory module allows the HP-41 to use its Alpha register as an Input/Output area for sending or receiving data and commands. Extended Memory can be used to store and transmit data or programs in complete blocks called files. All these features are more at home on a computer than on a calculator.

All this helps to explain why so many users call their HP-41 a computer. It is the reason why separate chapters have been devoted here to the Time

module, to the Extended Functions, and to other HP-41 system devices. Independent manufacturers make HP-IL compatible devices in the knowledge that HP-41 owners will be able to use them without having to buy additional control equipment. Yet if a faster or more powerful controller is required at a later date, the HP-41 can be replaced with an HP-71, an HP-75, or other more powerful computers. Modules such as the Extended I/O module, provide features in addition to those of the HP-IL module; these will be covered in Chapter 12 and mentioned again in Chapter 16.

HP-IL therefore provides a growing system of devices to be used with your HP-41. The Time and Extended Functions/Extended Memory modules expand the HP-41 as well as enhancing the whole HP-41 system. As these two modules are built into the HP-41CX they can be considered to be parts of the HP-41 itself, so I shall describe them first. The Time module provides a clock, a calendar and a stopwatch built into the HP-41, together with facilities for setting alarms. There are also functions for setting, calculating, and displaying times and dates.

The HP-41CX has all the features of the Time module and the Extended Functions module built in, except for some of the bugs. It also has a number of additional Time and Extended functions. These additional HP-41CX functions will be described at the end of each of the chapters on the Time functions, the Extended Functions, and the Extended Memory functions. The Time module manual is so good that it needs little if any addition. The Extended Functions manual by contrast is inadequate; two successful independent books have been written on the subject of these functions. The HP-41CX manuals cover them very well. The next three chapters are not intended simply to repeat the information in the manuals for the Time and Extended functions, they concentrate on how some of them can be used in practice and describe features not covered in the manuals.

9.2 Times and Dates.

The first thing to know about a clock or a calendar is how it displays the time or date. I shall begin by describing how the Time module uses the X

register to read or write times and dates and how it displays them. The Time module date functions start off by reading dates from X and writing them to X in the form MM.DDYYYY. The function **DMY** sets flag 31 and thereby tells the HP-41 to treat the contents of X as DD.MMYYYY. The function **MDY** clears flag 31 and goes back to MM.DDYYYY. Dates are displayed in two corresponding forms - either MM/DD/YYYY or DD.MM.YYYY - and when the display is set to show fewer than six digits after the decimal point then only the last two digits of the year are shown. Days of the week are shown as a three-letter abbreviation, but they are put into X as a number from 0 to 6, with 0 being Sunday. (This follows the ancient Jewish and Christian tradition of treating Sunday as the first day of the week, unfortunately it also follows the ancient HP tradition of using 0 instead of 1 as the first number of a set - it would be easier to treat Sunday as day 1.)

Times are initially displayed on a 12 hour clock, and are followed by the letters AM or PM, but the function **CLK24** sets a flag inside the Time module so that times are displayed on a 24 hour clock. **CLK12** switches back to a 12 hour clock. In addition, **CLKTD** sets another flag so that times are displayed as hours and minutes plus a date; **CLKT** switches back to an HH:MM:SS display. Regardless of the display format, times are always put into X by the Time module as HH.MMSSss on a 24 hour clock, ss being hundredths of a second. The module reads times <u>from</u> X such that values whose integer part is between 0 and 11 are treated as AM, and values with an integer part between -1 and -11 are treated as PM. Values with an integer part from 12 to 23 are also treated as PM.

Two small grumbles so far. X values that are negative but whose integer part is zero are interpreted as AM, not as PM. Better use a 24 hour clock with all times positive to avoid mistakes that can be caused by this. The function **DOW** puts a day number into X, and it also puts a three-letter abbreviation into the display, but not while a program is running, and these letters cannot be put into the Alpha register. It seems pretty silly to require an HP-41 program to interpret a number as a day; the information is already in the Time module. The functions **ATIME** and **ADATE** will convert times and dates in the X register into Alpha values, so it would have been

possible to provide an ADOW as well. Maybe HP preferred to let users write this function in their own languages: times and dates are international, but days of the week have local names. See Exercise 9.A.

Otherwise everything is very nice and you now know how to put times and dates into X or read them back from it. To set a date, you put it into X following the above rules and making sure that it is in the format you have chosen; **MDY** or **DMY**. Then execute the function **SETDATE**. To set the time, put a time into X then execute **SETIME** at that exact time. It is easier to be accurate if you assign SETIME to a key (clear the assignment afterwards or you will reset the clock accidentally). Remember that the function is spelled with one T, not two (this saves one letter but is awkward, on the HP-71 it is spelled more logically: SETTIME). There are two more time setting functions. **T+X** will change the time by the value in X, **CORRECT** will set a time in the same way as SETIME, but first it will check the previous setting of the clock and use the difference to work out a timing correction factor and store this. **RCLAF** lets you recall this **accuracy factor** to X and **SETAF** lets you set it manually (you may need to do this after changing the batteries). Do not forget that the HP-41 also has the built-in functions **HR**, **HMS**, **HMS+** and **HMS-** which can be used together with the Time module functions.

To recall the date or time to the X register, execute **DATE** or **TIME**. If executed from the keyboard, these functions not only put a number into X but also set the display to show a more readable form of the date or time. If a program is running then these more readable versions will not be displayed, but the functions ADATE, ATIME and also **ATIME24** can be used to turn the number in X into this form and append it to Alpha. As mentioned above, DOW converts a date in X into a day of the week, and displays the day in text form if a program is not running.

The function **CLOCK** displays a running clock without altering the value in register X. The clock will also be displayed if you press **SHIFT**, **ON**. How does this happen? When you press ON, the HP-41 prepares to turn itself off. Before going to sleep though it checks if any plugged-in module needs

something to be done. At this point the Time module breaks in, checks if the SHIFT flag is set, and if so then it wakes the HP-41 up again and displays the clock. Waking the HP-41 up is like turning it on; it clears flags 12 to 20, sets flag 26 (audio enable), clears flag 44 (the continuous ON flag), then sets the BAT flag if need be and sets flags 21 and 55 if a printer is connected. This clock display process seems to have been invented after the HP-41 had been designed. It is a very smart trick to use SHIFT, ON, but it can lead to problems with the flags (these problems will be described in Appendix C). The original Time module did this same procedure when the CLOCK function was executed, but the later version used in the HP-41CX will not turn the HP-41 off, so the flags will not be reset. The time display uses a lot of electricity unless the clock display is in **CLKTD** mode, in which case the display shows hours and minutes but not seconds and therefore changes only once a minute. To avoid wasting the batteries, the Time module cancels the clock display as soon as any key is pressed. There are just two exceptions; if either the SHIFT or the USER key are pressed and kept down for more than about a second, and then released, the clock will continue to be displayed. If it is not being displayed, and even when the HP-41 is off, the clock continues to run (and the date and stopwatch also continue to be updated).

In addition to these time and date functions which could be expected on a clock, the Time module has calculation functions as expected on a calculator. In addition to the time calculation functions described earlier there are two date calculation functions. **DATE+** adds a number of days in X to a date in Y and leaves the result in X, in date format. (This is like the built-in functions **HMS+** and **HMS-**, but days can only be subtracted by the trick of adding a negative number.) If two different dates are put in X and Y, **DDAYS** subtracts Y from X, drops the stack, and puts the result into X as a number of days. A negative result means that the date in Y came after the date in X. Be careful: this function works in the opposite way to normal RPN operations where you would expect X to be subtracted from Y. It is assumed instead that you will enter the earlier date first (which is fairly natural), and then the later date, so they will be in the order Y,X not X,Y.

The clock/calendar will only run for dates between Jan 1 1900 and Dec 31 2199 A.D. The date calculation functions however will work for dates from Oct 15 1582 to Sept 10 4320 A.D. All dates are according to the Gregorian calendar. (Oct 15 1582 was the day when the Julian calendar was replaced by the Gregorian calendar, although the British Empire, including the American Colonies, did not start using it until Sept 14 1752.) Remember that on this calendar century years (1900, 2000, 2100 ...) are leap years only if they can be divided exactly by 400. If you find the rule difficult to remember, this is rather like other years which are leap only if they are exactly divisible by 4. To use dates before or after this range (for example in astronomy) you may like to buy the PPC ROM (see Chapter 12) which has two calendar functions for all dates after March 1 of 1 B.C. and whose manual provides a routine for earlier dates and an article on dates.

9.3 Using the Stopwatch

The Time module provides the HP-41 with sophisticated stopwatch facilities. This is not surprising since several earlier HP calculators have had various stopwatch functions. (Not to mention the HP-01 which was a watch with calculator functions.) The HP-55 had a proper quartz timer; the HP-45 had several stopwatch facilities but they did not use a quartz timer so they were not very accurate and were not officially described as an HP-45 feature. Even the original scientific calculator, the HP-35, can be enticed to run as a timer, but this was also an unofficial feature, and would only count in milliseconds, using all ten mantissa digits.

On the HP-41 the stopwatch can be used in three ways. It can be run under full manual control, with the keyboard redefined to control the stopwatch functions and allowing the storage of split timings. It can be run under program control, for example to time execution of HP-41 functions. It can also be used as a simple alarm. The stopwatch will even run while the HP-41 is off, just like the clock, but independent of the clock. (The stopwatch does use the same accuracy factor as the clock though.)

The keyboard controlled stopwatch mode is entered by the execution of the function **SW** and is the most powerful of the stopwatch features. In this mode the keyboard acts as a controller; the keys are redefined to control the stopwatch features. (This is rather like the Alpha keyboard where the keys are redefined to enter text instead of executing functions.) A keyboard overlay is provided with the Time module and with the HP-41CX to show what the keys do in the stopwatch mode. This mode allows the taking of **split timings** (or **splits**) - times which are recorded while the stopwatch continues to run. One way that splits can be used is for timing several laps on a racetrack without stopping the cumulative stopwatch timing. At the end of the race it is possible to calculate individual lap times by recalling the split times for each lap in turn and subtracting the time at the end of the previous lap. The stopwatch mode can be entered from a running program if the program executes the function SW; the program stops running and the stopwatch keyboard becomes active until you exit from stopwatch mode, then the program continues to run. (You can leave the stopwatch running and go back to the main program if you press SHIFT, backarrow.)

As the keyboard stopwatch mode is of little use in most programs I shall not describe it in more detail. Indeed it can be a positive nuisance since it stores splits in data registers, starting at register 00 and overwriting information you may have wanted to keep in these registers. (The HP-41CX has a safer version of SW, called **SWPT**, or you can use the Extended Function REGMOVE to save the contents of the registers at risk. See Sections 9.5 and 10.3 for descriptions of these two functions.) For more details of the keyboard stopwatch, read the excellent descriptions in the Time module or HP-41CX manuals. The stopwatch keyboard functions are similar to the functions provided by a stopwatch that costs less than the HP-41 Time module on its own. The module is worth having only if you want to perform calculations on stopwatch values, or print these values out, or use other features of the module. Let us now turn to these other features.

Of more interest to the HP-41 programmer are the fully programmable

stopwatch functions. The first one to use in a program is **STOPSW**. This stops the stopwatch and should always be used in case the stopwatch is still running as a result of some previous operations. The next function is **SETSW** which sets the stopwatch to a time defined by the value in X. Any legal time from -99.59599999 (minus 99 hours, 59 minutes, 59.9999 seconds) to 99.59599999 can be set but only the first six digits after the decimal point will be used. Further digits after the decimal point will be ignored, not rounded; the stopwatch always works to the nearest hundredth of a second. (If you use SETSW while the stopwatch is running then it will carry on running from the newly set time.) After you have used SETSW, probably to set the time to zero, you can use **RUNSW** to start the stopwatch at any point you wish in your program. If the stopwatch is already running then RUNSW will not affect it, otherwise the stopwatch will start running at once. **RUNSW** will not enter the stopwatch keyboard mode or interrupt a running program in some other way. Finally **RCLSW** recalls the value of the stopwatch time to register X. RCLSW can be used while the stopwatch is running, and repeated uses of RCLSW followed by HMS- can be used to calculate stopwatch splits while a program is running. You must of course allow for the fact that RCLSW itself takes some time to execute (about 58 milliseconds, check it now or see Exercise 9.B). You may prefer to use STOPSW first, then RCLSW at a later place when you are ready for the calculations using the stopwatch value. You can then restart the stopwatch to continue timing, or you can reset it first and start timing a new process.

There are two important uses for these programmable stopwatch functions. The first is to allow the HP-41 to act as a controller for an external device. For example the HP-41 can start a process on a device attached via the HP-IL loop, then use RCLSW repeatedly until some fixed time has elapsed, then stop or interrupt the device or take a measurement from it. The same could be done with an alarm set by the Time module, but using the stopwatch gives different options. For example an alarm will interrupt a running program, whereas the stopwatch can be checked at a convenient point in a program which can continue to run in the meantime, and do something else without being interrupted. It is also possible to use the HP-41 and

HP-IL to control two entirely separate operations using alarms for an operation that must be executed at an exact time and using the stopwatch for an operation whose timing is not so crucial. (The alarm will interrupt whatever is happening and execute as soon as possible; the stopwatch can be checked by means of RCLSW at any suitable moment.) Some devices can send signals to the HP-41 when they start or stop an operation, and the HP-41 can use the stopwatch to time that operation.

The other important use of the programmable stopwatch functions is to time the HP-41 itself. You can write a program that starts the stopwatch, carries out a function or a set of functions, then stops the stopwatch and recalls the time taken. You can use this to see which of two alternative routines is the faster, or to time individual HP-41 functions. You must take a few precautions when timing HP-41 functions with the stopwatch.

i. Always PACK a program before timing it. Otherwise you will be timing any null bytes too, and these take about 4 milliseconds each, wasting time in a program that has been edited. If the program contains any local GTOs or XEQs then run it once to compile these before timing it, otherwise the first timing will give a different answer from later timings (since uncompiled GTOs and XEQs are slower than compiled ones).

ii. Time the timer itself. If you execute **CLX, SETSW, RUNSW, STOPSW, RCLSW** you will get an average time of about 25 milliseconds which is how long STOPSW takes to execute. Remember to subtract this from your timings.

iii. The timer only works to the nearest 10 milliseconds so it is not accurate enough to time a single execution of one of the faster functions. Functions also slow down slightly if they come in the last byte of a register, because the HP-41 has to spend some extra time reading the next program register (remember there are just seven program bytes in each register). For these two reasons it is always best to time an instruction by executing it not once but a multiple of

7 times (for example time the sequence **CLX, CLX, CLX, CLX, CLX, CLX, CLX** and divide the result by seven). This decreases the effect of stopwatch inaccuracies and allows for the fact that on average in a long program every instruction comes at the end of a word once in seven times. (Multi-byte functions can also straddle the end of a word, but this is automatically taken care of as well if you use a multiple of seven copies of the instruction. This only breaks down for instructions that are exactly seven bytes long, in that case put some other function between them, time the whole lot, then subtract the timing of the other function.) If you want to be very accurate, you should also allow for this when checking the timing of the stopwatch itself in point ii above.

iv. If you are comparing different functions on the same HP-41 then the above methods are sufficient to compare different ways of doing the same thing. Your timings will not be absolute though, because your HP-41 will run at different speeds depending on the state of its batteries and on the weather. Plugged-in modules can slow the HP-41 down as well, in particular a printer can slow down the HP-41 by as much as 50% even if the printer is turned off. The Time module slows down the HP-41 too; after each step of a program it checks the time and looks to see if an alarm is due to go off. The HP-41CX has been speeded up by circuit modifications to make up for this.

v. If you want to compare different HP-41s then you must allow for possible differences in speed due to slight variations in manufacturing. Before the Time module was introduced, functions were timed by seeing how many times they could be executed in 100 seconds (using a stopwatch). To compare different HP-41s the loop **LBL 01, +, GTO 01** was compiled (run for a short time so the GTO 01 could find the LBL 01 and store the distance to it), then it was run for 100 seconds with the number 1 stored in all four registers of the stack. A standard speed HP-41 was defined to be one that showed 1,700 in register X when R/S was pressed after 100 seconds. You can turn timings on your HP-41 into this standard by running the test, then

multiplying other timings on your HP-41 by 1,700 and dividing by the result of your 100 second test.

You can use the stopwatch to time any programmable functions you are interested in, but it requires some effort. Function timings standardised in the way described above are given in the books: HP-41/HP-IL System Dictionary, HP-41 Synthetic Programming Made Easy, HP-41 Extended Functions Made Easy, and in the HP-41 Synthetic Quick Reference Guide (see Appendix A). The speed of many functions depends on their arguments so their timings are given as a formula, or else an average time is given.

It was stated above that SETSW can set a negative time. Unless it is stopped the stopwatch will run forwards from a negative time until it goes through zero, it will continue to run forwards up to 99.595999, then it will restart at zero and run forwards again. If the stopwatch goes through zero from a negative time it sets off a **timer alarm**, or **countdown alarm**. If the stopwatch is being displayed this will merely sound a tone twice. If the stopwatch is not being displayed then two tones will sound, the message TIMER ALARM will be displayed, and a set of tones will repeat for up to 16 times unless you press a key to acknowledge the alarm. The stopwatch will continue to run. If a program is running then the alarm will interrupt the program, but the program will continue to run when the alarm finishes or when it is acknowledged. Only one alarm can be set in this way and it cannot do all the things that an ordinary alarm can do (we shall come to this in the next section), but it does not take up any space in memory.

9.4 Using the Alarms.

In addition to the one countdown alarm provided by the stopwatch, the Time module lets you create up to 253 alarms to go off at fixed times and dates. (The countdown alarm is fixed relative to the time it was started, not to a defined future time and date.) The different kinds of alarms will be described here before the functions for setting and cancelling alarms.

The simplest kind of alarm is a **message alarm**, similar to the countdown alarm. A message alarm displays a blinking message and sounds a set of tones at a chosen date and time. If a program is running, the alarm executes after the next function has been completed (this can be a long time for some functions, particularly the keyboard stopwatch), after the alarm finishes or is acknowledged the running program continues. When such an alarm goes off you need to acknowledge it while the display is blinking, by pressing any key except STO, backarrow or ON. When you press a key, the tones stop, the display stops blinking, and the second half of the message is shown if the message was more than 12 display places long. Backarrow or ON clear the alarm but also immediately clear the message from the display. Pressing STO stops the alarm, but does not clear it. If the alarm is not acknowledged at all, or if it is stopped by STO then it becomes a **past due alarm**. The purpose of a past due alarm is to remind you that an alarm has gone off but has been ignored, and you are reminded about any past due alarms every time you turn the HP-41 off or on. If no message is provided when the alarm is set, then it displays the time and date instead of a message.

The next kind of alarm is a **control alarm** (or **interrupting control alarm**). If you create a message alarm but the message starts with two up-arrow characters (↑ created by pressing SHIFT,N in ALPHA mode), then the message will not be displayed. Instead the HP-41 will go through CAT 1 and CAT 2 when the alarm comes due, looking for a label or function whose name is the same as the message after the two arrows. If the message is more than six characters long then only the first six will be used in the search. The whole operation is very similar to a XEQ IND instruction. If a corresponding label or function is found, then it is executed. A serious shortcoming is that CAT 3 is not searched, this too is like XEQ IND which cannot execute CAT 3 functions. As examples, you can see that ↑↑SIN will not execute the CAT 3 function SIN, and ↑↑RCLFLAG will not execute the Extended Function RCLFLAG, but will execute a program starting at the label LBL RCLFLA, if such a label exists. If you want to execute a CAT 3 function or any program or function with a seven-character name then you will have to write a short program, with a label of six or fewer letters,

containing the required function or XEQ the required label. This too is the same as with XEQ IND; see Section 6.4. If a control alarm goes off while a program is running, then it interrupts the running program after the current step, and the named program or function executes as if it was a subroutine called from the running program. Once again this is like an XEQ IND. The running program carries on when the interrupting function or program finishes (unless the function or program stops execution by doing a PROMPT or STOP or OFF). There is one danger here; if the running program has six pending returns at the time when the control alarm goes off then the top return will be lost if the control alarm executes a program, and returns will also be lost if the new program has some subroutine calls and the total number of pending returns exceeds six at any time. Losing the contents of the RPN stack (X, Y, Z, T, and L) is also a danger, but you are more likely to notice alteration of these values.

Control alarms can cause other nasty surprises. They may interrupt a running program and then stop it with the message NONEXISTENT. If you go into PRGM mode you will see a perfectly ordinary function which apparently caused the error for no obvious reason. The alarm is no longer available to check since it has already gone off and has been cleared, it can have caused the NONEXISTENT message if it tried to execute a CAT 3 function, or any name of more than six letters, or a deleted CAT 1 program. It could also be something on a module that has been removed, or a non-programmable function on a plug-in device, such as the printer function PRP. Just as bad, you might have written a new program with the same name as that used by a control alarm which you created long ago. The new name will be nearer the .END. than the old program, so it will be executed instead of the old program or CAT 2 function you had wanted to execute when setting the alarm. For all these reasons it is important to check your alarms occasionally. This can be done using the programmable function **ALMCAT**, or using **CAT 5** if you have an HP-41CX or a CCD module (see Chapter 12).

A control alarm whose message consists of two up-arrows but nothing more will run a program starting at the current step in the current program. This can be used for example if you want a program to turn itself off, then

restart the HP-41 and carry on running at a later time. To do this an HP-41 program can set an alarm, with a two-arrow message, to go off at a time when something should be done, then execute OFF. At the pre-selected time the HP-41 will turn on and carry on running the program. If an alarm like this goes off while a program is running then it just lets the program carry on running. This is one way to create a **conditional alarm** which is an alarm that goes off on condition that it does not interrupt something else (another program or a calculation being executed from the keyboard). A conditional alarm is like a control alarm except that it does not interrupt, so it can also be called a **non-interrupting alarm**. A two-arrow alarm will interrupt a keyboard calculation, whereas an ordinary conditional alarm will only interrupt a clock display.

The way to create a normal conditional alarm is to set up an alarm exactly like a control alarm, but with only one up-arrow at the beginning of the message. This alarm will behave like a control alarm when it goes off if the HP-41 is off or if it is displaying the clock. A conditional alarm will not stop a running program, it will just sound two tones and become a past-due message alarm. If the HP-41 is on but not running then the conditional alarm will behave exactly like a message alarm and will become a past-due alarm if it is not acknowledged.

The conditional alarms are most useful if you want to do something automatically, but without interrupting a running program. Say your HP-41 is using the HP-IL to measure a temperature and record it on a cassette once an hour. If you are also using the same HP-41 to calculate your test results then you will not want the automatic temperature measurement to interrupt your work. A conditional alarm will not interrupt your work, but you will be reminded when you try to turn the HP-41 off that you should run the measurement program manually.

Measuring the temperature once an hour could be done by having the measurement program set a new alarm every time it runs. However, every alarm can be set to automatically repeat at a specified interval. You could set a repeating alarm to remind you of your wedding anniversary once

a year. (You would have to set the repeat time to 365.25 days to avoid leap year problems since the repeat time is a fixed time. Your spouse might then wonder why you wake up in the middle of the night every fourth wedding anniversary... Maybe it's not such a good idea after all.) You could set a repeat alarm to wake you at the same time every morning, or to go off every five minutes in the morning like an alarm clock with a snooze button. And of course you could measure the temperature every hour.

The alarm repeat interval can be any length of time between 10 seconds and 10,000 hours (thirteen and a half months). On the HP-41CX the shortest repeat interval is 1 second. An alarm interval this short is rather inconvenient, the alarm is best cancelled by pressing SHIFT, C while the alarm is going off. After a repeating message alarm is acknowledged, or a repeating control alarm executes its task, it is reset. The resetting is done by adding the repeat time to the alarm time and making this the new alarm time. This new alarm time is compared with the clock and date. If the new alarm time has already passed then the repeat time is added again, and this is done until the alarm time is in the future. This makes sure that when you acknowledge an alarm that is past due or an alarm with a very short repeat interval then the next repetition will be in the future. You can acknowledge a repeat message alarm just like an ordinary message alarm, or you can press SHIFT,C to cancel the alarm and stop it from repeating.

If you do not acknowledge a repeating message alarm then it becomes past due and stops repeating. This is sensible for alarms that repeat once a day since you are reminded of the past due alarm when you next turn the HP-41 on or off. Unfortunately it is a nuisance for short repeat intervals such as a snooze alarm or an alarm that you set to go off every 10 minutes to remind you to make an urgent telephone call. Such an alarm becomes past due and fails to do its job if you do not acknowledge it. Instead it goes off unnecessarily when you next turn the HP-41 on or off, and then starts repeating again; you can avoid this on an HP-41CX by pressing SHIFT, C.

If the problem just described affects you then try the following. Instead of setting a repeating message alarm which goes off first at a given time,

set a non-repeating control alarm to go off at that time. (If you do not want the message to interrupt you then make this a conditional alarm.) The control alarm should run a program to set a non-repeating message alarm to go off within 3 seconds with the message you want. Then the same program should check the time and if necessary set the same control alarm to go off again within a fixed time. (For example a reminder to make a phone call should be set to go off again in ten minutes, but only if the time is in office hours.) This will make sure you are reminded at a fixed repeat interval even if you do not acknowledge the message. Unfortunately it may mean you get a lot of unacknowledged past due alarms at the end of the day. To avoid even this, you need an additional function or routine which cancels the most recent alarm. On the HP-41CX this is easily obtained by the two steps 1, CLALMX.

Using an alarm clearing function you can get rid of the previous alarm whether it is past due or not, then set a new alarm. An alternative method for dealing with past due alarms is provided by the **ALMNOW** function. Should a conditional alarm try to execute a program it will not succeed if the HP-41 is doing something else. Instead this alarm will become a past due alarm, but you can use ALMNOW in a program to execute this past-due alarm later on, if you know that a conditional alarm might go off while the main program was running. If a past due conditional control alarm does exist, ALMNOW will cause it to execute, using one subroutine level to execute it (as if it had been called as a subroutine). If no past due conditional control alarms are found, ALMNOW will do nothing, and no subroutine levels will be used. If more than one conditional past due alarm exists, ALMNOW will execute the oldest. Additional details of past due alarms are provided in the Time Module and the HP-41CX manuals. CLALMX (mentioned above) and three further functions to control alarms are provided by the HP-41CX. These will be described in Section 9.5.

Now that we have studied the uses of the alarms let us just quickly look at the way in which the various types of alarms can be set. You need to provide three arguments in registers X,Y and Z, then execute the function **XYZALM** (naturally!). You can also give an argument in the ALPHA register.

Remember that an argument is a value (a number or character string) provided for use by a function. The arguments are:

X register: Time when alarm is to go off.
In the case of a repeating alarm, this is the time when the alarm is to be activated for the first time.

Y register: Date when alarm is to go off, or zero to set the alarm to go off on the same day as it was set.

Z register: Repeat time for a repeating alarm, or zero for no repeat. Once the alarm is activated, the repeat time is added to the alarm time, and a new alarm is set to go off at the new time. If the alarm is delayed, then the repeat is reset only after the alarm goes off and a multiple of the repeat time is added to the original time so that the alarm is set for the first possible time in the future. The Time Module allows a minimum repeat time of 10 seconds, the HP-41CX a minimum repeat time of 1 second. Repeat times can be up to (but not including) 10,000 hours.

ALPHA: If empty the time will be displayed when the alarm goes off. If not empty, but the contents do not begin with an up-arrow, then the contents will be displayed as a message when the alarm goes off. If the ALPHA register contains more than twelve characters then as many characters as possible will be displayed when the alarm goes off, and the rest will be displayed when the alarm is acknowledged by a key other than ON or backarrow.
If the ALPHA register begins with ↑↑ then a control alarm is set, but remember that only the first six letters after the two arrows are used for the control alarm name. If the ALPHA register begins with ↑ then a conditional alarm is set, again using only the first six characters after the arrow.

(It is unwise to use up-arrows as names for programs to be set off by an alarm. The alarm-setting always uses the first two arrows as a control alarm, so you cannot set a conditional alarm to run a program whose label begins with an up-arrow. ↑↑PROG will set a control alarm for PROG, not a conditional alarm for ↑PROG.)

Times and dates for alarms should be set in registers X,Y and Z according to the rules laid out at the start of Section 9.1 and allowing for the current choice of DMY or MDY. The same limits on dates also apply. Seconds can only be set to the nearest tenth, not the nearest hundredth.

As a simple example of setting an alarm, try this. You need to make an urgent phone call as soon as possible after 4p.m. today. Set an alarm to go off every 5 minutes from 4p.m., displaying the message "CALL JO" :

```
1) .05 , ENTER                    ; put in 5 minute repeat time
2)  0  , ENTER                    ; put in 0 as alarm date
                                  ; (sets the alarm for today's date)
3)  16                            ; put the alarm time as 4p.m.
4) ALPHA, CALL JO, ALPHA          ; put in the message
5) XEQ, ALPHA, XYZALM, ALPHA      ; set the alarm
```

Once you have set a few alarms you may need to cancel some of them or to review them. The HP-41CX functions can be used if you have them and the special plug-in Data Acquisition Module also has an alarm cancelling function (see Chapter 12). You can also use Synthetic Programming to cancel alarms from within a program. Alternatively you can review and cancel alarms from the keyboard by using the alarm catalogue function ALMCAT. When you execute this from the keyboard or from a program, the HP-41 displays all the alarms in the order in which they are set to go off, then carries on running a program if ALMCAT was executed from a program. On the HP-41CX you can also list the alarms by pressing CAT 5, and CAT 5 is available on the plug-in CCD Module (see Chapter 12).

While the alarms are being displayed by ALMCAT or CAT 5, you can interrupt

the display by pressing R/S. The HP-41 stops with the keyboard redefined to let you review, reset or cancel the alarms. You can press SST to see the next alarm, BST to see the previous alarm or R/S to restart the catalogue. The backarrow key stops the catalogue and displays or continues a running program. The ON key turns the HP-41 off. You can also press M (the RCL key) to see the message associated with the alarm displayed. If the message is over 12 characters long, as many as possible will be displayed when you press M, and the rest will be displayed when you release M. If you have a control alarm or conditional alarm, then that will be displayed. Furthermore, you have the following time and date options available when you press the appropriate keys.

D	display date when alarm will go off
T	display time when alarm will go off
SHIFT T	display current time
R	display repeat interval
SHIFT R	reset a repeating alarm by a multiple of the repeat interval to the first time in the future after the current alarm time
SHIFT C	cancels the alarm displayed, and if it is a repeat alarm then it cancels the alarm completely, so that it will stop repeating.

SHIFT, C is an easy way to cancel an alarm with a very short reset time, since such an alarm will otherwise keep going off before you have time to cancel it. On the HP-41CX SHIFT,C can also be used to cancel a repeating alarm while it is going off. Press SHIFT,C instead of acknowledging the alarm, or after you have acknowledged it but while the message is still in the display, and the repeat alarm will be cancelled. The alarm times and dates use the HP-41 clock as a cue when you set them and when they go off. It is therefore important to check that the time and date in the HP-41 are correct when you set an alarm. An alarm that is to go off within a given time after the present time will of course not need the time and date; you can set such an alarm using the stopwatch timer alarm, or you can write a short program to do this (see Exercise 9.C for an example of this).

To avoid alarms going off in the middle of a running program you can move

the time into the past. If you execute **-1,T+X** then the clock is set back by an hour, and any alarms due to execute will be delayed by an hour. An hour's delay should be sufficient, but if your program is likely to run for longer, or if you need the correct time, then move the time back by a day. Of course you must correct the clock at the end of your program. Alternatively you could move the time forward, forcing any alarms due to go off in the near future to execute immediately.

The above description will be sufficient for most users of the Time Functions. For further details, particularly of multiple alarms and past due alarms, look in the Time Module Handbook or the HP-41CX Handbook.

9.5 Additional HP-41CX time functions.

An alarm which goes off at an awkward time, particularly in the middle of a running program can be a major nuisance. Alarms can be suspended by using **-1,T+X** as described above, but the Time Module does not provide any programmable way of cancelling an alarm. A programmable Synthetic subroutine to clear any buffer, including an alarm buffer, will be given in Chapter 16 but the HP-41CX provides the best solution; three alarm cancelling functions. First of all, **CLRALMS** clears all alarms, including past-due alarms. If this is too dramatic, **CLALMA** clears an alarm whose message or control string is the same as the string in the Alpha register, even if this is past-due. The contents of Alpha must match the alarm exactly, including the up-arrows in control and conditional alarms. The Alpha register can be left empty to cancel an alarm that was created with no message. If more than one alarm has the same Alpha string, then the alarm due to go off soonest (the first one shown by CAT 5) is cancelled. The third alternative is **CLALMX**. This uses the number in X to specify which alarm in the alarm catalogue should be cancelled. Unlike most HP-41 functions, CLALMX counts the first alarm as number 1, not zero. You have to be careful when programming this function because alarm numbers can change as new alarms are set and old ones go off. Both CLALMA and CLALMX give the error message NO SUCH ALM if the given alarm is not found.

If you want to save a few alarms before using CLRALMS, or if you want to check which alarm you need to cancel, then **RCLALM** comes to your rescue. Like CLALMX this uses the unsigned integer part of X to specify the number of the alarm to be recalled. It too counts alarms as in the alarm catalogue, starting from 1, and gives DATA ERROR if X is 0 or greater than 999. RCLALM sets the X, Y, Z and ALPHA registers to the time, date, repeat and message values that XYZALM would use to set the alarm. X is in 24-hour format and Y contains a date in the current date format (not a zero for the current date). The previous value of X is saved in L and the previous value of Y is lifted to T, even if stack lift was disabled. You can save these alarm values so as to set the same alarm or a similar one later. Saving all these values requires seven registers in general; they can be saved in a data file or even in a text file (the numbers can be brought to ALPHA by ARCL and can later be recreated by ANUM; see Section 10.2).

An alternative to cancelling alarms is to use conditional alarms. These will not go off while a program is running. The function ALMNOW can be used to set off any conditional alarm that has become past-due during the program.

The remaining new time function is **SWPT**. This is like SW but it lets you select the initial values of the storage register pointer and the recall register pointer. These two values are obtained from a value sss.rrr which has to be set in X before SWPT is executed. A negative value of sss.rrr also sets the stop watch to delta split mode instead of regular split mode. On exit from SWPT, X contains the current pointer values in sss.rrr, including a minus sign if delta split mode was in operation. If either pointer is undefined when the stopwatch is exited (for example it may time-out while you are resetting a pointer) then that pointer value is set to zero in X. The previous value of X is saved in L. A very small negative number sets the stopwatch to delta split mode with both pointers at zero, and is returned if SWPT is stopped with the stopwatch in this state. The stopwatch itself need not be stopped, the value in X can be saved, and then used again later to carry on recording times with SWPT.

SWPT is most useful for setting the stopwatch automatically, but it can be used in programs, for example to check how many splits have been taken. You must however be careful not to confuse those storage registers that contain stopwatch values with other storage registers that contain different values.

Exercises

9.A Try writing an **ADOW** routine to complement the functions ADATE and ATIME. Hint: one way to achieve this is to check whether the number in X lies between 0 and 6, then GTO IND X and have six labels each followed by a text string and a RTN.

9.B If you use the stopwatch functions to time other functions then you must know how much time the stopwatch functions take themselves. You can time RCLSW, but try the following alternative: write the routine

CLX, STOPSW, SETSW, RUNSW, RCLSW, STOPSW

Pack, then run the routine to give you its timing. Now you can insert between RUNSW and RCLSW seven copies of any instruction you wish to time. Seven copies and packing will ensure that the word alignment of RUNSW and RCLSW does not change. The difference between the two timings, divided by seven, gives a good estimate of the function's execution time. Now use this method to time RCLSW and STOPSW.

9.C It can be useful to set an alarm that will go off within a fixed time from the present (say in half an hour to let you have a short nap). Write a routine that sets an alarm to go off within the time specified by register X.

CHAPTER 10 - EXTENDED FUNCTIONS

10.1 Extending your control over the HP-41.

The bare bones HP-41C is just a calculator with nice additional features such as an alphabetic display and tones. These can be used for better communication with the user, but most instructions are mathematical functions. Programmers who want to use an HP-41 for more than just mathematical formulae are limited by their inability to control the HP-41 itself. A complicated program might need a lot of key assignments to be made before it is run. If the same keys are to be used for other purposes in different programs then the user must use ASN many times to assign the functions before running the program, and again to clear them after running the program. A programmable version of ASN or a programmable function to clear all key assignments would extend the programmer's ability to use the HP-41 by automating the process of assigning or clearing functions.

The Extended Functions Module, or the Extended Functions built into the HP-41CX provide this sort of extended control over the HP-41 itself. To continue with the above example, there is a programmable assign function and there is also a function to clear all key assignments. The Extended Functions do not include any mathematical functions at all. These Extended Functions are all used for controlling the HP-41 or for moving information from one place in the HP-41 memory to another place.

The Extended Functions Module also provides additional memory called Extended Memory, together with functions for moving information to and from this memory; they will be described in Chapter 11. Chapter 10 will cover those Extended Functions that do not deal with Extended Memory. The functions will be described as in Chapter 4, with emphasis on programming features that are not emphasised in the HP manuals. Much more detailed descriptions of the Extended Functions together with examples can be found in Keith Jarett's book "HP-41 Extended Functions Made Easy", and in "Data Processing on the HP-41C/CV" by William C. Phillips. Both are mentioned in the book list, Appendix A. The Extended Functions were apparently written

by someone who was not a member of the team that originally programmed the HP-41 itself, and contains a number of "bugs". Some of these are covered in Keith Jarett's book. More are described in Appendix C of this book. So long as you are careful in the use of the Extended Functions, they will help you get better use from your HP-41 without any trouble.

10.2 Alpha string control.

The ALPHA register and the text string handling features of the HP-41 were designed to provide convenient messages, prompts and program names, but little more. Operations such as interpretation of text strings or alphabetic sorting are very difficult or impossible, unless you use Synthetic Programming tricks which will be described in Chapter 14. Since the HP-41 does have some ALPHA features, and because these ALPHA features are used for the control of devices via HP-IL, it was logical to provide additional alpha string control functions in the Extended Functions Module. The functions are listed below. Remember that each character is stored as a single byte, between zero and 255. The **null** character is represented by the byte value zero and displayed as a line at the top of the HP-41 display. When the ALPHA register is empty it is actually filled with 24 null characters. The HP-41 was originally designed to treat a null character in the ALPHA register as indicating an empty ALPHA register, so null characters can produce odd behaviour. The results of using nulls will be mentioned in several places in the following descriptions. (The HP-41CX Owner's Manual and the Extended Functions module manual contain appendices describing the behaviour of null characters. You should read one of these to help you understand what is said here about nulls. The Extended I/O and the HP-IL Development Module manuals contain this information as well.)

ALENG This function works out the length of the text string in the ALPHA register by counting all the characters in it and puts this number in the X register, lifting the stack.

Another way of looking at this function is to say: the ALPHA register is 24 characters long, count the number of unused spaces

on its left and subtract this number from 24 to give ALENG. This means that null characters to the right of at least one non-null character are counted in ALENG. If the ALPHA register is empty then ALENG returns 0. This is the simplest of the alpha string functions, it does not alter the contents of the Alpha register.

A typical use for ALENG is in storing the text from the ALPHA register into some numbered data registers. You can store six characters at a time, then delete those six characters using ASHF, then use ALENG to check if there are any more characters left to be stored. The following shows how this can be done to store the ALPHA string into the registers starting at register 10:

```
1) 9              ; Register number last used, may already be in X
2) ENTER
3) LBL 12
4) SIGN           ; turn the contents of X into a one
5) +              ; adds the one to Y, and puts it into X
6) ASTO IND X     ; store 6 characters into register
7) ASHF           ; delete leftmost 6 characters
8) ALENG          ; obtain number of characters left
9) X≠0?           ; is this zero?
10) GTO 12        ; if not, repeat the cycle
11) RDN           ; otherwise get rid of the zero
```

These 11 lines can be included in a program which saves the current status of the HP-41 before changing the status. Register X can contain the number of the last register already used, these lines will store the ALPHA contents into one, two, three or four registers, and will finish by again leaving the number of the last register used in X. This number must also be saved somewhere so that the ALPHA register can be restored later.

XTOA This uses the contents of register X to append one or more characters to the right-hand end of the ALPHA register. If register X contains a text string, then this text string is added to ALPHA, in just the same way as ARCL X would work. (The text string would be put into X by an earlier ASTO X.) If register X contains a number though, then that number is interpreted as a byte code, and the character corresponding to that byte is appended to ALPHA. For example, the code for the character & is 38, so the instructions 38, XTOA would append the character & to the ALPHA register. Since & is not available on the ALPHA keyboard, this is the quickest way normally available for putting & into the ALPHA register. (Synthetic Programming provides an even better way which will be described in Section 14.3. The best way is to use the alternative ALPHA keyboard provided by the ZENROM module described in Chapter 12.) XTOA does not change the contents of X. If XTOA makes the Alpha register contain 24 or more characters then it beeps. If X contains zero, XTOA will append a null character to the right of the ALPHA register; this means that if ALPHA is already empty, it will stay empty, otherwise it will contain a null character which is displayed as an over-bar in the ALPHA register, but is not seen at all when viewed in a text string stored in a data register.

XTOA ignores the sign and the fractional part of X, and only works for numbers from 0 to 255. Table 8.3 showed how each number is interpreted in the HP-41 display. Any code that does not correspond to a character recognised by the HP-41 display will show up as a character with all fourteen segments turned on (often called a "boxed star" or "starburst").

The contents of the ALPHA register can be shown in the display, but they can also be sent to a printer, and they can be sent to any device connected to an HP-IL loop. Many of these devices, such as video displays or printers interpret characters according to the American Standard Code for Information Exchange (ASCII).

In this code some numbers are treated as characters to be printed while others are not printed but instead are treated as instructions to the device. It is worth noting here that XTOA can be used to put such instructions into Alpha for the purpose of sending them to a printer or a video display.

ATOX This function is nearly the opposite of XTOA. It removes the leftmost character from the ALPHA register, interprets it as a number from 1 to 255 and puts this number in the X register, lifting the stack. If the ALPHA register is empty then ATOX returns a zero to X. (ATOX cannot be used to find a null character in the ALPHA register, since it looks for the first non-null character at the left of the ALPHA register.)

ATOX can be used to interpret a text string, one character at a time, for example to interpret a string sent to the HP-41 from a device on the HP-IL loop. ATOX can also be used to compare two ALPHA values or put them in alphabetic order. Unfortunately, it destroys the ALPHA strings in the process, so copies should be made before the strings are checked. The HP-41CX indirect comparison functions, described in Section 10.6, and the PPC ROM program AL (see Chapter 12) compare text strings without destroying them.

AROT Rotates the characters in the ALPHA register left or right by the number of places given in X. If X contains a positive number, the rotation is to the left, and if the number is negative, the rotation is to the right. This is best illustrated by some examples. If the ALPHA register contains "SCAT" and you put 1 into X then execute AROT, you will find "CATS" in ALPHA. This is because 1, AROT rotates the contents of the ALPHA register by one character to the left. If you now put -1 in X and execute AROT again, then ALPHA will again contain "SCAT" since you have rotated one letter to the right. Now do -3, AROT

and you will again see "CATS" in ALPHA. This shows that rotating four letters one place to the left is the same as rotating them three places to the right.

A problem occurs if you rotate a text string containing null characters. Create the text string BAT followed by a null; you can do this by putting BAT into ALPHA, then executing 0, XTOA. You will see BAT¯ in ALPHA if you execute AVIEW or press ALPHA twice. Now do 1,AROT and you will see AT¯B in ALPHA. Do AROT again (the 1 stays in X) and you will see T¯BA. Do AROT one more time, and you will see BAT. The null character has been rotated to the front of the text string and has vanished because it has joined up with all the other null characters at the left of the ALPHA register. You must therefore be careful not to rotate null characters to the left-hand end of a text string since they can only be recognised if they have at least one non-null character to their left. A null character will also be lost if the characters to its left are removed by ATOX.

One important use for AROT is to correct for the fact that ATOX works at the right-hand end of ALPHA but XTOA works at the left-hand end. Suppose you want to check what the first letter in ALPHA is, without losing that letter. You can do the following:

```
ATOX ; take the left-hand end character from ALPHA
XTOA ; put the same character into ALPHA, at the right end
-1
AROT ; rotate this character back to the left-hand end
RDN  ; roll down the -1, leaving the character code in X, where
         you can now test it.
```

This is fairly quick, leaves register L unchanged, and saves the previous contents of registers X and Y in registers Y and Z. Unfortunately it does not work if ALPHA contains one or more nulls after the first character. In such a case it is best to

check whether null characters have been lost by using ALENG to see if the length of a character string has changed unexpectedly.

POSA This searches the ALPHA register for the text beginning with the string or character code contained in X. Say you are looking for the letter A and the ALPHA register contains the text string CATAPULT. You can put the number 65 in X (65 corresponds to the letter A, see Table 8.3), then execute POSA. Positions in the ALPHA register are counted from left to right, starting at 0, so you will see the result 1. (This counting system may look silly, but it works with AROT. If you want to find a character and put it at the front of the ALPHA register then you can do POSA, AROT, but see below for what happens if the character is not found.)

What if you wanted to find the second A in CATAPULT? You could use the fact that the second A is followed by a P, not a T. Store the text string "AP" in register X and do POSA. You will see 3 in register X, indicating that the text string "AP" is to be found in "CATAPULT" beginning at position 3. If you use POSA with a text string in X then obviously that text string cannot be more than six characters long.

If the character or string you are searching for is found, then the number or string in X is replaced by the position, and the previous contents of X are saved in register L (so that you can get them back with LASTX). If the target (character or string) is not found, then POSA returns -1 to register X, and again saves the previous value of X in register L. As with XTOA, POSA ignores the sign and fractional part of a number in X, and only works with numbers from 0 to 255. Since POSA works with 0, it can be used to check if the ALPHA register contains any nulls buried in the text string. This is a useful extra test for nulls that may later be lost because of ATOX or AROT as described above.

ANUM Searches the ALPHA register for a number and brings that number into register X, lifting the stack if stack lift is enabled. For example, if the ALPHA register contains "POWER = 125 WATTS", then ANUM will put 125.0 into register X.

This function is very useful because there is no other simple way of turning a set of text characters into a number. The HP-41 treats all text strings in the same way, whether they look like a string of letters "FRED", or a string of digits "17.530" or a combination "ANS=25". None of these can be used for arithmetic. If register X contains a number such as 17.530 (in FIX 3 mode) and you execute ARCL X then the ALPHA register will not contain the number 17.530, instead it will contain the text string "17.530". If you now execute ASTO X, register X will contain the text string "17.530", and if you try to do arithmetic with this string you will get the error message ALPHA DATA. Execute ANUM instead of ASTO X, to get the number 17.530 instead of the text string "17.530".

ANUM is used for two main purposes. The first is to recover numbers that have been turned into text strings by the ARCL function. The second is to extract numbers from the ALPHA register when the ALPHA register contains a string of text received from an HP-IL device. Your HP-41 could be connected to a telephone link which sends the vital message "NEW GOLD FIX=$950.352". A program could then use POSA to see if the word "GOLD" is in this string, then ANUM to find the number. (Then it could work out if you can still afford to be using an HP-41 to look after your finances.)

As with number entry from the keyboard, flag 22 is set if a number has been entered, and it remains unchanged if no number has been entered. (See Appendix D.) If you execute ANUM and the ALPHA register does not contain a number, then the stack remains unchanged and flag 22 remains clear if it was originally clear.

Your HP-41 could be connected to a telephone link and could use the following instructions to check each message for the presence of a number:

```
LBL 10      ; start subroutine to check for a number
INA         ; receive an alphabetic message from HP-IL
CF 22       ; clear flag 22 first
ANUM        ; try to read a number from ALPHA
FS? 22      ; check if a number has been used
GTO 20      ; go to deal with a number if one found
GTO 10      ; otherwise go and read the next message
```

ANUM sets flag 22 as if the number had been entered from the keyboard, and it also interprets the contents of the ALPHA register as if it were getting a number from the keyboard. This leads to the following rules: you can press the CHS key several times when you are entering a number from the keyboard; in the same way, ANUM changes the sign of a number each time it finds a minus sign, even if there are one or more minus signs in the middle of the number. Therefore the text string "-1--23-4" is interpreted as the number 1234.0, just as if you had pressed CHS four times while entering the number 1234 from the keyboard. (Pressing CHS before you start entering a number would of course just change the sign of the previous number in X.) Plus signs are simply ignored, wherever they may be in the number. In the same way, the first decimal point is treated as a decimal point, and any further decimal points are ignored, just as they would be if you pressed the decimal point more than once while entering a number from the keyboard. If the ALPHA string contains a row of more than ten digits, only the first ten digits will be used, just as with keyboard entry, and even if several digits at the front are zeroes they will be included in the count of ten digits. If a letter E followed by some numbers is found then it is treated as an exponent, but only if it comes after less than 9 digits, or if a decimal point comes before the eighth digit.

Combined with the previous rule this means that in 123456789E12 the E is ignored (because it comes after the ninth digit), so the number is made up of just the first ten digits and is 1234567891. (A sign after the E would also be treated as part of the number). However "12345.6789E12" becomes 12345.67890E12 or 1.234567890E16 since there was a decimal point before the 8th digit. The string "12345.67898765E12" is treated as 12345.67898E12 or 1.234567898E16 because the first ten digits before the E are used as a mantissa, the remaining digits up to the E are ignored, but the exponent is used. Decimal points, additional letters E, and all digits except the first two are ignored after the first letter E.

A number begins at the first numerical digit that is found in the ALPHA register, going from left to right. This means that "E12" is treated as 12.0 not 1E12. Decimal points and minus signs that come immediately before the first digit are treated as part of the number. Thus "-E-12" is treated as -12, because the -E is ignored. -E-.-12 is treated as 0.12 because the decimal point and both the minus signs after the E are included in the number. "1E-.12" becomes 1E-12 as only the point is ignored.

A number ends when any character is encountered that cannot be treated as part of the number. Additional E's or signs or points at the end of a number will not terminate the number, but a letter F or a dollar sign, or a null character will terminate numeric entry.

The behaviour of commas depends on the setting of flag 29 (the digit grouping flag). If flag 29 is set, then commas are treated as digit grouping marks, and are ignored no matter where they are and how many of them there are. If flag 29 is clear, then it is assumed that there will be no digit grouping marks, and a comma terminates numeric entry. For example 123,456.7 is interpreted as 123. if flag 29 is clear.

All of the above statements assume that flag 28 is set so that the radix mark which separates the fraction from the rest of the number is a decimal point. If you use European notation, with flag 28 clear, the roles of points and commas are reversed in all the above statements.

This is a truly amazing set of rules, but it follows logically from the assumption that ANUM should work in exactly the same way as numeric entry from the keyboard. In most cases you will only be interpreting simple numbers such as 12.70 anyway and these rules will not matter. The most awkward problem arises if the ALPHA register contains two numbers and you want to find both of them. This can only be done easily if you know in advance what character separates the two number strings. You can then do ANUM, use POSA to find the separator, use ATOX to delete all the characters before the separator, then use ANUM again. You can also use AROT to move the first number to the back of the Alpha register, but in that case make sure there is a non-numeric character following the second number, otherwise the first number may become part of the second one after the rotation. A space is suitable, since it is not treated as part of a number.

If problems of this nature are going to affect you seriously, you should consider buying the Extended I/O Module. This contains extra ALPHA control functions including ANUMDEL which works like ANUM, but then deletes the number just found from the ALPHA register. Other useful functions in this module work like XTOA and ATOX, but can be used at the left-hand end or the right-hand end or any position in the middle of an ALPHA string. See Chapters 12 and 16 for details of the Extended I/O Module and of the HP-IL Development module which contains similar functions, but not ANUMDEL.

10.3 Moving data and flags.

REGMOVE Moves the contents of a block of data registers to another block of registers. If you have important data in some

registers, say registers 00 to 08 but you need to use those registers for another purpose then you can move the data from registers 00 to 08 into some other registers with REGMOVE. Later on you could move the data back into registers 00 to 08.

In general, if you want to move a block of nnn registers, starting at register sss into a block of registers starting at register ddd then you put sss.dddnnn into X and execute REGMOVE. If you wanted to move the registers from 00 to 08 (9 registers in all so nnn = 009) as mentioned above you would use REGMOVE as follows. Select a block of registers to take the values, say registers 110 to 118. Then ddd is the first of these registers, namely 110. The source is the first address of the block 00 to 08, so sss is 000, and sss.dddnnn must be 000.110009. You should therefore put 000.110009 into X and execute REGMOVE. (Zeroes at the front and at the end can be left out so you could just put .110009 into X.) If nnn is zero, one register is copied.

Another use for REGMOVE is to make an extra copy of some data registers. For example you may want to invert a matrix, then check the inversion by multiplying the inverted matrix by the original matrix. Most matrix inversion programs, including the one in the Math Module destroy the original matrix they are inverting. You could therefore copy a matrix with REGMOVE, invert the original, then multiply the inverted matrix and the copy of the original one to check how good the inversion was.

Unlike some other Extended Functions, REGMOVE could be performed using just the functions of an ordinary HP-41, but this would disturb the stack whereas REGMOVE is faster and leaves the stack registers unchanged. If you find this description hard to understand, then follow it on the diagram below.

register number	register number
8	118
7	117
6	116
5	115
4	114
3	113
2	112
1	111
0	110

Lowest numbered source register is 000
Lowest numbered destination register is 110
Number of registers to be copied is 9
Therefore sss.ddd nnn is 000.110009
The copying is done by the two instructions .110009, REGMOVE.

REGSWAP This works exactly like REGMOVE except that the contents of the source and destination blocks of registers are swapped. When you put sss.dddnnn into X and execute REGSWAP then the contents of nnn registers beginning at sss are swapped with the contents of nnn registers beginning at ddd. We can take the same example as for REGMOVE; two subroutines each of which uses registers 00 to 08. Every time one of the subroutines is used following the other, REGSWAP could be used to bring its numbers into registers 00 to 08, while saving the numbers used by the other subroutine.

If the source or destination registers do not exist (because SIZE is too small) then REGMOVE and REGSWAP will normally display NONEXISTENT and do nothing. Unfortunately if you have all 319 data registers (a CV or a C with a quad memory module, or a CX), and an Extended Memory Module in Port 1 or Port 3 you may get into trouble. Early HP-41C's with Bug 2 (see Appendix C) have another problem. If your SIZE is smaller than S and you try to

REGMOVE or REGSWAP into registers above register number S (the first nonexistent data register) then you might write into the Extended Memory Module and destroy its contents. This is because the calculator does not check whether the top registers of the source and destination blocks are within the data register area; it only checks that these registers exist.

Normally you will be using REGMOVE or REGSWAP to move data between two separate blocks of registers. The two blocks could overlap though, by accident or on purpose (depending on what kind of programmer you are!). As a simple case, imagine doing REGMOVE with sss.dddnnn = 001.003004. This means copy registers 001 to 004 into registers 003 to 006, and the two blocks overlap. The diagram below shows what happens during the execution of REGMOVE. The source and destination are shown as two separate blocks even though they overlap. Originally each register contains a value equal to its register number. The operation is carried out as a series of four moves:

	source registers (sss)	content	content	destination registers (ddd)
A) before any moving	04	4	6	06
	03	3	5	05
	02	2	4	04
	01	1	3	03
B) first move	04	4	4	06
	03	3	5	05
	02	2	4	04
	01	1	3	03

C) second move	04	4	4	06
	03	3	3	05
	02	2	4	04
	01	1	3	03
D) third move	04	2	4	06
	03	3	3	05
	02	2	2	04
	01	1	3	03
E) fourth move	04	2	4	06
	03	1	3	05
	02	2	2	04
	01	1	1	03

As you can see, the contents of registers 03 and 04 have been changed. If ddd is greater than sss then the top (highest numbered) register is moved first. If sss is greater than ddd then the lowest numbered register is moved first. This means that you can reverse the effect of REGMOVE on the original source registers even if the source and the destination overlap, simply by exchanging sss with ddd and executing REGMOVE again. In the example above you could recover registers 01 to 04 by executing 3.001004, REGMOVE.

REGSWAP works in exactly the same way with overlapping blocks, but it only shuffles the values around without losing anything. This means that you can encipher a block of data by REGSWAPping overlapping parts of the block, and you can recover the original block if you know what sss.ddd nnn was, and execute REGSWAP again with sss and ddd exchanged.

An extremely clever use of REGSWAP is to rotate a block of registers to the left or to the right. This was suggested with

details in "HP-41 Extended Functions Made Easy". As an example you can rotate a block of nnn registers, starting at register sss, upwards by one register by using sss.(sss+1)(nnn-1) with REGSWAP. This moves the contents of every register up by one register in the block, and moves the contents of the top register to the bottom register. For more information about this, see "HP-41 Extended Functions Made Easy".

X<>F This function stores the status of flags 00 to 07 as a number in register X. At the same time it uses the number in register X to reset the status of flags 00 to 07. This makes flags 00 to 07 much more useful, since many separate sets of flags can be stored in them at different times. Any subroutine that uses flags 00 to 07 can save their initial contents before using them and restore the contents when necessary.

A particularly helpful feature of X<>F is that 0, X<>F clears all eight flags at once. The number 0 means that all flags are to be cleared. 1 means that flag 00 is to be set, 2 means that flag 01 is to be set, 4 means that flag 02 is to be set, 8 means that flag 03 is to be set and so on. In general flag n is set if register X contains 2↑n (two to the power n). If several flags are to be set, the appropriate numbers can be added together in the X register. For example you can set flags 00 and 03 (and clear all the others) by putting 9 in register X and executing X<>F (9 is equal to 1+8 so it sets flags 00 and 03). The largest number that can be used with X<>F is 255; this sets all 8 flags. Larger numbers cause an error. Signs and fractional parts are ignored, so using X<>F twice in succession obtains the absolute integer part of a number, without losing the value in register L.

X<>F can also be used to perform 8-bit arithmetic and logical operations, by putting an 8 bit number into flags 00 to 07, then treating individual flags as individual bits. X<>F can also be used to store a number temporarily in the flag register, if you

have no space left elsewhere in the HP-41 memory. There is one rather silly bug if you use X<>F with a conditional or control alarm. This is described in Appendix C but is unlikely to affect the average user.

RCLFLAG These two functions allow you to save the status of all the **STOFLAG** flags from 00 to 43 and later to restore some or all of these saved flags. RCLFLAG recalls the status of flags 00 to 43 and puts this into the X register as a text string. You can change the status of some flags and later put this flag text string into X and execute STOFLAG to restore all of flags 00 to 43. If you want to avoid losing the status of flags 11 to 20 then execute RCLFLAG before the HP-41 is turned off. When the HP-41 is turned back on it will clear flags 11 to 20 but you can use STOFLAG to put back their original status. If you have a program that uses a special display format (for example to round a number) then you can execute RCLFLAG at the beginning of the program and restore the original display format with STOFLAG at the end of the program.

To restore only some of the flags, put the flag text string into register Y and a number bb.ee into X. bb is the first flag to be restored and ee is the last flag to be restored. Should you wish to restore only flags 11 to 20 you should put the flag text string into register X, then put 11.20 into X (lifting the flag text string into Y), then execute STOFLAG. If you want to restore some more flags, press RDN, put a new value of bb.ee into X and execute STOFLAG again; you will need to do this if you are restoring some flags, but not other flags between them.

The sign of bb.ee is ignored, and additional fractional digits after the first two are also ignored. If ee is smaller than bb then only flag bb will be restored. This means that you can restore a single flag by just putting its number in X and executing STOFLAG while the flag text string is in Y.

STOFLAG can be used to play some interesting tricks. As an example, you can set FIX 10 mode with it as follows. First set FIX 2 mode and execute RCLFLAG. Then set FIX 8 mode. Next put 38.00 in X (lifting the flag text string into Y). Finally execute STOFLAG. If you look at the flag table (Appendix D) you will see that in FIX 2 flag 38 is set, and in FIX 8 flag 36 is set. The four steps above have set both flag 36 and flag 38, giving FIX 10. In this mode, the HP-41 is forced to display all ten digits of the mantissa, so it does not display the exponent at all; in effect FIX 10 is a "view mantissa" function. Unfortunately this does not work for negative exponents, nor for exponents that are 10, 11, 12 or 13 plus a multiple of 14. A better "view mantissa" function is provided by the program APX in Chapter 16. A more useful combination of flags is obtained by setting both of the flags 40 and 41, producing FIX/ENG mode in which large and small numbers overflow from FIX mode into ENG mode (rather than SCI mode). RCLFLAG and STOFLAG do not give access to flags 44 to 55; methods of accessing these will be described in Chapter 14 with a program to set or clear any flag.

You may wonder if there is any point in having X<>F as well as RCLFLAG and STOFLAG, but X<>F turns the flags into a number that can be used in numerical operations, whereas RCLFLAG would produce a number larger than the accuracy of the HP-41 can hold. Instead of a number, RCLFLAG produces a text string which cannot be used or altered. The text string can be stored in any data register and recalled for later use, but it cannot be stored in ALPHA and recalled from it. (This is because an ordinary text string contains the hexadecimal value 10 in its first byte, but a RCLFLAG text string is identified by 1FF in its first three nybbles. Bytes and nybbles were explained in Chapter 8.)

10.4 HP-41 status control

RCLFLAG and STOFLAG extend your control over the HP-41 status flags. This section describes other functions that extend your control over the HP-41 status or provide information about the status.

SIZE? It is impossible to select a SIZE that will suit all uses of an HP-41. By the time you have executed SIZE a few times and perhaps put in or taken out a memory module neither you nor your programs will know what SIZE the HP-41 is set to. The extended function SIZE? can be executed from the keyboard or from a program to check this; it returns the SIZE setting to X, lifts the stack (unless stack lift is disabled), and leaves register L unchanged. This is much simpler than finding your size setting by trial and error or with a program, and SIZE? is clearly a function that should have been built into the HP-41 itself if there had been room.

SIZE? is rather a poor choice of name for this function, for two reasons. To begin with, SIZE? ends with a question mark as do the test functions such as X=Y? or FS? Nevertheless, SIZE? does not skip a step, it just recalls the SIZE setting to X. Secondly, several plug-in program modules contain subroutines called SIZE? These subroutines do not work like the SIZE? function, they only help to remind the user to reset SIZE if necessary. If you have one of these program modules plugged in but you want to used the extended function SIZE? then you should take out the program module (Games, Real Estate, or Structural Analysis X), assign SIZE? to a key, then put the module back in. The HP-41 will know that the function assigned was the extended function SIZE? since that was the only one available when the assignment was made. If for some reason you do not want to take out a program module then you can still assign the extended function by using a key assignment program (such as GASN in Section 11.10) and giving it the decimal codes 166, 108. Once

the extended function has been assigned to a key it will be executed or stored when the key is pressed.

One more warning: SIZE? works by counting up from register 00 and stopping when it finds a nonexistent register. If you have a missing register, either because you have been playing with Synthetic Programming, or because part of your HP-41 memory is damaged, then SIZE? may not find any registers above the missing one, even though you can access these registers with STO and RCL.

PSIZE Once you have found the current size setting you may want to change it, from the keyboard or in a program. SIZE is not programmable, so the Extended Functions Module provides a programmable size function, PSIZE, which changes the size setting to the value specified by a number in X. Sign and fractional parts are ignored as usual, and a size that is too large for the current memory available causes an error. If the SIZE is increased, PSIZE will not allow any programs, key assignments or buffers to be deleted. If the SIZE is decreased though, it can delete data registers, even if they contain useful data. If you increase size by more than is available from the keyboard then both SIZE and PSIZE will pack memory and invite you to TRY AGAIN. If you do this from within a program though, then PSIZE will not pack, it will just display NO ROOM. Oddly enough, PASN (described below) does cause PACKING and TRY AGAIN, even from within a program.

A common use of PSIZE is to combine it with SIZE? so as to reset the SIZE in a program that uses data registers. Such a program could include the following steps:

SIZE?	Find current SIZE
nn	nn is the number of registers that the program needs
X>Y?	Check if this many registers are available
PSIZE	If not, change the SIZE setting.

You may think that the two steps nn, PSIZE would be enough, but these could decrease the SIZE setting and lose some valuable data, whereas the sequence above will only increase the size.

It may seem unnecessary for the HP-41CX to have both the SIZE function and the PSIZE function built into it, but since SIZE is a basic HP-41 function and PSIZE is part of the Extended Functions Module, both were left in on the CX. PSIZE is slightly faster during keyboard execution, because CAT 2 is searched before CAT 3, but it changes the stack.

PASN The programmable ASN function is similar to the programmable SIZE function; it allows a program to control its own destiny more completely. Before PASN came along, a program that needed several key assignments had to ask the user to make those key assignments; or it had to use the automatic assignment of the top two rows of keys. PASN makes this unnecessary; you just put a keycode in the X register, a function or routine name in ALPHA and XEQ "PASN" to make the assignment. As with the ASN function, PASN clears a key assignment if no name is given in ALPHA (that is, if ALPHA is clear). The key codes are the same as those used by ASN, the first digit is a row number, starting from 1 at the top row (not counting the toggle keys which are row 0), the second digit is the key number in the row, starting from 1 at the left. A minus sign denotes a shifted key. PASN cannot be used to make assignments to the toggle keys or to the SHIFT key. (The SHIFT key can be assigned by means of Synthetic Programming; see Chapter 14.)

In the next chapter we shall see that the ALPHA register used with Extended Memory functions can contain two parameters separated by a comma. It has been pointed out by Ulrich Jansen (PPCTN 13 P16) that the same rule applies for PASN. The function name is taken from the ALPHA register, but the first comma and everything after it are ignored. This means that PASN with a

comma as the first character in ALPHA will clear anything assigned to a key instead of giving NONEXISTENT as might be expected. This feature is an unexpected side-effect of the way in which other functions use ALPHA, but it can be useful. For example to clear key assignments without clearing the ALPHA register you can append a comma to ALPHA, then do -1, AROT, keycode, PASN, ATOX.

GETKEY This function can be used to identify a key for use with PASN or it can be used on its own as an alternative to using key assignments. When you execute GETKEY, the HP-41 waits for you to press a key, then puts the keycode (identified by row and column, see under PASN) into register X and lifts the stack if stack lift is enabled. If no key is pressed within ten seconds then a 0 is put into X. Obviously this can be used to get a keycode for use with PASN. You could write the following short subroutine to assign + to any key.

"WHICH KEY?", AVIEW, GETKEY, "+", PASN

Some warnings are necessary. The SHIFT key is just identified as key 31, and so you cannot obtain a shifted key identification. You must use CHS after GETKEY to do that. The SHIFT key cannot be used with PASN, and neither can the four toggle keys which GETKEY identifies as 01, 02, 03 and 04. Although these keys cannot be used with PASN, their use with GETKEY lets you redefine the whole HP-41 keyboard completely. You can for example follow GETKEY with GTO IND X. Then if you prefer to use the ENTER key to turn off the HP-41 (because it is the largest key on the keyboard?), you could have LBL 41, OFF in your program. You can do various novel things with GETKEY, e.g. assign short tunes to keys and use the HP-41 as an electric organ, or assign routines which append letters to keys in any sequence you like and touch-type on the HP-41. A final warning; you cannot use R/S with GETKEY to stop a program. A loop such as LBL01, GETKEY, GTO 01 is very difficult to stop. One way to stop it is to press any key except R/S, and keep that key pressed, then press R/S,

release the first key, and release R/S. This uses the "two key rollover" feature to delay the program. Should ten seconds seem too long a time to wait if you do not intend to press any key, then you can use the CX function, GETKEYX, described at the end of this chapter.

CLKEYS This function clears all key assignments. It does not use the ALPHA or X registers. Its main purpose is to provide an alternative to clearing keys one by one with PASN. There are two particular reasons for doing this. One is to save space when you are short of memory. Each key assignment takes up only half a register, but the other half cannot always be recovered. To avoid this waste, it is better to clear all key assignments, then make the necessary ones again. (This is not always worth the trouble since making key assignments in a program also uses up memory.) The second reason is to clear assignments on the top two rows of keys so that they can be used in their default mode to XEQ A , XEQ a , and so on.

Clearing all keys can be a bit too dramatic unless the assignments are saved somewhere first. They can be saved on a magnetic card with the WSTS function, or on a mass storage medium with WRTK. Unfortunately there is no function that saves key assignments in Extended Memory. Programs to do this can however be written with Synthetic Programming, and S.P. can also be used to suspend key assignments temporarily. The temporary suspension can of course be cancelled easily, or key assignments can be read back from cards, mass storage, or Extended Memory.

PCLPS This too is a programmable version of a non-programmable HP-41 function, CLP. There is one vital difference though; PCLPS clears the named program and <u>all</u> the programs that come after it in CAT 1. This can give you quite a shock! Should you execute PCLPS from the keyboard then if your HP-41 has the reset facility (see Section 4.1.iv) and you are quick you can cancel this

function by pressing ENTER and ON at the same time. PCLPS, like CLP, requires the name of an Alpha label and clears the program that contains this label. If no name is given in ALPHA, the current program is cleared (that means the program which contained the PCLPS function). Just like PASN, PCLPS expects the label name to be in the ALPHA register, and ignores a comma and everything that comes after it. There is a good reason why PCLPS should clear a lot of routines all at once. This is that the Extended Memory function GETSUB can fill up memory with a lot of routines copied from Extended Memory, and these routines can be interspersed with extra ENDS. These ENDS can be difficult to get rid of, but if you name the first routine copied with GETSUB, then PCLPS will delete that routine and all the routines that came after it, together with all the extra ENDS.

As mentioned above, a routine that executes PCLPS with the ALPHA register empty clears itself. The routine is replaced with .END. and then that .END. is executed. This stops a running program unless the routine had been called from another one, in which case the .END. acts as a RTN, and you return to the calling routine. This can be very useful since it lets you read subroutines from Extended Memory or from mass storage, and each routine can end with CLA, PCLPS which will clear the subroutine from memory and execute a RTN.

Two warnings. Firstly do not execute PCLPS while the program pointer is in a program module. For example you may have just finished using a program in the Statistics Module. If you then put a global label name into ALPHA and execute PCLPS from the keyboard, you are likely to delete most of your programs from memory, or even to get MEMORY LOST after a breathless pause. If you realise what is going on, you may save the day by pressing ENTER and then ON as well. Secondly, do not delete a routine to which you are going to return. If the last routine in memory contains XEQ "CLEAR", then the routine "CLEAR" must not

do a PCLPS, then a RTN. Otherwise the RTN will return to an empty space where the deleted routine was. The first thing following this empty space might be a buffer, or a timer alarm, or a key assignment, or something else. The program GASN in Chapter 11 uses this feature, but in general the results are unpredictable. This is one example of bug 11, described in Appendix C.

10.5 Additional HP-41CX functions and features

The HP-41CX comes with a Time Module and an Extended Functions Module built in. This means that all the functions described so far in this chapter and the previous one are available in an HP-41CX. An additional nineteen functions are available and some HP-41 features have been enhanced. The five extra Time functions and the Time function enhancements have already been described. Nine new Extended functions and two enhanced features will be described here; five new Extended Memory functions will be described in the next chapter. Some of these new functions can be replaced by short subroutines (using Synthetic Programming in certain cases) and a few such subroutines will be described in this chapter or later ones. So don't give up yet just because you do not have an HP-41CX.

The features that have been enhanced on the HP-41CX are CAT and the default SIZE. To begin with, a CAT display would be slowed down if you kept a key pressed (except for R/S and ON) on a 41-C or a 41-CV. Hewlett-Packard seem to have accepted the fact that most HP-41 users can read pretty fast and do not need this feature. On the HP-41CX, pressing any key (except R/S or ON) causes the catalogue to speed up. This helps you get more quickly to the point you want to reach. Should you really want to slow down a catalogue display, you can still stop it with R/S and then SST (or BST) through it.

CAT 2 has been speeded up in a second way. When started, it displays only those items in CAT 2 that are more than seven characters long. These are the titles of modules or of sections of modules. Two examples: the only name in the Machine Design Module that is more than seven characters long

is the module name "MACHINE 1A". The only function names that are over seven characters long in the Extended I/O Module are the four headers: "-X MASS 1A", "-X EXT FCN", "-X CTL FNS" and "-ADV CTL FN". When you reach the function group that you are interested in, you can press R/S to stop the display, then ENTER (wait until the display blinks after you push R/S, then push ENTER) to restart CAT 2 and display every function name. You can switch between the two display modes repeatedly by pressing R/S and ENTER. This certainly speeds up your search for a given function or program if you know which module it is in. There is just one problem with this scheme; a module whose name has seven characters or less will not show up in CAT 2 unless you stop it and press ENTER before you have reached that module. The only module known to be affected is MATH 1B.

You will notice that the Time Functions show up at the very beginning of CAT 2 and the Extended Functions show up at the very end (each is accompanied by a separate group containing the extra HP-41CX functions). The reason for this lies in the fact that these are ROM modules which have been wired into the HP-41; details were given in Section 8.3.

CAT 3 is unchanged, but CAT 1 also has an additional property; it shows the length of each program. At each END it displays the distance from the previous END in bytes. (The first END shows the distance from the start of program memory.) The .END. however shows the number of registers left free, so you cannot find the byte length of the last program in memory. On the HP-41C and HP-41CV you could get the same information by doing CAT 1 with a printer attached in TRACE mode. If you run CAT 1 on an HP-41CX with a printer attached in TRACE mode then the program lengths will be printed but will not be displayed. It is not obvious whether this was intended or whether it is a "bug".

Three additional catalogues are provided. CAT 4 is just another name for EMDIR and CAT 5 is an alternative for ALMCAT. The only difference is that CAT 4 and CAT 5 cannot be included in a program whereas EMDIR and ALMCAT can. CAT 6 provides a list of key assignments, showing the function name on the left of the display and the keycode on the right. You can use R/S

to stop CAT 6, and SST and BST to move through it. In addition you can use SHIFT C to cancel key assignments as with CAT 5 (ALMCAT). If you have managed to assign a function or program to the SHIFT key (see Chapter 14) then you can use CAT 6 in this way to cancel that assignment. CAT 6 provides a feature not previously available on the HP-41 itself, but it does much the same job as the printer function PRKEYS.

CATS 4, 5 and 6 use a lot of power so they time-out after two minutes if you have used R/S to stop them. They only use the standard keyboard, so you cannot reassign SST or BST and use the reassigned keys in USER mode with these catalogues. The idea of extending the CAT function is a good one, it has been taken even further by the CCD ROM Module (see Chapter 12).

The other feature that has been enhanced is the default SIZE value. On an HP-41C the SIZE after a MEMORY LOST depended on the number of memory modules plugged in. Forty six registers were set aside for programs, buffers and key assignments; all the rest were left for data, giving SIZE values between 17 on a bare 41C and 273 on an HP-41C with a quad memory module or an HP-41CV. On the HP-41CX the SIZE after MEMORY LOST is reset to 100; an easier and more convenient value.

Apart from indirect comparisons which will be described in the next sections, the CX contains three new general-purpose Extended Functions, ΣREG?, CLRGX and GETKEYX.

ΣREG? returns the address of the first statistics register. If you do ΣREG 21, ΣREG? then 21 will be returned to the X register, lifting the stack unless stack lift is disabled. If you do ΣREG 21, SIZE 010, ΣREG? then you will still get 21, even though the registers are nonexistent.

This function is most useful in general-purpose subroutines that use the results of statistical operations. You may for example need the value Σxy/n and this could be obtained by the following subroutine:

```
LBL "ΣXYN", ΣREG?, 5, +, RCL IND X,
ISG Y, LBL 00, RCL IND Y, /, RTN
```

The subroutine can be called from various programs and will work wherever the statistics registers may be. The LBL 00 does not do anything; it serves as a null operation after the ISG.

CLRGX lets you clear data registers selectively according to a control word of the form bbb.eeeii in X. This control word has the same form as the control word for ISG and DSE; bbb is the first (beginning) address to be cleared, eee is the last (end) address and ii is the increment. Signs and fractional parts after ii are ignored, ii is treated as 01 if it is zero. If eee is smaller than bbb or smaller than bbb+ii but eee and bbb both exist then bbb alone is cleared.

This function allows you to clear selected sections of memory without the wholesale destruction caused by CLRG which clears all registers. It is also very useful in clearing rows or columns of matrices; a single column of an m x n matrix can be cleared if you set ii equal to m.

CLRGX is one of the HP-41CX functions that can be easily performed on other HP-41s. Enter the subroutine shown below and execute it by doing: bbb.eeeii, XEQ "CLRGX"

```
LBL "CLRGX"
0
LBL 01
STO IND Y
ISG Y
GTO 01
RDN
RTN
```

This stores zeroes into all the registers specified by bbb.eeeii, and if eee is too small then it clears bbb only. The routine is not exactly like CLRGX because it does not check first whether register eee exists, this can be added with a few extra steps:

```
01 *LBL "CLRGX"
02  FRC
03  1 E3
04  ST* Y
05  RDN
06  X<> L
07  CF 25
08  RCL IND L
09  CLX
10  LBL 01
11  STO IND Y
12  ISG Y
13  GTO 01
14  RDN
15  RTN
```

The above stops at line 08 if register eee does not exist and stops at line 11 if register bbb does not exist.

GETKEYX "get key by X" is an improved version of GETKEY. Instead of waiting 10 seconds for a key to be pressed, GETKEYX waits the number of seconds specified as ss.s in X. This can be anything from 0 up to 99.9 seconds. Small values are only useful in games that demand a fast response, 1 second is much more useful than the unbearably long 10 seconds of GETKEY, longer intervals can be used to display a value for a specified period without stopping a program. An additional feature is that ss.s can be negative and this makes the HP-41CX react as soon as the key is pressed down, not when the key is released. If a key is already being pressed and X is negative then GETKEYX does not wait but carries on at

once. The user can then keep a key pressed down and a subroutine can keep using GETKEYX to check if the key is still down. When the key is released it acts in the normal way. This means that any key except R/S or ON is ignored if the program is still running. If the program has stopped though then the key behaves like a normal keyboard operation.

GETKEYX always alters the stack as follows. It lifts Y and Z into Z and T, it saves X in L, it puts the keycode into Y and it puts a character code into X. In ALPHA mode (flag 48 set) the character code is a number representing the ASCII code for a character, or it is a zero for any other key (such as ASTO or AVIEW). In normal mode (flag 48 clear), the character code represents the ASCII code for numbers, digits and CHS. If you press the dot key it returns 46 if flag 28 is set (decimal point) and 44 if flag 28 is clear (decimal comma). For any other key it returns zero in X. If no key is pressed, both X and Y contain zero. Finally, GETKEYX allows you to use SHIFT. This restarts the time interval and makes the key code negative, so that shifted keys can be used, for example with PASN. Even the toggle keys can be shifted.

10.6 Indirect comparisons.

The six indirect comparison tests are very powerful functions yet they are described in only eight lines of the HP-41CX manual. Originally, the HP-41C had five functions to compare X with Y and five to compare X with zero. These were clearly designed for mathematical use, in particular the comparisons with zero help you avoid errors such as dividing by zero or taking the logarithm of a non-positive number. Only X=Y? and X≠Y? allowed you to compare text strings, and those were clearly no good for sorting.

All the new functions are really indirect comparisons through Y. For example the original functions X=0? and X=Y? have a new companion X=IND Y? though this is called X=NN?. X=0? uses only one register (X), X=Y? uses

two registers (X and Y), X=NN? uses three registers (X, Y which contains an address, and the register addressed). To compare X with the value in register 17 you used to have to do RCL 17, X<>Y, X=Y?. The new function lets you do: 17, X<>Y, X=NN?. This takes one byte more but we shall soon see (point viii below) how useful it can be. Let us examine the various properties of these functions:

i) The six functions can be considered as three pairs of opposite tests:

X=NN?	X≠NN?
X<NN?	X>=NN?
X<=NN?	X>NN?

Remember that a test followed by two GTOs (one if true, the next if false) can be usually replaced by the opposite test followed by one GTO and then by the instructions to be executed if the original test was true. See Section 6.7.

ii) The functions are CAT 2 functions, so they take up two bytes each when stored in a program whereas the direct comparison functions are CAT 3 functions and only take one byte each. It is therefore better to use direct comparisons where they are available as an alternative to the indirect comparisons.

iii) The tests form a complete set, whereas the direct tests do not have X>=0? or X>=Y?. (The HP-41CX manual has an error in the list of direct tests, it gives X>=Y? instead of X≠Y?.) However X>=Y? can be simulated by X<Y?, X=Y? (this technique was described in Section 6.7; the second test is always false if executed). You can also use X<>Y, X<Y? but in this case the contents of registers X and Y are reversed after the test and may need to be exchanged again.

iv) All the tests allow comparison of numbers or text strings. Alpha strings are always greater than numeric data (so that the text string "1.4" is greater than the number 1.4). Alpha strings are compared one

character at a time, starting from the left, and comparing the ASCII values of the characters. Thus "a" (ASCII value 97) is greater than "A" (ASCII value 65). The ASCII values were given in Table 8.1. "AB" is greater than "A" because shorter strings are compared as if they had nulls attached at their right end. (An exception to this rule is that a string followed by a real null character is less than a string which is identical but has no null character at the end.) All this allows for alphabetic sorting which can be used in word games or in applications such as checking lists of customers' names.

v) As might be expected, the functions execute just like direct comparisons. From the keyboard they display the answers "YES" or "NO". When executed in a program they obey the rule: "do next step if true, otherwise skip it."

vi) Instead of a data register number, the Y register can also contain one of the text strings "X", "Y", "Z", "T" or "L". These allow the user to compare register X with any of the stack registers, but how much use is this? Let us go through the stack registers one by one.

X allows you to perform the tests X=X?, X≠X? and so on. Since X is equal to itself, X=X? just provides a very esoteric NOP (a do-nothing function, see Section 6.8). X<=X?, X>=X? are also NOPs, the other three functions force you to skip the next step, but this can be done by much simpler tests that are always false, for example FS? 53.

Y provides a roundabout alternative to the tests X=Y? and so on, but register Y must contain the letter "Y". These tests are therefore of little use unless you want to compare other text strings with the letter "Y".

Z can be used when you want to make comparisons in the stack alone, without using any data registers. This may be of some help when you have a very large program or are even forced to use SIZE 000. For numeric values it is usually easier to rearrange the stack so as to

use a test comparing X and Y, but text strings cannot be compared this way (except for X=Y? and X≠Y?) so this method does have uses.

L can be used in the same way as Z, but it can also be used specifically to compare X with a value saved in L by an arithmetic function. Mind you, the same test could nearly always be made by doing LASTX, X=Y? and register Y would not have to be used just to hold an "L". However there may be times when you want to make such a test without moving the contents of the stack.

T can be used in the same way as Z. Another use for T is to check a set of values, particularly in a data file. Since the pointer is automatically updated after each GETX, you could set the pointer, put the test value in X and "T" in register Y then use the loop:

```
LBL 01
GETX
RDN
X<=NN?
GTO 01
```

This loop would extract values from the data file one at a time, put them in T and compare them with X. When the test becomes false (any test can be selected) then the program leaves the loop. This loop is useful for a data file that contains text strings. If the data file contains only numeric values then a direct test would be just as easy, as follows:

```
LBL 02
GETX
X<>Y
X>Y?
GTO 02
```

This works exactly like the LBL 01 loop above and is also one byte shorter, but X>Y? does not work for text strings. If you use non-normalised numbers (NNNs, see Section 8.2 and Chapter 14) then you will know that recalling them (even for a test) normalises them. The indirect stack comparisons are therefore very useful if you need to compare NNNs, since they do not normalise anything.

In conclusion using the stack for indirect tests can be helpful if you are comparing text strings in the stack (without using numbered data registers) and vital if you are comparing non normalised numbers.

vii) If Y contains any alpha value other than "X", "Y", "Z", "T" or "L" then the DATA ERROR error occurs. If Y contains a number, then its sign and fractional part are ignored. If the number is 1000 or more, then you get a DATA ERROR. If it is less than 1000 but greater than SIZE-1 then you get NONEXISTENT.

viii) The indirect comparison functions are particularly useful if you want to search through a set of registers. As an example, you may wish to check if the value contained in register X has already been saved in one of the registers 20 to 30. You can search through these registers one by one, using the index 20.030 in register Y and incrementing it by use of ISG Y. You could then have a loop as follows:

```
101  20.03            ; Set up the loop index
102  X<>Y             ; and put it in register Y
103  LBL 03           ; Begin loop
104  X=NN?            ; Is X equal to the next indexed register ?
105  GTO 04           ; If yes then go to say it is found
106  ISG Y            ; If not then
107  GTO 03           ; go and try the next index value
108  "NOT FOUND"      ; Get here at end of index list, so display
109  PROMPT           ; NOT FOUND
110  LBL 04           ; Get here if value was found, so display
111  "FOUND AT "      ; where it was found.
112  ARCL Y
113  PROMPT
```

Of course the index can be anything you choose, and you can check every second or third or nth value, not every single value. You can use any test you like, and you can take specific actions instead of just displaying a message at the end. Without the indirect comparison functions you could still make the same tests, but you would have to replace lines 104 and 105 with something like the following:

```
104  RCL IND Y
105  X<>Y
106  CF 10
107  X=Y?
108  SF 10
109  X<>Y
110  RDN
111  FS?C 10
112  GTO 04
```

This mimicks lines 104 and 105 exactly but is 7 lines and 10 bytes longer. The two X<>Y lines (105 and 109) are not necessary if the test is X=Y? or X=Y?, but they are necessary for the other tests.

ix) Use of the indirect comparison functions for sorting has already been mentioned but an additional sorting technique called **index sorting** can be used with these functions. Imagine you have a list of customers, and for each customer you store the following information:

/Customer no./name/tel. no./no. of next customer in alphabetic order/

If you store this information in order of customer number you may not need to store the customer number itself (it could be that customer 1 is in register 51, customer 2 is in register 52 and so customer n is in register 50+n). But what if you want to print a list in alphabetical order? You use the "next customer" pointer to go from one customer to the next, not in numeric order but in alphabetic order. These pointers (or indices) provide you with a "linked list". You can have more than one set of pointers, another set could be used to point backwards to the previous customer, another could start from the customer who owes the most and end at the one with the best credit and so on. Each linked list needs special pointers to show which is the last register of the list, which element is the first in the sequence, and also a special pointer value which says "there are no more items, this is the last one in the sequence". In certain cases it is better to use a "ring" with no first and last item. For the purposes of an example, take a simple list in which each element contains one word, and a pointer to the next word in alphabetical order. The words are in registers starting at 51 and their corresponding pointers are in registers starting at 101. Consider a list containing just the words "A", "AND", "BUT", "ΣΣΣΣΣΣ

Register number	Value in register	Register number	Pointer to next word
50	54 (last register)	100	52 (first word in list)
51	BUT	101	53 (points to ΣΣΣΣΣΣ
52	A	102	54 (points to AND)
53	ΣΣΣΣΣΣ	103	999 (last word in list)
54	AND	104	51 (points to BUT)

Each word in the list "owns" a pointer which is stored 50 registers beyond the word and which points to the next word in alphabetical order. You can add a new word to the list by putting in the next register (55 in this case), and by rearranging the pointers, without moving the words. For example, say you want to add the word "IS" to the list at the next address, and to rearrange the pointers. Put the word in the Alpha register and call the subroutine:

```
01 *LBL "ADDWRD"; Subroutine to add a new word to the list
02  ISG 50       ; Increment the address of the last word in the list
03  LBL 00       ; A NOP to follow the ISG
04  ASTO IND 50  ; Store the new word at this address
05  100          ; Register where address of first word of sequence is
06  RCL IND X    ; Get address of first word in sequence
07  ENTER        ; Lift stack so next step does not lose address
08  ASTO X       ; Put word to be tested into X
09  LBL 01       ; Start loop to compare word in X with words in list
10  X<NN?        ; Is new word less than current word in list
11  GTO 02       ; Yes, so go to put its pointer into pointer list
12  X<>Y         ; No, get word address to X
13  50           ; Add 50 to it,
14  +            ; to get address of this word's pointer
15  X<>Y         ; Put test word back in X
16  RCL IND Y    ; Fetch pointer to next word in sequence
17  X<>Y         ; Put this pointer in Y
18  GTO 01       ; Go back to repeat the test
19  LBL 02       ; Get here if new word belongs before word tested
20  RDN          ; Roll the word out of the way (it was in X)
21  RCL 50       ; Recall address of the new word
22  STO IND Z    ; Make the sequentially  previous word point to it
23  50           ; Add 50 to address of the new word,
24  +            ; to get location of the pointer it owns
25  X<>Y         ; Put this pointer address in Y and
26  STO IND Y    ; store address of sequentially next word there;
27  RTN          ; (this was the last word tested) Finished, so return.
```

Try setting up the short word list and its index list, then putting "IN" into the Alpha register and running the routine. It is much quicker than a sort that needs to move values to make space for a new one, and it works with alphabetic or numeric values. As the word "IN" fits alphabetically into the sequence between "BUT" and "THE" the pointer belonging to "BUT" (in register 101) should now point to "IN" (address 55). The pointer belonging to "IN" (in register 105) should point to "THE" (in register 105). The whole sorting operation just changes the values in registers 101 and 105. Check that this is what has happened. The original word is still in the Alpha register. If you want to sort words of more than six letters then you can write a very similar program using a text file.

The routine is rather long as it needs to set up initial and final values, but the central part is the short loop in lines 09 to 18. Various index sorting programs can be built around this loop. Index sorting is faster than normal sorting because there is no need to rearrange a set of values, only an index is changed, but it takes twice as much space as normal sorting since registers are needed for the pointers as well as for the data values themselves.

Exercises

10.A Write a routine to convert a string of upper case letters in the Alpha register into lower case letters. You will need to use ALENG, ATOX, X<>F and XTOA. If you cannot work out how to do this then look at the beginning of Section 16.3. This operation is useful for work with printers that do not use flag 13 to set lower case, and to encode text strings so they can not be read in the display.

10.B If you have done Exercise 10.A then try writing a routine to convert from lower case back to upper case. Change only those characters that are lower case letters. See if you can deal with null characters too.

10.C Write a routine using the functions STOFLAG and RCLFLAG to display a number in register X without showing its fractional part or decimal point, and then to restore the original display mode.

10.D If you have an HP-41CX try writing a program to sort a list of words which are entered into the Alpha register one at a time. Use the index sorting routine shown in Section 10.6. When you store the first word you will need to initialise the list by pointing to that word from the list start and by making the word point to a nonexistent "next word" by setting a large pointer value, such as 999 used in the example.

10.E If you have done Exercise 10.D then try writing a program to print the list of words, both in the order in which they came, and in alphabetical order. Programs like this are most useful if the words or items to be sorted come from outside the HP-41, for instance from a telephone or other device connected via HP-IL.

CHAPTER 11 - EXTENDED MEMORY

11.1 What is Extended Memory?

The largest group of functions in the Extended Functions/Extended Memory module is that concerned with using Extended Memory. The normal RAM memory of the HP-41, or **Main Memory**, as described in Chapter 8 provides numbered data registers. It is also used for programs and so on, of course. Each register can be used individually, for storing and recalling data and for doing arithmetic. In general there is no particular connection between the registers; a subroutine can use any combination of registers that seems convenient: a subroutine that uses four registers could use 2, 3, 5, and 77, just as well as registers 21, 22, 23, and 24. Sometimes though it makes sense to treat a block of several registers as one unit, for example the six statistics registers, or the registers that contain a complete program in program memory. Extended memory on the HP-41 is RAM memory used specially for handling blocks of registers, in units called **files**. A file could be made up of six registers, holding the results of some statistical calculations, it could be twenty registers long and hold a set of telephone numbers, or it could hold a program for which there is not enough room available in main memory. Each file begins with a header consisting of two registers, one to hold its name (up to 7 characters long) and the second to hold the file type, length, and the file pointer. The last file is followed by a **partition register** (or **End-of-Memory marker**).

The Extended Functions Module (and HP-41CX) contains 127 usable registers of Extended Memory. You can also plug in one or two Extended Memory modules each of which provides another 238 usable registers. If you plug in two Extended Memory modules, one must go into Port 1 or Port 3, and the other must go into Port 2 or Port 4. Never plug in two Extended Memory modules one above the other. The Extended Functions module can go into any free port though. With a full Main Memory and full Extended Memory, an HP-41 has 6,458 bytes of usable RAM in addition to the stack registers and the ROM memory. Details of how the Extended Memory fits into the RAM memory layout of the HP-41, and of how Extended Memory can be treated as if it

were ordinary Main Memory, will be given in Chapter 16.

11.2 Creating and deleting files

The HP-41 Extended functions use three types of files. Data files contain data in registers just like Main Memory, Program files contain complete programs copied from Main Memory, and text files (or ASCII files) contain text strings like those in the ALPHA register but up to 254 characters (bytes) long. This section describes how files are created and deleted. After a file has been created, it is immediately ready to be used with additional commands, so it is called the **working file** or **current file.** If you want to work with a different file then you have to use a command to select the chosen file as your current file. The "current file" in Extended Memory is rather like the "current program" in Main Memory; it is the one that you are working with (using or altering it). There is only one "current file" at a time. The safest commands to use for selecting a current file are RCLPTA, SEEKPTA, FLSIZE or EMDIR because they do not alter the file.

SAVEP is the simplest function to create a file. It copies a program directly from Main Memory into Extended Memory without changing it, creating a file just long enough to hold the whole program. The X register is not used, but the ALPHA register should contain two names separated by a comma:

NAME1, NAME2

The first name should be the name of any global label in the program you want to save. The second name should be the name of the file that you want to create. If you omit the comma and the second name then the file will have the same name as the label specified. If you omit the first name but leave the comma and the second name, then the current program will be saved in a file of the given name. If the ALPHA register is completely clear, or if there is nothing after the comma then you will get a "NAME ERR" message. The names can be more than seven characters long, but all characters after the seventh will be ignored. A second

comma and anything after that will be ignored too. A file name less than 7 characters long will be filled out with blanks on the right in Extended Memory (the Alpha register is not altered). As names shorter than seven characters are filled out with blank spaces on the right, file names such as "X " and "X " are exactly the same as "X". If more than one program contains a label corresponding to the name in ALPHA, then the one nearest to the .END. (bottom of CAT 1) will be copied. If no such label exists, you will get the NAME ERR message. If a data file or a text file with the same name exists, you will get a "DUP FL" error message, but if a program file with the same name exists, then be warned; it will be deleted and the new one will be added at the end of Extended Memory. Once a program has been saved in Extended Memory it can be cleared from Main Memory (with CLP or PCLPS) to provide more room for other purposes. If you have the older type of Extended Functions Module (EXT FCN 1B) then you should never execute SAVEP while the current program is in a plug-in module. For instance if you have been using a program in a module such as the Maths Module or one of the printing programs in the printer or HP-IL modules, always do a CAT 1 or GTO.. before doing SAVEP, otherwise you may get MEMORY LOST.

CRFLD Creates a data file in Extended Memory. The length of the data file (the number of registers in it) is taken from the value in X (sign and fraction ignored as usual). When you work out the length of the file, count just the number of registers you need; the header length is added automatically. The name of the data file is taken from the first seven characters in ALPHA. As with SAVEP, characters beyond the seventh are ignored, spaces are used for padding out a short name, and commas are treated as separators. CRFLD will not replace a file that already exists.

CRFLAS Creates a text file (or ASCII file) in Extended Memory. The name and length are given in ALPHA and X as for CRFLD, and CRFLAS will not replace a file that exists. If you want to work out exactly

how long a text file needs to be then proceed as follows.

i) Add up the number of all the characters in all the text strings you will use.
ii) Add 1 for each text string and add 1 to the final result.
iii) Divide by 7 and round up to the next whole number.

This gives the number of registers you will need, but if you are not sure exactly how much space you will need then increase your estimate by about a quarter. You can increase a file's size on the HP-41CX or with Synthetic Programming, but otherwise you will have to use one of the following roundabout methods:

A) Create a larger file and copy the text to it from the first one, using the Alpha register to copy 24 bytes at a time.
B) Copy the file to a mass medium HP-IL device, delete the old file in Extended Memory, create a new file, and copy back to the new file. See the SAVEAS and GETAS functions.

PURFL Purges (deletes) the file whose name is given in ALPHA. A name must be given, the rules about name length, use of blanks and commas apply as for SAVEP. If you have one of the older Extended Functions Modules (most of those that show up as -EXT FCN 1B in CAT 2) then PURFL has the following nasty bug. After executing PURFL you must make a file into the working file, otherwise you lose all files in Extended Memory. The bug is described in detail in Appendix C, and a way to recover from it is also described. This can be used as a quick way to clear all Extended Memory for re-use.

11.3 File pointers.

In Main Memory each data register can be identified just by its number, but in Extended Memory the registers have to be numbered separately in each file. To speed up the handling of data in the files, each file contains a **pointer** to the **current record**. In data files, each record is simply a

single data register and the pointer is the number of the register that will be read or written next. After every read or write the pointer number is increased by 1: to access any record except the next one in a data file you must reset the pointer. The first record in a file is record number 0 just as the first register in Main Memory is register 00.

In text files, the data are not stored one register at a time, and the pointers are used in a different way. A text file contains records, each of which can contain from 1 to 254 characters. A record is rather like one line on a page; it may have a few letters or a lot. You can add text to each record, or delete text, rather like adding to lines on a page, or crossing out part of a line or a whole line. Just as you can pick any letter on a page by giving the line number and saying which letter it is in the line, so you can point to a character in a record by specifying the record number and the character number in the record. The first record is number 000, and the first character in a record, counting from the left is again 000.

In general then a data file pointer is specified as a register number:

rrr

and a text file pointer is specified as a record and a character position:

rrr.ccc

In program files, the pointer cannot be used, since the file can only be read or written as a whole. A pointer is set up when a program file is written though. When a program is copied to a program file, the values of all the bytes are added together and the remainder of dividing this number by 256 is calculated. This becomes a **checksum byte** and is stored after the last byte of the program. The file pointer is set to hold the number of bytes in the program, for later use. Thus the pointer in a program file contains the length of the program in bytes. When a program is to be copied back from Extended Memory to Main Memory, the checksum is recalculated, and if it differs from the value stored then the HP-41 assumes the program file has been damaged and refuses to copy the program back. Instead it displays CHKSUM ERR. Once again, Synthetic Programming will be able to help you recover from such a problem. If you are unsure

about the meaning of bytes, reread Chapter 8. Just as the current file is the one you are working with, so the **current pointer** is the position of the pointer in that file.

RCLPT is used to recall the pointer value of the current file to register X. The contents of the ALPHA register are ignored, the stack is lifted unless stack lift is disabled. If the current file is a data file then the current register number rrr will be recalled. If the current file is a text file then the current record and character pointer rrr.ccc will be recalled, and if the current file is a program file then the program length in bytes will be recalled.

RCLPTA works like RCLPT except that it first changes the current file to that named in the ALPHA register. The file naming rules apply as usual except that the ALPHA register contents must not begin with a comma. If no name is given then the current file remains the same, so RCLPTA acts exactly like RCLPT.

SEEKPT resets the pointer in the current file to the value specified in X. Minus signs and irrelevant fractional parts are ignored. SEEKPT cannot be used with program files; it gives FL TYPE ERR, which is just as well, otherwise the checksum byte address would be lost.

SEEKPTA works like SEEKPT except that it first changes the current file to that named in the ALPHA register. The file naming rules apply as for RCLPTA, and as with RCLPTA no name makes SEEKPTA the same as SEEKPT. If ALPHA contains a valid text file or data file name but the pointer cannot be positioned as required then the file becomes the current file but its pointer position is not changed.

It should be clear that recalling the pointer or resetting it are necessary operations and we shall come back to these functions in the next two

sections. When a new data file or text file is created, its pointer is set to the very start of the file. If a different file becomes the current file, the position of the pointer in the previous current file does not change, so you can go back to the same place in the file later with RCLPTA.

11.4 Using data files.

Data can be stored in data files, and retrieved from them, in three ways; one register at a time, or in blocks of registers, or the whole of Main Memory data at once.

SAVEX does what it says: it saves the contents of register X into the current register in a data file. The pointer is then incremented by one, so that the next SAVEX would save X into the next register in the data file.

GETX gets the data in the current register and copies it from the data file to register X, lifting the stack unless stack lift is disabled. The current pointer is incremented by one.

If the last register in the data file is accessed by SAVEX or GETX then the pointer is incremented again so that its value becomes larger than the number of registers in the file. The next SAVEX or GETX will then result in "END OF FL" unless the pointer is reset first.

SAVERX saves a block of Main Memory registers according to the value in X. The value in X should be of the form bbb.eee where bbb is the register at the beginning of the block and eee is the register at the end of the block. The registers are copied to the current file (if it is a data file) starting at the current pointer. When SAVERX is finished, the pointer is set to the next value past the end of the registers saved. Minus signs and irrelevant fractional parts of bbb.eee are ignored. If eee is smaller than bbb then only register bbb is saved in the data file.

GETRX uses a value bbb.eee as in SAVERX to copy a block of registers from a data file into registers bbb to eee. The first register copied is the one at the current pointer, and the pointer is reset to the first register past the end of the block. If the data file does not contain enough registers to fill from bbb to eee then no registers are copied. (This is contrary to the statement in the Extended Functions Manual, but in agreement with the HP-41CX Manual.) If eee is less than bbb then information is only copied into register bbb.

If eee is greater than the last register in Main Memory, then SAVERX and GETRX display NONEXISTENT and do nothing.

SAVER copies all the data registers from Main Memory to the file named in the ALPHA register. If no file is named, the current file is used (if it is a data file). SAVER sets the pointer to 000 before the copying starts and sets the pointer to the register above the last one used when it finishes. This is different from SAVEX and SAVERX but it is sensible because you are likely to use SAVER only once in a data file, since there are so many registers to copy. If the file is not big enough to hold all the data registers in Main Memory then an END OF FL error occurs, no data are moved, and the pointer is not changed. This makes SAVER very safe to use, since no data can be lost, but it also is very restrictive.

GETR copies data registers from the file named in ALPHA (if it is a data file) or the current file (if ALPHA is empty) to Main Memory, starting at register 000 of the data file and at register 00 in Main Memory. The file pointer is not used, but at the end it is set to the register after the last one copied. GETR carries on copying registers until it comes to the end of the Main Memory registers or to the end of the data file. Unlike SAVER, GETR works even if there are fewer registers in the data file than in Main Memory. This allows you to change SIZE after

doing SAVER, but since it lets you lose data, maybe SAVER need not have been so strict about having to save all the registers.

Let us now look at a few uses of the three types of data saving.

i) SAVER and GETR are helpful when you want to store all or most of your data registers before changing SIZE or running a program that would destroy the data. They are easy to use because they do not use pointers and do not require you to use a separate command for selecting the current file (but the file must have been created beforehand). Consider the following: you need to set fifty registers to zero to use them for a long calculation, but at the end you want to reset them to their original values, without losing the new contents of the remaining registers. You decide to set SIZE to 100, save the contents of registers 50 to 99, clear registers 50 to 99 and use them for the calculations, then restore registers 50 to 99 to their original values, without changing the new values of registers 00 to 49.
Using SAVER and GETR you can do this as follows:

```
line   step          comments
no.
101  100          ; Set SIZE
102  PSIZE        ; to 100
103  "TMP"        ;
104  CRFLD        ; Create a data file 100 registers long
105  SAVER        ; Save all register contents
106  50           ;
107  PSIZE        ; Reduce SIZE to 50
108  X<>Y         ;
109  PSIZE        ; Increase it to 100 again, so that
                    registers 50-99 are clear.
```

At this stage you can do your calculations, using registers 50-99, for example a matrix summation and inversion. At the end you want to recover the original values of registers 50-99, without

losing the new contents of registers 00-49.

```
201  50          ;
202  PSIZE       ; Save the current contents of registers 00-49,
203  "TMP"       ; placing them in registers 00-49 of the data
204  SAVER       ; file "TMP". Registers 50-99 of the file
205  100         ; are not changed.
206  PSIZE       ; Reset the SIZE to 100
207  GETR        ; This restores registers 00-49 to their
208  PURFL       ; latest values, and 50-99 to their original
                   values. Then we delete the data file.
```

ii) The main disadvantage of SAVER and GETR is that they need a large file in Extended Memory. SAVERX and GETRX are more selective and allow you to save or recover only those registers that need to be copied. In a way SAVERX and GETRX are like REGMOVE; they copy a block of registers, but to and from Extended Memory, not to another region of Main Memory. Since the example above really only needs fifty registers, let us see how it can be worked with SAVERX and GETRX using a file only fifty registers long.

```
101  100
102  PSIZE
103  50.099      ;
104  "TMP"       ; Create a data file 50 registers long.
105  CRFLD       ; CRFLD ignores the fractional part.
106  SAVERX      ; Save registers 50 to 99 in the file.
107  PSIZE       ; Clear registers 50 to 99 by
108  100         ; setting SIZE to 50, then back
109  PSIZE       ; to 100 again.
```

(If you have a 41CX you can replace lines 107-109 with CLRGX.) When you have finished your calculations you can recover registers 50 to 99 as follows:

```
201 0
202 "TMP"        ; Select the "TMP" data file as the current
203 SEEKPTA      ; file, and set the pointer to 0.
204 50.099       ;
205 GETRX        ; Recover registers 50 to 99.
206 PURFL        ; Delete the file.
```

This takes two program lines fewer than the first version, but it is 4 bytes longer (unless you have an HP-41CX, in which case it is 1 byte shorter).

iii) SAVEX and GETX are more useful when you want to store one value at a time, not a whole block of registers. For example you may want to save a set of results, calculated one at a time, but in a place that will not be in danger of being altered by subsequent calculations. A data file built up by SAVEX would be ideal for this.

Another use for SAVEX and GETX is to save the contents of the stack in a data file. The stack cannot be treated as a block of registers (unless you use Synthetic Programming), so it cannot be saved with SAVERX. The following can be used instead to save the stack:

```
101 SIGN         ; Save X in register L
102 RDN
103 4
104 "STAK"       ; Create a file of 4 registers
105 CRFLD
106 R↑
107 SAVEX        ; Save T
108 R↑
109 SAVEX        ; Save Z
110 R↑
111 SAVEX        ; Save Y
112 LASTX
113 SAVEX        ; Recover X from L and save it
```

At this stage registers X, Y, Z, T are unchanged and have been saved in the data file "STAK". Note that register L is altered and is not saved.

To recover the stack use:

```
201 0
202 "STAK"    ; Select the "STAK" data file and set the
203 SEEKPTA   ; pointer at its start.
204 GETX
205 GETX
206 GETX      ; Get T, Z, Y, X.
207 GETX
208 PURFL     ; Delete the file.
```

At this point, X, Y, Z and T have their former values again. Note that register L is not recovered.

CLFL Should you wish to delete the contents of a data file and replace them with all zeroes then put the file name in ALPHA and execute CLFL (clear file). The named file becomes the current file and the pointer is set to zero. CLFL also clears text files; this will be described under POSFL in the next section. CLFL needs a file name; it will not just clear the working file.

11.5 Using text files.

You can change a text file much as you can change a page of text. Just as you would change a few letters in a line, so you can use functions that change a few characters in a record. Just as you can remove or add a few lines on a page so you can use functions to remove or insert records.

APPREC This adds (appends) the contents of the ALPHA register as a new record at the end of the current file (if it is a text file). You can use this to put some records into a newly created file which is empty to begin with. You can also use it to add new records at the end of a file that already exists. You do not

need to reset the pointer before using APPREC, but afterwards it is positioned at one character past the end of the last record in the file. This allows you to add more text to the last record without resetting the pointer. You may need to do this often since the ALPHA register can only hold 24 characters at a time which may not be enough for a whole text record. Since the contents of ALPHA are used to add text to text files, they cannot be used at the same time to select the text file as the current file. You must use a separate command such as RCLPTA, SEEKPTA, or FLSIZE. On the HP-41CX you can use EMDIRX too. (The HP-41CX contains an additional "text-editor" function ED which you may find easier to use than the functions described here. ED is described in Section 11.9. It can only be used for manual alteration of a text file.)

INSREC This inserts the contents of the ALPHA register as a new record immediately before the current record (the one pointed to by the pointer in the current file, assuming it is a text file). INSREC can be used to insert extra records at any time, but it is particularly useful when you are adding records to a file with APPREC and realise you have just missed a record. Without resetting the pointers you can put the record you skipped into ALPHA, then execute INSREC, and the record will be put into the file as the next-to-last record. After INSREC the pointer is positioned at one character past the end of the record inserted, so that you can add more text to the same record.

DELREC This deletes the record pointed to by the pointer. The pointer is then set to the first character of the next record which means the record number is not changed but the character pointer is set to 0. Another DELREC will therefore delete the next record (if there is one), but an INSREC will put a new record in the place of the one just deleted. You may think that it would be difficult to find the record you want to delete, but wait until you read about POSFL.

The next three functions work like APPREC, INSREC and DELREC but handle characters within a record. All work on the current file, but only if it is a text file.

APPCHR appends the contents of the ALPHA register to the end of the current record in the current file and moves the pointer to one character past the end of the current record. This function lets you build up records longer than 24 characters, by using APPREC to create a new record, and APPCHR to add to it.

INSCHR inserts the contents of the ALPHA register before (to the left of) the current pointer. The current pointer points at the same character as it did before INSCHR was executed. If the current pointer is at the end of a record (one character past the last character the in record) then INSCHR works just like APPCHR. Like APPCHR, INSCHR can be used repeatedly to add characters to a record, but at any point in the record.

DELCHR A little different from the other functions because it uses the number in the X register and deletes that many characters from the current record, starting at the character pointed to by the current pointer. The sign and fractional part of X are ignored as usual. The current pointer value does not change, so after DELCHR it is pointing to the first character past those deleted. If there are fewer characters available to delete than the number in X then all the characters from the current pointer to the end of the record are deleted and the pointer is set one character past the end of the record. (No error message is generated.) If the number in X is greater than or equal to 1000 then no characters are deleted and the "DATA ERROR" message appears. (This fact is worth noting here because it does not appear in the manuals.)

If any of the functions APPREC, INSREC, APPCHR, INSCHR fail because

there is not enough room in a text file then they generate END OF FL. The pointer is not changed.

POSFL You may have been wondering how you can find the place at which you wish to make changes in a file. This is accomplished by use of the function POSFL. POSFL goes through the current file, starting at the current pointer and searches for a text string that matches the contents of ALPHA. If a matching text string is found, the pointer is set to the first character of that string, and the pointer value is recalled to X, lifting the stack (unless stack lift is disabled).

If you want to change SMITH to SMYTHE in the current file you can do the following:

i)	0,SEEKPT	to go to the start of the file
ii)	"SMITH", POSFL	to find the offending text string
iii)	5,DELCHR	to delete the string (you can use ALENG instead of 5 if you do not know the length of the string)
iv)	"SMYTHE", INSCHR	put in the new string.

Note that the pointer operations are designed to let you do all this without having to reset the pointer yourself.

If more than one string matches the contents of ALPHA, POSFL locates the first matching string at or after the current position. If you have found a copy of the text string and want to locate a second copy then you must move the pointer forwards. Otherwise you will find the first copy again and the pointer will not have moved.

If the string is not found then the pointer is not changed and POSFL returns -1 to X. This happens if the string does not exist in the file. It also happens if the ALPHA register is empty, or if the search starts at a point in the file past the last occurrence of the string. POSFL should put this -1 into X and lift the stack in order to change the stack in the same way whether the string is found or not. Unfortunately this does not happen if you have the older EXT FCN 1B module. In this case, the -1 overwrites

register X and puts the previous value of X into register L. This means that you have to be very careful if you are using POSFL while some important values are stored in the stack. Any program you write can be made to work with POSFL on a given HP-41, but it may fail on another HP-41 which has a different version of the Extended Functions and therefore alters the stack in a different way. To get around this execute ENTER immediately before POSFL. The stack will lift, stack lift will be disabled, and the pointer value or -1 will always be written into X without a second stack lift. Only the value in register L will differ from one HP-41 to another, the rest of the stack will be the same whether you have an older module, a newer module or an HP-41CX. If the string in ALPHA is not found then the pointer position is not changed.

POSFL will not find a string which is split up between two records. Say one record ends in the word SAD, and the next begins with AMAZING. The result is that four consecutive characters spell out the word ADAM, but POSFL will not find this, because it only searches one record at a time. POSFL also gives up searching a record as soon as the piece remaining to be examined is shorter than the contents of the ALPHA register. If you are looking for the name FRED and a record contains the word DOMINIC then POSFL will make just four tests, comparing FRED against DOMI, OMIN, MINI, and INIC. On some computers, you would save time by testing a shorter string, just FR for example. With POSFL this actually takes longer because six unsuccessful tests will be made instead of the four if you used FRED. You will therefore only save time testing short strings in preference to long ones if you use very long records in your text files, or if the text string to be found is near to the current position. As with data files, you can delete the contents of a text file by putting the file name in the ALPHA register and executing CLFL. A file name is required. The number of records in the file is set to zero, the pointer is set to 0.0 and the named file becomes the current one.

How can you get the contents of a text file back into the HP-41? The obvious answer is to use the ALPHA register again, but the records you want to recall may be over 24 characters long. The two functions that let you

recall text therefore put a whole record into ALPHA if they can, and clear flag 17 if they have got to the end of a record. If they have filled ALPHA but there is more text still left in the record then flag 17 is set. You can use this to check if a record is finished, and you can also use it with the HP-IL function OUTA. HP-IL devices such as printers check if flag 17 is set, and attach a carriage return and line feed (for printers) to the end of a character string only if flag 17 is clear.

GETREC, ARCLREC GETREC clears the ALPHA register first and then copies characters to ALPHA, starting at the pointer and finishing after 24 characters or at the end of the record if that comes sooner. ARCLREC works like GETREC except that it does not clear the ALPHA register first. ARCLREC is therefore similar to ARCL which also appends to what is already in ALPHA. (ARCL does not stop when the ALPHA register is full though.) Obviously if the ALPHA register is already full ARCLREC will not change the ALPHA register or the pointer.

ARCLREC and GETREC reset the pointer to one character past the last character copied. If the end of the record has been reached they do not change the record pointer but set the character pointer to one character past the end of the record. If the pointer is not changed in the meantime, then the next call to ARCLREC or GETREC will begin copying from the start of the next record (if there is one).

SAVEAS GETAS Instead of copying from a text file to the ALPHA register piece by piece you can copy the whole of a text file directly to mass media on an HP-IL device. Later on you can copy the whole file back to Extended Memory. To copy a text file to mass media you must first create a data file on the medium using the HP-IL function CREATE. (Put the file size, as a number of registers, in X, and the file name in ALPHA, then XEQ "CREATE". Note that a text file copied to Mass Storage might need up to 2 more bytes per record than the file in Extended Memory.) You could also use

a file that already exists on the mass medium. Then put the Extended Memory text file name in ALPHA followed by a comma and the mass medium file name and execute SAVEAS. The text file will be copied to the mass medium, starting at the beginning of the source file (in EM) and at the beginning of the destination (mass medium) file. To read a text file from a mass medium to Extended Memory simply reverse the procedure. First create an Extended Memory text file to copy the source file into (use CRFLAS unless the file already exists). Then put the source file name, a comma, and the destination file name in the ALPHA register and execute GETAS. Note the following points:

i) A source file name must be given. The destination file name can be omitted completely if it is the same as the source file name. A comma on its own after the source file name gives rise to NAME ERR.

ii) Copying begins at the start of the source and destination files and continues up to the end of the data in the source file. If the destination file is too small then as many complete records are copied as possible, then an END OF FL error occurs.

iii) The previous contents of the destination file are lost. If the destination file on a mass storage device was previously a data file (DA) then it becomes an ASCII file (AS).

iv) SAVEAS makes the source file into the current file but it does not change the position of the pointer in the Extended Memory file. GETAS makes the destination file the current file and leaves its pointer set at the beginning of the file.

v) If an HP-IL module is not plugged into the HP-41 or if there is no mass storage device attached to the HP-IL then SAVEAS and GETAS stop with the error NO DRIVE.

It may seem odd that these two functions which use HP-IL are part of the Extended Memory functions. In fact though, the HP-IL Module works only with HP-41 Main Memory, saving data registers, key assignments, programs, status registers or all memory. Program files and data files can be copied from Extended Memory to Main Memory and from these to mass storage, but text files are not easily copied to Main Memory. That is why these two extra functions are provided with the Extended Functions. No corresponding functions have been provided for copying text files to the Card Reader, but many people have used Synthetic Programming to achieve this (once again, see Chapter 16).

Instead of using SAVEAS to save a text file you may want to print or view the file. You can view a text file on the HP-41CX by using the text editor ED, but it is important to be able to print a file if you have been using ED to make notes to yourself. Here is a short program that will print any text file on any printer, using the full page width. The file will also be displayed 24 characters at a time, so you can use the same program with and without a printer; flags 21 and 55 are used to make sure the viewing and printing do not interfere with each other. To use the program just put the file name in the Alpha register and execute PRFL. The program first advances the paper by one line to print anything that may be left in the print buffer. Then it prints the file name, advances another line, and prints the file contents. Each record is separated from the next one by a blank line; this is done by using flag 17 which is set unless the end of a record is found by GETREC. The program is designed to be as short as possible - it does not print record numbers, and records that do not fit on one line overflow onto the next line or lines without hyphenation. Flag 25 is cleared when the program tries to read past the end of the file; at that stage the printer is advanced again and the program stops. A zero is left in register T but the rest of the stack and register L are unchanged, flags 17 and 21 are cleared.

```
01♦LBL "PRFL"
02 CF 25
03 CF 21
04 0
05 SEEKPTA
06 RDN
07 AVIEW
08 SF 21
09 ADV
10 FS? 55
11 PRA
12 ADV
13 SF 25
14♦LBL 01
15 CF 21
16 GETREC
17 FC? 25
18 GTO 02
19 AVIEW
20 SF 21
21 FS? 55
22 ACA
23 FS? 17
24 GTO 01
25 FS? 55
26 PRBUF
27 ADV
28 GTO 01
29♦LBL 02
30 ADV
31 END      59 BYTES
```

11.6 Using program files

The only things you can do with a program file is create it, delete it, find its size or copy it back to main memory. The first three have already been dealt with, the last will be dealt with here.

GETSUB To copy a program back to main memory, put the file name into the ALPHA register and execute GETSUB. This adds an END to the last program in memory (even if the last program was empty and contained only the .END.). The .END. is converted to an END, with a new .END. being created. GETSUB copies the file to program memory, if the file exists, and is a program file. If there is not enough room left in Main Memory to hold the program, then nothing is copied but the program file becomes the current file. In a running program, the message NO ROOM appears, but if you execute GETSUB from the keyboard then you get PACKING and TRY AGAIN.

When you save a program, some of its labels may be assigned to various keys. This information is stored with the program. If the HP-41 is in USER mode while GETSUB is executed then these key assignments are restored, but if the HP-41 is not in USER mode then the current key assignments are not changed.

GETP GETSUB puts the copied program after the last program in memory and if it is executed by a program then that program carries on running as usual. GETP puts the copied program in the place of the last program in memory. This can be used to save space if a program uses various subroutines and gets each one in turn from Extended Memory. Each time GETP is used, the last program is deleted from the bottom of program memory and the program read from Extended Memory is put in its place. If you do GTO.. before using GETP the first time then the bottom program in memory contains only the .END., and it is this empty program that is replaced when GETP executes. Normally you would execute GETP from a program other than the last one in memory. If you execute GETP from the last program in memory though, that program will be replaced and program execution will continue from the first line of the new program. (This is rather like a program using PCLPS to clear itself and then continuing execution at the first step of the replacement program, which is the .END.) One thing to be careful of is calling a subroutine from the last program in memory, then using GETP to replace the last program, then using RTN to get back to the last program. If you do this then you can finish up anywhere in the replacement program, or even beyond the .END. . Apart from these differences, GETP acts in the same way as GETSUB.

You can use GETSUB and GETP in various combinations to call up different subroutines and execute them one after another. You may want to do this if there is not enough room in Main Memory for all your routines at the same time, or if more than one subroutine has the same label and you want to be sure which one is used. Subroutines which call one another can use GETP to delete themselves and get the next one at the same time. As was mentioned under PCLPS, a subroutine can also delete itself and return to the calling program by executing CLA, PCLPS.

11.7 Checking the contents of Extended Memory

The last two Extended Memory functions let the user check the contents of Extended Memory. When you have created several files you may need to check what they are called, or how large they are, or how much unused Extended Memory is left.

EMDIR (Extended Memory DIRectory) lets you check all this information. When you execute EMDIR you get a list of files in the same way as you get a list of programs when you execute CAT 1. (Indeed CAT 4 on the HP-41CX is an alternative name for EMDIR.) The files are listed in the order in which they were created; this is also the order in which they are stored in Extended Memory. Each file name is followed by its type (D for data, A for ASCII or text, P for program), and its length in registers (not including the two header registers). If you have no files in Extended Memory then you get the message DIR EMPTY. (This is not an error, only a message.) When EMDIR has finished, it puts the number of free registers left into X, lifting the stack unless stack lift was disabled. The number of free registers is also put into X if the directory is empty; it already allows for two header registers for the next file, and is the largest number of registers you can use with CRFLD or CRFLAS. If you want to create as large a data file as possible (e.g. for use with SAVER) then do EMDIR, "NAME", CRFLD. You will get the result zero if you have 0, 1 or 2 free registers left because in all three cases there is not enough room for a header and data. If you interrupt EMDIR (see below) the number of free registers will not be put into X. When you execute EMDIR with a printer attached and set to TRACE mode you will get a printed version of the EMDIR listing.

You can interrupt EMDIR on an HP-41CX by pressing R/S. You can then use SST and BST to move among the files, or R/S to let the list run again. After you have stopped the listing you can also press the backarrow key to terminate EMDIR and to make the last

file displayed into the current file. If you press any other key except ON you will speed up the EMDIR list. Pressing ON terminates the directory and turns off the HP-41. If you allow EMDIR to finish, the current file will be the same one as before.

On an HP-41C or CV EMDIR behaves a little differently. Pressing R/S or ON terminates the listing and makes the last file displayed become the current file (ON also turns off the HP-41). Pressing any other key "freezes" the display but EMDIR continues when you release the key. If EMDIR is allowed to run till it finishes then the last file will become the current file.

FLSIZE finds the size of a file in registers (excluding the two header registers) and puts this number in X, lifting the stack unless stack lift is disabled. The name of the file should be put in Alpha, and this file becomes the current file. If you leave the Alpha register empty then the size of the current file is returned. If ALPHA is empty and there is no current file, or if Alpha does not contain a file name then you get FL NOT FOUND. FLSIZE is a good way to select a file as the current one because it does not affect the file in any way.

11.8 Additional HP-41CX Extended Memory functions.

There are four additional Extended Memory functions, plus a text editor which will be described in Section 11.9.

EMROOM returns to X the number of unused Extended Memory registers. This is the same number as would be put in X if EMDIR (or CAT 4) were executed and allowed to run to the end. You are saved the time you would otherwise have to wait for EMDIR to execute.

EMDIRX provides information about one particular Extended Memory file, that whose number, n, is specified in X. As with RCLALM, the first item is number 1, not number 0. EMDIRX clears the ALPHA register and returns the file name to ALPHA (if the file exists). The number n is put into register L and a two-letter code is returned to X. This is PR for a program file, DA for a data file, or AS for a text (ASCII) file. If the file is not found, 0 is put in X. Unfortunately, it is possible (Synthetic Programming again) to create file types other than PR, DA or AS, and in that case the file type is returned as a number; 0 or 4 to 15. This means that a zero value in X does not always mean the file does not exist. It is wise to use ALENG to check if a file name has been returned to ALPHA. If the file is found, it becomes the current file, otherwise the current file is not changed. It may seem silly to use letters instead of numbers to describe file types, but this was done for the purpose of consistency with the HP-IL function DIR. It is also consistent with the Extended I/O Module function FLTYPE which uses the same two-letter codes.

ASROOM allows you to check how much unused room there is left in a text (ASCII) file. You can therefore check if there is a reasonable amount of room in a text file before adding to it, and so avoid running out of space during an operation. ASROOM returns the number of free bytes in the current file to register X, lifting the stack unless stack lift is disabled. The current file must be a text file. If ASROOM tells you that there is not enough room left, you can increase the size of the file by using RESZFL.

RESZFL This function changes the size of the current file to the number of registers specified in X. The size of a program file cannot be changed. The size of a text file can be increased or decreased but only unused space in a text file can be removed; an attempt to remove registers that contain text or the end of file marker results in FL SIZE ERR. The size of a data file can also

be increased or decreased. If the file size is being decreased registers are removed from the end of the file and it is possible that some data registers containing useful values will be deleted; to force RESZFL to delete non-zero data registers you must put a negative value in X.

11.9 The HP-41CX Text Editor

The text file functions let you write programs to create and edit text files, but it is difficult to see what is happening to the file. The HP-41CX has an "interactive" text editor which lets you examine and edit a text file, showing you the result of each command as you execute it. In principle a text file can contain lists of names, information, or even a letter you are writing, rather like a text file on a full-sized computer. In practice, the HP-41 is not at all well suited for writing letters or books since it does not have a typewriter keyboard, it has only a one-line 12-letter display, it has limited memory and is rather slow. A full Extended Memory could hold about two pages of text, but it would take about 10 minutes to read it all, and up to 20 seconds to insert or change a single letter. In any case the text would either have to be in upper case letters, or it would be illegible since most lower-case letters only show up as boxed stars. (See Appendix E though.)

This does not mean that the Text Editor is useless, but only that it is most suitable for purposes that fit the HP-41 design. These are, typically, keeping lists of names, notes (an electronic pocket notebook) or short memos (which can be typed out on a printer if necessary see program PRFL at the end of Section 11.5.)

A separate use is for writing programs in a language such as BASIC, LISP or FORTH. Each record can hold one statement in the chosen language, and the record number can be used as a program line number. A long HP-41 program which reads the statements one by one from a text file, and executes them, can be stored in Main Memory, so the HP-41 can be made to operate on programs in a variety of languages. Hewlett Packard has mentioned a

"language ROM" for the HP-41, but nothing has come of this. User group members however, have written their own interpreters for BASIC, LISP and FORTH. These interpreters can be obtained from User groups and a FORTH language ROM has been considered by a user group in France.

The Text Editor function ED can be executed from within a program or directly from the keyboard. Once the Text Editor has begun to operate, the keyboard is redefined to act as described on the plate at the back of the HP-41CX. This is very similar to the ALPHA keyboard layout, but it allows for the creation and deletion of records. The ON key is used to exit from ED, and if ED was executed from within a program then that program continues to execute. (ED only responds to keyboard entry, so the program will have been suspended while ED was active.)

The keyboard functions of ED are well described in the CX manuals and summarised on the plate at the back of the CX. I shall therefore not describe the operations of ED in detail, but shall mention some of its quirks. First of all, ED normally operates in ALPHA mode where all keys are available for entering text, but there is also a numeric keypad mode. This can be entered by pressing the ALPHA key, or by executing ED from within a program without having ALPHA mode set. Most people will rarely use the numeric keypad and may find it a nuisance that ED falls into this mode when executed from a program. It is generally best to precede ED in a program with AON and to follow it with AOFF.

Secondly, ED is automatically set to add characters to a record, so it does not need the Alpha Append character (Shift K). It treats this character (⊦) like any other character on the ED keyboard. A second non-standard character (⊤ called the text symbol or super-tee) is also provided by ED, more by accident than on purpose. This is used as an "empty record indicator" whenever a new record is created, but it is actually a real character. If you type some text into a new record it will replace the super-tee, but if you go into Insert mode (press SHIFT L) then the super-tee will stay in the record. This appears to be either bad design or an oversight, since an empty record indicator should be just that; an

indicator not a real character that takes up space. If you create an empty text record by pressing R/S while using ED then that record would have zero length; the Extended Functions do not expect zero length text records and this may be why a real character had to be used.

The two characters ⊢ and ⊤ are the only non-standard characters provided by the Text Editor but not by the ordinary ALPHA keyboard. Other characters such as &, ",) and (would have been more useful. To enter these characters into a text record you must exit from ED, type in their character number (as given in the ASCII table), then execute XTOA, INSCHR (or APPCHR) and go back to ED. If you need to do this often it is best to write a short program which does it all and assign it to a key.

Thirdly, if you have the cursor positioned in the middle of a record, then pressing the backarrow key deletes the current character and moves the next character into its place. Repeated pressing of the delete key therefore deletes characters towards the right. When the cursor is at the end of a record though, it deletes characters to the left, because there are none to its right. This behaviour may seem inconsistent and you should be aware of it.

Fourthly, ED can be very slow if you are editing a record near the beginning of a long file. This is because ED moves all characters after the one that has been changed, and it uses the rather cumbersome Extended Functions instead of providing its own rapid text shifting function. You should therefore avoid editing records near the beginning of a long file. It may even be simpler to create the edited text in the Alpha register and then put it all at once into the file using INSCHR. Another problem is that of getting to the start of a long record; pressing SHIFT, USER repeatedly can take a long time. You should remember the advice given in the manual, press SST, SHIFT, SST. Getting to the end of a long record is more difficult, one method is given in the answer to Exercise 11.D.

Some people consider ED an unnecessary luxury; to others it is a very useful tool. If you do not have an HP-41CX you may wish to write your own

text editor or get one from another source; several good ones have been published in user group journals. A very powerful one by Frank Wales is in DATAFILE V2N2, p23-26 (barcode was provided). If you get the PPC CJ then you will find a good short editor by Arend van den Brug in V11N6p28-30.

Even if you do have an HP-41CX, you may find it helpful to write a program around ED. This could prompt for the name of the file to be edited and create the file if it does not already exist, then execute ED. On exiting from ED, the program could insert a non-standard character, resize the file, or even print it. (A short text file printer was described in Section 11.5.) ED is versatile but you, the user, can be more versatile.

11.10 Generalised Key Assignment program - GASP

Figure 8.6 showed that a Key Assignment Register can hold a two-byte ROM instruction as well as a one-byte CAT 3 instruction. Soon after the introduction of the HP-41, Richard Nelson (founder of the User Group PPC) suggested that two-byte CAT 3 instructions could perhaps also be stored in a KAR. Instead of having a filler (04) and a prefix (such as STO), why not put a prefix and a postfix into the two available bytes?

It took a few months to do, but it worked! Not only can you assign RCL to a key, but you can also assign RCL 55 to a key if you like. This may not seem very useful, but it does save a couple of keystrokes. Assigning GTO IND X to an unshifted key saves four keystrokes (SHIFT, GTO, SHIFT, ., 6 is replaced by one keystroke). Assigning X<>IND Y saves ten keystrokes compared with XEQ, ALPHA, X, SHIFT, COS, SHIFT, TAN, ALPHA, SHIFT, ., *; this is achieved by storing the bytes CE, F2 in a KAR. The key can be used for keyboard calculations or when entering a program.

The design of the HP-41 allows this to happen, but the designers did not plan it. Any two-byte key assignment is therefore displayed as an XROM function during the key preview, since XROM functions were meant to be the only two-byte functions assigned. When the key is released, the correct function is executed, and in a program it is stored as the expected

instruction. Sometimes the XROM number corresponds to a function on a plugged-in module, and then that function is previewed, but the correct function is still executed or entered into a program. The worst that can happen is for the XROM number to correspond to a non-programmable function on a plug-in module. These are PRP and LIST on the printers, VER, WALL and WPRV on the card reader, and NEWM on the HP-IL module. If a two-byte function displays as one of these (or any other non-programmable function on a plug-in ROM - such as ZENROM - see Chapter 12), then unplug the device while you enter the function in a program.

But how do we achieve these non-standard two-byte assignments? The usual method involves Synthetic Programming as will be described in Chapter 14, but some users may want to try two-byte assignments before they read that far. The programmable version of ASN, called PASN, has already been described in Section 10.4. This section will provide a generalised version of PASN called GASN. For those users who want to call GASN from another program there is another version called GASP (Generalised ASN Programmable. Why, what did you think it meant?). This is a "generalised" version of PASN because it assigns any two-byte function, not just two-byte CAT 2 functions and one-byte CAT 3 functions.

The program does not use any Synthetic instructions but does require the use of an HP-41CX or an HP-41C or CV with both an Extended Functions Module and a Timer Module. If you do not have these you will have to wait till Chapter 14 to see how to create two-byte key assignments. To use GASN/GASP first enter the program into memory from the listing given here, or read it in from the barcode in Appendix F. GASP is not really a separate program, it is just a different label in GASN - called an "alternative entry point".

The following notes contain advice on entering and running GASN, if you are in a hurry, read note 7 which tells you how to use the program:

```
01 RTN
02↑LBL "GASETUP"
03 CLA
04 117
05 XTOA
06 SIGN
07 206
08 XTOA
09 125
10 XTOA
11 166
12 XTOA
13 "+RC"
14 RDN
15 144
16 XTOA
17 118
18 XTOA
19 R↑
20 XTOA
21 R↑
22 XTOA
23 R↑
24 XTOA
25 LASTX
26 XTOA
27 "+0"
28 CLST
29 1.0121
30 R↑
31 XYZALM
32 "GASETUP"
33 SAVEP
34 RTN
35↑LBL "GASN"
36 "PRE↑POST↑KEY"
37 PROMPT
38↑LBL "GASP"
39 INT
40 ABS
41 CHS
42 "+"
43 PASN
44 CHS
45 PASN
46 "GASETUP##1"
47 105
48 XTOA
49 RDN
50 12
51 XTOA
52 RDN
53 43
54 X<>Y
55 -
56 ST* X
57 1
58 XTOA
59 XTOA
60 X<Y?
61 ST- L
62 RDN
63 X<> L
64 10
65 /
66 FRC
67 160
68 ST* Y
69 X<> L
70 INT
71 +
72 240
73 XTOA
74 R↑
75 XTOA
76 R↑
77 XTOA
78 R↑
79 XTOA
80 R↑
81 R↑
82 XTOA
83 R↑
84 XTOA
85 R↑
86 8
87 +
88 XTOA
89 XEQ 01
90↑LBL 01
91 PCLPS
92 END      212 BYTES
```

1) The program uses R↑ and XTOA a lot so you will find it useful to assign these functions to two keys if you are entering the program from the keyboard. Once you have entered the whole program, check carefully that it is exactly as on the printout. Be particularly sure that lines 13, 27, 32, 42 and 46 are entered correctly as text lines.

2) PACK or GTO.. after you have entered the program to eliminate all the null bytes. If you like, you can save the program on an HP-IL device, or on a magnetic card (it only takes up one card).

3) The first part of the program, lines 02 to 34 is a setup routine. You only need to run this once. Execute "GASETUP", and a copy of the whole program will be saved as a file called "GASETUP" in Extended Memory. This routine also creates a timer alarm with a very unusual message. The contents of this message will actually be executed as a subroutine by GASP. The alarm is set to go off on Jan 1 2100. This should make sure that the alarm message is not lost unless you take out the Time Module or do a MEMORY LOST, and that no other alarms come later than it. If you do have an alarm in your HP-41 set to go off after the year 2099 then change line 29 of the program to 1.012199.

4) When the program runs, it deletes itself with PCLPS, then the subroutine in the alarm replaces it with GETP. Unfortunately, PCLPS will delete any other programs that come after GASP in main memory. GASP should therefore be the last program in CAT 1 unless you are happy to lose subsequent programs. You can put programs before GASP by doing GTO GASP, GTO.000, entering the new program, and following it with an END (XEQ "END"). This will create a program before GASP, separated from it by an END. You can also delete GASP with CLP or PCLPS, put any new programs at the end of memory, then get GASP back by putting "GASETUP" into the ALPHA register and executing GETSUB.

5) GASP will not work if any buffers other than the timer buffer exist in memory. Buffers were described in Section 8.3, they can be created by the Plotter, HP-IL Development and CCD Modules. If you use any of these regularly, you should make sure that you delete all buffers that these modules create by putting the necessary instructions after line 38 of the program GASETUP. For a Plotter Module, put PCLBUF between lines 38 and 39. For an HP-IL Development Module, put 0, BSIZEX, RDN in the same place. The CCD module function CLB can delete any buffer.

6) The program first uses PASN to create two normal key assignments, then replaces the last KAR with an alternative KAR set up to produce the

chosen two-byte assignment. Normally the last KAR contains one or both of the normal assignments just made so the special assignment will successfully replace the normal assignments. This will not work if you have any space left from cancelled key assignments, or if you use GASP to assign to a key already used for an assignment. In either of these cases, the assignments made by PASN may not go into the last KAR. The last KAR will then be replaced by the alternative KAR, so the last two assignments will be lost. Finally GETP will try to tidy up the key assignments in case there are any global label assignments in the program just read. This will nullify the two that were originally in the last KAR and also any assignments to the key used for GASP if GASP has been assigned.

You can check if the assignment has worked by pressing the key in USER mode. If the assignment has not worked, you should try again by re-running GASP. When you have rerun GASP enough times to fill in all your cancelled key assignments, the assignment you want to make will work. Alternatively you could cancel all previous key assignments before running GASP; just use CLKEYS. Once you have made the required key assignments with GASP you can recreate any other assignments in the normal way.

7) Now for the actual running of GASP (or GASN). Before you use GASP or GASN you must run the setup routine. Simply XEQ "GASETUP". [Step (1) above explains what this does; GASETUP can be called as a subroutine.] You will not need to do this again unless you clear the alarm, remove the Time Module or do a Memory Lost, or reach the date 1.012100. It will not cause any trouble if you run GASETUP a second time, but will waste space by creating a second alarm. If in doubt, run GASETUP again.

You can then run GASN, which is best run from the keyboard, or GASP which is more suitable for use as a subroutine called by another program. GASP does exactly the same as GASN but it expects the input values to be in the X, Y and Z registers whereas GASN prompts for

these values. To run GASN press XEQ "GASN", or if you have just run GASETUP from the keyboard you can simply press R/S.

GASN prompts for a prefix, a postfix and a key code. The prompt PRE↑POST↑KEY is a very concise way of doing this, copied from a key assignment program in the PPC ROM. (See Chapter 12; the PPC ROM is full of excellent ideas of this sort. This particular idea came from earlier programs by W. Wickes.) You should put in a prefix number, press ENTER↑, put in a postfix number, press ENTER↑ again, put in the keycode, and press R/S. For example, to assign RCL IND X you should first look up the prefix in Table 8.1 and the postfix in Table 8.2. The prefix for RCL is hex 90, which is decimal 144. The postfix for IND X is hex F3, decimal 243. If you want to assign RCL IND X to the RCL key, you can check that this key is row 3, column 4, keycode 34. Therefore you should press 144, ENTER↑, 243, ENTER↑, 34, R/S. GASN will run for 20 seconds, then stop with an odd-looking value in the display. If you have deleted the file "GASETUP" from Extended Memory, then GASN will stop with a NONEXISTENT or DATA ERROR mesage. Press GTO.. or use Catalog 1 or an Alpha GTO to get back to program memory. The assignment will have been made despite the absence of the "GASETUP" file.

To check if this assignment has worked go into PRGM mode, then press and release the RCL key. When you press the key, you should see XROM 03,51. When you release the key, you should see RCL IND X. Delete this step, otherwise it will mess up the program. As another example, assign XEQ IND Y to the XEQ key. From Table 8.1 the prefix for XEQ IND is hex AE, decimal 174. The postfix for XEQ IND Y is decimal 242 as you can see from Table 8.2. (Remember that the same prefix is used for GTO IND and XEQ IND, but the postfix for GTO IND is from the first half of the byte table. GTO IND Y would be 174, 114.) The XEQ key is row 3, column 2, so the keycode is 32. To make the assignment, XEQ "GASN", reply to the prompt by 174, ENTER↑, 242, ENTER↑, 32, R/S. Once again you can check the assignment by pressing the key in PRGM mode, but remember to delete the step afterwards.

8) GASN is also generalised because it can assign instructions made up of unusual combinations of prefixes and postfixes. When you press FIX on the keyboard, you can only follow it with one of the keys 0 to 9. GASN frees you from this limitation. You can create key assignments such as FIX e or TONE X. Try using the programmable version GASP to create the key assignments FIX 20 and TONE E. Delete the GASN program from memory (XEQ, "CLP", "GASN") and put in the following program.

LBL "DONE", "GASETUP", GETSUB, 156, ENTER↑, 20, ENTER↑, 62, XEQ "GASP", 159, ENTER↑, 106, ENTER↑, 63, XEQ "GASP", "DONE", AVIEW, PCLPS, .END.

(You should replace 62 and 63 by keycodes for two keys that you have not yet used.)

Run this program, then go into USER mode. Press TONE E to obtain a lower TONE than is normally available. Press FIX 20 four times and see the angle mode toggle through RAD, GRAD, RAD, DEG and over again while the display stays set to four digits.

9) The four key assignments used in the examples are all used often enough by some people that they may want to assign them. RCL IND X can be used whenever two levels of indirect addressing are needed. To sort an array you can create a second array of numbers giving the addresses of the items in sequence, without sorting the items themselves. (This is much faster than sorting a long list of items; it is called "index sorting" and was described in detail at the end of Chapter 10.) To get the nth item, you can put n into X, press RCL IND X once to get the address of the item (assuming the list of addresses starts in register 1), then press RCL IND X a second time to get the nth item itself. If the list of addresses starts at an address other than 1, just add that address to n.

XEQ IND Y can be used to execute a chosen subroutine which uses the value in X. For example you may wish to calculate SINH X at one time and ASINH X at another time but in the same place in a program. Put

"SINH" or "ASINH" into Y, put the value in X, and do XEQ IND Y. The subroutine name in Y can also be a local label number. XEQ IND Y is better than XEQ IND nn if you do not want to alter any numbered data registers.

FIX 20 works like FIX 4 but toggles the angle mode. You can use your favourite FIX n, ENG n, or SCI n display and by adding 16 to n you can make it serve the extra purpose of toggling the angle mode. This is very useful for keyboard calculations. Press the assigned key once to obtain your preferred display setting, and press it a few more times to get the desired angle mode. All on one key! The postfix 20 is not ideal, because it can cause changes between FIX, SCI and ENG modes. See Section 15.2 for more details and a better postfix. TONE E will startle some people both by its display and by the sound it makes. Hewlett-Packard consider such tones as frivolous and indeed their main use is in games. But why should a serious HP-41 user not be allowed to play games from time to time? TONE E does also have a serious use. On older HP-41s it sounds for about as long as the normal tones, on HP-41s made more recently it lasts 2.2 seconds. This means that TONE E can be used as a quick test of the date of manufacture of the HP-41 internal ROMs, for instance to check if the ROMs have been updated during a repair. Special tones have also been used for serious purposes such as a darkroom timer or acoustic control of equipment.

10) GASN can be used to assign any combination of prefix and postfix and then to execute it or put it in a program. Further uses to which such combinations can be put will be described in Chapter 14. For now you can test the results of such combinations, but avoid the prefixes 145 or 206 followed by postfixes 122 to 127. The use of these operations can easily lead to MEMORY LOST. A few other combinations can sometimes cause trouble by disrupting the global linkage in CAT 1. If you make any mistakes in entering the program, or if you accidentally alter it before running it you may also get into trouble. The moral is: always make a copy of the HP-41 memory contents before you start

exploring. Best of all do a WALL onto magnetic cards, WRTA onto an IL storage device, or at least print out any important programs or data. If you want to follow GASN or GASP you can SST through the programs. Always SST until you come back to the line 01 RTN, and do not stop part of the way through. Sometimes you will have to wait a few seconds for some steps to work. Do not stop when you reach the .END., press SST again and see some unusual steps. It is most important that you do not alter the stack between the first time you see X<>c and the second time it occurs.

11) Whether you give a keycode for a shifted key or an unshifted key, GASN assigns the function both to the shifted key and to the unshifted key. This is a compromise to make the program shorter and faster. When you have made all the assignments you want to make with GASN, you can cancel any unwanted assignments, or assign something else to these keys using ASN or PASN.

12) A note for experienced Synthetic Programmers. This program may seem inelegant and some features such as (11) above may look inconvenient. It cannot be used to make assignments to the SHIFT key. The point of this program though is that it can be entered by any user since it contains no synthetic instructions and it is not so long as to discourage new users. It only uses one bug (Bug 11, see Appendix D), which HP are very unlikely to correct; the same bug even exists in the HP-71B (see Error Number 23 in the HP-71 Manual). The program works by clearing itself and falling into a program in an alarm message, so make sure there is nothing between the .END. and the top of the Alarm buffer.

13) Nothing has yet been said about possible problems other than point 5. The first problem that can occur is that there may not be enough room for the alarm created by the setup routine. The alarm requires three registers, or five if no other alarms exist. Make this space by reducing SIZE by three (or five), or by deleting some alarms, or an unwanted program.

The second problem may be that there is not enough room in Extended Memory to save the program. It takes 31 registers, plus a two register header. If you are very short of space in Extended Memory you can take lines 03 to 34 out of the program and put them in memory as a separate program (with a different label, not "GASETUP"). Do this by deleting line 1, replacing line 2 with a different label, putting an END after line 33, then putting LBL "GASETUP" between the RTN and LBL "GASN". The shorter version of GASN is missing the setup routine, but it is now 21 registers long and the setup routine need only be run once.

Next you may give an illegal keycode to GASN. If the program stops with "KEYCODE ERR", press the backarrow key to remove the message, and again to delete the illegal keycode, then put in the correct keycode and press R/S. If you are not sure how to work out a keycode, use the Extended Function GETKEY. Note that keycodes or prefixes and postfixes can have fractional parts or be negative, but the sign and fraction will simply be ignored except for the sign of the keycode.

There may not be enough room in memory to put in the extra assignments. In this case you will see PACKING, TRY AGAIN. Press R/S and the program will continue if the PACKING created enough space. If not, reduce the SIZE by at least one, or delete a program and rerun GASN (or GASP) from the beginning.

If the prefix or the postfix are greater than 255, the program will stop with DATA ERROR. Run the program from the start, making sure that the prefix and postfix values are valid. A few other problems may turn up, such as running GASN without the alarm (because you have removed the Time Module and lost the alarms), or without an Extended Functions Module in the HP-41. Alarms are deleted whenever the Time Module is removed and the HP-41 is turned on. The alarm can also be lost if you cancel it or accidentally set the date to be past the year 2099. You should avoid all these conditions, and rerun GASETUP if need be, otherwise you will get some thoroughly unpredictable results.

All the above may seem a lot of work, but in practice you should be able to run GASN easily enough after just reading Step 7 of the instructions.

An explanation of the way GASN works and of the Synthetic Programming principles used will be given in Section 16.4. Yes, SP is involved, even though it is not directly visible in the program listing.

Exercises

11.A Write your own routine to create a data file long enough to save the stack, and to save the stack in it. Test your routine and compare it with the one in Section 11.4, and with the one in Section 13 of the HP-41CX manual if you have one.

11.B Write a routine which prompts the user for the name of a subroutine which is stored in a program file, then gets that subroutine, executes it, and clears it. You will find a routine like this useful if you write long programs which do not fit into Main Memory all at once and have to get subroutines from Extended Memory.

11.C Write a subroutine to insert the character "&" at the pointer position in a text file. Then rewrite the subroutine to insert any character whose ASCII code is given in register X.

11.D Do you remember how to get to the beginning of a long record while using the HP-41CX text editor? Can you think of an alternative? To get to the last character of a long record you may have to push SHIFT, PRGM up to 21 times. One way that is quicker is to exit from ED, put a character in the Alpha register, then execute APPCHR, CLA, ED. It may be worth writing this as a short routine and assigning it to a key. Once you are back in ED you can delete this last character unless you need it. Can you think of a different way to get to the end of a long record quickly?

CHAPTER 12 - PERIPHERALS AND PLUG-IN MODULES

12.1 More programs, more equipment.

You can often avoid the effort of writing a long and complicated program by buying a plug-in module which contains a program that does the required job. This type of module is small and light, it does not take up any extra space since it plugs completely into the HP-41, and the programs in it are ready to be used at once. The more complicated programs even come with keyboard overlays that tell you which keys to press. If you buy a fairly expensive piece of equipment such as an HP-41, then it makes good sense to buy a relatively less expensive module which will let you start using the HP-41 in your own special field immediately. A question that may bother you is "which module or modules do I need?". Sections 12.5 to 12.9 describe the program modules and other other types of module available.

If you need to print, plot or display some information, record some data or read it then you need an Input/Output device. These were introduced in Section 2.7 but more details are needed. You can even send data or use your HP-41 to control laboratory or field equipment through the Interface Loop. Sections 12.2 to 12.4 provide information for prospective buyers and useful details, especially those that are hard to find in manuals.

12.2 Printers and display devices.

The original HP-41 printer, the HP82143A, plugs directly into an I/O port and acts as if it was part of the HP-41. It prints on blue or black thermal paper which can take a maximum of 24 characters per line. Paper sold by companies other than HP can be used, but unless the right kind of paper is used then the print head will wear out prematurely and you may pay more to repair it than you saved on the paper. Black paper produces copies that are more permanent and photocopy more easily, but all thermal paper fades with time, so it is best to photocopy any important printouts or plots. Follow the advice in the handbook when loading paper, a sharp crease with the heat sensitive surface on the outside and at least one inch

folded over helps you to push the paper into the slot. Avoid using the beginning of the roll with glue on it; this may seem wasteful but it is less of a waste than paying for the print head to be repaired. If the paper jams you can slide out the clear window and pull out the paper, but be careful not to damage the print head. Black paper is less likely to jam because it is stuck down with just a small piece of tape at the beginning of the roll.

The printer has three kinds of function - direct print functions, print buffer functions which store data in a special memory area built into the printer, then print the data from that buffer, and graphics functions which create and print special characters and graphs.

Four flags are used with the printer. Flag 55 is set to inform the HP-41 that a printer is present, flag 21 is set to enable print functions; if it is cleared then running programs ignore these functions, but they can still be carried out from the keyboard. If flag 21 is set but flag 55 is clear then the HP-41 assumes that you want to make a record of any data that is displayed, but that there is no printer; it therefore halts execution of a running program at every VIEW or AVIEW. Flags 55 and 21 are set by the HP-41 if a printer is present, even if it is turned off. Printer instructions are recorded in a program, even if the printer is off, but it is unwise to attach a printer and leave it off; this can lead to a PRINTER OFF error message and stops running programs. When flag 55 is set the HP-41 sends everything to the printer in case it is in TRACE mode (see below). This slows down running programs; Synthetic Programming can clear flag 55 and disable print operations to speed things up. Setting flag 12 causes characters to be printed double wide. Setting flag 13 causes letters to be printed in lower case mode - other characters are not affected. Characters that are put into the print buffer will print according to the way these flags were set when the characters were put into the buffer.

The printer has a switch to set one of three modes, MANual, TRACE, and NORMal. In MAN mode nothing is printed unless a print, VIEW, AVIEW or

PROMPT is executed from the keyboard, or from a program with flag 21 set. In NORM mode, numbers, text and instructions are also printed when they are keyed in, even if you are entering a program. In TRACE mode everything that is keyed in is printed, along with intermediate and final answers, and with each step of a running program in RAM. Running programs in ROM modules are not printed - you cannot change them anyway, so why trace them? (If you must trace them use SST.) Output from CAT functions (and EMDIR and ALMCAT) is printed if you set TRACE mode.

The following are direct print functions. **PRX, PRA, PRSTK, PRΣ** print the contents of X, of the Alpha register, of the four stack registers, and of the six statistics registers. **PRKEYS** prints all the key assignments, and **PRFLAGS** prints the status of all the flags together with other status information - the SIZE, the position of the first statistics register, the angle mode and the display mode. **PRREG** prints the contents of all the data registers (the program REGS in Section 7.2 uses less paper because it prints only the non-zero registers). **PRREGX** uses a number of the form **bbb.eee** in X to print the contents of data registers from bbb to eee.

The function **PRP** prompts for a global LBL name and prints the whole of the program which contains that label. If no name is given then the current program is printed. **LIST** prompts for a three-digit number (which can be turned into four digits by use of EEX) and prints that many program lines, starting at the current line of the current program. LIST stops at an END if it reaches one before finishing, and PRP is actually a LIST which is set to print FFF (4095) lines, so it always stops at the END. Neither PRP nor LIST will work in a program, which means you cannot get one program to print another; a programmable PRP will be presented in Chapter 16. The layout depends on the print mode; MAN, NORM or TRACE. TRACE will print programs even if flag 55 has been cleared, but in an untidy layout.

Seven functions copy data into the print buffer. One advantage of using the buffer is that it automatically prints a line whenever it becomes full. It can store a maximum of 44 bytes, but one of these bytes is used for printer control when you start using the buffer and every time you change

the print type, by setting or clearing flags 12 and 13, or by changing between the use of whole characters and single print columns. A program can keep putting information into the buffer and it will be printed automatically without the need for print instructions. Once a line has been printed, the characters in it are removed from the buffer, making room for more. You can also position characters exactly where you want them in the buffer, and you can make up your own special characters or symbols and print them from the buffer. **ACX** and **ACA** "accumulate" the contents of X and Alpha in the buffer which means that they add them to whatever is already in the buffer. **ACCHR** accumulates a character whose number, from 0 to 127, is in register X. **ACSPEC** accumulates a special character built up by **BLDSPEC**. Each character is made up of seven printer columns with seven dots in each column. **ACCOL** lets you accumulate one column, with the dots to be printed specified by a number in X. **SKPCOL** and **SKPCHR** let you specify a number of columns or characters to be left empty in the buffer before more columns or characters are put in it.

Two functions print the present contents of the buffer and advance the paper by one line, even if the buffer is empty. The HP-41 function **ADV** prints the buffer such that its rightmost character is in the rightmost print position ("right justified"), the printer function **PRBUF** prints the buffer left justified. Once the information is printed, the buffer is empty. The buffer should only be used in MAN mode, because its contents will be printed whenever anything else is printed, and TRACE or NORM can print information part of the way through an accumulation. To clear the buffer without printing its contents turn the printer off and on again.

The remaining functions are for printer graphics. **BLDSPEC** is used to specify the exact shape of a character seven dots wide and seven dots high. BLDSPEC actually creates a text string up to six bytes long, and the printer interprets the text string as a special character. You can use BLDSPEC to create display characters not available from the Alpha keyboard; use CLX, ENTER - then put the character number, 0 to 127, into X and execute BLDSPEC. You can store this character or put it into the Alpha register by using ARCL. This method was used before synthetic methods or the Extended

function XTOA were available. For byte values greater than 127, use CLX, ENTER, 1, BLDSPEC, c-128, BLDSPEC, where c is the character number. A character created by BLDSPEC and stored in register 03 will be used by the graph plotting functions to specify the character used for plotting instead of a simple x.

PRAXIS is used to print the Y axis and the units for a graph. It uses values from registers 00 to 04 to specify the axis parameters. **REGPLOT** prints a single line for a graph, using the function value in X and plot parameters in registers 00 to 02. If you do not want to use registers 00 to 02, for example because your program uses them, then **STKPLOT** lets you print a single line using values in the stack (in this case the plot value is in register T, not in X). **PRPLOT** is a complete program written in FOCAL which asks the user for plot parameters and then uses the printer functions to plot a complete graph. **PRPLOTP** is a routine which does not prompt for plot parameters but expects them to be set up by a program which then calls PRPLOTP to produce a plot as would PRPLOT. PRPLOT and PRPLOTP expect the user to provide a routine to calculate function values; this routine must not do any printing of its own or it will interfere with the plotting. No graphics should be carried out in TRACE mode.

Buffer accumulation functions send a control character to the printer before the printer data, whenever the type of printing changes. All control codes are bytes greater than 127 (hexadecimal 7F). The normal printer functions ignore the top bit of any byte sent to the printer, so control codes cannot be sent by the user. However, all 8 bits are sent when a program is printed by PRP or by LIST. The printer will not print these bytes, but those which are control codes can make the program printout suddenly go into a different mode or even turn into illegible hieroglyphics. The bytes that do this are shaded in the Byte Table (Table 14.2). Only bytes from rows A, B, D, and E of the Byte Table affect the 82143A printer. Details were given in the PPC Calculator Journal V7N6P19-22, and are also in the Synthetic Quick Reference guide. Only Synthetic programs can affect the printer this way because only they can have control characters in text strings.

The HP82162A printer is an HP-IL version of the HP82143A. It has to be connected to the HP-41 through an HP-IL module but it has all the same features as the older printer. It responds to one extra printer function, **FMT** which is provided with the other printer functions on the HP-IL module. The printer functions are built into the HP-IL module, not into the printer itself, and they are described in the HP-IL module manual, not in the manuals for the various HP-IL printers. FMT can be used to justify two character strings, one on the right and one on the left, or to centre a single character string. It is actually sent to the printer as a control byte, C0. The HP82162A printer, like all other HP-IL printers, changes a few of the characters from those printed by the HP82143A, but in its Eight-Bit mode it uses a character set very similar to that of the HP82143A. The HP82162A can however be set to a different mode, called Escape Mode, in which it prints characters differently, according to the standard ASCII code. The print buffer can hold 101 bytes before it is full and prints a line, not 44 as the HP82143A.

The manual for the HP82162A gives details of the control characters to be used in both Eight-Bit mode and Escape mode. These include characters to print barcode and graphics, but the manual does not make it sufficiently clear how to go from one mode to the other, or how to select barcode or parse modes. The printer normally starts up in Escape mode, but the HP-IL module used with the HP-41 sets it to Eight-Bit mode. To get back to Escape mode (which is needed for Parse mode for example) you should select it as the primary device (this needs care if you have more than one printer on the same HP-IL loop), then send it the decimal byte 252 (or 253), followed by the bytes to be sent in Escape mode. If the HP82162A is the only printer then you can select it as the primary device and set it to Parse mode by executing the instructions shown below. Each byte is equivalent to the instruction above it, and all these instructions are in the Eight-Bit and Escape Instruction tables, but you need some examples like this one to understand how these instructions are to be used. Be careful not to confuse the letter "l" (lower case L) with the number "1" (one) in the tables and remember to send the ASCII codes for "0" and "1", not just the numbers.

Set

Esc Esc & k 1 H

CLA,PRA,252,XTOA,27,XTOA,38,XTOA,107,XTOA,49,XTOA,72,XTOA,OUTA

Following this you can print text in Parse mode (each line ends at a blank space, not in the middle of a word) by sending characters to the printer with ACA. A line will be printed whenever the buffer is full. As soon as you execute a function that instructs the printer to print something, not to accumulate it, the printer will go back to Eight-Bit mode. You can send any sequence of bytes to the printer by using OUTA, except for null bytes, which OUTA ignores. You can send a zero byte by executing the following (you may need SF 25 to avoid a TRANSMIT ERR):

SF 25, 0, ACCOL

(Use ACCHR instead of ACCOL if you are accumulating characters.)
ACCOL and ACCHR will send out a null byte, but will not send the top bit of a byte, so OUTA should be used for sending bytes greater than 127. In Escape mode you send instructions to the printer in much the same way, with the decimal value 27 as the first byte; 27 is the code for the Esc byte. As the 82162A uses more control codes than the 82143A you have more chance of getting an illegible program printout if you use Synthetic Programming. You can clear the buffer by turning the printer on and off, by executing the HP-IL function STOPIO, or by executing PWRDN,PWRUP. Turning off and on is safest; it does not alter the status of other loop devices. To print a program containing text strings with control bytes use LIST to print up to the step with the text string, then LIST from the line after the text string. The printer is reset every time a printing instruction is executed, so the second LIST will set the printer back to normal. It may be simpler to replace the string with a different one that does not contain control characters, print the program, then put back the special text.

If you do not have an HP-IL module then you will not know what all this is about, but if you have used one then you might agree with me that things are much easier when you have an Extended I/O module, except that it is expensive. "The HP-IL System: An Introductory Guide to the Hewlett-Packard

Interface Loop" may be of help if you have problems, but your nearest user group could help more.

All HP-IL printers print the time and date at the beginning of a program listing if the HP-41 has a Time module. HP-IL is less careful with flag 55 than the older printer, but this does not affect printing; if you execute a printer function or FS? 55 and flag 55 is clear then the HP-41 will check for the presence of a printer and reset flag 55 if necessary.

The larger printers, such as the HP82905B, or the Thinkjet, print on normal-sized paper, but use ordinary ASCII characters which are not quite compatible with the HP-41 characters. The worst problem is that the Sigma and Append characters are printed differently, and these are used a fair deal in HP-41 program listings. These printers will not respond correctly to the graphics or plotting functions either.

Should you wish to show what your HP-41 is doing to a group of people, or to someone who does not want to peer at a display, then you can connect your HP-41 to a "display device" via HP-IL. This device is normally a video monitor of the type used with computers. A display connected to an HP-41 will act rather like a printer, showing standard ASCII characters and not doing graphics. The display devices are driven by video interfaces sold by HP, or by interfaces made by independent manufacturers. Some of the latter can have more characters per line, and some graphics capabilities. Most video interfaces are made to work with computer monitors, not home TV sets, so they need an extra circuit if they are to be attached to an ordinary TV set.

Display devices are intended mainly to let a group of people follow an operation on the HP-41 (or some other computer such as an HP-71), for example during a lesson or seminar. To help do this the display devices and larger printers can be set to a "stack trace" mode in which they display the contents of the whole stack after each operation. Since printers and monitors do not have mode switches at the front, the print mode is selected by flags 15 and 16. The modes are set as in Table 12.1.

Mode	Flag 15	Flag 16
MAN	clear	clear
NORM	clear	set
TRACE	set	clear
ST/TRACE	set	set

Table 12.1 Flag settings for HP-IL print modes

If you want to do a lot of printing you may find the extra functions provided by the PPC ROM, the Paname ROM or the CCD Module useful. These will be described in Sections 12.7 and 12.8.

12.3 Card Reader and Wand.

The Card Reader lets you save data, programs, assignments and other status information on small magnetic cards, and read this information back onto the same HP-41 or onto another one. Most card reading operations can be carried out by just putting a card into the Card Reader, because it recognises different types of information on a card and knows how to use them. Most writing has to be done using special functions. During reading and writing operations the Card Reader will display messages asking for more cards if it needs them.

The function **WDTA** copies all the data registers onto as many cards as are needed. (Each card has two sides, or "tracks", and each track can hold up to 16 registers.) This is rather like the printer function PRREG. A second function, **WDTAX** lets you copy selected registers beginning with register bbb and ending at register eee; the number bbb.eee must be in register X. This function is similar to PRREGX.

WSTS writes one or more status cards; the first track is a record of the stack registers, the Alpha register, flags, the SIZE and the location of the summation registers. Key assignments of functions from CAT 2 and CAT 3

are recorded on track 2, and on further tracks if one is not enough. The first track actually records most of the information in the lowest block of 16 registers (see Chapter 8), which is why these are called the Status Registers. You can record just this information by pressing backarrow after writing the first WSTS track. You can read back only this information by reading the first track of a WSTS set and then pressing backarrow. Reading back the first track will reset the SIZE, so you may lose important data, so be careful. You can read back a set of key assignments, without altering the status registers by reading track 2 and any further tracks, without reading track 1. Ignore the prompt for track 1 and press backarrow when you have read all the key assignment tracks. If you want to use WSTS to record only a set of key assignments then you can write the first track, then record the second track over it on a card, saving one track. WSTS does not record CAT 1 assignments, and such assignments will not be changed when you read a set of key assignments, except that a CAT 2 or 3 assignment on a status card will replace a CAT 1 assignment to the same key.

You can cancel all your assignments, pack, make one assignment, cancel that, and record the cancelled assignment on a card (record track 02 over track 01). This idea is due to R.H. Hall (HP Key Notes V5N1P11). Whenever you read this **assignment cancelling card** it will cancel all CAT 2 and CAT 3 key assignments. You might find such a card very useful.

The WALL function saves "all" the information in an HP-41. In practice this means all data, all status registers, buffers, and key assignments, and all programs. This takes up a lot of cards, but it does not include any Extended Memory registers. You can use several WALL sets of cards, for example if several people are using the same HP-41 and each person needs to save their work. It is unsafe to interrupt the reading of a WALL set, but it does not always lead to MEMORY LOST as the manual says. Synthetic programs to save buffers and Extended Memory files on cards have been written; programs to do this are given in Chapter 10 of the book "HP-41 Extended Functions Made Easy".

To write a program onto cards you position the HP-41 anywhere in the program, set PRGM mode, and feed cards through until the Card Reader stops asking for more. It is best to PACK before saving a program, as this removes unnecessary nulls which will otherwise take up space on the cards. To read a program back, you should go to the .END. (use GTO..) and read the cards, with PRGM mode off. The program replaces the last program in memory - if you have just executed GTO.. then nothing is overwritten because the last program contains only the .END.. You can read a program without going to the .END. and it will still be read and will replace the last one. Key assignments of global labels in the program are recorded with the program; if you read it back with USER mode set then these assignments will be read back and will replace any other assignments to the same keys. If you want to know how many cards a program needs before writing it then execute WPRV (see below); this will prompt you for the first track and tell you how many tracks you need. Another way is to insert a card in PRGM mode and quickly remove it before the motor starts. Press backarrow to remove the prompt.

It is easy to forget which of these operations requires PRGM mode on and which requires PRGM mode off. Either way it can be disastrous; you may overwrite a program you are trying to save, or you may write a program on a card which contains an important program you were trying to read. The second problem can be avoided if you clip the corners of all cards containing important programs. (Such cards can be written on again only if you set flag 14. This flag will be cleared as soon as a write operation has been completed so that you will not accidentally overwrite further cards.) The easiest way to avoid overwriting programs on cards or in memory is to remember:

WRITE ON - READ OFF (Mnemonic: right on!)

You write programs <u>on</u> cards with PRGM mode <u>on</u>, and you read programs <u>off</u> cards with PRGM mode <u>off</u>. (You can also remember that you <u>r</u>ead programs to <u>r</u>un them in <u>r</u>un mode.)

To record a program on cards so that it can be read back but cannot be examined you use the **WPRV** function. Go to the program and execute WPRV;

you do not have to set PRGM mode, but you can execute WPRV in PRGM mode as it is a non-programmable function. When you read back this card the program will be **PRIVATE**. You will be able to execute it, but you will not be able to view it, alter it, copy it to another card, SST through it, or GTO any line number. You will be able to GTO and XEQ labels in the program. The main use of PRIVATE is to protect a program from accidental alteration; it could be used to protect commercial programs but synthetic programmers have found many ways to overcome this protection.

Although you can read programs and data back just by feeding the cards into the reader, you may want to carry out these operations as part of a program. **RDTA** reads a complete set of data cards and puts the data from them into memory, starting at register 00. You can read a selected number of registers, either by writing them to cards with WDTAX, or by interrupting RDTA before all the cards have been read, but the data read in begin at register 00, unless you do not read the first track. **RDTAX** uses a number bbb.eee in register X to specify where data are to be put; it can read cards recorded by WDTA or by WDTAX. Remember eee must be three digits long; use 009 for register 9 and so on. If you want to keep all the data you have in memory, but create ten empty registers at the bottom of memory, then you can use the following instructions:

```
WDTA      ; save all your data on cards
now check the SIZE and increase it by 10
7CLREG    ; use this Card Reader function to clear registers 0-9
10.eee    ; eee is the old SIZE plus 10
RDTAX     ; read back the data, starting at register 10
```

You could do all this by writing a program to copy the registers one by one, but this method may be faster. If you have the Extended Functions then you can move registers by using REGMOVE instead.

You can also read program cards under program control. **RSUB** reads a program card, but if it is executed while the HP-41 is positioned at the last program in memory it adds the new program after the last program in

memory, instead of using it to replace the last program in memory. This means that a running program can ask for a subroutine to be read from a card, but this subroutine will not replace the running program. **MRG** reads a program from cards and uses it to replace all lines in the current program after the current line. If you are very short of program space you can have a program which holds some control statements at the beginning and is followed by MRG. Every time MRG is executed you can read in a selected subroutine which will immediately follow the MRG and will be executed after the MRG has been completed. MRG only works correctly if it is executed while you are positioned at the last program in memory. MRG clears the subroutine return stack, so if you call a subroutine, execute MRG, and try to return to the calling program then the subroutine will stop instead of returning. You can use MRG to read a PRIVATE card, but this becomes part of your program so the whole program becomes PRIVATE which means you cannot alter it and therefore cannot MRG any more cards. See Bug 11 in Appendix C for another trap that affects RSUB.

VER verifies any card; it tells you what type of card it is, and which track you are reading of a set. If the card is blank, dirty, faulty, or does not have HP-41 or HP-67 or 97 information on it then you will see **CARD ERR** or perhaps **CHECKSUM ERR** or **MALFUNCTION**. Check that the card is what you think it is, then breathe gently on the black surface and rub it on a soft clean cloth (actually your shirt may do). If this still does not work then try cleaning the black surface with alcohol (use methylated spirit - "denatured alcohol"). It is wise to write a description of each card on it - a soft pencil writes well and can be erased later.

Data and program cards written on an HP-67 or an HP-97 can be read, and programs will be automatically translated to run on the HP-41. As the HP-41 does not have some HP-67/97 functions, the Card Reader provides 67/97 **compatibility functions**. These need not be used only in translated programs, they can be used in ordinary HP-41 programs as well, but the Card Reader handbook does not give much explanation of what they do. Here are some details for those HP-41 owners who do not have an HP-67 or 97 Guide.

7ENG, **7FIX** and **7SCI** set the display mode without changing the number of digits shown. **7DSP0** to **7DSP9** change the number of digits without changing the mode. The HP-67 and 97 use only one register for indirect operations, on the HP-41 the Card Reader uses register 25 for these operations. **7DSPI** is used to translate the indirect DSP instruction of the HP-67/97; it could be called DSP IND 25 on the HP-41. Like HP-41 functions it ignores the sign and fractional parts and gives DATA ERROR if the number is outside the range 0 to 9. The direct increment and decrement functions are **7ISZ** and **7DSZ**. **7ISZI** and **7DSZI** are indirect via register 25; unlike the HP-41 functions all four ignore the fractional part of the control number. They were described at the end of Section 6.8. The translation of statistics functions, and the use of **7RCLΣ** have also been described already, at the end of Section 5.4.

The HP-67/97 used register I for indirect GTO and GSB instructions; these could be translated to GTO IND 25 and XEQ IND 25, but negative values had a special meaning on the 67 and 97. Positive numbers from 0 to 19 were used for going to the 20 possible labels of the HP-67/97, negative numbers were used to skip backwards over the specified number of program steps. This was called "rapid reverse branching" and made it possible to use GTO and GSB without labels. If the program skipped past the beginning of program memory it would go to the end and carry on up, so forward skipping was possible too. Translation of this feature to the HP-41 would be very difficult, and use of GTO IND 25 or XEQ IND 25 would execute a label ignoring the sign, which would be wrong. The Card Reader translates indirect GTO and GSB instructions into **7GTOI** and **7GSBI** which produce a NONEXISTENT message if register 25 contains a negative number or a text string. These functions are not perfect translations; they allow GTO or XEQ to any label from 20 to 99, whereas this would have caused an error on an HP-67 or an HP-97. You may find these two instructions useful if you want an HP-41 program to treat negative numbers as illegal labels.

7P<>S exchanges registers 00 to 09 with registers 10 to 19; this can be useful if you have two subroutines which both use registers 00 to 09. You

could also use 7P<>S if you are short of memory and want to use short-form RCL and STO steps which will not access registers 15 to 19. P<>S uses two bytes, so if you have more than four STO or RCL instructions using registers above 14 then it can be better to use P<>S twice and STO or RCL registers 05 to 09 instead.

7PRREG will print all HP-67/97 registers if a printer is attached. The HP-67/97 registers are translated to HP-41 registers 00 to 09 and 20 to 25. The secondary registers, 10 to 19, are not printed. If you do not have a printer then the HP-41 displays each of these registers, showing "Rnn=" followed by the register contents. **7PRSTK** will either print the stack contents (using the printer function PRSTK), or display them if there is no printer. The registers are displayed in the order T,Z,Y,X but without an indication of which is which. The function **7PRTX** prints X if a printer is attached (using PRX), or else executes VIEW X.

A few more notes on HP-67/97 translation. Other indirect operations are translated into IND 25 functions. Flag 3 which is used to test for numeric entry on the HP-67/97 is translated as flag 22. The flag test F? on the HP-67/97 cleared flags 2 and 3, so it is translated as FS?C 02 and FS?C 22. The other translations are explained in the Card Reader handbook.

The functions WALL, WPRV and VER are supposedly non-programmable. This means that if you XEQ them in PRGM mode they are executed at once, not stored as a program step. If you assign any of these functions to a key, remove the Card Reader, turn the HP-41 on, go into PRGM mode and press the key, then the function will be recorded in the program. Turn off, attach the Card Reader again, and see the function recorded in the program. When the Card Reader was not attached the HP-41 could not check that the function was non-programmable so it recorded it. These three functions will actually work in a program. The same trick can be used for PRP and LIST but they will not work properly because this method does not let them prompt for parameters.

The Optical Wand lets you read data, programs, or single instructions from

barcode. Data consists of single numbers which are read into the X register, or text strings which are read into the Alpha register. Single instructions are provided on a "paper keyboard" which contains all the normal HP-41 functions and Alpha characters, together with Printer, Wand and Card Reader functions, but not the Card Reader HP-67/97 compatibility functions. Each function can be scanned and will execute exactly as if it had been assigned to a key and that key had been pressed. Functions that are not on this keyboard can be spelled out, as with Alpha execution. Barcode for all Extended and Time functions, including the new HP-41CX functions is provided at the beginning of Appendix F. This also contains barcode for all Card Reader functions including the compatibility functions, for all printer and HP-IL module functions, and for the HP-IL Development module functions. The barcode at the top of each set gives the module identifier and can be included in a program followed by FS? 25 to check if a module is attached to the HP-41. A complete set of Alpha characters is provided as well. Byte 0 acts as a backarrow, a separate code is provided to append nulls, but this only works in PRGM mode. Byte 127 acts as an APPEND function; it re-enables text entry, so a barcode which appends byte 127 has been provided. The exact behaviour of byte 0 (and bytes 28, 37, 42, 43, 45, 47, 127) depends on the version of the Wand you have, and on whether you have PRGM and ALPHA mode set. You can find this out for your own Wand by experimenting - a table is given in the Synthetic Quick Reference Guide, along with an Alpha barcode table.

If you want to use these barcodes a lot you will have to protect them, or photocopy them (remember about the copyright of this book; you can make copies for personal use). If they are too close together then you can cut them out to make individual labels, like those that come with the Wand.

Complete programs can be read in much the same way as programs on cards. No initial instruction is required and the program replaces the last program in memory. The Wand prompts for each row in turn, if you cannot read one row then press SST and the Wand will prompt for the next row. If you have done this then all the steps that were in the missing row will have to be entered from the keyboard later. Appendix A in the Wand manual

gives details of this procedure. You can have trouble reading barcode for various reasons given in the manual. It pays to try reading difficult barcode in both directions, at various speeds, and using a straight edge to guide the Wand. One reason why barcode can be difficult to read is that strong light can interfere with the light reflected by the Wand; try to read barcode away from windows or strong electric lights if you have unexplained problems. Electric currents can also interfere; do not read barcode near AC power cables.

Two Wand functions let you read program barcode from a running program. **WNDSUB** prompts for a subroutine much as the Card Reader function RSUB. **WNDLNK** reads a subroutine in the same way but then immediately executes the subroutine it has read, and then returns to the calling program unless the subroutine stops instead of returning. **WNDDTA** lets a program prompt for a single data or text entry to be scanned; this goes into X or Alpha and the program resumes. **WNDDTX** prompts for data that will be read into registers specified by a number bbb.eee in X (like RDTAX).

WNDDTX normally accepts data barcode in any order that you read it, so you can have a collection of numbers and text strings which you can read in the order you choose. You can also use "Sequenced Data" barcode which specifies the order of each data item. This sequencing can be used to make sure a user reads data in the correct order. Early Wands (those which show up as WAND 1E in CAT 2) are unable to read sequenced data.

The function **WNDSCN** reads one to 16 bytes and stores them in registers beginning with register 01. Each byte is converted into a decimal number and stored in a register. These bytes can be normal keyboard, program or data bytes in which case everything including barcode type identification and checksums are read in and stored. They can also be single bytes from the stick-on label sheet, intended purely for use with WNDSCN, to identify data according to your own scheme. For example you can have a store which holds different types of equipment and uses these barcode labels to identify various items. The Wand provides one program, **WNDTST**, which uses WNDSCN to read a row of barcode and analyse it; you can use this to check

if the Wand is reading exactly what it should, or to read and interpret a row of barcode that is giving CKSUM ERR.

HP publishes an additional manual "Creating Your Own HP-41 Bar Code" which tells you how to design and print barcode, and includes BASIC programs to print barcode. This manual is worth reading even if you do not plan to print barcode; you can hand-draw acceptable barcode without much trouble if you need a small quantity. The manual tells you how barcode is designed if you are interested; it tells you what you need to know if you cannot read a piece of barcode and want to see what it should contain. Synthetic instructions cannot be coded as individual barcode keyboard functions, but complete programs containing Synthetic instructions can be read because the Wand does not check each individual byte when it reads a program.

12.4 HP-IL and other peripherals.

When you plug an HP-IL module into your HP-41 your horizons suddenly expand. You can use printers and display devices as already described but the HP-IL system is designed to recognise 16 classes of device, with 16 types in each class. A single HP-IL loop is designed to address 31 devices but each of these can have up to 31 secondary addresses. HP produces many devices that work with HP-IL (the best source of information on these is the catalogue of HP equipment published each year), independent companies produce HP-IL compatible equipment, and EPSON has now started to produce HP-IL equipment. On the other hand your HP-41 system gradually becomes less portable; individual items are portable but it is difficult to carry many, say a portable Thinkjet printer and a portable cassette drive as well as your smaller accessories and the HP-41 itself. If this book were to cover all HP-IL devices then it would not be portable either! I shall therefore only give brief descriptions in this section.

The HP-IL module provides three sets of functions: printer functions which were described above, mass storage functions, and more general-purpose interface control functions. The HP82162A printer and the HP82161A

cassette drive are the most popular examples of the first two device types, I shall only describe others briefly. The general-purpose control functions, particularly when used with Synthetic Programming, provide a fair degree of control. More complete control of HP-IL devices is provided by special modules which will be described later in this chapter.

The first HP-IL device class is 0, used for controllers. The HP-41 with an HP-IL module attached is one of these. Some controllers insist on controlling the loop to which they are attached, others can be made to relinquish control. Unless you have an HP-IL Development ROM your HP-41 will insist on trying to control the loop, so you will not be able to use it with another controller (for instance a second HP-41). HP series 70, series 80, and series 100 computers can also be controllers.

Class 1 devices are Mass Storage devices. The standard HP-41 Mass Storage device is the Cassette Drive, but the HP9114A 3 1/2 inch disk drive can be used as well and is much faster. The HP-IL module functions are designed for the cassette drive and cannot control all the features of the disk. The Cassette Drive has been studied in great detail by members of user groups who want total control of it; for instance it can be used as a single large data file instead of being treated as a collection of data files, each of which has to be accessed through the directory at the front of the cassette. Articles about the cassette drive have been published in PPC CJ V9N4P42-44, V9N6P4, V10N6P28-29, V10N10P9-11, V11N9P55-56, in the PPCJ V12N1P6-8, V12N2P15-17, and in PPC Conference Proceedings 6 and 7. Articles about using the disk drive with an HP-41 have come out in the CHHU Chronicle V1N3P12-14 and V2N1P28-29. This list is not complete but will give you enough to read for a while.

The HP-IL module functions to control Mass Storage devices are as follows. **NEWM** lets you initialise a new cassette or disk; this destroys the previous contents, so it is not programmable. NEWM can be recorded in a program using the same trick as for the non-programmable Card Reader functions, but it is unwise to do this. **DIR** provides a directory of the files on a cassette or tape. Data files can be created by **CREATE**, and the Extended

function SAVEAS can be used to turn a data file into a text file and save text in it. Programs can be copied to program files by **WRTP** and **WRTPV** (which writes PRIVATE program files in the same way as WPRV on the Card Reader). Programs can be read back by READP and READSUB. Unlike the Card Reader and Wand, the Mass Storage devices must be specifically commanded to read and write programs and data. Data can be written to a file by **WRTR** and **WRTRX**, which are similar to WDTA and WDTAX. It can be read back by **READR** and **READRX**. READRX need not start at the first register in a data file; you can select any register by means of **SEEKR**. Following SEEKR you can use WRTRX to write more data to a file as well. **ZERO** fills a data file with zeroes; CREATE does this anyway when you create a new file. **WRTS** and **READS** write and read the status registers; the same information as the first track of a status card. **WRTK** and **READK** save and retrieve the key assignments, as do tracks 2 and up of a set of status cards. **VERIFY** checks if a file can be read, rather like VER. **RENAME** changes the name of a file, **SEC** secures a file, preventing it from being purged (see below), renamed, or altered. **UNSEC** cancels this. Finally **PURGE** deletes a file from a tape. The space used by the file is still there, and can only be reused if a file that is smaller or the same size is created later. File names, where required, are given in Alpha as "source,destination" in the same way as for Extended Memory operations.

Additional commands to examine and copy Mass Storage files are provided by the Auto Start/Duplication and Extended I/O modules described later. The Extended Function GETAS reads text files back into Extended Memory; it can be used to access the directory entry of any file, and the directory can then be studied. If you want to study a cassette tape directory then do:

1) Select a file in the part of the directory that you want to study.
2) Put "filename," into the Alpha register.
3) SF 25, GETAS - this will read that part of the directory which contains the file, but will then stop because of a NAME ERR due to the comma after the file name.
4) You can now read the file, or use INA to read the directory.

The above is adapted from advice given by Mike Markov in the PPC 10th Anniversary Conference Proceedings. These are available directly from PPC, but if you are interested maybe someone in your local user group has a copy. A good deal of information on cassette tape formats is given on page 9 of the Synthetic Quick Reference Guide.

Do not get upset if the above does not interest you; it interests some users and you may want to come back to it later. To begin with you can use the Mass Storage devices by following directions in the IL module manual.

Back to HP-IL device classes; 2 covers HP-IL printers with and without graphics. These have already been covered in sufficient detail for the purposes of this book.

Class 3 covers displays. The HP 82163A Video Interface was the first device of this class, but HP now sells the HP 92198A 80-column Interface made by Mountain Computer, Inc. Video interfaces are made by other companies as well. The Paname ROM, described later, provides additional functions to control displays.

Class 4 is for interfaces. This includes the HP 82168A HP-IL modem which is used for remote data transfer over telephone lines. The transmission rate is 300 baud. Other interfaces are the HP 82165A GPIO interface, the 82164A RS-232C interface, and the 82169A HP-IB interface. GPIO interfaces are also available in kit form for users who wish to build them into their own equipment. The kit part number is HP82166 followed by a letter denoting the current kit type; at present this is C for a kit which includes four interface units with development software.

Class 5 describes electronic instrumentation. This can include various devices such as instrumentation controllers, but note that some instruments are included in class 7.

Class 6 devices are Graphic I/O devices, like the HP 7470A option 003

plotter which can accept plot data but also send digitising information to a controller.

Class 7 covers Analytical and Scientific Instrumentation. An example of this is the HP 3421A Data Acquisition/Control unit. A special HP-41 module has been produced to let the HP-41 control this device and others such as the HP 3468A Digital Multimeter.

Classes 8 to D were left undefined when the HP-IL system specifications were published.

Class E was used to cover devices that do not fit into other classes. EPROM programmers were included in this class.

Class F was left for "Extended class" devices.

A list of HP-IL products made by independent companies was published in PPC CJ V10N2P32. Of these the Mountain Computer EPROM programmer was made in only a limited quantity. New products are advertised in HP related publications such as the user group journals and Personal Computing, and in the specialist press related to the products.

Some other peripherals should be mentioned here. Port Extenders, as described in Section 2.7, are important if you need to connect more than 4 modules or devices directly into the HP-41. The best-known kind is produced by AME and sold through EduCALC (see Appendix B), others are also made, some are advertised through user group journals and by Boekhandel Prins (see Appendix B again). Devices that let you write programs onto memory that the HP-41 will treat as plug-in ROMs, and devices that will let the HP-41 read instructions from EPROMs instead of ROMs will be described in Chapter 17. The Port-X-Lite plugs into a port like a module and has a small bulb on a stalk, so that you can read the HP-41 display in the dark, it was made by AME and advertised in the EduCALC catalogue. The original description was in PPC CJ V9N4P15. Carrying cases, power supplies and magnetic card holders are all sometimes advertised in user group journals.

12.5 Plug-in modules and XROM conflicts.

Plug-in modules for the HP-41 will be divided into four types and described in the next four sections, after a few general comments here. The first type are the "application program" modules (also called "Application Pacs" by Hewlett-Packard). These are modules such as "Circuit Analysis" or "Home Management" which contain programs for one specific application of the HP-41. The programs in these modules look just like user-written programs but they can not be altered and they do not take up any space in the HP-41 RAM memory. The second type is a "utility programs" module. The Mathematics module and the Standard Applications module belong in this class; they provide useful programs which can be applied in a variety of fields. The third type can be called a "system enhancement" module. The Time module is a good example, it does not contain programs but provides a built-in clock and functions that let you use it. Among the other "system enhancement" modules are the Memory modules, the Extended Memory modules, and the Auto-Start Module. All these provide features which enhance the HP-41 system and which would be difficult or impossible to achieve with programs alone. A fourth type are the modules used by HP Repair Centres to check the HP-41. Modules are also used to connect plug-in devices into the HP-41 and to control either a single device, or in the case of the HP-IL Module, to control many devices; these were described above.

This division into four types will help in further discussion but it should be noted that the separation is not always very clear. For example the Mathematics Module provides hyperbolic function routines which many people would consider an essential part of a scientific calculator, so the Mathematics Module could be treated as a system enhancement module. Again, three application modules contain functions and routines which provide a sophisticated set of unit conversions. These functions will be discussed later in Section 12.6 and could be considered to be a system enhancement.

Another point is that Hewlett-Packard manufactures modules for other companies. Several petroleum companies for example have written HP-41

programs for their own use and have put them into special modules so that employees can use the same programs throughout the company. A few organisations have also ordered "system enhancement" modules of their own design. All these modules are manufactured by Hewlett-Packard to their normal high standard, and the software in these modules is often also of a high standard. A few of these modules are on sale to the public and some will be mentioned in this chapter. Other groups, who do not wish to invest in manufacturing a plug-in ROM, write HP-41 programs onto EPROMs (Erasable Programmable ROMs) which are cheaper for small numbers of copies than ROMs. Such EPROMs can be easily copied and are connected to the HP-41 in specially-made "EPROM boxes". These EPROM boxes and similar items will be mentioned in Chapter 17; the reader should be aware that a great deal of specialist software is available only on EPROMs.

Each plug-in module and device is identified by an ID number. This is the first of the two numbers that show up when the module is removed and programs or functions show up as XROM id,fn. Only the numbers 1 to 31 can be used for id's, but far more than 31 plug-ins have been made, so there are bound to be cases where several modules have the same ID number and there are conflicts. If you look back to Chapter 8 you will see that some modules contain two 4K blocks, and each of these has a separate ID, so cases can arise where one of the two numbers is the same as the ID number of another module; "module" because HP has been careful that devices such as the Card Reader do not have the same number as anything else.

If you have two modules with the same number connected to your HP-41 then all references to functions with that number are checked only in the first module found by CAT 2, so you will be unable to use any programs or functions from the module in a higher port number. The synthetic program XRO in Example 4 of Section 15.5 will let you execute a program (but not a function) by jumping straight to a selected address in a module. This does not use an XROM number so you can use XRO to execute a program from a module which is in a higher port number than another module with the same ID. Take an example: you want a program to use the function JDAY which is in the Securities module and has the number XROM 19,15. For some reason

the same program must also use a t-distribution function, and there is one in the Clinical Lab module. Unfortunately the Clinical Lab module programs are also identified as XROM 19,nn so you could not normally do this. If you plug the Securities module into a lower port number than the Clinical Lab module and enter XEQ "TDIST" then TDIST will be found to be XROM 19,14 in the Clinical Lab module, so XROM 19,14 will be recorded in your program, but this will be interpreted as XROM "ATP", which is XROM 19,14 in the Securities module. You can however remove the Securities module for a while and record the address of TDIST. Then you could use XRO to jump directly to this address without using an XROM number at all. Note that this would not work if the Clinical Lab module was in a lower port number than the Securities module; you cannot use XRO to jump to a function. If you want to use this method, read the details in Chapter 15.

Table 12.2 gives a list of XROM numbers and the modules that use each number. Modules that appear twice in the table are the ones with two 4K blocks. Two 4K modules occupy the upper 4K of their port, not the lower 4K as is usual. HP makes modules to order for independent companies: the XROM ID's are written on the label at the bottom of these modules. Most such modules use 31, or 21 and 31 if they are 8K modules; the Oilwell module is given as an example of them. The Synthetic Programming Quick Reference Guide gives a list like this and individual function names and numbers too.

XROM number	Module
1	Math; functions in 1C have different numbers than 1A and 1B
2	Statistics module
3	Surveying
4	Finance
5	Standard Pac, ZENROM(upper 4K only), Paname lower 4K
6	Circuit analysis
7	Structural Analysis lower 4K
8	Stress Analysis
9	Home Management, Paname upper 4K, CCD module lower 4K
10	Auto/Dup(upper 4K only), Games, PPC ROM lower 4K
11	Real Estate, CCD module upper 4K
12	Machine Design
13	Thermal and Transport Science
14	Navigation
15	Petroleum lower 4K
16	Petroleum upper 4K
17	Plotter ROM lower 4K
18	Plotter ROM upper 4K
19	Aviation, Clinical Lab, Securities, Structural upper 4K *
20	PPC ROM upper 4K
21	Data Logger lower 4K,Daly Oilwell module lower 4K
22	HP-IL Devel module lower 4K, Advantage lower 4K
23	Extended I/O
24	HP-IL Devel module upper 4K, Advantage upper 4K
25	Extended Functions, including HP-41CX Extended Functions
26	Time module, including HP-41CX Time functions
27	Wand
28	HP-IL Mass Storage and Control functions
29	Printer and HP-IL printer functions
30	Card Reader
31	Data Logger upper 4K, Daly Oilwell module upper 4K

*The X on the label tells you to avoid using 2 of these at the same time.

Table 12.2 XROM numbers used for plug-ins

If you write a program which uses functions from one module, say the Games ROM, and later run the same program without this module connected the program will normally stop with the message NONEXISTENT. However if a different module with the same ID, say the PPC ROM, is connected then the program may run and you might not notice anything was wrong till later, so be careful. If you assign an XROM routine or function to a key, then plug in a different ROM with the same XROM number into a lower port, or remove the first module altogether, then the corresponding function from the second module will be executed. A particularly confusing thing can happen in cases like the following: you have an Aviation module in port 4 and a Structural Analysis module in port 2. You enter XEQ "CRUISE" to execute an Aviation module program; the HP-41 searches for this function by name, finds it in port 4 and notes that it is XROM 19,11. Then it tries to execute XROM 19,11 and finds the Structural module, numbered XROM 19 in port 2. It tries to execute the 11th XROM 19 function in this module, but the highest number in this module is only XROM 19,07. You therefore see NONEXISTENT, even though the HP-41 had already found the function you wanted. This can happen with any two modules having the same XROM numbers.

A second difficulty is that certain function or program names are used by more than one module. The worst case is SIZE? which is used as a routine name in the Stress, Structural, Games and Real Estate modules. It is also used as a function name in the Extended Functions module and in some other modules which contain a few Extended Functions, for example the Data Logger module. All of the SIZE? routines check whether the current SIZE setting is sufficient for a program to work and complain if it is not; the SIZE? function returns the SIZE setting to register X. If you just enter XEQ "SIZE?", which one will be executed?

Names like this are searched for in CAT 1 first, then in system devices in CAT 2, then in other plug-ins, starting with port 1. If you execute SIZE? then the first SIZE? to be found will be executed, so you have to check which ports your modules are plugged into. If you have an HP-41CX then the

built-in Extended Functions come right at the end of CAT 2, so any other SIZE? will be found before it and will be executed. The same CAT 2 search rule applies if you are assigning a function, the first function found will be assigned. Different functions with the same name can be assigned to different keys by using this. Remove all plug-in modules from an HP-41CX and assign SIZE? to the X<>Y key. Then plug in a module with a SIZE? routine and assign SIZE? to the TAN key. Now you have two different SIZE? key assignments, and each one will execute the chosen SIZE? ; you even have a mnemonic - the function is at the left and the routine at the right.

12.6 Application Program Modules.

General Information.

A few general points concerning Application Program modules should be made first. To begin with, you should always read the manual, preferably all of it. That is the only way to make sure you will use the module correctly. You should also do some of the examples to get practice, and to see if the manual has any misprints. Hewlett-Packard provide Addendum Cards for modules or manuals which contain possible faults or errors, but these cards can only be produced after the problem has been discovered. The Hewlett-Packard magazine "Key Notes" used to reprint the information from these cards but it is no longer published. One way to get this information now is from a user group. Modules are occasionally updated, but Hewlett-Packard will not normally exchange an old module for a new one in Europe.

Application Program Modules are designed to be used by people who never program an HP-41, as well as by experienced programmers. The instruction manuals tell the users how to obtain a particular result, keyboard overlays remind them which keys to press, and the programs in the modules ask helpful questions. These questions are valuable to the non-programmer but they often annoy programmers who want to use only part of the program or perhaps to design their own questions.

You can copy programs from a ROM module to the HP-41 RAM memory (using the

COPY function) and then alter the copy in the same way as you would edit any other RAM program (see Chapter 8 for a comparison of ROM and RAM). A program that has been edited takes up room in the HP-41 memory, it cannot be copied back into the module, nor can programs in a program module be edited. The copy in RAM memory is in CAT 1 and will be executed instead of the version in the module which is in CAT 2 because CAT 1 is searched before CAT 2. The copy in RAM may still refer to other programs in the module though. If you want to change any of these programs or if you want to remove the module entirely, then you must COPY them to RAM too. You must also delete every program step that says XROM "routine" and replace it with XEQ "routine" after you have copied the routines to RAM. An exception to this are the Petroleum Fluids calculation subroutines which have XEQ instructions in them already. Details of COPY are given in Section 6.5.

In some of the modules, the initialisations and questions are put into separate subroutines so that programmers can use just the routines that do the calculations. Several modules have separate subroutines which do useful jobs such as asking yes/no questions. It is a fault of most manuals that these useful subroutines are not mentioned at all. If you want to skip the questions at the start of a program which contains calculations and questions you can use the synthetic program XRO, mentioned above, to jump directly into the middle of a program in a module.

Many of the programs in the Application modules have been adapted from programs written for the HP-65 or the HP-67 and HP-97. Indeed some of the HP-41 modules are nearly identical copies of application packs available for the older calculators. These programs have the great advantage of being long-used products which are less likely to contain catastrophic errors than new programs. The adaptation to HP-41 use has not always been efficient, for example many programs finish with the two steps RTN, END. END does exactly the same as RTN, so the only reason to have both is in case a user presses R/S accidentally after a program has stopped. Never mind, a program in a ROM runs faster than the same program in RAM anyway, so some inefficiency is acceptable. Most of the programs are designed to print their results on a printer if one is attached.

A lot of programs in the Application Modules work to U.S. Standards. This means that some will be of little help to people in the United Kingdom or elsewhere who have to work to their own national standards. A few programs can be copied to RAM and easily modified, but others are of no use. Warnings about this will be given where necessary. Several modules have a mixture of some programs that are useful to everyone and others that apply only in the U.S. - it can be worth buying these modules just for the latter programs.

It is worth examining Application Modules outside your own field of work. For example the Petroleum Fluids Module may be worth buying just for its unit conversions, or the Clinical Medicine Module may be worth buying because it has some statistical programs that are not available in the Statistics Module. In some cases (unit conversions or vector operations in particular) several modules may have the functions you need so it is worth checking which one best suits your needs (or which one is cheapest).

In some cases, several modules contain programs with the same XROM numbers, and you should only plug one such module into your HP-41 at a time. The table above listed such conflicting modules. If you are not sure what XROM numbers are, reread Section 3.5.

Unit Management.

The **Unit Management** or unit conversion functions are a very powerful feature of three modules, and it is surprising that Hewlett-Packard have advertised them so little. The general rules for Unit Management are that you put a conversion equation into the ALPHA register, and the number to be converted into the X register. The conversion equation consists of an input and an output separated by a dash. The input and output are each a formula in terms of acceptable units, with powers from 1 to 9 also allowed. You can omit the output in which case the conversion is to S.I. units. You then execute the conversion function (these functions are written in machine code, so they are fast). The functions first check if they know

the units, and if the units are consistent. (You cannot convert centuries to centimetres!) The Petroleum Fluids Module allows 82 units and provides 6 additional input and output routines to prompt for the inputs and display or print the outputs. These routines can also store the names of your input and output units separately. They can also record whether you are working in English or S.I. units and can ask for the units of input and output values. The Petroleum Fluids Module comes with an excellent manual, but it is more expensive than other modules. The Thermal and Transport Science Module allows for 55 units and provides one input and one output subroutine. The Machine Design Module allows for 42 units and has no input /output subroutines, but it allows for some units that are not recognised by the Petroleum Fluids Module.

Units allowed for can be English, S.I. or other common units. For example the Petroleum Fluids Module allows pressure to be specified in fourteen different units; of pounds per square inch, pounds per square foot, atmospheres, bars, millibars, megapascals, pascals, kilopascals, kip per square inch, Torr (and also the nearly equivalent millimetres of mercury), inches of mercury, feet of water, and inches of water. Use of the Unit Management System can save you a lot of time and space. Executing a conversion from UK gallons to US gallons for instance is simple and quick; you do not need to look up the conversion factor or store it in the HP-41 memory. A serious shortcoming (for many users) is that electrical and magnetic units are not allowed for. Hewlett-Packard had originally planned to extend the Unit Management System. Even though these extensions never reached the HP-41 users, the Unit Management System is still very much a "good thing" in its present form.

Aviation.

This module turns the HP-41 into a "flight computer", but one best suited for use with twin-engined planes (particularly Cessnas). If you fly wide-bodied jets then you have more powerful computers on board anyway. If you fly Beechcraft planes you may prefer the Beechcraft module (see below). The Aviation Module is designed for flight planning (FCC rules limit its

use to pre-flight calculations only), and it certainly does this job, but you may prefer to obtain a Navigation Module as well (see below again). A few general-purpose subroutines are available in this module. "360+" simply adds 360 to the number in X without altering registers Y, Z or T. (This is quite different from "+360" in the Machine Design Module.) "*T" is another useful subroutine - it displays a time or angle in hours (degrees): minutes: seconds and prevents displays of 60 minutes or seconds.

"Beechcraft"

This module is not sold by Hewlett-Packard; it was written under contract for Beechcraft Corporation. It comes with the documentation for Beechcraft Super King aeroplanes, or you can purchase it (without the plane) from Beechcraft.

Circuit Analysis

This provides 63 routines split up into two general sections; general network analysis and ladder network analysis. A circuit is described in terms of branches and nodes together with a list of component values. The circuit response can then be studied (and printed or plotted if you have a printer) and programs that use the circuit analysis pack can vary one parameter or several. These programs could even use minimisation functions (available in the Math ROM and in the PPC ROM) to find best values for components.

Two useful subroutines in this module are "*C*" (complex multiplication of two numbers in the stack) and "*C+" (complex addition of two numbers in the stack). These subroutines are described in Appendix B of the manual. You should also be a little careful of the routine labelled "PI", or you may confuse it with the function PI.

Clinical Lab. and Nuclear Medicine.

The six Clinical Lab. programs in this module are Beer's Law, Body Surface Area, Creatinine Clearance, Blood Acid-Base Status, Oxygen Saturation and Content, and Red Cell Indices. This is such a variety of programs that you will need to look at the manual before deciding whether you should buy this module. The more-commonly used of these values are now routinely provided by the hospital labs using widely- available apparatus. The practicing clinician may therefore have little use for the programs, but research workers may find some of them useful.

The four Nuclear Medicine programs are "Total Blood Volume", "Thyroid Uptake", "Radioactive Decay Correction" and "Radioimmunoassay". You may want to read the next chapter to find out about other possible sources of similar programs. The radioactive decay correction program contains data for 15 isotopes and can be run with other isotopes. It may be useful to people who perform decay calculations in fields other than medicine, for example in Carbon-14 dating.

The module also provides programs for basic statistics, chi-square evaluation and distribution, t-statistics and t-distribution. The last of these is particularly useful since it is not available in the Statistics Module. Be warned: the first two use CLRG and will destroy all your data.

Financial Decisions

This module provides several useful programs which in effect let you calculate the future value of money. The programs are designed for the United States, so if you plan to use the module elsewhere you might have to copy and alter the programs or write additional routines. (One example is that the interest rate on mortgages is calculated differently in some European countries.) This module also contains a general-purpose "Days between Dates" calculation, but the Time Module provides a faster version called DDAYS. Several of the financial programs are also available in the Real Estate Module.

Two additional financial modules, programmed by independent companies are available; they too are clearly designed for the U.S. market. One is a "Financial Planner" module; the other is a "Market Forecaster". This provides an 80-day forecast of the Dow Jones Average (with a correlation coefficient of r=0.78 for an eighteen year test period). Both can be ordered from EduCALC (address in Appendix B). There is an excellent finance program in the PPC ROM; this can do non-U.S. calculations.

If you really want to do a great deal of financial calculation you may prefer to purchase a financial calculator (such as the HP-12C). Hewlett-Packard is at last beginning to think about non-U.S. financial calculations in response to requests from Hewlett-Packard in Europe but we shall have to wait and see if there will be any definite results.

Games

Why use your HP-41 for work only? Some of the best HP-41 programs are in the Games Module. There is a number guessing game (inspired by MASTERMIND), and there is a word-guessing game (which requires two players). There is the almost obligatory "Biorhythms", and there is a "subhunt" game and a "spacewar" game. These two are played on a square grid which is provided separately because the HP-41 has only a one-line display. There is a simple dice game, and finally there is a wonderful pinball simulation. This provides eight scoring devices with sound effects, two flippers, and even a "tilt" mechanism. It seems to be extremely well-suited to the HP-41.

If you need an excuse to buy this module, it has two random-number generating routines.

Home Management.

This module is a great aid for those HP-41 users who want to run their home like a business. It helps with home budgeting, travel expenses and

chequebook balancing. It also helps with investment calculations, including taxation, mortgages and insurance (suited to United States applications of course). Several of the programs are similar to ones in the Financial Decisions, Real Estate or Securities modules but the programs selected for this module are for a typical (U.S.) householder rather than an expert in one of the other fields. If you want a financial calculator and do not need the other features of the module then you could buy a financial-only calculator, or you could use the PPC ROM or Advantage ROM which provide a financial program along with scientific and engineering functions.

The Home Budgeting and Travel Expense Record programs use a fair amount of stored data (as does the Stock Portfolio program). It is useful to have a Card Reader to store all this data while it is not being used, the programs have built-in facilities for using a Card Reader if you have one. These programs can be used outside the United States with no trouble since you can just replace Dollars with Pounds or Francs or whatever. One last point: the manual for this module provides a useful introduction to some of the financial problems tackled.

Machine Design

This module provides programs for design and analysis in mechanical engineering. Topics include the design and analysis of four bar systems, slider cranks, circular cams and various gears. There is a well-described helpful ("user-friendly") helical spring design program. The examples for this are good but the first answer to the first example is printed wrongly. The write-up includes parameters for seven types of wire (American Standards of course), for other types of wire you have to know the parameters. This program works either in newtons and millimetres or pounds and inches, but the module contains unit conversion ("Unit Management") routines to let you use other units. Fewer units are recognised than by the Petroleum Fluids Module, but five units not in the Petroleum module are included in the Machine Design one because they are used in mechanical engineering. (The Thermal and Transport Science Module provides all the units that are in the Machine module and some more.)

There is also a program which provides numerical solutions for a damped oscillator driven by a force. This force can be specified by a separate routine (its label name must be 6 characters long or shorter though this is not stated). The program solves a problem which is given in many engineering and physics degree courses, I hope that students who own this module will try the problems first without using the module! Finally there are three geometrical programs: co-ordinate transformations, circle lying on three specified points, and regularly spaced points on a circle.

Apart from these programs, the module contains several useful subroutines. "KEY" prompts the user to press one of the keys in the top two rows, "+360" brings an angle to within the range 0 to 360 degrees (it also works in radians and gradians modes). "IN" checks for the existence of a printer and if the SIZE setting is sufficient for a program. "*?" asks the question Y/N (yes/no?). Any reply other than "Y" is taken to mean no; this information should have been provided somewhere in the manual, since "*?" is used by several programs. Overall this module does a very good job if you have a use for it.

Navigation

The people who use this module consider it one of Hewlett-Packard's best. It provides the necessary data and calculations for Great Circle and Rhumb Line Navigation. The calculation routines are all independent of the routines that prompt for input. This means that users can write their own programs to call these subroutines, which is particularly important for air navigation. The almanac routines are also very impressive, they contain information on all 58 objects in "The Nautical Almanac" including the Sun, Moon and four navigational planets. As usual the module was designed with American users in mind, but a British navigator was a prime mover behind its good design, so it is fairly international. Some users object to the American-style emphasis on extensive prompting which meant that several very useful calculation routines originally planned for it were left out but you cannot have everything.

This module also provides a collection of subroutines for use in the user's own programs. The routines cover such subjects as Date and Time including Sidereal Time, Co-ordinate calculations, and position analysis.

Additional Navigation programs are being collected together by the Marine Navigation group within HPCC. A large number of programs, and even a module are produced by ADM Systems in the United States. (Their products are available from EduCALC see Appendix B.)

Petroleum Fluids

This module provides programs concerning reservoirs of hydrocarbon fuels. It can be considered as four parts; natural gases, oils (and gas-oil mixtures), reservoir brines and rock compressibility, and utilities (including unit conversions). The programs, including the unit conversions, are divided into separate subroutines for prompting and for calculations. Both the program material and the manual are exceptionally well-designed and useful. The manual is 200 pages long, it contains more detailed instructions concerning program use than do other manuals, at least two worked examples for each program (in each case the first example shows how to use the program as a whole), a special section on using subroutines from the module and annotated listings of the programs. Only two general-purpose subroutines are available apart from the unit conversions; one prints a program title and checks if the SIZE is sufficient, the other asks Yes/No questions, as usual any answer except Y is taken to mean No.

Both the module and the manual are wonderful, but you are unlikely to have an excuse for buying them unless you work in the petroleum industry or need the unit conversions. If you do work in the petroleum industry you may find it worth getting some of the other modules such as the Oilwell module made by a UK company, Daly Drilling Enterprises (see Appendix B).

Real Estate.

The ten programs in this module are all related to cash flows and purchase or renting of real estate in the United States. Those programs most likely to be of use outside the United States are also available in other modules such as Home Management. Several subroutines which calculate financial formulae can be called from your own programs, but users should really check the manual before deciding whether to buy the module. The manual is helpful in that it does describe the general-purpose routines separately from the programs.

Securities.

The ten programs in this module allow you to make calculations related to investments, mostly on the U.S. market, in bonds, notes and stocks. Many people who have a great deal of money invested or work in the financial markets will find this a helpful module. Those readers who have barely scraped together the cash for an HP-41 will not need this module (yet). Once again, the advice has to be that you look at the manual and decide for yourself. The manual contains an appendix with financial formulae, and the module contains four general-purpose routines and a days-between-dates function which works on a 360-day financial year. The module helps you to save data on magnetic cards for future use.

Stress Analysis.

Eight programs intended for use by Mechanical Engineers are provided. Much of the module is concerned with beams, cantilever beams, and columns. Programs on Section Properties and Mohr Circle Analysis help with this. Additional programs help with strain gauge data reduction and the solution for the unknown variable in Soderberg's equation for fatigue. The last program allows the HP-41 to be used as an RPN calculator for vectors. Each vector is represented as a magnitude and an angle, there is a four-level stack, with a LASTX, and there are standard vector operations. (The results of the cross product and dot product calculations are left in the

ordinary HP-41 stack, but not in the "vector stack".) A useful module and many engineers will want either this or the Structural Analysis Module.

Structural Analysis.

This is similar to the Stress Analysis Module but is designed particularly for Civil Engineers. Half of the programs are for analysis of structures and the other half are for design (the module is sometimes called "Structural Design"). The design programs are specifically for United States specifications. The analysis programs can be useful anywhere and some engineers have told me that this module is vital for their work, particularly the beam programs. An RPN vector program is also provided, as in the Stress Analysis Module.

Surveying.

This module is very clearly intended for land surveying according to United States practice. Those parts that calculate land area and geometrical properties can nevertheless be used anywhere. There is also a Co-ordinate Transformations program, but this is available in the Mathematics Module as well. As with several other modules I would advise potential users, particularly outside the United States, to check the module manual before buying. One problem is that the module uses CLRG in its initialisation operations, so that it deletes all the data you had previously stored. You may also consider purchasing a Survey Module programmed by a company independent of Hewlett-Packard, there are at least two such modules at the time of writing. Read the specialist press and check if companies such as EduCALC sell a module that suits your purposes.

Thermal and Transport Science.

The "Transport Science" concerns the transport of fluids, in line with the rest of the module (not Public Transport!). The whole module is designed for Mechanical Engineers who work with the flow, heating and compression or expansion of fluids. A black body thermal radiation program is included; a

slightly less sophisticated version with a smaller choice of units is available in the Physics Solutions book. The Thermal and Transport Module can provide a wide range of units because it contains Unit Management functions, including seven units that are not allowed for in the Petroleum Fluids Module. The manual provides a good description of the Unit Management System.

Turning back to the programs, you will find that these cover equations of state for gases (Ideal and Redlich-Kwong model), compression, expansion and flow of ideal gases, and fluid flow in conduits (with an automatic option for water). There are also programs for energy equations in steady flow (such as a water tower), and analysis of flow in heat exchangers.

This is rather a specialised module, useful in its field and possibly of interest to other people because of the equations of state program and the Unit Management functions and routines.

Other non-HP modules.

As already mentioned HP make modules for users who supply their own programs. A few modules have been mentioned above, most are ordered by companies for internal use. Some are for customisation of HP-41s which may then be resold or included in larger pieces of equipment, but a few have been made for public sale. These include a hydraulics module, a module for U.S. and Canadian horse-racing followers, and an Astrology Module. Look for advertisements in specialist publications, or ask EduCALC.

12.7 Utility program modules.

This section covers five modules which contain programs that do not apply in just one field.

Mathematics

This was the first module designed for the HP-41. Ten programs were included in it, they were selected as programs that many HP-41 owners would find useful. Eight were taken from the HP-67/97 Math Pack, one was taken from the HP-67/97 Standard Pack, and one from the HP-67/97 Electrical Engineering Pack. (The selection was apparently made with an eye on the standard plug-in module for the Texas Instruments TI58 and TI59 calculators which is not surprising, because these were the main competition for the HP-41.)

The module and programs are described here in more detail than others because the Math Module is the one with the most applications.

The programs are as follows:

1) Calculates the determinant and inverse of a matrix, or solves a system of simultaneous equations. Matrices up to 16x16 can be solved if you have a quad memory module or an HP-41CV or CX: 19x19 if you have Extended Memory and use Synthetic Programming; see Chapter 16.

2) Finds a value of X that will provide a zero value of a specified function. The program actually finds a value of X such that f(X)=0 to the accuracy of the display.

3) Finds roots of polynomials of degree 5 or below and evaluates polynomials. In version B of the module incorrect roots may be calculated for 4th and 5th order polynomials. Check the roots by evaluating the function using them as inputs.

4) Performs numerical integration of functions whose values are known at a set of equally spaced points, or whose functional form is known explicitly.

5) Solves first and second order differential equations of the form y'=f(x,y) or y"=f(x,y,y').

6) Computes up to ten consecutive pairs of coefficients of a discrete Fourier sine and cosine expansion of a function. Any set of ten consecutive coefficients can be selected, provided enough data points are given. The second part of the program computes the value of the Fourier series at a selected point. This is potentially the most powerful program in the module; it can be used for many purposes such as smoothing a data series. However the program was copied from the HP-67/97 Electrical Engineering pack, and in Revision A of the module allowance was not made for the fact that these calculators automatically clear flag 2 when they test it. The unfortunate result is that incorrect values for the Fourier series are computed if the zero-coefficient (a_0) is included. This can be corrected if you copy the program to program memory and change line 160 to FS?C 02 but that uses up a lot of memory which the module is supposed to be saving. Alternatively you can halve the zero-order coefficient of the expansion. (The correction card says you should halve both parts of the zero-order coefficient, but in fact the second part ought to be zero.) If you live in the United States you might be able to get a replacement module for one with an error like this, but elsewhere this is rarely possible.

7) Allows calculations involving complex numbers. Seventeen functions of one or two complex numbers are provided, and if the result is a complex number then it can be used in subsequent calculations. The complex numbers must be in rectangular form.

8) Computes hyperbolic sines, cosines and tangents, and also their inverses. The program given in Chapter 7 is slightly more accurate for tanh and atanh, and it does not use any numbered registers, but it takes up some memory space.

9) Provides triangle solutions and calculates the unknown values for a triangle if one side and two other values (sides or angles) are given.

10) Calculates two- or three-dimensional co-ordinate transformations and rotations. If you have revision B of the module you must make sure flag 01 is clear before you attempt to do any transformations.

The errors in Math ROM revisions A and B have been pointed out above. You can check which revision you have by doing CAT 2. On an HP-41C or CV you will see MATH or MATH 1A if you have revision A, MATH 1B if you have revision B, and MATH 1C if you have revision C. On a 41-CX you should see the same except that 1B will not show up at all unless you press the ENTER key to get a list of all CAT 2 functions. Although revision C corrects the earlier errors, it has the programs in a different order. This means that programs written with a revision A or B module will not work correctly with a revision C module. For example XEQ "ACOSH" is recorded as XROM 01,37 if you have a revision A or B module. However XROM 01,37 appears as XROM "ATANH" when a revision C Mathematics Module is plugged into an HP-41. If you obtain any software using Mathematics Module programs you must check that the right programs are called when you plug your module in.

If you need only a few of the Mathematics Module programs then you may find similar programs in one of the other modules. The PPC ROM in particular has several useful mathematical programs.

Statistics.

Like the Mathematics Module this provides a collection of general-purpose routines. A very brief description of each program is given here.

1) Basic statistics for two variables, grouped or ungrouped. The values calculated are mean, sample standard deviation, population standard deviation, coefficients of variation, correlation coefficient and the standard summations provided by the HP-41 built-in statistical functions.

2) Moments (first to fourth), and coefficients of skewness and curtosis for grouped or ungrouped data.

3) One-way analysis of variance. Generates all twelve values for an ANOVA table.

4) Two-way analysis of variance for the case where row and column effects are separate and each cell in the set of data has only one observation.

5) One-way analysis of covariance.

6) Curve fitting. Fits data to one of four possible curves; straight line, exponential, logarithmic or power curve. Calculates coefficients of regression and determination. The same program is available in the Standard Applications, and one which requires less data entry is available in the PPC ROM. The Advantage ROM has a good curve fit routine too.

7) Multiple linear regression. Fits (by least-squares) a line of the form t=a+bx+cy+dz to a set of points t and three corresponding variables x, y, z. Also works for two variables x and y. Calculates coefficients of regression and determination.

8) Polynomial regression. Fits sets of data points x, y to a second-order or a third-order polynomial. Calculates coefficients of regression and determination.

9) Calculates t statistics for paired observations or two independent sets of observations (two means).

10) Calculates chi-squared (goodness of fit) for sets of observations. Allows for unequal expected frequencies of observation and for equal expected frequencies.

11) Calculates chi-squared and Pearson's coefficient of contingency for 2xK and 3xK contingency tables. Can also be used to calculate column sums, row sums, and table totals.

12) Calculates Spearman's Rank Correlation coefficient. If a set of observations (or individuals) is arranged in some order (ranked) two times, this coefficient is an estimate of how similar the rankings are in the two cases.

13) Calculates the normal density function, its integral, and its inverse.

14) Calculates the chi-squared density and its integral for a given number of degrees of freedom.

Standard Applications

In order to provide some programming examples for the new owner of a programmable calculator, Hewlett-Packard often provides a book of Standard Applications programs. A book like this, containing ten programs, was once provided with each HP-41C and HP-41CV. If you do not want to enter the example programs from the keyboard then you can buy a Standard Applications module which contains these programs. The module is sold without the book, since you are supposed to have received the book together with your HP-41.

If you buy an HP-41CX, bad luck! You do not get a Standard Applications Handbook because the HP-41CX manual itself has five example programs, including barcodes. (You get barcode for the Standard Applications programs in the Wand manual.) You can buy the Standard Applications module for use with your HP-41CX but you will have to try to find a book for it separately and pay extra for it. (You do get a card with some brief

instructions.) In case you are interested in this module and do not have the handbook or want to check for errors in it, here is a list of the programs. The RPN primer uses the Alpha features of the HP-41 to show you how the RPN stack and LASTX are manipulated. The Calendar functions program calculates dates, days of the week, and days between dates. (It is correct from March 1 1900 to February 28 2100). There is a word guessing game which uses the alphabetic features of the HP-41; you need a second player to provide the words to be guessed. The Arithmetic Teacher can be used to help teach your children to do simple arithmetic. (This uses the same random number generator as the Games Module and the PPC ROM.) A hexadecimal-decimal program converts integers between these two number systems. The Financial Calculations program is similar to the one in the Home Management Module, but has had some features removed. It solves for the unknowns in a cashflow situation involving present value, interest rate, number of payment periods, payment amount per period, and future value of the loan or savings. (The PPC ROM provides a financial program more sophisticated than either of the other two.) The Root Finder program searches for a root of f(x)=0, this is similar to the second program in the Math Module, but less sophisticated. (Lines 44 and 45 are the wrong way round in the original handbook, they should be 44 1E-8, 45 X>Y? Check if this has been corrected in your book or module. You can also try another guess by pressing A in User mode.) The Curve Fitting program is the same as the one in the Statistics Module. There is a Vector Operations program which adds, subtracts, multiplies, or divides two vectors. Only the stack is used, the vectors must be two-dimensional vectors (or complex numbers) in rectangular form. Finally there is a Blackjack game in which you play against the HP-41.

HP-41 Advantage Module

After six years on the market the HP-41 continues to sell very well, but newer calculators and pocket computers have been introduced. The HP-41 Advantage module (Advanced Solutions Pack) designed to give the HP-41 back its advantage over the competition was introduced in August 1985. It gives the HP-41 additional functions and programs in three main fields:

Mathematics and Matrix operations (similar to features on the HP-15C), Computer Science (providing some functions of the HP-16C) and Financial Calculations (giving some of the capabilities of the HP-12C).

Some of the functions and programs are rewritten versions of ones in other modules. For example there is a Time Value of Money program that is very similar to the main financial program of the Real Estate module (and also much like the Home Management module "Financial Calculator" program or the similar program in the Financial Decisions module). However like most of the others, this program runs from a "menu" which is displayed when the program is started and can be displayed again whenever necessary, so there is no need for a keyboard overlay. The program description in the manual is clear and explains what subroutines are used by the main program and how you can call them directly from your own programs. This enhances the manual since it lets you take full advantage of all the subroutines and functions in the module for use in your own programs.

The manual has twelve chapters. The first describes complete programs for creating and using real and complex matrices. The second chapter describes the forty-five functions that are used by these programs. There are many utility functions but also functions that can be called from the user's programs to multiply two matrices, find determinants, to invert and transpose matrices, and to solve systems of equations. Previously these operations were only available as programs in FOCAL; the functions are much faster and finally make the HP-41 suitable for serious work with large matrices. As matrices can be stored in Extended Memory as well as in Main Memory a single matrix could be up to 599 registers long (one register is used as a status header). Most of the matrix handling utility functions come from the CCD module, while some of the matrix mathematics routines come from the HP-15C. One additional general-purpose function, **AIP** (Append Integer Part), is described with these functions. It appends the absolute value of the integer part of a number in X to the contents of Alpha without a decimal point, simpler than INT, RCLFLAG, FIX 0, ARCL Y, STOFLAG, RDN, X<>L.

The third chapter describes a function SOLVE to find solutions to the general equation f(x)=0; I shall describe this below. The fourth chapter describes a program to find roots of polynomials of degree 2 to 5, and to evaluate polynomials of degree 2 to 20. The fifth chapter describes a function INTEG to perform numerical integration, also described below. The sixth chapter covers a program for numerical solution of first and second order differential equations using a fourth-order Runge-Kutta method to step from one point to the next. The step size can be changed at any time.

A set of routines to operate on complex numbers is described in the seventh chapter, routines to handle 2- or 3-dimensional vectors are in the eighth chapter, and routines for three-dimensional coordinate transformations and rotations are described in the ninth. These three sets of operations are inter-related, for example a complex number can be considered to be a 2-dimensional vector. Some of these calculations are available in other modules, but the vector operations are arranged differently from those in the Stress and Structural Analysis modules.

A set of functions for number conversion and Boolean logic comes in the tenth chapter. The number functions allow entry and display of numbers in binary, octal and hexadecimal using the keys A-F for the digits above 9. With the Boolean functions they allow computer science operations to be carried out on the 41. As they have been copied from the HP-IL Development module, nice features of the CCD module functions, such as different sign conventions, variable word length, and the ability to recall hex values to Alpha, are not available. The bug in the BIT? function (which checks if bit X of word Y is set) has been corrected but the BININ and OCTIN bug (described in Appendix C under the HP-IL Development module) has not. The eleventh chapter describes a curve fitting program similar to CV in the PPC ROM; it fits one of four types of curve or finds the best type, without destroying the data, and lets you estimate values using the fitted curve. The last chapter describes the Financial program.

Actually over half the above are improved versions of programs in the Math module; however SOLVE and INTEG are functions not programs, and so are

faster. They are copied from the HP-15C; the 41 uses larger batteries so they run faster than on the 15C as well. To return to a machine language function from a FOCAL subroutine they use the program SIRTN and the function SILOOP in the Advantage module; as these are intended to be used only for this they are not described in the manual. Both functions set up a buffer with the identifier EE (buffers are described in Sections 8.3, 8.4 and 16.7). The buffer always begins at absolute address 0C0, below the key assignmemts and other buffers; this means that SOLVE and INTEG can use certain defined absolute addresses without looking for the buffer. If SOLVE is used on its own then this buffer is 13 registers long, if INTEG is used then the buffer is 32 registers long, with 12 of these reserved for use by SOLVE. If you move this buffer then SOLVE or INTEG fail in SIRTN. The buffer is deleted as soon as these functions complete their work but there must be enough room below the .END. for it. SOLVE acts as a test function; if executed from the keyboard it displays NO if it finds no solution, and in a program it skips the next step. This is very easy to forget when you are writing a program; try to think of the function name as SOLVE? in analogy with X=Y? or BIT?. To compare INTEG with other integration programs I ran it on 1/x↑2 in the same way as in Section 7.3. The results were:

	FIX 4 mode		SCI 4 mode	
upper limit	result	error estimate	result	error estimate
10	0.9000	0.0005	9.0000E-1	1.4301E-5
100	0.9905	0.0050	9.9000E-1	1.5614E-5
1000	0.0679	0.0500	9.9900E-1	1.5662E-5
1 E90	1.0E90	1.0E90	0.0000E00	5.0000E-14

The 1E90 result in the bottom row is wrong, and the stack is not tidied up. The other values in the last two rows are not due to a bug, they are a result of the way INTEG works. The HP-15C gives the same values for the third row and an overflow error in the fourth row. Try to read the HP-15C manual which describes this and SOLVE very well. Evidently it is better to use SCI or ENG modes to specify the required accuracy as a relative value, not FIX to specify it absolutely. One more point to beware of; all the flags 0 to 10 are used somewhere in this module.

To fit all the 117 routines and functions into one module, page-switching is used to make the module into a 12K ROM (see Section 8.3 for an explanation of page-switching). The XROM numbers used are 22 and 24, the same as for the HP-IL Development Module. The Advantage is a useful module for the HP-41, particularly for new owners and people who are short of ports and so want one module with commonly used utilities. It provides a good selection of functions and routines and it should be considered an addition to the range of HP-41 modules, not a replacement for any of them.

PPC ROM

This is a utilities module written not by Hewlett-Packard but by a users' group. It contains those routines that a large cross-section of users wanted most, in the light of their first two years' experience. It is called the PPC ROM simply because it was created by the user group PPC and because it is a Read Only Memory module.

Several types of routines and programs are provided in the PPC ROM. First of all, there are improved versions of programs that already exist in other modules. These are an integrator, a curve fitting program, a financial program, a root finder, complex arithmetic and a random number generator. Then there are many additional programs of the same type; base conversions, differentiation, Julian day conversions, alphabetic and numeric data sorting programs, and a Gaussian random number generator are examples.

Thirdly there is a collection of routines and programs to handle blocks of data and matrices. Blocks can be cleared, viewed, moved, rotated, matrix rows can be initialised, added, multiplied or exchanged. Single elements can be found and examined, maximum and minimum values can be found, individual items can be removed from blocks or inserted in them. There is also a block statistics program which can be used to find dot products.

Next there are several programs that enhance the use of HP-41 peripherals. For the printer, there are histogram plotting routines and high resolution

graph plotting routines, including a routine to plot up to nine different functions on the same graph. There is also a routine to arrange printer output neatly in columns. For the Wand, there is a barcode analyser.

A range of mathematical functions come fifth. Some of these are the kind of function that could have been included in the Mathematics Module, for example combinations and permutations, operations with fractions and fraction conversions, or prime number calculations. Others should really have been built into the HP-41 itself if there had been room for them, for example a modulus function that gives both the quotient and the remainder, or a mantissa view function which displays all ten digits of the number in the X register. Both those functions, and others like them, affect the stack minimally, in the way that a true HP-41 function would.

These last two functions can really be considered to be HP-41 extensions, not utilities, because they provide instructions that the HP-41 could well have had built into it. In fact about half of the PPC ROM is made up of HP-41 extension routines, and they will be discussed in the next section.

One of the good points of a users' group (whether they be bulldozer users or computer users) is that it provides the individual member with access to a pool of experience, expertise and enthusiasm. The PPC ROM is a good example of the workings of such a group provided it has dedicated leadership; no single user would have provided such a product but a group could create it. As a result, every member can get the module, take advantage of other users' knowledge, and contribute in turn by suggesting uses for the routines or improvements to them. Over 40 members did in fact contribute in this way by writing sections of the user manual. The PPC ROM manual is 492 pages long, and the pages are twice as large as those of an HP program module manual. Reading the manual itself (it can be purchased separately) provides inspiration, education, and perhaps exhaustion.

12.8 System extension modules.

Memory-modules.

The first system extension modules provided for the HP-41 were the memory modules. They were "extension modules" because they provided additional memory for the HP-41C, not programs of any sort. Each module contained 64 data registers which could be used for storing programs or data. Since the HP-41 has four ports, a maximum of four such modules could be plugged in, providing a maximum of 256 extra registers. The HP-41 was designed with this maximum in mind, and the HP-41CV and the HP-41CX already have all this memory built in, so nothing can be gained by plugging memory modules into them.

Most program modules and extension modules can be plugged into any I/O port, but the memory modules must be plugged into ports 1, 2, 3 and 4 in that order. If for example you have three memory modules you must plug them into ports 1, 2 and 3. Port 4 is left free for another module, and the Card Reader plugs into port 4 just because this leaves room for three memory modules.

Quad Memory Module.

It rapidly became obvious that the single memory modules were very limited. A user who needed a Card Reader, a Printer and a Mathematics Module could only plug in one memory module, yet it was just this kind of system that needed more memory. The quad memory module was introduced to overcome this problem. It contains four times as much memory as a single memory module, and it can be plugged into any port. Users of other equipment could now have three ports for additional items, and still extend the HP-41 memory to the full. Their only problem was what to do with their unusable single memory modules (HP no longer makes single memory modules, but there is a market for these in the user clubs because people need the module shells). Hewlett-Packard also introduced the HP-41CV with the quad memory already built in, so that the user had the full memory and four free ports.

The story so far is a repeat of what has already been said in Chapter 8. Things got more complicated when Extended Memory was introduced, we shall come to this soon.

The PPC ROM again

Some of the programs in this module have already been mentioned, but it also contains many functions that are really system extensions. Two simple examples are functions that replace X with its mantissa or its exponent (power of ten). The original value of X is saved in L and registers Y, Z and T remain unchanged. (The view mantissa function described earlier does not change the stack at all.) Another very useful routine takes you directly to the last program in memory. This lets you interrupt the writing of a new program, then go back to that program even if it does not have a global label. Other functions test the display, delete the last character in the ALPHA register, find the SIZE and the location of the statistics registers. There is an alternative to the BEEP function. There is a function to let you jump into the middle of a program in ROM, at any selected point; this lets you skip over questions which may be unnecessary. A very useful pair of functions let you extend the RTN stack to more than six pending returns. Other routines let you analyse, alter and save the ALPHA register, save and restore the display mode, suspend, reactivate, and view the key assignments, compress or decompress data, view the flag settings and perform other similar useful operations.

Most of the above routines use Synthetic Programming to achieve results that would otherwise be impossible without microcode programming. The final section of the PPC ROM routines helps you to do your own Synthetic Programming. Some routines achieve easily understood results such as executing unusual TONEs or inverting the status of any flag (including flags 30-55, most of which cannot normally be altered). Others may be more difficult to understand at once, for instance routines to create or decode non normalised numbers, or routines to assign any function to any key (read Chapter 8 if you do not understand these operations). Finally there is a

set of routines to give the user maximum control of memory, interchanging programs and data, and even using one program to let the user automatically write another program.

Most, but not all, of the PPC ROM routines have now been mentioned. A few can be replaced by Extended Functions (such as SIZE?) or functions in other modules, but the majority are still only available in the PPC ROM. Some of the routines will be described in more detail in the chapters on Synthetic Programming. The PPC ROM would be a good investment for many owners who want to extend their HP-41.

After the PPC ROM had been made, it became possible to write machine code programs on EPROM memory. Several collections of programs on EPROM have been made and circulated within PPC and other user groups. Some of these are called PPC EPROMs and should not be confused with the PPC ROM. There has been a long-standing project to develop a second PPC ROM. This is generally called PPC ROM II, but it has not been completed and perhaps it never will be.

Timer, Extended Functions and Extended Memory.

These have already been described but it is worth remembering that they are HP-41 extension modules. It is also important to remember that a maximum of two Extended Memory modules can be plugged into an HP-41. If one is plugged in it can go into any port, but if two are plugged in then one must go into port 1 or 3, and the other must go into port 2 or 4. You should make sure that you understand the differences between the various types of memory modules or you may buy the wrong one.

ZENROM

This is a system extension module programmed for the HP-41 by Zengrange Ltd., a U.K. company. Its main purpose is to make Synthetic Programming easy on the HP-41. In brief Synthetic Programming means the use of numbers and instructions which cannot be created from the keyboard, but must

instead be put together (synthesised). Synthetic Programming will be explained fully in Chapters 14 and 15; SP greatly extends the power of the HP-41 but it is not easy to synthesise the required instructions. The ZENROM makes it possible to create synthetic instructions directly from the HP-41 keyboard as if they were ordinary instructions (hence its slogan "direct key synthetics"). You will probably need to read Chapter 14 to realise just how useful this is. The second purpose of ZENROM is to allow its owner to program the HP-41 in its own machine language, but a special type of memory must also be purchased to store the machine language instructions. We shall come back to this in Chapter 17.

ZENROM provides other useful features. A special Alpha keyboard mode lets you enter all the special display characters which cannot be keyed from the normal keyboard. This includes parentheses, square brackets, the ampersand, hash sign, underscore, and all lower-case letters. Most of the lower-case letters do not show up properly in the display, but they are worth using if you have a printer. The pound sign is also provided, obviously useful in the U.K. There are functions to clear main memory or Extended Memory separately (unlike a master clear which clears everything at the same time). There are functions to encode and decode non-normalised numbers, to go to the last program in memory and to toggle a flag; all these are similar to PPC ROM functions but faster because they are written in machine code. Other functions allow the storing and recalling of non-normalised numbers, and byte-by-byte editing of the contents of RAM memory.

In some ways ZENROM is an alternative to the PPC ROM; in other ways it provides facilities not otherwise available, particularly the machine-code writing and analysis section. As a compromise, I always have both modules plugged into my HP-41!

The ZENROM manual is worth reading on its own. It provides a good introduction to Synthetic Programming and M-code programming. For information on ordering ZENROM, see Appendix B.

Modules for use with HP-IL.

Various HP-IL devices need many different commands, but the HP-IL Module itself can only send a limited number of commands. The ALPHA register is often used to send or receive loop messages, so control of the loop requires complete control of the ALPHA register. Several modules have therefore been produced to allow improved control of HP-IL devices and of the ALPHA register. Most of these modules provide other functions which can be used with HP-IL or to enhance the use of the HP-41 itself. (All of these modules plug into the HP-41, which controls the Interface Loop, not into the individual loop devices.) Seven such modules are available at present as described below.

Extended I/O Module.

This is a general-purpose IL control module. Its main purpose is to provide the commands needed to control IL devices such as modems, non-HP printers and video monitors. A second group of functions is available to control Mass Storage devices. These functions let you get a directory listing of the files on a cassette or a disk, examine individual files, and copy individual files. They also let you use all the space on a disk, not just a part of it as is the case if you use only an HP-IL module. There are functions to copy all the files from one Mass Storage unit into up to 30 other units, to copy all the files and make the copies PRIVATE, and to verify that the copies can be read successfully. Example programs are provided in the manual so that the owner of this module can use it at once.

The module also provides functions called extensions to the Extended Functions, because they are similar to those in the Extended Functions Module. These "Extended Extended Functions" can be very helpful for Alpha operations even if you do not use HP-IL. One such function is ANUMDEL which works like ANUM but deletes the number it has found (and everything to its left) from the Alpha register, so that other numbers can be found in Alpha if ANUMDEL is executed again. All these functions are particularly useful in Synthetic Programming and are therefore described in Chapter 16.

Auto-Start and Duplication Module.

This module serves two fairly specialised purposes. If it is plugged into an HP-41 and the HP-41 is turned on, then it immediately tries to run a Catalogue 1 program called "RECOVER". Failing this, it tries to run a CAT 2 program called "RECOVER". The module itself contains a program of that name so this is normally executed; it tries to reinitialise the HP-41 memory from a Mass Storage write-all file called "AUTOST". These two operations are similar to running a program automatically by turning on the HP-41 with flag 11 set, but they do not require you even to set flag 11. The main use of this auto-start facility is if you want the HP-41 to do some job completely automatically; you just turn the HP-41 on and it does everything according to a pre-written program (it could even turn itself off when it finishes). This can be worthwhile if you want to sell a customer a complete unit including an HP-41 to do some job and the customer wants to have everything done automatically.

The other use of this module is for copying complete Mass Storage devices; the module contains the same three Mass Storage copy and verify functions as the Extended I/O module. The module also contains a program to automate the task of copying and checking a cassette or disk, but this program is provided in the manual for the Extended I/O Module. If you need these functions it is therefore better to buy the Extended I/O Module since it provides so many other useful functions.

An interesting feature of this module is that the auto-start can be disabled by setting flag 35, but no way is provided for setting or clearing this flag. In effect this acknowledges the need for Synthetic Programming, or the PPC ROM, or ZENROM, or the CCD ROM, which are the only means available for setting flag 35.

HP-IL Development Module.

This module provides features to allow complete low-level control of the HP-IL module and loop. It also allows the HP-41 to be used to monitor messages and commands sent round an HP-IL loop controlled by some other device (another HP-41 or an HP-75 for example). As its name implies, these features allow the module to be used by someone who is trying to develop a specific use of HP-IL, for instance a complicated new program or a new piece of HP-IL equipment.

Other people may also find the module very useful. It provides even more complete control of HP-IL devices than does the Extended I/O Module. It provides the same ALPHA register control functions as the Extended I/O Module except for ANUMDEL. It provides a set of functions to enter numbers in binary, octal and hexadecimal. The numbers are treated as if they were unsigned thirty-two bit integers, but they are stored as decimal numbers, so that decimal arithmetic can be performed on them. The following logical (Boolean) operations can also be carried out on these numbers: AND, OR, XOR, NOT, ROTXY (rotate to the right by n bits), and test bit n. This last function, called BIT? will skip a step if the specified bit is clear, not if it is set as stated on page 53 of the manual. The manual should also point out that the least significant (rightmost) bit is bit number zero, and the most significant bit is bit 31. In all these logical operations, the sign and fractional parts of the decimal representation are ignored, so that they are lost except in the case of the function BIT? There are also functions to display numbers in binary, octal or hexadecimal, so the results of all the arithmetic and logical operations can be displayed in the required mode. BIT? does not always display the answer when executed from the keyboard: see the list of bugs in Appendix C.

In order to provide full control of the IL loop, this module has functions to copy, read, receive, analyse and compare bytes. These bytes can be in the stack, in the ALPHA register, or in a special buffer which the module can create (see Sections 8.3 and 8.4 if you do not know what a buffer is). The buffer can be up to 253 registers long (1771 bytes), so it can be used

to send or receive much longer messages and commands than the ALPHA register will hold. Any particular byte in the buffer can be identified by a pointer similar to the pointers used in Extended Memory files. Numbers can be extracted from the buffer in various ways, including a function similar to ANUM but using a specified number of bytes. The contents of the buffer can be printed or displayed as a string of bytes or of IL instructions. Since the buffer only exists so long as the Development Module is plugged into the HP-41, it is assumed that its contents will not get scrambled, so its contents are never normalised. (Normalisation was described in Section 8.2.)

All these features of the Development Module buffer, and the functions to control it, make it an ideal tool for Synthetic Programming which is largely involved with the creation, analysis and storage of bytes and non-normalised numbers. Some people who do not use HP-IL at all have bought Development Modules just to use the non-decimal numbers, logical operations, or buffer operations. The module also contains a function identical to X<>F, and a function to test if a plug-in ROM is faulty (this is actually done by calculating the checksum for the ROM and comparing it with a value saved in that ROM). Finally the Development Module can be used to set the HP-41 so that it turns off, but will wake up and run a user-written program (called INTR) if some other device on the HP-IL loop requests to be serviced. This is done by setting flag 18 before the HP-41 is turned off, and it is similar to the flag 11 auto-start of the HP-41 and to the action of the Auto-Start Module. In a way it is even more powerful, since it allows the HP-41 to be turned on from some other device.

The HP-IL Development Module is an extremely powerful tool for control of the HP-41 and of an HP-IL loop. This power was obtained at the cost of some complication and a few bugs (described in Appendix C). For these reasons and because of its name, it is nicknamed the "devil" ROM. An early version of this module was copied onto EPROM and released to members of PPC. This version is called PPC EPROM 5A or the Monitor ROM. It is like the Development Module but contains fewer functions and more bugs.

Plotter Module

This lets you do two things: create barcode for printing on printers or plotters (including the 82162A printer), and control the HP-7470A plotter. The plotter commands could be generated one byte at a time with the HP-IL Development Module, but that would be terribly time consuming. The Plotter Module therefore provides functions each of which makes the plotter perform a complete operation. These functions are compatible with the Hewlett-Packard Graphics Language (HP-GL) used for plotting by larger computers. A special function allows you to send the pen position from the plotter to the HP-41 ALPHA register, where it can be interpreted by ANUM or ANUMDEL. Repeated use of this function allows you to use the plotter as a digitiser, sending a pattern to the HP-41 which can analyse or redraw it. There are also some programs to create a complete plot making a line graph or a bar chart.

Four kinds of barcode, including HP-41 barcode, can be generated using functions in the module. These functions use a special buffer which can be created and deleted using plotter module functions. The top register of the buffer contains barcode parameters which are set to default values when the buffer is created and which can be altered. This buffer is of less use for general purpose operations than is the Development Module buffer, but it is much easier to generate barcode with this module and the buffer than in any other way.

The module's manual provides some programs which let you use the module and the HP-41 straight away to begin plotting or generating barcodes. You should certainly buy this module if you use your HP-41 with an HP7470A plotter or if you want to produce barcode without using Synthetic Programming.

Data Acquisition Module.

A little-known (by HP-41 users) but powerful Hewlett-Packard instrument is the HP3421A Data Acquisition/Control Unit. It is not normally sold in the

U.K. but can be ordered specially; it is a fairly low cost unit for automatic measurement of voltages, resistances and temperatures. The unit is normally sold with a Hewlett-Packard Series 80 computer to drive it and store the results, but it can also be controlled from an HP-IL loop to make it portable. The HP-44468A Data Acquisition Module provides functions and programs that let an HP-41 control the HP3421A unit. With a Time Module in the HP-41, a mass storage device and printer also on the HP-IL loop, and internal batteries in the HP3421A a data collection unit for field use can be set up. The Data Acquisition Module can also be used to control the more modest HP3468A/B Digital Multimeter.

This module and its manual allow people who have not previously used an HP-41 to treat it as a front panel for the Data Acquisition/Control Unit, creating and editing sequences of commands for data collection and analysis on up to 10 channels. The module contains some general-purpose functions worth describing. Seventeen are just copied from the Extended Functions Module. There are two extra functions called CLKY and AK; these are alternative names for CLKEYS and PASN which are also in the module. Two functions FINDT and FINDX are similar to the Extended I/O function FINDAID for searching the HP-IL loop for a device of a particular type or class. FINDT is actually described wrongly in the manual, it searches the loop for a device whose type is given by MOD(ABS(X),16). Most interesting is the function CLAL which is identical to the HP-41CX alarm clearing function CLALMA. If you do not have an HP-41CX but need this function then you may consider buying the Data Acquisition Module.

Paname ROM.

This is an HP-IL utilities and data array handling module written by members of a user group in Paris. (Hence the module name: **Pa**ris **Name** or just Paname; Paname is also a slang name for Paris.) It contains 122 functions all written in HP-41 machine code. The functions are grouped as follows:

1) 13 general-purpose HP-IL commands, some identical to Extended I/O Module functions.
2) 16 functions to control a video monitor. These include HOME (move the cursor to the top left), cursor movement and scrolling.
3) 7 functions to simplify control of the 82162A thermal printer.
4) 10 functions to control the 82905B impact printer.
5) 19 plotter functions for use with a Tandy CGP110 plotter or the Canon equivalent. This is a cheaper and smaller alternative to using the HP 7470A plotter but requires an HP-IL interface.
6) 12 ALPHA register control functions, including the Extended Functions and Extended I/O Functions such as ANUMDEL.
7) 11 Extended Functions and CX Extended Functions, including all the indirect comparisons.
8) 8 register arithmetic functions, including a function that prompts for the other functions.
9) 13 functions to allow the setting up, initialisation, and control of arrays and matrices. These include the building and analysis of ISG and DSE counters, and initialisation of matrices by name. (For example you can initialise a matrix to contain the names MON, TUE, WED, etc., you can then prompt for values to replace each day of the week.)
10) 13 other utility functions, including integer division, data sorting, Yes/No question, a NOP, reading and writing Extended Memory to Mass Storage. A very clever idea is the CHFLAG function. When this is entered into a program, the Paname Module automatically records the status of all the flags as a text string following CHFLAG in the program. If the program is executed later, the CHFLAG function uses this text string to set all the flags to their status at the time when the program was written.

This module is not designed as an aid to Synthetic Programming; it should be seen clearly as a utilities module. As such it is amazingly successful, combining the most useful features of the Extended Functions, Extended I/O and CX Extended Functions together with others. It is particularly worth buying if you want to upgrade your

HP-41CV into a CX without actually buying one. All the more surprising is that the Paris group managed to write this module despite the unhelpful attitude of Hewlett-Packard in France. (Hewlett-Packard reactions to people who write modules vary widely from country to country. The Paris group were particularly unlucky, but this need not be seen as a criticism of Hewlett-Packard, whose employees have different expertise in different countries.) One more point; the manual for this module will be available in English. See Appendix B for information about ordering.

CCD module

This is another utilities module, written by members of the users' club CCD in West Germany. It concentrates on HP-41 enhancements including Synthetic Programming, and has less emphasis on HP-IL than the Paname Module. The functions can be considered in four blocks; HP-41 system enhancements, utilities, mathematical functions and I/O operations.

The system enhancements are rather like the ZENROM enhancements, but not the same. There is an extra Alpha keyboard, and 16 catalogues (0-F, CAT 0 is an HP-IL device catalogue for example) are provided. XEQ and ASN allow direct entry of any two-byte function (see Chapter 8). Not all the direct-key synthetic features of ZENROM are available.

The utilities allow direct access at the byte and register levels to all of the HP-41 RAM. This too is similar to the ZENROM functions, but is implemented in a different way, by PEEK and POKE functions (familiar to BASIC programmers). Other utilities provide a programmable Alpha back-arrow, save, get and merge key assignments and provide a permanent auto-start (unlike setting flag 11 which gets cleared). You can save key assignments and buffers in two special types of Extended Memory file, and recover them from these files; key assignments can be merged with assignments that are already present. Registers and files can be sorted.

The mathematical functions include a random number generator and logical

(Boolean) functions. There is also a set of functions for entering matrices and performing matrix arithmetic and comparisons. Matrices can be stored in Main Memory or as a new type of Extended Memory file.

The input/output operations include a set of functions to help a program prompt for data and check the legality of the input (check for lower and upper bounds, check for integer values, use default values). Output operations include functions to put results into the right format and send results to an HP-IL device.

There is a set of hexadecimal and logical functions that is more complete than the set in the IL Development module. These allow various word sizes, up to 32 bits, and various sign modes.

A set of "advanced functions" allow control of the return stack and finding of the lengths and addresses of programs. A few functions not mentioned in the manual provide additional utilities; one allows entry into M-code execution in much the same way as the program XRO (mentioned in Section 12.5) allows entry into any address of a FOCAL program in a ROM module.

You may prefer the selection of functions in this module to those in ZENROM or the Paname ROM. You should be warned that this module contains more bugs than ZENROM and is not designed to be compatible with ZENROM (which came out earlier and should really have been allowed for). The manual is not yet available in English (at the time when this book was printed). Ordering information is provided in Appendix B. The worst bug is described in Appendix C.

12.9 Diagnostic and Service Modules

A number of HP-41 modules are made purely for use by HP Repair Centres. There is an HP-41C Service module which tests the HP-41C and also the HP-41CV, Card Reader and plug-in modules for hardware faults. There is an HP-41CX Diagnostic ROM module which tests the HP-41CX and plug-in modules for faults. Some of the tests in this module can also be used to check the HP-

41CV, Timer Module, Extended Functions Module and Extended Memory Modules. Further modules test the HP-41 printer and other peripherals.

In addition to these modules, Hewlett-Packard prints their own service manuals which describe the hardware and how the modules can be used to test it. Neither the modules nor the manuals are available for sale to the public, but a few copies have found their way into the hands of some user groups. You may like to see if your local group has any; they are very interesting to the really dedicated user, and they can help you check if there is anything genuinely wrong with your equipment before you send it to a Repair Centre.

Exercises.

Exercise 12.A Use a Wand (you may be able to borrow one) to read the Alpha barcodes in Appendix F and make the text strings:

Y" = f(X,Y,Y') "JACK & JILL"

Is this easier than other ways of making such strings?

Exercise 12.B Use a printer to print the two strings from Exercise 12.A. You can make the strings by using a Wand, or by using ACCHR.

Exercise 12.C If you have bought any plug-in program modules and have used them, try to assess their value to you. Do they save a lot of space that would otherwise be taken up by programs in memory? Would you have done as well to buy just a couple of programs from the User Library (see next Chapter)?

Exercise 12.D Now that you have read about the system extension modules, do you think you need any of them? If you have some of these modules, do you think you could do without them in your HP-41?

Exercise 12.E Can you see any obvious features that you think the HP-41 should have but which are not provided by any of the modules described in this Chapter? See the sections in Chapter 16 on non-programmable functions, and the section in Chapter 17 on missing functions, for some comments. The "wish lists" compiled by HP-65 and 67/97 user groups were important in defining the HP-41; today's "wish lists" may be fulfilled in a future HP product. The HP-41 Advantage module gives the HP-41 some of the advanced features of other HP calculators. However only a new model can produce new hardware features, such as a large display with graph-plotting as is provided by the Casio FX-7000G calculator.

CHAPTER 13 - ADVANCED PROGRAMS AND USER GROUPS

13.1 Advanced programs?

The previous Chapter described the advanced programs provided by plug-in modules for the HP-41. This Chapter will continue the theme of advanced programming by describing other sources of programs and by providing advice on writing your own advanced or complicated programs.

You can greatly extend the usefulness of your HP-41 by knowing how best to go about writing complicated programs for yourself, but it can be far easier to combine your own programs with ones that have been written by other people to do a particular job and tested by many users. It is therefore important to realise that you can write advanced programs that use ideas or complete subroutines provided by other people in books, journals and elsewhere. This is one of the reasons why joining a user group can be very helpful: you can borrow programs from other users who are specialists in a particular subject, other users can help you by testing your programs, and groups of users can work together on problems that no single member could tackle efficiently.

Advanced programming can lead you to try out operations not described in the manuals or not even foreseen by the manufacturers. The results of some operations like this can be unexpected and very useful; the members of a user group can provide a pool of knowledge and interest to develop the use of such operations. We shall come back to this in the next Chapter.

13.2 Books and Journals.

Programs and programming tips for the HP-41 can be found in many kinds of books. Some of these are written especially for the HP-41 such as "Calculator Tips and Routines Especially for the HP-41C/CV" by John Dearing. Some are written for the HP-41 and other pocket computers, for example "Curve Fitting for Programmable Calculators" by William Kolb. Others are written for RPN calculators in general ("Algorithms for RPN

Calculators" by John Ball) or indeed for all advanced calculators ("Scientific Analysis on the Pocket Calculator" by Jon Smith). There are also a great many introductory books for calculator users, but most of these are for calculators simpler than the HP-41. The manuals for the PPC ROM and for ZENROM provide a great deal of useful information about the HP-41 as well as about the modules themselves.

Ideas for programming the HP-41 can also be found in various types of computer science publications. Books on programming style, algorithm design or data structures often cover the subjects in such a way that their suggestions can be applied to HP-41 programming. If the HP-41 is your only programmable machine then you need not go out of your way to look at computer science books, but if you have any reason for looking at such books or journals, then look out for ideas that will apply to the HP-41.

An entirely different kind of programming book is that designed to cover a particular application; flight planning, hydraulic engineering or interior decoration for instance. Some of these books are written specifically for calculator users, others provide programs almost in passing. Professional journals may also publish HP-41 programs for specific applications. Some of these carry advertisements for HP-41 programs written by professional companies.

Of special interest are magazines and journals devoted solely to HP products. Hewlett-Packard publishes in-house journals, which are sometimes shown to users of their equipment, and the HP Digest which contains articles about the design and use of new Hewlett-Packard equipment. HP Key Notes was a particularly useful magazine especially for users of HP calculators and portable computers, but this was discontinued when the editor retired. A commercial magazine for Hewlett-Packard computer users, called "Professional Computing" was intended to replace this; the original publishers, John Wiley, withdrew after six months, fortunately Camden Communications Inc. (Highland Mill-PO Box 250, Camden, Maine 04843, USA) now publishes "Professional Computing" and have continued to print columns for series 40 and series 70 computers. Old copies of both these

publications are well worth looking at, Key Notes in particular was full of clever ideas submitted by HP calculator users. An alternative to Key Notes is provided by the journals of the various user groups; see Section 13.6.

It would be impossible to list all relevant books and journals here, let alone describe them, but a list of books and journals known to the author is given in Appendix A. EduCALC (see Appendix B) regularly publishes a catalogue providing up-to-date information about equipment and books, it is a good idea to obtain this from time to time.

13.3 Adapting programs from other calculators.

You will often find that someone has already written a program to do the same task as you need to do, or a program that does something similar. It can be distressing to find out that the program has been written for a brand X, or model Y, calculator - not for the HP-41. It may be worth learning how to translate programs onto the HP-41, then you will provide yourself with many new sources of programs; books, articles and program libraries or user clubs which specialise in other calculators.

Translating programs from the HP-67 and HP-97 is no trouble at all if you have an HP-41 Card Reader. This will read and translate program cards from these calculators directly for use on the HP-41. A few incompatibilities remain; these were described in Section 12.3.

Many of the programs in the HP-41 program modules and solutions books are adapted versions of HP-67/97 programs. If you do not have a Card Reader, or if you want to translate a program from some other RPN calculator then the main problem is to find which HP-41 functions correspond to which functions on other calculators. For example, most earlier calculators used the name GSB (Go to subroutine) whereas the HP-41 calls the same function XEQ (Execute subroutine). The best solution is to get a manual for the other calculator and an HP-41 manual, then make up a translation table. The book "Algorithms for RPN calculators" by John Ball contains a long list of translations and differences between the HP-45 and various other RPN

calculators. You could use these translations, then make up your own translation table from the HP-45 to the HP-41. Your table should include warnings about features that cannot be translated directly, for example the HP-65 skips the next two steps, not just one, if a test is false. This particular feature can be replaced by inverting the test and following it with a GTO label, then putting that label after the next two steps. Most features of earlier calculators can be translated in a similar fashion, but programs from some specialised calculators can be much more difficult, because these calculators have specific features for their field. Financial functions from the HP-12C can usually be replaced by programs in the HP-41 Financial Decisions Module, or one of the other financial modules. Matrix arithmetic and complex arithmetic on the HP-15C can be performed by programs or functions from plug-in modules such as the Mathematics ROM, the PPC ROM or the CCD ROM. Logical operations and calculations in non-decimal bases on the HP-16C can be simulated by means of functions on the HP-IL Development Module. Many of the above features are provided in the new Advantage Module. All these modules were described in Chapter 12; a module can be cheaper than buying another calculator.

Programs from calculators that do not use RPN, or from computers, can be much more difficult to translate. If the program works out an equation then it is best to find out what the equation is (by deciphering the program if necessary), then writing it down and turning it into an HP-41 program. A similar method should be used if the program is more complicated; the program should be deciphered and written out as a set of instructions or as a **flowchart**. (A "flowchart" is a diagram which uses boxes of different shapes to describe different operations, with the operations described in words inside the boxes. Operations that follow one another are connected by lines with arrows to show the order in which they are carried out.) The set of instructions or the flowchart can then be rewritten as an HP-41 program. One of the advantages of using a flowchart is that it can give you an overall view of a program, helping you to redesign it for more efficient use on a different machine.

Once you have translated a program and put it into the HP-41 you must check

that it gives the same results as did the original program. Run a few examples on both machines if you have both. If you are translating a program from a printed article, run the program on the HP-41 and also do the calculation going step by step through the original program on paper if necessary. Do more than one test and use realistic numbers (not just 0, 1 and pi). Do not be alarmed if the answers are not exactly the same to ten significant figures. Many calculators are less accurate than the HP-41, but the TI-58 and the TI-59 are more accurate on some functions because they have two extra guard digits which are not rounded off after each instruction (whereas they are on the HP-41). If the results of several tests are the same to seven significant figures on the original program and the HP-41 program, then the translation is probably correct.

You may of course have to adapt an HP-41 program for use on (horror) another calculator! The principles outlined above will also work, but what if the other calculator does not have one of the functions you need? If you are a computer scientist who uses an HP-16C most of the time but you sometimes run HP-41 programs which calculate a sine then you might need a short program to calculate SIN on the HP-16C. The easiest way out is to use a truncated Taylor or MacLaurin series, if you know how to work out the terms. If you do not then you could look up the series somewhere. The book "Scientific Analysis on the Pocket Calculator" by Jon Smith includes a list of series approximations to logarithmic, exponential and trigonometric functions in Table 1-4.

13.4 Hewlett-Packard Solutions Books.

Apart from making plug-in program modules for the HP-41, Hewlett-Packard has also published a range of programs in HP-41 Users' Library Solutions books. Each book contains a selection of fairly specialised programs, taken from the Users' Library. An example is the High-Level Math book which provides programs to calculate special functions that are not available in the Math Module. Each book contains about 10 programs which can be covered by the same general title, although some users may find only one or two of these programs to be much use. Each program comes with a

program description, user instructions, some examples, a listing and barcode (except for very early editions which do not contain program barcode). Some books have odd program steps or unusual instructions; these were explained in point i. of Section 3.6.

There have been at least 29 Solutions books published for the HP-41, but some are out of print (e.g. Cardiac/Pulmonary) and others have never been widely sold or advertised (Applied Statistics I and II for example). New books are also sometimes published (for example Games II), so a list is not given in this book since it would be neither complete nor up to date. It is best to obtain a Hewlett-Packard leaflet (these are updated every 6 months or more often) or ask your dealer what is available. You may also find it worth getting one of the HP-67/97 solutions books (40 were published) or application packs (there were 10, all with magnetic program cards which can be read by the HP-41) although most of these can now only be obtained second-hand. Some of the HP-41 advertising leaflets contain a list of the programs available in each book and it is worth checking which book or books contain programs of interest to you. Since the solutions books are much cheaper than the plug-in program modules it may well be worth buying one or two to provide some useful programs. You may want to use the programs just as they are given, or you may want to change them, or use parts of them in your own programs.

13.5 User Libraries and User Clubs.

In addition to providing program modules and solutions books, Hewlett-Packard maintains libraries of user-written programs. This is a useful service for people who need specialised programs but do not have the time or means to write their own. Authors are encouraged to submit their programs by being rewarded with coupons that can be exchanged for Hewlett-Packard products or for other programs. People who want programs from the library can subscribe to it on an annual basis. Subscribers obtain programs on payment of a fee for handling, postage, duplication (of documentation and barcode) plus the cost of magnetic cards or cassettes (the programs can be recorded on these if required). New programs and

their documentation must meet certain requirements (stated in a book that you get when you join) before they are accepted for inclusion in the library, but they are provided purely on an "as is" basis. This means that users must check the programs for themselves and neither Hewlett-Packard nor the contributor shall be liable for any incidental or consequential damages. (Neither Hewlett-Packard nor the contributors could afford to run a library of this sort at a relatively low cost if everyone had to be insured against possible claims for damages.) The book that you receive when you join the library, called the "Users' Library Programmer's Reference Guide" can be bought separately from Hewlett-Packard for US$10. It is well worth having because it provides a collection of useful advice about writing and documenting programs. On joining the library you also receive a catalogue of contributed programs so that you know what can be ordered, and this is updated twice a year.

The Users' Library has separate sections for different HP programmable calculators and computers, but the HP-41 and HP-67/97 libraries were kept together. This meant that HP-41 users could order HP-67/97 programs and translate them for their own use. The HP-41 Card Reader's ability to translate HP-67/97 programs was advertised as a feature to allow easy use of the many programs written for the HP-67 and HP-97 so it seemed sensible to provide these programs to HP-41 users. Unfortunately, the library decided to close down the HP-67/97 section when manufacturing of the HP-67 was wound down; this was obviously inconvenient for those HP-41 owners who had bought a Card Reader for the very reason that they would be able to use HP-67/97 programs. At present there is a combined library for HP-41 and HP-71 programs, because the translation module lets HP-71 owners use most HP-41 programs. HP-41 owners used to have access to a separate Users' Program Library in Europe (UPLE), but this took the more logical (though even more inconvenient) step of closing down entirely (HP-67/97, 41; the lot) in October 1983. Among reasons given for the closure was the widespread availability of programs sold by many independent companies; but have you ever tried to get a list of all these companies from your local HP office?

The loss of the libraries was not complete though, because Hewlett-Packard promised to let user groups have access to the sections that were closed. In the United States, the HP-67/97 programs were transferred to the largest group, PPC. If you know of a program you want from this section you can order it from PPC at: 67/97 Library, P.O. Box 90579, Long Beach, CA 90809, U.S.A. They have however disposed of programs that no one has ordered for over 5 years. If you want to obtain such a program then your best chance is to ask for it through the HELP columns of user group newsletters. The European Library was to have been divided up by languages, and programs in each language were offered to a user group using that language. The UK group provided HP with a list of the English language programs that seemed likely to be required, but as of August 1985 they had still not arrived. If you need a particular program it is best to ask at your local group if one of the members has it.

This brings me back to the subject of user groups themselves. The members of an HP equipment user group are usually expert in some subject and enthusiastic to share their expertise and exchange ideas. The groups are not financially supported by Hewlett-Packard or organised by HP though HP employees or consultants are sometimes active members. This means that the members of the groups are users who are sufficiently motivated that they go out and join, or even set up, such a group themselves. (I am not writing about large computer users whose employers usually pay to belong to a manufacturer-supported group.)

13.6 The benefits of belonging to a User Club.

Up till now, I have described "User Groups" as if they were collections of dedicated people programming calculators, treating it all terribly seriously. This is indeed one thing that user groups are for: they allow people to exchange ideas and help each other write better programs, but it is not their only purpose. It is often more accurate to think of them as clubs whose members share a common interest in using a particular make of calculator, but then exchange ideas and help on a variety of subjects. Club meetings can be informative and amusing or serious. You can discuss

the electronics of an HP-41, or electronics in general; you may find yourself talking about econometrics on an HP-41 or the predicted forthcoming collapse of the world economy. Some people have found new jobs through user clubs, either with companies that use Hewlett-Packard equipment or with HP themselves.

Help is available at all levels: a new user can find out what is meant by something that is not clear in a manual, and a company director can find a professional programmer to write a specialist application program. (A special register of programming consultants has been printed - see Section 13.7.) All of them can learn of special reductions on equipment prices or buy second-hand items from other members. User clubs often hear first of new products, and some groups of users design specialist equipment, for their own use or for sale to other members.

These groups vary in size and nature. Some are just small and temporary, for example a few students at one college with a shared interest in using one type of calculator. Others are large organisations with hundreds or thousands of members. Some large groups have local sections (known as "Chapters") or smaller locally based affiliated clubs, or just local meetings (for instance the British club currently holds local meetings in London and Leeds). Large groups will provide application forms and information about themselves if you write to them at the addresses given below and enclose a self addressed envelope with stamps or International Reply Coupons. Smaller groups advertise locally, or you can obtain a list of them from CHHU or PPC in the United States. The larger clubs can provide advice if you want to set up a local section.

Some groups publish their own newsletters or journals. These are particularly valuable for people who cannot attend meetings but want to stay informed and to share in the constant exchange of information provided by clubs. Back issues of journals, particularly the PPC Journal (previously known as HP-65 Notes, later published for a time as the PPC Calculator Journal, with a separate Computer Journal), contain a wealth of information and show how certain subjects have evolved. Journals do not

contain all the information on a given subject, additional information is published in Conference notes and reports, or is simply repeated by word of mouth. Occasionally there will be sufficient interest in a particular topic that a group will twist the arm of a member until they extract a promise to write a book. This book is the outcome of such an exercise; the British group wanted one book answering common questions and problems, and another book describing recent advances in Synthetic Programming, particularly those connected with the use of Extended Functions. They got this one book, covering both subjects and filling in the spaces between! One problem with such a book is that it has to refer to material which has been published in journals; but without infringing copyright. In this book I have referred to items from journals but the references are generally in review form; "you can obtain information of such-and-such a type in this journal". This means that the book contains a wealth of introductory material but for details of techniques or program listings you may have to refer to the journals; another reason for joining a club, since you should be able to borrow journals from members or a club library.

Overall then, a club provides programs and expertise, but it provides much more; what comes out depends on what the members put in. Maybe you should become a member too. The address of the UK club, HPCC, is given below, followed by an alphabetical list (by country) of the national clubs that have official contacts with HPCC: the addresses are correct at the time of going to press. (Many other clubs exist, particularly in Europe. Apologies to clubs I have left out, I used the HPCC list - please let me know if you want to be included if I reprint the book.) The clubs are not commercial organisations, so do not expect that a reply will always come by return post, and remember to include a self-addressed envelope plus stamps or International Reply Coupons to cover the cost of return postage.

The British club is **HPCC** (Handheld and Portable Computer Club):
HPCC
Geggs Lodge
Hampton Road
Deddington, Oxford OX5 4QG
United Kingdom
[Journal - Datafile]

Australia

PP Melbourne
Box 512
Ringwood, Victoria 3134
Australia
[Journal - PPM Technical Notes]
[Formerly PPC Technical Notes]

CHHU-SYDNEY
c/o K. Besley
Charley Business Services
22 Elsie St.
Burwood, N.S.W. 2134
Australia

Austria

CCA, Computerclub Austria
P.O. Box 50
A-1111 Wien
Austria
[Journal - CCA Journal]

Belgium

PCX-Belgium
Postbus 205
B-8000 Brugge 1
Belgium
[Journal - PCX]

Denmark

PPC-Danmark
c/o S. Petersen
G1. Landvej 19, 2620 Albertslund
Denmark
[Journal - USER]

France

PPC-PARIS
B.P. 604
75028 Paris Cedex 01
France
[Journal - JPC]

PPC-T
now closed

West Germany

CCD e.V.
Postfach 110411
Schwalbacherstr. 50 Hhs.
D-6000 Frankfurt 1
West Germany
[Journal - PRISMA]

Holland

Contact the bookshop
TH Boekhandel Prins -
(address in Appendix B) or
HP-GC, Quellinjnstraat 47-3
1072 XP Amsterdam
The Netherlands

U S A

Handheld Programming Exchange
P.O. Box 566727
Atlanta
GA 30356
USA
[Journal - HPX Exchange]

PPC (Personal Programming Center)
POB 90579
Long Beach
CA 90809
USA
[Journal - PPC Journal]

PPC has not published any journals for a year, but runs a bulletin board and sells back issues.

The clubs are international in nature, you can join several or choose one you like.

Incidentally, some of you may be familiar with PPC but not with HPX. HPX is the successor to CHHU. CHHU was founded by Richard Nelson, and the CHHU Chronicle proved to be an excellent source of information on both the HP-41 and the HP-71. Back issues of CHHU are available through HPX. HPX was founded by Brian Walsh in 1986, and like CHHU is a volunteer, member supported noncommercial club of Hewlett-Packard handheld users focusing on the selection, application and use of today's true personal computers.

13.7 Buying and ordering programs.

Should you need a program that is not available through one of the sources mentioned so far then you have three remaining choices. You can buy the program from a company, you can pay someone to write the program, or you can write the program yourself. A few companies sell specialist software; they usually advertise in an appropriate journal or magazine, for example the "Oil and Gas Journal". You may also be able to obtain the names and addresses of these companies from your local HP sales offices.

If you want a really specialised program then you may need to get someone to write it for you. Large companies can give the task to their own workers, or they can employ people specially to do the job (they might even look for people at the nearest user group). Smaller companies and individuals usually need to find just one suitable programmer whom they can seek through personal contacts or local advertising. A local user group can be a good contact in these cases too, both for those needing a programmer (often part-time or for just one particular job), and for those seeking work. The American group PPC has taken this a step further by printing a list of people willing to undertake specialist work in various fields. This is available from PPC (address above) at a price of $4.25 plus the cost of mailing the 9 oz document. The list is called the "PPC Consultant Register" and contains details of the kind of work, task size undertaken and relevant experience of PPC members willing to take on programming and allied tasks on HP portables. You may find it worth getting a copy if you are looking for someone to do a specialist programming job. Some entries represent not individuals but companies willing to undertake extensive programming tasks.

13.8 Writing advanced programs yourself.

If you cannot find a program to suit your needs, and cannot find anyone to write the program for you, then you may have to write the program yourself. You may well be able to build most of it from pieces of other programs (which can be obtained in the ways described above). Maybe you want to write your own program, but you may dread the thought. In either case, some helpful advice will do no harm, so here it is.

Look for similar programs because you may be able to re-write one instead of starting from scratch. Look for several programs which each do part of the task and see if you can put them all together, perhaps by using them as subroutines called from your own program. You could for example obtain one program that calculated Value Added Tax and a second program that converts between various European currencies, then combine them to calculate the total tax you expect to pay on a product which you plan to sell throughout the EEC (European Economic Community -- the Common Market).

Write programs efficiently. Look at published programs, see what they do well and avoid what you think is wrong in them. If you are writing a long program you should make sure you fully understand the problem you are dealing with and the answers you require. You should then plan out how you will design the program, following the five points given in Section 6.10 (initialisation, input, processing, output, tidying). Once this has been done, write the program as a set of **program modules** each of which does one part of the job. "Module" as used here means a distinct piece of a program, not a plug-in ROM module. A module can be a part of a program that you call as a subroutine, or it could be a piece that does a particular job, then goes straight on to the next section. Some of the modules could be taken from other programs. Test each module separately, then put them together, this is really a restatement of the advice in Section 6.10; you should reread that section.

When you have finished writing the program and making it efficient, you will need to test it. The first test is "does it give the right answers?"

This requires you to check how the program works on problems to which you already know the answers. You should also check what will happen if you use unlikely or meaningless values. (Will your orbital calculation program allow a satellite to move faster than the speed of light without any warning?) You must write a final version of the program documentation (see Section 6.10 again). Serious problems often show up when you write the documentation since you are forced to think about the whole program while you are describing it, and it is far better to deal with these problems now than later on. It is at this stage that you may decide to redesign parts of the program, for example assign various functions to a conveniently laid out set of keys or combine some subroutines to save space. Many programmers imagine that they will be able to document their programs months after these have been written, but few actually have the talent to do this without rechecking the whole program from scratch. The moral is: once you have written an advanced program, document it while you are still proud of it and remember how it works.

Exercises

Exercise 13.A Find a list of the programs available in HP-41 solutions books. Lists are available in pamphlets advertising the HP-41 series, and in Users' Library Catalogues. Go through the list and check how many programs are useful in your own application. (You may find there are none, for example the Physics solutions book covers thermal physics, mechanics including relativity, and nuclear physics, but nothing about solid state physics or plasma physics.)

Exercise 13.B Pick a fairly long program, either in this book or elsewhere, and read through it. Look for clever programming tricks, but also for inefficient and wasteful programming. It is much easier to criticise someone else's programs than your own. Now choose one of your own programs, look for similar faults, and try to improve it by applying some of the tricks you have seen in the other program.

Exercise 13.C A type of program required by many HP-41 users is a polynomial fitting routine. User club journals publish many such programs; if you need one try POLYFIT by Julian Perry in DATAFILE V2N5P24. If you do not have DATAFILE, try to find some polynomial programs in your local group journals.

PART IV

Synthetic Programming

CHAPTER 14 - SYNTHETIC PROGRAMMING

14.1 How many bytes make a million ?

In Section 8.8 I described three ways of writing the number one million in an HP-41 program. The first was to put in a step consisting of **1,000,000** which takes seven bytes. The second was 1E6 which takes three bytes, and the third was 6 , 10↑X which takes just two bytes but is slower. 1E6 takes the least time in a running program, but it wastes one byte. Is the 1 at the front really necessary though? After all, it is a common practice to write 10^6 instead of $1*10^6$ in arithmetic expressions. Removing the 1 would make a worthwhile saving in a long program, but could the HP-41 be made to use E6 as a program step instead of 1E6 ?

The answer to this question may not seem to be important, yet it is an example of saving space and it leads us to the whole subject of programming techniques which the HP-41 designers made possible without themselves realising it. In this chapter we shall delve into the area of operations that cannot be executed immediately from the keyboard and yet can be performed with just a few extra keystrokes. The power of some operations described later should impress you if you have not met them before, but let us first go back to the simple example above.

Can E6 be created and will it work? If you look at some of the HP-41 Solutions Books you will find programs that contain steps of exactly this sort. Two examples: line 62 of the Bessel and Error Function program in the High-Level Math book is E-9, line 56 of the Ventilator program in the Cardiac/Pulmonary book is E2. If you try to enter these programs from the keyboard you will only be able to get 1E-9 and 1E2, but if you have an Optical Wand and read the barcode given in the books then you will get the exponents on their own and the programs will work normally. It is clear that exponents without a 1 in front can be created. In fact both the programs mentioned were copied from HP-67/97 programs. The HP-67 and HP-97 do allow exponents to stand on their own, and the HP-41 Card Reader will translate this directly to an HP-41 program as an exponent on its own.

(The HP-41 Card Reader makes no attempt to alter exponents and sign changes in an HP-67/97 program. This can cause problems; see Bug 6 in Appendix C.) If you have a Card Reader and an HP-67 or an HP-97 then you can write a 67 or 97 program containing the two lines EEX , 6. You could then save this program on a card, read the card into your HP-41, and have the step E6. Alternatively, if you have a Wand you could look for a Solutions Book containing a program with the step E6, then read that program and delete everything except the E6.

Even if you do have the equipment to do this, you would have to repeat the whole process to create a different step, such as E7. It would be much simpler if you could create 1E6, or 1E7, or any other exponent, and then delete the 1 at the front. This would be rather like editing a program with the function DEL, except that DEL deletes whole lines whereas we now want to delete a single byte. What we need is a new program editing function that works one byte at a time. Does such a function exist? It is not described in the HP-41 manuals, but neither is E6. You will soon see that a function which lets you delete one byte at a time does exist and can be assigned to a key for use in USER mode, just like DEL.

14.2 Non-Normalised Numbers, tumble dryers and cement mixers.

Before we go on to byte editing, two warnings are necessary. First, you must realise that you are going to alter the contents of the HP-41 memory in ways that the HP-41 designers did not plan for. The HP-41 manuals are therefore not going to be much help. You should read Chapter 8 instead; it describes exactly how numbers, programs, key assignments and so on are laid out in the HP-41 memory. What you are going to do is to create new combinations of bytes in the HP-41 memory. A new combination might create the step E6, or RCL 100. The keyboard will not allow you to enter either of these directly. A new combination of bytes in the key assignment area can create entirely new functions, as was mentioned at the end of Section 8.6. A new combination of bytes in the X register might create the number 5.2E270. This is a non-normalised number larger than the HP-41 can normally handle but sometimes useful in itself. For example you can take

the LOG of this number and get the right answer. All these combinations of bytes are impossible to create directly from the keyboard, but they can all be put together, or **synthesised**, using well-understood techniques. Any such unusual combination of bytes can be referred to as a **Non-Normalised Number** (**NNN** for short), and the techniques of creating NNNs, storing them in useful places, and using them are called **Synthetic Programming**. Chapter 8 laid the foundations for an understanding of Synthetic Programming (or **SP**) by explaining the layout of the memory. This chapter, together with the next two, will take you through the fundamentals of SP. You will not learn everything about SP from this one book (nor from any other one book), but it will introduce the ideas and will then provide new information on how Extended Functions can be used with SP. The original SP book "Synthetic Programming on the HP-41" by W.Wickes is always worth reading and there are other books too. "Synthetic Programming Made Easy" by Keith Jarett provides a good introduction to the subject. These books are described in Appendix A.

The second warning concerns Hewlett-Packard's attitude to Synthetic Programming. Imagine, for a moment, that a construction engineer has bought a new tumble dryer. After a few weeks of use, the tumble dryer air pipe gets blocked and the machine begins to make strange noises. The owner notices that these noises sound like a cement mixer in action and decides to investigate further. Lo and behold, the damaged tumble dryer turns out to work perfectly well as a cement mixer; what is more it is far cheaper to buy than a cement mixer. Our construction engineer tells a few friends and soon there is a brisk business in tumble dryers which are destined for use as cement mixers. Now the company that makes these tumble dryers is delighted at the upturn in sales, but they are not specialists in cement mixing. They happily sell their tumble dryers to anyone, but they refer all questions about cement mixing to the person who pioneered this new use of their machines.

The attitude of Hewlett-Packard to Synthetic Programming on the HP-41 is similar. They designed and built the HP-41 as an advanced calculator with a well-defined instruction set, and they wrote the manuals with this in

mind. If the users want to do something extra, that's fine and it helps to sell more HP-41s, but Hewlett-Packard do not employ a team of Synthetic Programming specialists. Instead they refer users to the groups which pioneered Synthetic Programming and to the books which describe SP.

I hasten to add that Synthetic Programming does not damage the HP-41 (unlike cement mixing which is likely to impair the normal functioning of a tumble dryer quite severely). The worst that can happen is that you might get a MEMORY LOST, or a hang-up which can be dealt with as described in Chapter 4. This is usually caused by improper storage of an NNN into a register used by the HP-41 **operating system**. (The operating system is the set of built-in programs that control the internal workings of a computer.) Neither a MEMORY LOST nor a hang-up will harm your HP-41, but losing some information can be painful so you should always make a separate record of any important programs, data, or alarms before you embark on some SP experimenting. If you do have any problems with SP you must bear in mind the second warning above; SP is not an activity that the manufacturer will help with. You must contact a user club, not HP. This is expressed by the acronym **NOMAS** (NOt MAnufacturer Supported). Several documents which are used by Synthetic Programmers have been given to the user groups by HP on this understanding. Users are welcome to read these documents but they are stamped "NOMAS - recipient agrees **NOT** to contact manufacturer". This is not a severe imposition since the SP experts are members of user groups and can be contacted through these groups anyway, we can thank Richard Nelson for the NOMAS arrangement between the user groups and HP.

None of this is to say that HP has anything against Synthetic Programming. After all, it does help to sell even more HP-41s. Programs containing Synthetic instructions are accepted by the User Library, and Synthetic Programming is discussed in publications about HP products. You can use SP quite happily and so join the many other people who have learned to extend their HP-41 through Synthetic Programming. So let's get to it!

14.3 Your first synthetic tool.

You can begin Synthetic Programming by creating just one instruction. This will be used to create other instructions and then complete programs. These further instructions and programs can become your tools for more complicated Synthetic Programming, but everything is "bootstrapped" from just one instruction.

Some users will not really need to go through this bootstrapping process. Owners of the PPC ROM will have the tools already available, and the ZENROM provides not only the tools but even direct key synthetics (so that the synthetic instructions can be entered directly from the keyboard like ordinary instructions). Owners of Card Readers or Wands can read in the programs from magnetic cards or from barcode in books. But even these lucky people can learn something by starting from scratch; this section is worth reading whether or not you already have access to synthetics.

Instead of a function to delete bytes one at a time (rather like DEL), we shall use a function that splits up a row of bytes, chopping off the bytes one at a time. This is more useful than deleting because the bytes can be put back later if necessary. To see how that can be done, you need a detailed understanding of how HP-41 programs are edited. Let us start with a subroutine made up of the program steps:

LBL 21 , 1E6 , + , RTN

If you have entered the above subroutine from the keyboard, it will be represented in memory by these bytes (the byte values here are in decimal):

207 , 21 , 0 , 17 , 27 , 22 , 64 , 133

If you are lost then go back to Chapter 8 and you will see from Table 8.1 that each program step is represented in memory by a group of bytes or a single byte. For example LBL 21 is made up of the prefix 207 (which stands for LBL) and the suffix (or postfix) 21. The bytes in a program normally

come one after another without any spare bytes in between, but numbers (like 1E6 above) are created with a spare byte (equal to zero) in front of them. This is to prevent two numbers from being merged into one (see Section 8.5). In this particular example, the zero (or **null**) byte before the 1 is not actually necessary. (The 1 is represented by a byte value of 17.) You could get rid of this null byte by packing, or you could replace it with an extra one-byte instruction. If you decided to round the number to which this subroutine will add a million, you would go to the line showing LBL 21 and key in XEQ "RND". The program would become:

LBL 21 , RND , 1E6 , + , RTN

represented by the bytes:

207 , 21 , 110 , 17 , 27 , 22 , 64 , 133

The null is replaced by 110, which stands for RND. The HP-41 always tries to fill out unused bytes in this way if you edit a program. What happens though if there is no spare room for the new step? Say you decide to save the number that is in X before adding a million to it. You go to the line which shows RND and you press STO 30. The program becomes:

LBL 21 , RND , STO 30 , 1E6 , + , RTN

The HP-41 had to make room for the new step. It did this by providing an empty register (or seven bytes) after the RND. To do this, the HP-41 moved everything below the RND down by seven bytes, starting at 1E6 and finishing with the .END.; this left seven empty bytes between RND and 1E6. Two of these were filled in by STO 30, the rest remained unused, so the bytes representing the subroutine are now:

```
207, 21,  110,  145, 30,   0,  0,  0,  0,  0,    17, 27, 22,   64,  133
\___/       \___/                                 \_____/
```

I have used braces to mark off each group of bytes that belongs together in one instruction. You can see clearly that the new instruction, STO 30, has taken up just two bytes (145,30), leaving five bytes free. What would happen if we did not use a two-byte instruction but were to put in an instruction that seems to be eight bytes long? Would it fill up all the seven free bytes and grab the next byte too?

Indeed this might happen if an apparently eight-byte-long instruction was put in all at once, but in fact an eight-byte instruction (such as a text string) is usually created one byte at a time, and after the first seven bytes have been put in another seven bytes are opened up for use. The way to avoid this is to create a complete eight-byte-long instruction all at once. If you could assign an apparently eight-byte-long instruction to a key, then press that key, you would open up a new register, fill all seven bytes, and grab the next byte too. Is there an instruction that seems to be eight bytes long and that can be assigned to a key?

Most instructions are one, two, or three bytes long, but a text string containing seven characters would be eight bytes long. It would contain the seven characters, and an extra header byte (247) at the front to warn the HP-41 that the next seven bytes are all one text string, not a series of functions. You therefore need to make a key assignment of something that seems to be a seven-byte text string. This can be done by assigning the byte 247, followed by one text character, to a key. Any byte from 14 to 255 will work as the text character, but some values above 143 can cause trouble. The byte 63 (which represents the question mark) is a good choice. This assignment will only be two bytes long, but when it is copied into program memory it will look like an eight-byte long instruction. You must now create this key assignment before you can try using it. Three methods of obtaining the assignment will be described before its use is explained; you only need to use one of these methods but it worth reading about all three.

An assignment of the sort described is obviously an unusual one and cannot be made by using the ASN or PASN functions. It can however be made with

the more general GASN program which was introduced in Section 11.10, or with similar programs (MK or 1K in the PPC ROM for example). To make the assignment to the LN key (key 15) using one of these programs do:

247 , ENTER , 63 , ENTER , 15 , XEQ "program"

The "program" should be "1K" if you have a PPC ROM or "GASP" if you use the program from Chapter 11. If you use GASP then remember that you must have executed the initialisation routine GASETUP beforehand. **If you use the PPC ROM's "MK" or "1K", clear your Time module alarms first, or they will be turned into junk.** Other key assignment programs of this type can be found in the books by Wickes and Jarett and also in the PPC ROM manual; they do not use the Extended Functions (except for the program in Jarett's Extended Functions book) or Time Functions but they do themselves contain Synthetic instructions so they cannot be created until you already have some Synthetic tools (or a Wand to read them). By the way, if you have a CCD ROM you can make this assignment directly with the ASN function (unless HP updates the part of the HP-41 operating system used by the CCD ROM).

A second way of getting the key assignment is to read it in from a magnetic card or an HP-IL mass storage device if someone has already made the assignment and recorded it for you. After you create the assignment for yourself you should copy it to a card (using the WSTS function) or to an HP-IL device (using the WRTK function) if you have either of these devices. You will then be able to read it back into your HP-41 if you ever lose or cancel the assignment.

A third way of obtaining this assignment is to get into the key assignment area and to edit this area as if it was a program. To obtain the desired result you need to replace an ordinary key assignment with a seven-character text string. This method requires some precautions, firstly you must find the right assignment to replace, and secondly you must delete enough bytes to leave room for the text string; otherwise the HP-41 operating system will try to move the contents of memory down by one register to make room for the text string. Unfortunately the key

assignments are stored below the .END. so the HP-41 will not stop moving registers until it comes to the very bottom of memory. The whole of memory, including some registers used by the operating system, will be moved down. This will result in MEMORY LOST.

The simplest but most dramatic way to avoid problems is to do a Master Clear yourself (MC for short; turn the HP-41 off, press the backarrow key and keep it down, press the ON key, release both keys, see MEMORY LOST). This makes the rest of the process much easier, but it destroys everything in your HP-41 except the time and date. When you start experimenting with Synthetic Programming for yourself you will probably accidentally get MEMORY LOST anyway, so you may as well start from this state. (In any case remember the advice in the previous section; make a copy of any important information on paper or on a device like a Card Reader.) If you do not want to clear your HP-41 memory then you will need to take some extra steps which will be described in the appropriate places and will be marked with a *. Now proceed as follows.

1. Do a Master Clear and go on to step 2.
   * If you do not want to MC then do all of the following five steps.
   a. Create an extra label at the front of the first program in memory. To do this, do CAT 1, press R/S at once, press RTN, go into PRGM mode, press LBL "T", and exit from PRGM mode.
   b. Delete all key assignments, except for global labels (those labels in CAT 1 that are assigned to keys). You can delete the assignments one at a time by assigning nothing to each key, or you can delete your key assignments all at once with the Extended Function CLKEYS. You can also clear all assignments except those of global labels by using a "clear assignments" magnetic card (described in Section 12.3).

c. If you have an Extended Functions module or an HP-41CX then make sure there is at least one file in Extended Memory. Make sure as well that the name of the first file in Extended Memory does not contain the character "≠" as one of its first five characters. (In fact the bytes 30,31 or any value above 143 must also be avoided, but it seems unlikely that anyone would normally use them in a file name.) If the first file does have such a name then you must copy the file contents from Extended Memory to main memory, delete the file, then copy its contents back to Extended Memory, and make sure the first file now has a name without these characters; otherwise repeat the process.

d. If you have any module that creates buffers (HP-IL Development module, Plotter module or CCD module) then take it out or delete any buffers it has created.

e. Pack memory by executing GTO.. or PACK.

2. Assign LBL to the LN key and then DEL to the LOG key. The LBL assignment will appear as LBL 65 when you come to edit the assignment area; later on you will delete it and replace it by the new assignment. The DEL will be useful when you need to do some deleting.

3. Go into USER mode and PRGM mode.

4. Execute CAT 1 and stop it at the first entry by pressing R/S at once. If you do not press R/S soon enough then execute CAT 1 again and try to press R/S more quickly. If you have done an MC then you will see the .END., and if you do not press R/S soon enough then the display will blink. You must press R/S before the display blinks.

5. Now press ALPHA, backarrow, ALPHA. You have deleted the first line in program memory and the HP-41 has tried to go to the previous line - which does not exist. Due to a bug in the operating system (see Bug 9 in Appendix C), the HP-41 has gone to the bottom of the key assignment area instead and you see a line numbered 4094. You can now edit the key assignment area almost as if it was a piece of a program. If you

do not see a line numbered 4094 then go back to step 1 and try again. This may still fail if Hewlett-Packard ever removes Bug 9; in that case you will have to use one of the other methods of obtaining the key assignment. Fortunately, Bug 9 is only likely to be removed in an updated version of the HP-41CX - which may also have a new name - and you should be able to run the GASN program on such an updated HP-41. If Bug 9 is removed you may still be able to use one of the alternatives described with Bug 9 in Appendix C - Bug 9R is the easiest.

6. Press GTO.005 and wait until you see a step numbered 05. This will take you to the point at which you need to delete an assignment and replace it with a text string.
   * If you have not done an MC and you have some timer alarms then you need the following three additional steps to find the key assignment before deleting it.
   a. When you see the step numbered 05, press PRGM to get out of program mode.
   b. Execute GTO 65 to reach the LBL which was assigned at step 2. If the HP-41 responds with NONEXISTENT then you will have to delete your alarms and go back to step 1 above.
   c. Go back to PRGM mode and press BST. After a few seconds you should see a step LBL 03. If you see anything else then SST twice and go back to step b above.

7. Press DEL 002 (push the LOG key and then the 1/X key). This deletes the first key assignment that you made in step 2.
   * If you have not done an MC and you have an HP-41CX, or an Extended Functions module in your HP-41, then press DEL 005 as well (push LOG and LN), for a total of 7 lines deleted.

8. You have now made enough room to fit a seven character text string in the last three bytes of the key assignment area. Create this string by pressing ALPHA, ?, A, A, A, A, A, A, ALPHA. If you do not have any Extended Memory then you will see the text string "?A‾‾‾‾‾" (?A

followed by five nulls which display as overline characters; you have pressed the letter A but there is no memory to store it so it is lost and a null appears in its place). If you have some Extended Memory then you will see "?AAAAAA", and the last five As will have replaced the first five letters of the name of the first file in your Extended Memory (you will have to use this new name from now on; check what it is by doing an EMDIR or CAT 4).

9. To tidy up and get out of the key assignment area press PRGM and GTO.. - then press the LN key and hold it down. You should see XROM 28,63 and then NULL. If you see anything else then go back to step 1 and try again. XROM 28,63 is a **pseudo-XROM number** - the HP-41 previews any two-byte key assignment using an XROM number even if it is not a real XROM function.

You have now assigned something that seems to be a seven character string to the LN key. Later on you will be able to use a different key (if you prefer to leave the top row free) by using the key assignment program in Section 14.10. Let us go back to the subroutine we have been using as an example and see what this new key assignment will actually do. The subroutine was:

```
 LBL 21 , RND ,   1E6    , + ,   RTN
\____/        \_____/
 207,21,   110,   17,27,22,  64,   133
```

Delete the STO 30 if you put it in earlier, or if you have not yet put this into your HP-41 then do so now, and remember to PACK so as to get rid of any spare (null) bytes. Go to the RND step, make sure you are in USER and PRGM mode, then press the LN key. You will see a very odd seven-byte long text string and the subroutine will have become:

```
 LBL 21 , RND ,      ?      , E6   , + ,   RTN
\____/        \____________/ \___/
 207,21,   110,    247,0,63,0,0,0,0,17,   27,22,  64,   133
```

When you pressed the LN key, the HP-41 moved everything down by one register and provided seven extra bytes for the new step. It then copied the assigned bytes 247 and 63 into the first and third bytes available. Four spare null bytes were left beyond the three bytes that were used up when the key was pressed, so the HP-41 did not open up (i.e. insert) any more registers. This looks fine until the result of pressing the key is examined in detail. It turns out that the byte 247 is the header of a seven-byte text string, so it takes over all of the next seven bytes, producing an instruction eight bytes long, and therefore grabbing the first byte of the next instruction.

To confirm this, just press SST and see the E6 standing on its own. The 1 at the front of 1E6 has been grabbed by the preceding text line. This is what you have been working towards all this time. The new key assignment is called a **Byte Grabber** for reasons which are obvious by now. The Byte Grabber is a program editing tool, rather like DEL as I mentioned above. It does not do anything very useful in a running program but it is your first and fundamental tool for editing programs to create Synthetic instructions. We shall continue in the next section by studying some examples of the use of the Byte Grabber.

First though I should acknowledge the work done by many members of user groups to develop Synthetic Programming as a whole, and the use of the Byte Grabber in particular. A great deal of very clever and very hard work went into the development of SP. Most of the pioneering work first reached a wide audience through the pages of the PPC Calculator Journal. Bill Wickes, encouraged by Richard Nelson and Henry Horn, published the first SP book; this contains detailed descriptions of the achievements of the first year of Synthetic Programming, many of the achievements are Bill's own. The Byte Grabber and its uses were only discovered after his book had been published. The first Byte Grabber (or **Prefix Masker** - an alternative name) was invented by Jack Baldridge, written up in the PPC Journal by Clifford Stern as soon as he heard of Baldridge's ideas, and also written up by Baldrige (with extra notes by John McGechie) in PPC Technical Notes and

then in the PPC Journal. It used a 3-character text assignment. The simplified 7-character byte grabbing technique was developed by Erwin Gosteli, and Phi Trinh invented the first method of creating the the BG assignment in an easy fashion. Roger Hill, who had independently duplicated Erwin Gosteli's extension of the Baldrige technique, thought of the catchy name "Byte Grabber". The first book that included a description of the Byte Grabber assignment was published by Keith Jarett ("HP-41 Synthetic Programming Made Easy" -- the assignment method described used ideas provided by Keith Kendall).

If you are really interested in this sort of history then you should read PPC TN5; pages 50-60 describe Bill Wickes' crucial role in the development of SP. The list of names given here is not complete; perhaps someone will one day write a history of the HP-41 and the people who used it! In the meantime let us go on to some uses of the Byte Grabber, while reverently muttering "I have stood on the shoulders of giants".

14.4 Using the Byte Grabber.

Your first use of the Byte Grabber has produced the subroutine:

```
 LBL 21 , RND ,      ┬ ?----▩      ,   E6  , + ,   RTN
\_____/        \_____________/ \___/
 207,21,   110,  247,0,63,0,0,0,0,17,     27,22,  64,  33
```

Notice that the byte 17 which was immediately before the two bytes 27,22 (E6) is still there, but it is now part of the text string created by the Byte Grabber. Since the 17 is now in a text string, it is displayed as a text character, not as the numeral 1. If you look back to Chapter 8 you will see that the byte value 17 prints as the Greek letter capital omega but it is shown in a text string as a boxed star ▩. Most byte values actually show up in the display as a boxed star (a character with all 14 segments turned on). This is the default for any byte which is not a symbol recognised by the display. Some bytes will show up as a recognisable character in a Byte Grabber string, but most will not. This

is why the question mark was chosen in the previous section as the second byte in the Byte Grabber key assignment: a seven-character text string which begins with a null and a question mark will be easily recognised as the result of a Byte Grabber operation, not a normal text string.

If you ever see such a string in a program you are editing you can be fairly sure that it is a Byte Grabber string. In the example above you wanted to get rid of the 1 in front of the E6, so you can now delete the text string by pressing the backarrow key, leaving the subroutine:

LBL 21	**, RND ,**		**E6**	**, + ,**	**RTN**
207,21	110 ,	0,0,0,0,0,0,0	27,22	64	133

The entire deleted string has been replaced by nulls, which do not show up as program steps. When you pack memory, these eight nulls will be removed, and the program will move up in memory. Previously it moved down seven bytes to make room for the new instruction, now it moves up eight bytes. You have therefore saved a byte in program memory. The new instruction E6 takes a byte less than 1E6, and it takes 33 milliseconds less execution time. Not much on its own, but in a long program you could save several whole registers and a few seconds too if you converted every occurrence of 1En into the corresponding **short-form exponent** En. You could also create the step 1E by just pressing EEX, then convert it into E. The symbol E on its own acts exactly like a 1 on its own but takes 12 milliseconds less; people who want to save every millisecond therefore use E instead of 1 in their programs. (It also helps to confuse anyone who is trying to steal a program from you.)

What should you do if you decided not to delete the 1 in the 1E6 after all? You would have to remove the byte grabber string, but leave the 1 behind. This is easy to do if you have not yet deleted the byte grabber string. (If you have already deleted it then delete the E6 too, put in 1E6, pack, go to the RND step and press byte grabber.) Backstep to the line before the byte grabber string and press the byte grabber key again. The result will be as below:

LBL 21,	RND,	┬ ─?─────▒,	STO 15,	,	1E6	,	+ ,	RTN
207,21	110	247,0,63,0,0,0,0,247	0	63	0,0,0,0	17,27,22	64	133

You have byte grabbed the Byte Grabber! The new byte grabber string has grabbed the byte 247 of the previous byte grabber string. This has released all of the bytes from the old string. The 17 has become part of 1E6 again and the 63 (? in a text string) is now a STO 15 when left to stand on its own in a program. The other bytes were all nulls so they do not show up as program steps (the null steps are shown here as blank spaces above zeroes). If you delete the STO 15 and the byte grabber string and pack then you will be back to the original state of the subroutine.

Byte grabbing a byte grabber can therefore be used to recover from an unwanted use of the byte grabber key in a program, not just to create new combinations of bytes. There is one exception; you may not be able to recover easily if you byte grab the .END.. The first thing to do if you accidentally grab the .END. is to BST once and byte grab again. If you SST twice and see the .END. then you are safe. If you see anything else then SST once more, next press the backarrow key four times to delete the previous four instructions, and finally press GTO.. to tidy up. This will sometimes work; you can check by doing a CAT 1. At other times it will not work and you will have to do a Master Clear, so don't byte grab the .END.!

You may have found the last few paragraphs confusing because they describe several different ideas. If you just want to create a short form exponent such as E6 then here is a review of how to do it with the byte grabber.

1) Put a program into the HP-41 memory step by step in the normal manner.
2) When you come to the exponent, put it is as usual, 1E6.
3) Backstep once (BST) to the line before.
4) Press the byte grabber twice (once to grab the null in front of the number, and once to grab the 1).

5) Press backarrow twice to delete the two byte grabber strings.
6) Press SST to see the short form exponent E6 (or Enn or E-nn; the method works for any exponent).
7) Remember to GTO.. or PACK in order to recover all the null bytes left behind. If you are writing a program in which you are using the byte grabber several times it is simpler to PACK just once, when you have written the whole program.

From now on I shall often use the abbreviation **BG** to mean "push the Byte Grabber key in USER and PRGM mode." After studying one use of the byte grabber in detail you will want to see some other examples. Here is an easy one. Have you ever wanted to use a few more registers than 100? You can use the byte grabber to create instructions such as STO 102 and RCL 102. To create STO 102 you have to put the postfix 102 behind a STO prefix. This means that you must create some other instruction such as STO 99, then use the byte grabber to separate the prefix and the postfix, and then put in a new postfix. Do the following, all in program mode:

1) Put the step STO 99 into a program, then XEQ "PACK" to make sure there are no null bytes to complicate matters.
2) Backstep once to get to the line before STO 99.
3) Press byte grabber to grab the STO prefix and leave the 99 postfix on its own.
4) SST once. You will see X≠0? From Table 8.1 you can see that the byte 99 stands for X≠0? if it is on its own and not a part of some other instruction.
5) Delete the X≠0? line and put in X<0? in its place. From Table 8.1 you can see that X<0? is byte 102.
6) BST twice and BG. This grabs the first byte grabber (as we saw in the case of 1E6 above) freeing everything after it.

7) SST once and see STO 15. This is the question mark released from the first byte grabber. SST again and you will see STO A. Why STO A instead of STO 102? It just happens to be the case that the postfix A (such as in LBL A) is represented by the number 102. Thus STO A means STO 102. The STO prefix has been released and the X<0? after it has become a 102 postfix (which is shown as the letter A).
8) If you want to put a synthetic instruction at the beginning of a program, it is easiest to GTO.000 then put in a dummy step (say LBL 00) and then create the synthetic step after it. After you create this synthetic instruction, you can BST and delete the dummy step. If you do not have room to put in a dummy step, go to step 000 to make the synthetic step and go to 000 again to byte grab it. A **dummy** step is one that has no purpose of its own but is used to fill a space for a time.

To check that STO A is really STO 102, delete the unnecessary instructions which were left over by the byte grabbers and write the program:

PI , STO A , CLX , 102 , RCL IND X , STOP

Do SIZE 103, then run the program (you will not be able to do this if you have an HP-41C and no memory modules). You will see that RCL IND X which recalls the contents of register 102 has brought back PI, and the first two steps put PI into register A, so A and 102 are the same register.

You can create other instructions such as RCL 102 and X<>102 in exactly the same way. Follow steps 1 to 7 above, but begin with the instructions RCL 99 or X<>99. The results, RCL 102 or X<>102, will show up as RCL A or X<>A. This method will let you write instructions to use any register up to 111. Why only 111? Back in Chapter 8, Table 8.2 showed the postfix values of each byte from 0 to 255. Bytes 0 to 99 behave in the normal way, bytes 102 to 111 represent the letters A to J (used in local labels). Bytes 112 to 116 represent the stack registers T, Z, Y, X, L, so they cannot be used to address the registers numbered 112 to 116. Bytes 117 to 127 will be explained in the next section, and bytes greater than 127 are used for indirect addressing. Thus 111 is the highest numbered register

that can be addressed directly even if you use Synthetics. A simpler method of creating instructions such as STO A will be described in the next section.

It has just been shown that the postfix 102 shows up as a letter A; the postfixes 103 to 111 show up as B to J, but what about postfixes 100 and 101? If you use these with STO (use X>0? or LN1+X instead of X<0? in step 5 above) then you will see STO 00 or STO 01. This is because STO only expects two-digit suffixes and simply drops the 1 at the front. The letters A to J are only one character long so nothing gets dropped. It is better to avoid STO 100 and STO 101 if possible because you cannot tell them from STO 00 and STO 01 just by looking at a program. Even if you avoid these two, Synthetic Programming has given you direct access to 10% more registers.

STO is not the only function that drops the 1 at the front of a 3-digit suffix. RCL, X<>, ISG, DSE will all do the same. So will the flag functions such as SF and CF, but there are no flags greater than 55, and the only purpose of creating FS?100 or CF J seems to be to confuse people. Some instructions expect only a one-digit suffix though, so they drop two digits. For example TONE 10, TONE 20, and so on up to TONE 100 can all be created synthetically, but they all appear as TONE 0. It is easy to check if they are the real TONE 0, since they all make different sounds when executed by means of SST. As an example of a synthetic tone, try to create TONE 87. The method is very similar to that described for STO 102. Create a TONE 7 step, pack, BG the TONE prefix, delete the 7 postfix (it shows up as LBL 06), replace it with an 87 postfix (press shift, 10↑X), BST twice, then grab the first byte grabber and SST twice to see TONE 7. Press PRGM to get out of program mode, then press SST to hear the synthetic TONE. It has exactly the same pitch as TONE 7, but is just half the duration (a semi-quaver instead of a quaver). You may like to use this instead of TONE 7 to provide a shorter attention-getting note. You could also use it together with TONE 7 to provide Morse Code dots and dashes, or a short and long pair of notes for audio control of equipment (such as automatic telephone dialling). Other synthetic tones can be very short (TONE 89 is

one fifth the duration of TONE 9 but at the same pitch) or long (TONE 65 has the same frequency but is eight times as long as TONE 1) or at a lower pitch. TONE 26 is more than an octave below TONE 2 and seventeen times longer! To create TONE 26, create TONE 6 first, BST and BG, SST and delete the suffix (LBL 07), then put in a decimal point (byte 26). After this you have to pack to get rid of the null that is always put in front of any number (otherwise this null byte will become the suffix, instead of the byte 26). Now you can BST twice, BG, SST twice and see TONE 6, which is actually a TONE 26. If you execute TONE 26 it lasts almost 5 seconds (unless flag 26 is cleared).

Synthetic TONEs can be used in a variety of applications, for example as timing signals in a photographic darkroom, and of course there is their obvious use in games. Tables of Tone frequency and duration can be found in Wickes' book, the PPC ROM manual, Synthetic Programming Made Easy, and in the "Synthetic Quick Reference Guide". A few tones vary significantly from one HP-41 version to another. This is covered everywhere except in Wickes' book.

One more simple example before we go on to the next section; this one is a useful tip for people who use the statistics registers. It is easy to store a value accidentally in one of the statistics registers and thus destroy some data you summed earlier. Typically you may execute ΣREG 14, CLΣ and do some summations, then do STO 19 later on, forgetting that register 19 is one of the statistics registers. You can make this far less likely if you do ΣREG 104, since STO 109 cannot be done accidentally; you have to use a synthetic instruction or indirect addressing. (ΣREG 104 is displayed as ΣREG C of course.) If you are working from the keyboard only, then naturally you can do 104, ΣREG IND X (or with a ZENROM, do ΣREG, EEX, LOG or 04), so the synthetic instruction ΣREG C is most useful when you are writing a long program in several parts. By the time you write the last part, you may forget where the statistics registers are, but if you start the program with ΣREG C then you will not accidentally write an instruction that destroys the results of some summations.

You will find this instruction helpful if you use the statistics registers, if you change their position from time to time, and you want to avoid the risk of losing some data (either by overwriting the summation registers or by using a statistics function to overwrite a data register). You create the instruction by writing a program with an ordinary Σ REG, then byte grabbing the Σ REG prefix, deleting the postfix and putting in an appropriate postfix (INT for 104), then grabbing the first byte grabber to release the prefix and so producing a synthetic ΣREG. When you use this Σ REG, make sure you set a sufficient SIZE; you need SIZE 110 if you are going to use Σ REG C.

14.5 The Byte Table and the Status Registers.

When the byte grabber is used to split up a function containing several bytes, the bytes that are left free can change from postfixes to prefixes. In the previous section, you were referred to Tables 8.1 and 8.2 to see which byte did what. You may also need to use the information from Table 8.3 which shows how bytes are related to characters in text strings. It would be much easier to have the information from all three tables put together in one place (and in this chapter so that you do not have to keep turning back many pages).

The Byte Table, Table 14.1, contains all the required information. Like Tables 8.1 and 8.2 it has 16 rows and 16 columns, marked in hexadecimal. Each box contains the prefix (or one-byte) function corresponding to that byte, below that the postfix value of that byte and its display value, and below that the printer character which that byte produces on an HP82143A printer. Next to this, each box also contains the decimal equivalent of the hexadecimal number describing the box.

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8	DEG IND 00 128 ◆	RAD IND 01 129 ×	GRAD IND 02 130 ⁻	ENTER↑ IND 03 131 ←	STOP IND 04 132 α	RTN IND 05 133 ß	BEEP IND 06 134 Γ	CLA IND 07 135 ↓	ASHF IND 08 136 Δ	PSE IND 09 137 σ	CLRG IND 10 138 ◆	AOFF IND 11 139 ↖	AON IND 12 140 μ	OFF IND 13 141 ∠	PROMPT IND 14 142 π	ADV IND 15 143 ⇕	8
9	RCL IND 16 144 θ	STO IND 17 145 Ω	ST+ IND 18 146 δ	ST− IND 19 147 ∞	ST ✳ IND 20 148 ∂	ST/ IND 21 149 ∀	ISG IND 22 150 ∃	DSE IND 23 151 ∅	VIEW IND 24 152 ∈	Σ REG IND 25 153 ∪	ASTO IND 26 154 ∪	ARCL IND 27 155 ∉	FIX IND 28 156 ∝	SCI IND 29 157 ≠	ENG IND 30 158 £	TONE IND 31 159 ⁑	9
A	XR 0-3 IND 32 160	XR 4-7 IND 33 161 !	XR8-11 IND 34 162 "	X12-15 IND 35 163 #	X16-19 IND 36 164 $	X20-23 IND 37 165 %	X24-27 IND 38 166 &	X28-31 IND 39 167 '	SF IND 40 168 (	CF IND 41 169)	FS?C IND 42 170 *	FC?C IND 43 171 +	FS? IND 44 172 ,	FC? IND 45 173 −	GTO XEQ IND IND 46 174 .	SPARE IND 47 175 /	A
B	SPARE IND 48 176 0	GTO 00 IND 49 177 1	GTO 01 IND 50 178 2	GTO 02 IND 51 179 3	GTO 03 IND 52 180 4	GTO 04 IND 53 181 5	GTO 05 IND 54 182 6	GTO 06 IND 55 183 7	GTO 07 IND 56 184 8	GTO 08 IND 57 185 9	GTO 09 IND 58 186 :	GTO 10 IND 59 187 ;	GTO 11 IND 60 188 <	GTO 12 IND 61 189 =	GTO 13 IND 62 190 >	GTO 14 IND 63 191 ?	B
C	GLOBAL IND 64 192 @	GLOBAL IND 65 193 A	GLOBAL IND 66 194 B	GLOBAL IND 67 195 C	GLOBAL IND 68 196 D	GLOBAL IND 69 197 E	GLOBAL IND 70 198 F	GLOBAL IND 71 199 G	GLOBAL IND 72 200 H	GLOBAL IND 73 201 I	GLOBAL IND 74 202 J	GLOBAL IND 75 203 K	GLOBAL IND 76 204 L	GLOBAL IND 77 205 M	X<>-- IND 78 206 N	LBL -- IND 79 207 O	C
D	GTO -- IND 80 208 P	GTO -- IND 81 209 Q	GTO -- IND 82 210 R	GTO -- IND 83 211 S	GTO -- IND 84 212 T	GTO -- IND 85 213 U	GTO -- IND 86 214 V	GTO -- IND 87 215 W	GTO -- IND 88 216 X	GTO -- IND 89 217 Y	GTO -- IND 90 218 Z	GTO -- IND 91 219 [	GTO -- IND 92 220 \	GTO -- IND 93 221]	GTO -- IND 94 222 ↑	GTO -- IND 95 223 _	D
E	XEQ -- IND 96 224 ▪	XEQ -- IND 97 225 a	XEQ -- IND 98 226 b	XEQ -- IND 99 227 c	XEQ -- IND100 228 d	XEQ -- IND101 229 e	XEQ -- IND102 230 f	XEQ -- IND103 231 g	XEQ -- IND104 232 h	XEQ -- IND105 233 i	XEQ -- IND106 234 j	XEQ -- IND107 235 k	XEQ -- IND108 236 l	XEQ -- IND109 237 m	XEQ -- IND110 238 n	XEQ -- IND111 239 o	E
F	TEXT 0 IND T 240 p	TEXT 1 IND Z 241 q	TEXT 2 IND Y 242 r	TEXT 3 IND X 243 s	TEXT 4 IND L 244 t	TEXT 5 IND M [245 u	TEXT 6 IND N \ 246 v	TEXT 7 IND O] 247 w	TEXT 8 IND P ↑ 248 x	TEXT 9 IND Q _ 249 y	TEXT10 IND ⊢ 250 z	TEXT11 IND a 251 π	TEXT12 IND b 252 |	TEXT13 IND c 253 →	TEXT14 IND d 254 Σ	TEXT15 IND e 255 ⊢	F
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

For price information and a list of dealers in your area, send a self-addressed stamped envelope to: SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, CA 90266, USA

Table 14.1 The Byte Table (first half -- rows 0 to 7)

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	CAT	@c (GTO..)	DEL	COPY	CLP	R/S	SIZE	BST	SST	ON	PACK	←(PRGM)	USR/P/A	2 __	SHIFT	ASN	
0	NULL 00 − 0 ◆	LBL 00 01 ⌅ 1 ×	LBL 01 02 ▩ 2 ⌆	LBL 02 03 ▩ 3 ←	LBL 03 04 ⌅ 4 α	LBL 04 05 ⌅ 5 ß	LBL 05 06 ⌅ 6 Γ	LBL 06 07 ▩ 7 ↓	LBL 07 08 ▩ 8 ∡	LBL 08 09 ▩ 9 σ	LBL 09 10 ▩ 10 ◆	LBL 10 11 ▩ 11 ⌅	LBL 11 12 ⌅ 12 ⌅	LBL 12 13 ⌅ 13 ⌅	LBL 13 14 ▩ 14 ⌅	LBL 14 15 ▩ 15 ⌅	0
1	0 16 ▩ 16 ⊟	1 17 ▩ 17 Ω	2 18 ▩ 18 δ	3 19 ▩ 19 ⌅	4 20 ▩ 20 ⌅	5 21 ▩ 21 ⌅	6 22 ▩ 22 ⌅	7 23 ▩ 23 Ö	8 24 ▩ 24 ⌅	9 25 ▩ 25 Ü	. 26 ▩ 26 Ü	EEX 27 ▩ 27 Æ	NEG 28 ▩ 28 æ	GTO ᵀ 29 ⌅ 29 ≠	XEQ ᵀ 30 ▩ 30 £	W ᵀ 31 ▩ 31 ▩	1
2	RCL 00 32 32	RCL 01 33 ; 33 !	RCL 02 34 " 34 "	RCL 03 35 ⌅ 35 #	RCL 04 36 ⌅ 36 $	RCL 05 37 % 37 %	RCL 06 38 ⌅ 38 &	RCL 07 39 ' 39 '	RCL 08 40 (40 (	RCL 09 41) 41)	RCL 10 42 * 42 *	RCL 11 43 + 43 +	RCL 12 44 , 44 ,	RCL 13 45 - 45 −	RCL 14 46 . 46 -	RCL 15 47 / 47 /	2
3	STO 00 48 ⌅ 48 0	STO 01 49 ⌅ 49 1	STO 02 50 ⌅ 50 2	STO 03 51 ⌅ 51 3	STO 04 52 ⌅ 52 4	STO 05 53 ⌅ 53 5	STO 06 54 ⌅ 54 6	STO 07 55 ⌅ 55 7	STO 08 56 ⌅ 56 8	STO 09 57 ⌅ 57 9	STO 10 58 : 58 :	STO 11 59 ; 59 ;	STO 12 60 ⌅ 60 <	STO 13 61 ⌅ 61 =	STO 14 62 ⌅ 62 >	STO 15 63 ⌅ 63 ?	3
4	+ 64 ⌅ 64 @	− 65 ⌅ 65 A	✳ 66 ⌅ 66 B	/ 67 ⌅ 67 C	X<Y? 68 ⌅ 68 D	X>Y? 69 ⌅ 69 E	X≤Y? 70 ⌅ 70 F	Σ+ 71 ⌅ 71 G	Σ− 72 ⌅ 72 H	HMS+ 73 ⌅ 73 I	HMS− 74 ⌅ 74 J	MOD 75 ⌅ 75 K	% 76 ⌅ 76 L	%CH 77 ⌅ 77 M	P→R 78 ⌅ 78 N	R→P 79 ⌅ 79 O	4
5	LN 80 ⌅ 80 P	X↑2 81 ⌅ 81 Q	SQRT 82 ⌅ 82 R	Y↑X 83 ⌅ 83 S	CHS 84 ⌅ 84 T	E↑X 85 ⌅ 85 U	LOG 86 ⌅ 86 V	10↑X 87 ⌅ 87 W	E↑X-1 88 ⌅ 88 X	SIN 89 ⌅ 89 Y	COS 90 ⌅ 90 Z	TAN 91 ⌅ 91 [	ASIN 92 ⌅ 92 \	ACOS 93 ⌅ 93]	ATAN 94 ⌅ 94 ↑	→DEC 95 ⌅ 95 _	5
6	1/X 96 ᵀ 96 ᵀ	ABS 97 ⌅ 97 a	FACT 98 ⌅ 98 b	X≠0? 99 ⌅ 99 c	X>0? 100 ⌅ 100 d	LN1+X 101 ⌅ 101 e	X<0? A ▩ 102 f	X=0? B ▩ 103 g	INT C ▩ 104 h	FRC D ▩ 105 i	D→R E ▩ 106 j	R→D F ▩ 107 k	→HMS G ▩ 108 l	→HR H ▩ 109 m	RND I ▩ 110 n	→OCT J ▩ 111 o	6
7	CLΣ T ▩ 112 p	X<>Y Z ▩ 113 q	PI Y ▩ 114 r	CLST X ▩ 115 s	R↑ L ▩ 116 t	RDN M [▩ 117 u	LASTX N \ ▩ 118 v	CLX O] ▩ 119 w	X=Y? P↑ ▩ 120 x	X≠Y? Q_ ▩ 121 y	SIGN ⊢ ᵀ ▩ 122 z	X≤0? α ▩ 123 π	MEAN b ▩ 124 ⌅	SDEV c ▩ 125 →	AVIEW d Σ 126 Σ	CLD e ⊢ 127 ⊢	7
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

00 01 02 03 | 04 05 06 07 | 08 09 10 11 | 12 13 14 15 | 16 17 18 19 | 20 21 22 23 | 24 25 26 27 | 28 29 30 31 | 32 33 34 35 | 36 37 38 39 | 40 41 42 43 | 44 45 46 47 | 48 49 50 51 | 52 53 54 55 ← bit numbers in a 7-byte register

Table 14.1 The Byte Table (second half -- rows 8 to F)

The Table is presented in two halves because it is an enlarged copy of a pocket-sized plastic card which has half the table on each side. This plastic card is made by Synthetix (See Appendix B for their present address). It fits into the HP-41 case so that you can have it available whenever you are programming the HP-41. Unless you can memorise the entire contents of the Byte Table you would be well advised to obtain one of these plastic cards and carry it with your HP-41. Some useful additional information is provided: the bottom row of each half contains the binary equivalents of the hexadecimal digits 0 to F. This can be used to convert hexadecimal numbers into "bit patterns" (rows of bits representing a number in binary notation), for example you can use this to translate the hexadecimal number E1 into the binary number 1110 0001. Below the bottom row of the first half there is a list of bit numbers in a register, this can be used to locate which nybble of a register contains a given bit. This row of numbers can also be used to help identify pseudo-XROM numbers (see point 9 in Section 14.3); the method is described on page 14 of the Synthetic Quick Reference Guide (on page 12 if you have the 1982 edition).

The second half of the table does not show how the bytes look in the display as part of a text string because all these characters show up as boxed stars. However normal print functions (such as PRA) print them on the Hewlett-Packard thermal printers in the same way as the corresponding bytes in the first half of the table, so this is shown. Some of the printer characters in the second half of the table are shaded, these bytes are used as "control characters" by the 82143A thermal printer. If a byte from the second half of the Byte Table is included in a text string and the program containing the text string is printed (using the functions PRP or LIST) then any of these characters will disrupt the printer's operation but other bytes from the second half will not be printed. This was covered in Section 12.2, and it was mentioned that the HP-IL printer, the HP82162A, reacts to more control characters. The card produced by Synthetix also has a list of the HP-41 flags and what they do, and a description of the structure of multi-byte instructions.

Important Note: Now that we have introduced the full byte table, the rest of this book will express byte values in **hexadecimal**. The 2-digit hexadecimal form lets you find the byte you want very easily. The first digit is the row number, and the second digit is the colum number.

The first five bytes in the eighth row (row 7 in the Table) provide the stack register postfixes T, Z, Y, X and L. These are used by functions such as VIEW Z (hexadecimal bytes 98,71). All the other bytes of row 7 can also be used as postfixes, though Hewlett-Packard did not plan for this to be done. You may remember from Chapter 8 that the HP-41 registers come in blocks of sixteen, and that the five stack registers are part of the block of sixteen status registers. The other eleven status registers are used by the HP-41 operating system. It is these registers that are accessed if you use the bytes of row 7 as postfixes.

Before going on to study these registers, let us create a program step that uses a status register postfix. If you look at the byte table you will see that the two bytes 91, 74 make up the function STO L. We can use the byte grabber to make a similar instruction 91, 75. This could be done using the method shown in the previous section, grabbing the byte 91, deleting the byte 74 (R↑), replacing it with the byte 75 (RDN), and grabbing the byte grabber. A simpler method is available though:

1) Create the two program steps RCL IND 17, RDN. This puts the three bytes 90, 91, 75 into a program.
2) PACK; this is only necessary to remove nulls, so you may not need to do it, but it is always safer. Then BST twice to get to the step before RCL IND 17.
3) Press the Byte Grabber key (in USER mode of course). This grabs the byte 90 and leaves the two bytes 91, 75 on their own.
4) Press the backarrow key to delete the byte grabber string, then SST and see the program step STO M.

In the previous section we used well-known prefixes and postfixes but we put them together in unusual combinations. STO M however contains a

completely new postfix. M just happens to be the next letter after L, so the next postfix after L turns out to be M. Since the HP-41 designers did not plan for this postfix to be used, they did not design the print functions to interpret this postfix properly, and program listings will produce the line STO [instead of STO M. The byte table therefore shows both M and [for the postfix use of byte 75. You can check this if you have a printer; print the program containing the STO M step we have just made. The next four postfixes after M show up as N, O, P, Q in the display and they print as \,], ↑, _. Byte 7A shows up as the append sign ⊢ in the display, and prints as the text symbol ⊤. Some people refer to it as R (particularly if they do not have a ⊢ symbol available), and the ZENROM (see Section 12.8) uses the R key to provide this postfix. The remaining postfixes from this row are the familiar lower case a to e, normally used for local Alpha labels, but also able to provide access to the status registers.

Most program steps containing these row 7 synthetic postfixes can be created using the method just shown for creating STO M. For example to create VIEW Q you need to produce the bytes 98, 79. You can do this by using any two-byte function with IND 24 as the second byte and following this with X≠Y?, then byte grabbing the first byte of the two-byte function. One way would be to produce the two steps TONE IND 24, X≠Y?, then BST twice, byte grab, delete the byte grabber string, SST, and see VIEW Q. These synthetic postfixes can be used for functions such as TONE or FIX or LBL (you can create additional local Alpha LBLs such as LBL X or LBL M, and their corresponding GTO functions). Their most interesting use though is with the register access functions STO, RCL, and X<> . The next few sections will provide examples to show what the status register access functions will let you do.

Indirect functions such as RCL IND P can also be created. The simplest way to make these postfixes is to use text strings of the appropriate length in combination with the byte grabber. An example will help again; to produce RCL IND Q (90, F9) do the following:

1) Create the program steps RCL IND 16, "ABCDEFGHI". This provides the bytes 90, 90, F9, 41, 42, 43, 44, 45, 46, 47, 48, 49. We only need the bytes 90,F9 so any text string of nine bytes would do.
2) BST twice, BG, and delete the byte grabber text string.
3) SST and see the instruction RCL IND Q. If you see something else then you probably left some nulls in the program at step 1), and should have PACKed. Go back to step 1) and try again.
4) Delete all the unnecessary instructions after the RCL IND Q. If you got it right the first time then you will have nine bytes left from the text string so press SST once and then execute DEL 009.

The method of creating synthetic functions described in this section relies on the use of the Byte Table. You first use the table to select a combination of prefixes and postfixes, then grab one prefix so that its postfix becomes a prefix and the next byte becomes a postfix. The next sections will assume that you can use this method to create any synthetic function to access the status registers. If you want to practice creating a few more of these functions then try the following examples. Each function is followed by one method of creating it. BG stands for "press the Byte Grabber key in USER and PRGM mode". Remember to tidy up and check each function after you have created it by pressing backarrow and SST.

TONE T	(STO IND 31, CL Σ, BST, BST, BG)
LBL X	(RCL IND 79, CLST, BST, BST, BG)
GTO L	(STO IND 80, R↑, R↑, BST, BST, BST, BG) As GTO is a 3-byte function, it needs <u>two</u> postfix bytes. The first one can be any byte, the second one must be R↑ for the postfix L. A synthetic GTO created in this way has a compiled distance in it but this will be removed when you finish the editing, unless your HP-41 has bug 8 (see Appendix C).
X<>Q	(STO IND 78, X≠Y?, BST, BST, BG)
RCL d	(STO IND 16, AVIEW, BST, BST, BG)

ST* IND M (RCL IND 20, "ABCDE", BST, BST, BG)
ISG IND O (RCL IND 22, "ABCDEFG", BST, BST, BG) Be careful not to confuse register O ("Oh") with register 00 (zero, zero)
FIX IND e (TONE IND 28, "ABCDEFGHIJKLMNO", BST, BST, BG)

In the last three examples remember to delete bytes left by the text string. If you are going to do this a lot then you should assign DEL to a key. I have suggested the use of STO or RCL to produce the first prefix in all except the last example. This is just because STO and RCL need only one keystroke; you can use other prefixes as is shown by the last example.

Figure 14.1 shows the sixteen status registers, together with the name of each and its absolute address. The lowest five are the familiar stack registers. The other eleven will be described in the next six sections.

14.6 Status Registers M, N, O, P.

The last three bytes of status register P together with the whole of registers O, N and M provide the 24 bytes of the ALPHA register as shown in Figure 14.1. The previous section showed you how to create the function STO M. Now try out that function; first CLA, then put 1.4 into register X, then go to the STO M step and execute it by pressing SST while you are in run mode (not in PRGM mode). Go into Alpha mode and you will see that the Alpha register (which you had just cleared) now contains the text string π□-----. From Chapter 8 you can see that the number 1.4 is stored in register X as the hexadecimal bytes 01, 40, 00, 00, 00, 00, 00. The byte table will show you that the same bytes can be interpreted as the text string just shown. When you put these bytes into register M and view Alpha they are naturally interpreted as a text string. This is one way of putting "non-keyable" characters into the ALPHA register. Non-keyable characters are the one that are not available on the Alpha keyboard. The last five characters in this text string are nulls. If you try to delete a null character with the backarrow key in append mode, then the HP-41 assumes that the whole Alpha register contains nulls and it executes the CLA function.

Register name	C O N T E N T S	Absolute dec reg no.	Absolute hex reg no.
e	<----shifted key assignments ------> Scratch Pgm line no	15	0F
d	<---------- F l a g s 00 to 55 ---------->	14	0E
c	Stat Pointer |Scratch| 1 6 9 |R00 Pointer| .END. ptr.	13	0D
b	3a 2nd 1st current address	12	0C
	---------- **Subroutine Return Stack** ----------		
a	6th 5th 4th 3b	11	0B
⊢,⊤	<----unshifted key assignments-----><-----Scratch----->	10	0A
Q,_	S C R A T C H R E G I S T E R	09	09
P,↑	Extended Alpha/Scratch 24 23 22	08	08
O,]	21 20 19 18 17 16 15	07	07
	ALPHA Register Characters 1 to 24		
N,\	14 13 12 11 10 9 8	06	06
M,[	7 6 5 4 3 2 1	05	05
L	L A S T X	04	04
X	S T A C K	03	03
Y		02	02
Z	M A N T I S S A EXPONENT	01	01
T	SIGN 1 2 3 4 5 6 7 8 9 10 SIGN 1 2	00	00
	13 12 11 10 9 8 7 6 5 4 3 2 1 0		Nybbles
	Byte 6 Byte 5 Byte 4 Byte 3 Byte 2 Byte 1 Byte 0		Bytes

Figure 14.1 The HP-41 Status Registers

This example has shown you that register M is part of the ALPHA register and that it can be used to store numbers. It can also be used for arithmetic. Create the two program steps ST+ M and RCL M (use the methods described in the previous section), then execute them by using SST. ST+ M should add 1.4 to the 1.4 that you have just put into M, and RCL M should recall the result into X. Indeed you will see 2.8 in X. The three registers M, N, O can be used for storing numbers, doing arithmetic on them, and recalling them just like ordinary registers. Of course you cannot use the Alpha register for other purposes while you are doing arithmetic in it, but it provides you with three extra registers. This can be very helpful, particularly if you have an HP-41C with only 64 registers. If you need only one register then you can use register O, and still keep a maximum of 14 characters in the Alpha register without disturbing them since they all fit into registers M and N.

Registers M, N and O can be used if you need to save a number or do some arithmetic without disturbing any of the numbered data registers. Say you need a function to calculate 1/X↑3 without disturbing registers Y, Z or T and saving the old X in L. This should work just like the ordinary 1/X function and should not disturb any of the numbered data registers, which you may be using for something else. The following short routine will do the job.

```
01 *LBL "RECUBE"    (reciprocal cube routine)
02  1/X             (calculate 1/X and put X into L)
03  STO O           (store 1/X in register O)
04  ST* O           (this step and the next one...)
05  ST* O           (...put 1/X↑3 in register O)
06  CLX             (put a zero in X)
07  X<> O           (get the result in X and clear register O)
08  END             (registers M and N are unchanged and O is clear)
```

The last three bytes of register P are also used to hold Alpha characters, but the first four bytes are often used for other purposes. (An area like this is called a **scratch** area, since it is used like a scratchpad.) In particular the CAT functions (including EMDIR, ALMCAT and the CX catalogues

4, 5 and 6) use these bytes when a CAT is executed or searched (by a GTO or XEQ operation). They are also used whenever a number is entered into register X or when a number is displayed (showing X or using VIEW or PSE) or register X is ready for a number to be entered into it. Further details are given in the book Synthetic Programming Made Easy. If you stay in ALPHA mode or display a text string in register X and do not enter any numbers from a program then you can use register P just like registers M, N and O. Another very useful feature is that characters pushed out of the lowest 24 positions in the Alpha register are pushed into the top four bytes of register P. Therefore you can store up to 28 characters in the Alpha register so long as you do not alter register P by displaying or entering a number or executing a catalogue. The Card Reader function WSTS also changes register P. The top four characters will not be shown when you use AVIEW, but they will be there and you can recall them by using RCL P. These top four characters are moved and altered when you use backarrow to make corrections or alterations to the ALPHA register. Remember that you can edit the ALPHA register at any time by entering Alpha mode and pressing APPEND (Shift K).

Since register P is altered whenever a number is put into a program, you cannot safely use a set of steps such as

STO P, 5.795432E-11, RCL P

because the 5.7... entry alters the first four bytes of P. If you recall the number from a register then P will not be affected, so you could use the steps

5.795432E-11, STO 10 ..... STO P, RCL 10, RCL P

Sometimes you might not have a free register to hold the number, or you may not know which registers are free. Where can you put the number then? Why not use the ALPHA register itself? You can always use RCL M to get a number from the ALPHA register, and you can even use a text string to put the number into Alpha. Continuing with the example of 5.795432E-11, try translating it into a string of bytes. From Chapter 8 and the Byte Table

you can see that this number is stored as:

	+5 79 54 32 00 0- 89
or as the hexadecimal bytes:	05 79 54 32 00 09 89

How can you create a text string containing these bytes? First create an ordinary program text string with seven characters.

" A B T 2 E F G "

This is made up of the bytes F7, 41, 42, 54, 32, 45, 46, 47. Now go back to the program step before this string, PACK and press the Byte Grabber key. The F7 byte will be grabbed and the string of bytes 41, 42, 54, 32, 45, 46, 47 will appear as a string of instructions: -, *, CHS, STO 02, X>Y?, X<=Y?, Σ+. If you compare this with the string of bytes above then you will see that you need to replace a few bytes with different instructions.

1) SST past the BG, delete the - (byte 41) and put in LBL 04 (byte 05).
2) SST again, delete * (byte 42) and put in X≠Y? (byte 79).
3) SST again. Byte 54 is already in the right place, provided by the T.
4) SST again. Byte 32 is already there.
5) SST again. We want to replace X>Y? with a null byte, but not yet.
6) SST again, delete X<=Y? and put in LBL 08 (byte 09).
7) SST again, delete Σ+ and put in PSE (byte 89).
8) Now BST twice and delete the X>Y? This becomes a null byte 00. We could not do this earlier, because deleting bytes X>Y? and X<=Y? would have meant that LBL 08 would have gone into the first empty byte, not the second one. You must not PACK now, or this null byte will be lost.
9) BST five more times, and BG the Byte Grabber string. This releases the F7 byte so our new text string is ready.
10) Delete the BG string, SST and delete the next line, then SST again to see the new text string instruction. (It is safe to PACK now since the null byte created at step 8 is now safely hidden inside the text string.)

If you SST this instruction and then execute RCL M (put a RCL M after the text string in your program) then you will find the number 5.795432E-11 in register X.

Now you can see that the program steps:

"⊼▩T2¯▩▩"
STO P
RCL M
RCL P

will let you save a number in register P, put a new number into register X, and later get the other number back from P without changing P or using any numbered data registers.

You might also notice that the text string and RCL M use a total of 10 bytes whereas the program instruction 5.795432E-11 takes twelve bytes. The method just described will let you save several bytes on any long number that you want to enter from a program, and it is much quicker. In this case 5.795432E-11 takes .37 seconds, whereas the text string followed by RCL M takes .11 seconds, so you save a quarter of a second. This can be quite important in a long program which enters many numbers, but you must have the patience to build up the text strings required. In this particular case you could have saved a byte by replacing 5.795432E-11 with 5795432E-17 but the synthetic method saves more bytes and much more time. By using longer text strings you can put numbers into registers N and O as well, then recall them to X. Instead of building up text strings you can use a Q-loader - see Section 15.6.

The method just described for creating a text string with non-keyable characters can be used to create other text strings too. For example the text string " μAMPS" can be created by first making the text string "MAMPS", then byte grabbing the text byte, replacing the first character M (byte 4D, instruction %CH) with the character μ (byte 0C, instruction LBL 11), and finally byte grabbing the previous byte grabber. The same trick

can be used to create text strings with lower-case characters. The lower case characters (except a to e) will show up as boxed stars, but will print correctly on the Hewlett-Packard printers, without the need for setting and clearing flag 13.

Later on in this Chapter we shall use combinations of instructions like STO M and RCL N with ordinary ALPHA operations like ASTO and APPEND. These combinations will give us a new class of ALPHA operations, useful particularly in manipulating non-normalised numbers. First let us study some other status registers.

14.7 Register Q.

This is the next register up from M, N, O and P. You can see from Figure 14.1 that Q is the only complete status register available for temporary storage, or **scratch** purposes. This register is therefore used by many HP-41 functions and it is rarely safe to store values in it. If a printer is attached, then Q is altered by every step of a RAM program; so normally you cannot use Q in any program that might be run with a printer. If you clear flag 55 (see next section) then the HP-41 will not know that a printer is attached and you will sometimes be able to use Q.

Other functions that use Q are as follows:

Number entry: whenever a number is entered from the keyboard or from a program it is built up in register Q and copied to register X only when number entry is terminated. Register Q is not used when numbers are recalled to X, so the text line number entry method described in the previous section does not alter Q.

Alpha execution: whenever you spell out a function or program name, the name is stored in register Q. For example CLA, create the program steps RCL Q, STO M, then key in XEQ "SIGN". Now BST to the line RCL Q, go out of PRGM mode and press SST twice. The first step will recall the contents of register Q, the second will put them in register M. If you set ALPHA mode

you will see the text string NGIS. This is just "SIGN" spelt backwards; names are always copied backwards into register Q.

GTO and XEQ: in addition to alpha execution, any direct or indirect GTO or XEQ that spells out a program or function name will put that name into register Q. The name is always put in Q, even if it turns out to be NONEXISTENT. When you create a global alpha label by pressing LBL and spelling out a name, that name is also put into register Q. Other functions which demand an Alpha parameter also use Q. For example the printer function PRP uses register Q to store the name of the program to print. In all these cases the name is stored backwards, and the length of all names is limited to seven characters because register Q is seven bytes long (like all other data registers).

ALPHA: Text entry does not alter register Q, but viewing the contents of Alpha (with AVIEW, PROMPT or by stopping a program in ALPHA mode) does. So you can only keep Q unchanged while entering text if you do this in a running program.

Mathematical functions: Y↑X, R-P, P-R, SDEV alter Q, and so do SIN, COS, ASIN, and ACOS. TAN and ATAN do not.

Extended functions: SAVEP and a few others use register Q as well. The CCD module extended catalogue function **CAT'** uses register Q, and so do some of its other extended operations.

From the above you will see that register Q can rarely be used for storage and arithmetic, but this does not mean that it is of no use. In Chapter 16 I shall describe a program which uses register Q to provide a programmable version of the printer function PRP. Register Q can also be used for writing synthetic programs. There is a whole class of synthetic instructions called Q-loaders; these are used to "load" the contents of register Q into a program, creating unusual text lines, labels, GTO and XEQ instructions. Q-loaders can only be used if they are assigned to a key or scanned with a wand, so they will be described after a key assignment

program has been developed that will work on any HP-41.

14.8 Register d, the flag register.

I shall now go on to register d because it is often used in combination with registers M, N, O and P. Section 8.1 mentioned that a single register can be treated as 56 bits; each of these bits can be used as a flag - either set (and equal to 1) or cleared (and equal to 0). Register d contains all 56 HP-41 flags. These are described in Appendix D and their positions in register d are given at the bottom of the first half of the Byte Table. If you execute RCL d then you recall into register X a value that gives the setting of every flag. You can change some of the flags, and later restore their original status by using STO d to copy the original value from X back to register d. Say I want a program to recall the number in register 02 and display it without showing its fractional part or the decimal point, but then I want the program to go back to its normal way of displaying other numbers. If I knew that only one person would read this book and that this one person always used ENG 3 mode with digits grouped into threes, then I could use the four instructions

FIX 0, CF 29, RCL 02, PSE

to recall the number and display it without a fraction or a decimal point. After this I could reset the normal display mode by using SF 29, ENG 3. However I hope that more than one person will read the book (at least two have promised to do so!), so I cannot just set ENG 3 mode and hope everyone is satisfied. Instead of this I could use

RCL d, FIX 0, CF 29, RCL 02, VIEW X, PSE

to save the original display status. I could then replace the display status by using

X<>Y, STO d, RDN

The X<>Y gets the old contents of register d into X, STO d puts this back

into the flag register, and RDN puts the flag register value into T, so that the contents of registers X, Y, Z are as in the previous example.

Another occasion when RCL d is useful arises if you have some of flags 11 to 20 set and you want to turn off your HP-41 or you want to leave it unused for more than 10 minutes. Turning the HP-41 off and on again clears all of flags 11 to 20, but if you do RCL d before the HP-41 turns off then you can save their status. After the HP-41 turns on again, you should use STO d to put back the original settings. The contents of register d rarely look like an ordinary number or text string; they are usually an NNN (non-normalised number; see Section 8.2). This means that they can look very unusual in the display, and it also means that you cannot store them in a numbered data register and recall them unchanged. Remember that NNNs are normalised when they are recalled from a numbered data register (see Section 8.2). Numbers are not normalised when they are recalled from any of the status registers, otherwise even the instruction RCL d would not work properly, so you can save the contents of register d in any of the stack registers or registers M, N, O, then recall them from that register before doing a STO d.

```
01♦LBL "TRYd"
02 ENG 3
03 PI
04 VIEW X
05 RCL d
06 FIX 0
07 CF 29
08 VIEW Y
09 STO d
10 RDN
11 END
```

Before going on to the rest of this section, you should try out the example above or the one to the left. Enter this program (it is just a modified version of the steps described previously) and run it to see PI displayed first in ENG 3 mode, then without the fractional part or a decimal point. Then press the backarrow key to cancel the VIEW function and see the contents of X in ENG 3 mode. Instead of VIEW Y you could print a number, or ARCL it to make it into part of a message. You can use any display mode you like, STO d will always restore the original display mode. Just make sure that the previous contents of d really are in register

X, otherwise you can get some odd results which may need to be tidied up by the setting or clearing of a lot of flags. If you cannot remember how to create the steps RCL d and STO d, refer back to the examples at the end of Section 14.5. Use STO IND 16, AVIEW to produce RCL d; use STO IND 17, AVIEW to produce STO d.

Of the flags in register d, only 0-29 can be altered directly by the user. Flags 30-55 can be tested, and some can be set or cleared by special functions. For instance flag 48 can be set in a program by AON and cleared by AOFF. For various reasons it can be useful to set or clear some of the flags 30-55 even though Hewlett-Packard did not consider it wise. Let us begin with flag 55. This is set whenever the HP-41 detects the presence of a printer. The HP82143A printer, which is designed for use only with the HP-41, always sets flag 55, whereas printers attached via HP-IL might not set flag 55 until they are used. Once flag 55 is set though, the HP-41 knows a printer is present, and when a program is running in RAM every program step is copied into register Q in case the printer is in TRACE mode and wants to print that step. This means that register Q cannot be used for synthetic programming (as mentioned in the previous section), and that programs are considerably slowed down. Armed with our synthetic programming techniques, we can try to clear flag 55 so that the presence of a printer will not slow down programs or affect register Q. If we look at the bottom of the first half of the Byte Table, we can see that flag 55 is one of the eight flags in the last (rightmost) byte of register d. Storing a zero in that byte will clear flag 55 (and all the other flags 48-54). Normal HP-41 operations only allow us to alter complete registers, but the ALPHA register can be altered one character, and therefore one byte, at a time. The program below uses synthetic instructions and ALPHA operations to clear flag 55. It clears flags 48 to 54 as well, but this does not really matter as will be explained later. Even if you do not have a printer it is worth studying the techniques used by this program. If you are unsure how to enter the program, a detailed method will be given later.

01	*LBL "C55"	Program name - clear flag 55 - a three letter abbreviation is usually sufficient and saves space.
02	RCL d	First copy register d into register X.
03	STO M	Copy the flag set from register X into the rightmost seven bytes of the ALPHA register.
04	"⊦-ABCDEF"	Append six characters to the ALPHA register. This pushes the six leftmost bytes of the flag set into register N, and leaves the rightmost byte (flags 48-55) in register M, followed by six letters.
05	CLX	Put a zero into register X. This leaves all fifty-six bits containing zeroes.
06	STO M	Copy the zeroes into register M. The rightmost byte of the flag set, which contained the settings of flags 48-55, now contains all zeroes.
07	"⊦-G"	Append one more character to the ALPHA register. The former contents of register M have now been pushed a total of seven bytes to the left, so they are entirely in register N, except that the last byte has been replaced with a zero.
08	X<> N	Put a zero into register N, and copy the altered flag set back into X.
09	STO d	Store the altered flag set into the flag register. The last eight bits are all zeroes, so flags 48-55 are all clear.
10	RDN	Replace the original contents of registers X, Y, Z. L is also unchanged. T, M and N have been changed. The previous contents of N are now in O and can be recalled from there.
11	END	

You can use this routine to clear flag 55 wherever a part of a program does not need a printer. Simply XEQ "C55" and the printer will be disabled, register Q will not be changed at each step, and the program will speed up. This is a short yet effective example of the use of synthetic programming. Flags 48-55 are all cleared in just 9 steps (not counting the label and

END). The settings of flags 49-54 do not really matter; flag 49 is set by a low battery and will be reset at the next step if necessary, flag 50 is set if a message is being viewed which is not necessary in this case. Flags 51 to 54 are normally clear anyway in a running program. Clearing flag 48 clears ALPHA mode, so if you run the program by pressing R/S while in ALPHA mode then you will finish out of ALPHA mode, but this can be corrected by an AON step in the program if it is necessary. Flag 55 will stay clear until a PSE or until the program stops running if you have an 82143A printer and it is turned on. With an IL printer, flag 55 will be set if you try to execute a printer function, or a function such as VIEW which uses the printer, or after a flag test function. Be careful to clear flag 21 if necessary; when it is set but flag 55 is clear then VIEW and AVIEW will stop a running program.

If you are still not comfortable creating synthetic functions in a program then the following step-by-step instructions should help you.

```
01 *LBL "C55"
02  RCL IND 16
03  AVIEW
04  RCL IND 17
05  RDN
06  "←ABCDEF"
07  CLX
08  RCL IND 17
09  RDN
10  "├-G"
11  RCL IND 78
12  LASTX
13  RCL IND 17
14  AVIEW
15  RDN
16  END
```

First enter the program printed alongside. Check that it is correct and PACK. Next SST to the label, make sure you are in USER and PRGM mode, BG (press your Byte Grabber key), and press the backarrow key to delete the BG text string. SST once and see the line RCL d. BG and backarrow again, and SST three times, until you see CLX. BG and backarrow, SST twice. Then BG, backarrow, and SST once. BG, backarrow the last time, SST and see the last synthetic line, STO d. PACK again and the program is ready. To try it out, even if you have no printer, leave PRGM mode, press RTN, ALPHA and R/S. After about one quarter of a second the program stops with ALPHA cleared. If you made a mistake then your flags may not be set correctly - you will have to reset some of them, correct the program, and try again.

Although this is a simple example of synthetic programming, it is not perfect. For one thing, the program could be made a little shorter though this might make it less clear. A shorter version is given in the book Synthetic Programming Made Easy. Secondly, eight flags are altered all at once. If you do not have a printer you may still want to alter some other flags. A more useful, but more complicated program would set or clear any one of the flags 00 to 55.

The next program, SCF, can be used to set or clear any one flag. The flag number should be put into register X, and then SCF should be executed from the keyboard or from a program. If the flag number is positive then the program sets the flag, if it is negative then the program clears the flag. Since the HP-41 does not allow for a negative zero, use any negative fraction between 0 and -1 to clear flag 00. Apart from this, the fractional part of X is ignored. If only flags 00 to 29 were used, then the whole job could be done by the three steps (lines 12 to 14)

SF IND X, X<0?, CF IND X

To deal with flags above 29, the program shifts the flag register contents two bytes to the left (lines 23 to 27), and tries again. This will still not work for flags 40-55. In that case the program shifts the flags another two bytes to the left. After the desired flag has been set or cleared, the flag register is shifted two bytes to the right (lines 29 to 36) once or twice as necessary, and the program stops at the RTN. In case register X contains a number larger than 55, or a text string, the program tries to test the flag number at line 03. If flag 25 is set then an error at line 03 will be ignored, so the test is repeated at line 04; this time an illegal value will stop the program with an error message. If X contains a legal value then line 05 will usually be executed, but in some unusual cases it may not be, so it should be a NOP. The LBL 01 acts as a NOP at this stage, and later in the program it is used by a GTO as well. Lines 05 to 15 try to set or clear the flag; if the value is 30 or over then they jump to line 16 so that the flag register can be shifted left and these lines can be executed as a subroutine. LBL 03 is used so that the

program will not stop in the middle, where restarting could produce incorrect results.

The program leaves registers X, Y, Z and N unchanged, and it does not use any numbered data registers. In order to make this possible, the stack and ALPHA registers are used a lot, and some unusual arithmetic operations are carried out, particularly at steps 18 to 22 which decrease the absolute value of the flag number by 16. The flag number is decreased by 16 each time the flag register is shifted two bytes (sixteen bits) to the left.

```
01♦LBL "SCF"
02 STO [
03 FS? IND X
04 FS? IND X
05♦LBL 01
06 ABS
07 30
08 X<=Y?
09 GTO 02
10 RDN
11 X<> L
12 SF IND X
13 X<0?
14 CF IND X
15 GTO 03
16♦LBL 02
17 RDN
18 16
19 ST- Y
20 X<> L
21 SIGN
22 *
23 X<> d
24 X<> [
25 "⊢PP"
26 X<> [
27 X<> d
28 XEQ 01
29 X<> d
30 X<> [
31 "⊢ABCDE"
32 STO [
33 RCL ↑
34 X<> ]
35 X<> \
36 STO d
37 RDN
38♦LBL 03
39 RTN
40 END    82 BYTES
```

You should have had enough practice by now to be able to enter the program from the listing given here. (If you are lazy and have a wand then you can use the barcode in Appendix F.) As was mentioned in Section 14.5, registers M, N, O, P print as [, \,], ↑. If you need detailed instructions, then read through the next paragraph, otherwise skip over it.

Lines 02 and 32 are displayed as STO M. They can be created by entering the two steps RCL IND 17, RDN, then backstepping twice, pressing your BG key, and pressing backarrow. Lines 23, 27 and 29 can each be created by entering RCL IND 78, AVIEW, backstepping twice, pressing BG and backarrow. Lines 24, 26 and 30 are X<>M created from RCL IND 78, RDN, followed as usual by the four steps BST, BST, BG, backarrow. Line 36 is made in the usual way from RCL IND 17, AVIEW. Lines 33, 34 and 35 are RCL P, X<>O and X<>N. They are made from RCL IND 16, X=Y?, from

RCL IND 78, CLX and from RCL IND 78, LASTX respectively. As you go through these steps, you should follow them on the Byte Table (Table 14.1) so as to see how they are made. If any instruction fails to come out as expected then repeat the steps given, and remember to PACK.

To check if you have entered the program correctly, put 49 into X and execute SCF. The BAT annunciator should come on. You can clear it by pressing CHS and running the program again, or by turning your HP-41 off.

The program will not work for flag 53 and may create problems with some others. Flag 53 is used to check if any I/O device needs attention; it is checked and cleared after every program step. Thus flag 53 will be cleared as soon as you set it. Whenever flag 53 is cleared, flag 54 is also cleared if it was set. Moreover if flags 53 and 45 are both set then flags 30, 45, 47, 50, 53 and 54 will all be cleared. When flag 53 is cleared in a running program then the current program line number (see Section 14.10) will be reset as well.

Flag 54 is the PSE flag; in a running program it does not execute a PSE until the program comes to a stop at a RTN or END. Then the program does a PSE, clears flag 54, and starts running again from the next step (after a RTN) or from the beginning of the program (after an END). If you call SCF as a subroutine and set flag 54 then your program might not stop where you expect it to, unless you clear flag 54 later by calling SCF again with -54 in X. If you execute SCF from the keyboard then after setting flag 54 it would not stop at an END but would run again. For this reason there is an extra RTN at line 39; SCF can do a PSE at this line and stop at the next one. If you set flag 52 then the HP-41 gets the idea that it is in PRGM mode and that you are writing a program. Normally this does not matter, but if flag 52 is set when a number entry line is executed, then the HP-41 begins programming itself, repeatedly putting the first digit of that number into program memory. This may leave you a lot of tidying up to do, so be careful with flag 52. One more warning: SCF may use one or two subroutine calls internally so take care not to exceed a total of six pending subroutine calls while you use it.

The program SCF can be used for a variety of purposes, including exploration of the HP-41. As an example you could set flag 30 (the Catalogue flag), then press R/S to restart a catalogue and see which catalogue you have gotten into. If you have an HP-IL module you can disable all IL operations by setting flag 33, and if you have an Auto Start/Duplication Module then you can disable the Auto Start by setting flag 35. You can set flag 44 (continuous ON) or flag 47 (SHIFT). Normally these are non-programmable operations. If flag 40 is set alone then the HP-41 is in FIX mode which displays numbers without an exponent if possible, and otherwise displays them in SCI mode. You can use SCF to set flag 41 as well as flag 40, giving FIX/ENG mode which displays numbers in FIX mode if possible, and otherwise in ENG mode.

Some other books contain programs similar to SCF. These usually invert (or toggle) the flag whose number is in X - if the flag is set then they clear it and if it is clear then they set it. SCF lets you explicitly set or clear a flag, this makes it a little longer. The other programs usually use clever tricks to make them shorter and faster, I have avoided this to make the program easier to enter and simpler to understand. After you have read this book you may want to try the programs in other books. The description of the program IF in the PPC ROM manual is particularly worth reading, and the program is worth studying as it uses several clever methods to make it as short as possible (it is only two-thirds of the length of SCF). The ZENROM function TOGF does the same job as IF.

SCF is very useful for setting one flag, but rather slow if you want to set a whole lot of flags. In such a case it is better to build up a non-normalised number corresponding to the required flag settings, put this number in X, and then STO d. How do we build up an NNN though? Maybe you can guess - we use register d. By setting and clearing flags 00 to 07 we can build up any required combination of bits in a byte. Then we use ALPHA operations to append this byte to register M. By repeating this seven times, we can append seven bytes to register M, then RCL M to provide a complete NNN. This can be stored in register d to provide a flag set, or

it can be used for other synthetic purposes. A routine which builds one byte and appends it to Alpha can also be used to create non-keyable characters such as quotation marks or lower case letters. The program BAB, "Build a Byte", shown below takes a number between 0 and 255 in register X and appends the corresponding byte to register M.

```
01♦LBL "BAB"
02 STO ]
03 ABS
04 ABS
05 256
06 R↑
07 STO \
08 RDN
09 .007
10 R↑
11 STO ↑
12 CLX
13 X<> d
14 X<>Y
15 RDN
16 RDN
17♦LBL 02
18 MOD
19 LASTX
20 1/X
21 ST+ X
22 1/X
23 X<=Y?
24 SF IND Z
25 ISG Z
26 GTO 02
27 R↑
28 X<> d
29 X<> [
30 X<> \
31 RCL ↑
32 RCL ]
33 "+A"
34 RCL \
35 CLA
36 STO [
37 RDN
38 END
```

Line 02 saves the original X value, lines 03 and 04 check if it is a number. The test is made twice (as in SCF) in case flag 25 is set, and it leaves a positive value in X. Line 05 puts 256 into the stack for later use, lines 06 to 08 save the value in Z, line 09 stores an ISG counter in the stack: this will be used for a loop to set eight flags. Lines 10 and 11 save Y in register P; this could not be done earlier because number entries alter P. Lines 12 and 13 put zero in register d so that all the flags are clear and the original register d value is saved in the stack. Lines 15 and 16 put 256 back in the X register. The absolute byte value is now in register Y, the flag counter is in register Z, and the original contents of register d are in T. Line 17 starts the loop which will turn the byte value into a set of powers of 2, setting each of the flags 00 to 07 if the corresponding power of 2 is present in the byte value. Lines 18 and 19 obtain the remainder of dividing the byte value by the required power of 2. At the first pass through the loop this makes sure that the byte value being set is not larger than 255. There are no error checking steps in this program except for lines 03 and 04 and these two lines. Lines 20 to 22 halve the number in X. The steps 2, / cannot be used because they would push the value in T off the top of the stack, and would

alter register P. Lines 23 and 24 set the required flag if X is not greater than Y, meaning that Y contains the power of 2 that is stored in X. Lines 25 and 26 increment the flag counter, and go back to repeat the loop if necessary. At the end of the loop, flags 00 to 07 contain the byte we need. Lines 27 and 28 get this into register X and restore the original contents of register d. Lines 29 and 30 put the new byte at the left of register M and put register M into register N while putting the previous register N (the original Z) back in the stack. Lines 31 and 32 also recover Y and X from P and O. Line 33 moves the byte from the lefthand end of register M to the righthand end of register N. After this step register N contains the previous value of M with the new byte appended at the right. Line 34 recovers this value and line 35 clears out the ALPHA register. Line 36 puts the value back into M and line 37 restores the original contents of X, Y and Z.

When the program finishes, registers X, Y and Z are unchanged. Register M has lost its leftmost byte and gained a new byte at the right. This byte is byte number X. If X did not contain a positive integer between 0 and 255, the byte is ABS(X) MOD 256, with the fractional part of X ignored. If X contained a non-numeric value, then the program should have stopped at line 03 or 04. The ALPHA register is cleared except for register M; if you do CLA before executing BAB then you can use BAB to put a single non-keyable character into the ALPHA register and then ASTO this character in a register for later use.

Lines 02, 07, 11, 13, 28, 29, 30, 31, 32, 34 and 36 are all synthetic lines of the sort you have made in the previous examples. Use the same methods to create these lines. If you have entered this program, you can try it out by doing CLA, putting 12 into register X, and running the program. At the end you should still see 12 in register X, but the ALPHA register should now contain the Greek letter mu. You can ASTO this in a data register, or you can add more characters to it, for example add the & character by putting 38 into X and running the program again. Use BAB up to seven times to build a NNN in register M, then RCL M and put it where you want, for example STO d. If you begin with an empty register M and add

n bytes, then the leftmost 7-n bytes of M will still contain zeroes, so you do not need to use BAB to create these null bytes. Several programs exist to automate the whole procedure of converting fourteen nybbles (entered in hexadecimal in Alpha) into a seven-byte NNN. These programs are usually called CODE or ENCODE. Programs have also been written to DECODE an NNN and display it as fourteen nybbles. Such programs are available in Wickes' book, the PPC ROM and in ZENROM. A program to decode bytes one at a time will be given in Section 15.3.

Be careful of three things. First of all BAB expects the byte number in X to be a decimal value. Should you want to put the hexadecimal byte number FC into register M, then you will have to check in the Byte Table, see that the decimal equivalent of FC is 252, and put this number in X when you are using BAB. Secondly, avoid running this program (or any other synthetic program) with a printer attached and set to TRACE mode; in this mode the printer can normalise numbers in X before printing them. Avoid single-stepping through the lines 11 to 31 because this changes register P. In general it is unwise to SST through any synthetic program that uses registers P, d, or c; far better to run the program and wait for it to stop.

Instead of using BAB, you can use the PPC ROM program DC, if you have a PPC ROM. Byte values from 0 to 127 can also be entered from barcode if you have an optical wand. Barcodes for these characters are given in the Appendix F. Byte values from 128 to 255 cannot be entered directly into Alpha with a wand. The astute reader will have noticed that BAB does the same job as the Extended Function XTOA. BAB has been included here for those users who do not have the Extended Functions, and as an example of how register d can be used in Synthetic Programming. I shall now go on to some other status registers, but you should keep the programs SCF and BAB in your HP-41 as they will be used again later.

14.9 Register c, a vital register.

All sixteen registers accessed by bytes from row 7 can be called "status registers", but register c is the status register. It records the positions of the .END., of the "curtain", and of the statistics registers, each as a three-digit pointer. Register c also contains a three-digit number whose value is frequently checked; if the value of this number changes then the HP-41 assumes something has gone dramatically wrong and initiates a MEMORY LOST. Figure 14.2 shows how these numbers are arranged within register c.

13	12	11	10	9	8	7	6	5	4	3	2	1	0	Nybble
s	s	s			1	6	9	c	c	c	e	e	e	Contents
6			5		4			3			2		1	0	Byte

Figure 14.2 Register c contents

Each pointer is a three-digit hexadecimal number giving the absolute address of the first statistics register, of register 00, or of the register containing the .END. (the .END. is always stored in bytes 0, 1 and 2 of a register so the register number alone is sufficient to locate it). As was explained in Chapter 8, absolute addresses are fixed to the HP-41 hardware, for example register T is always at absolute register 000. The pointers in register c allow the HP-41 to use **relative** addresses, for example data register 10 is 10 registers above data register 00; the position of register 10 is known only relative to the position of register 00. On an HP-41CV the highest absolute register of normal memory is 1FF hexadecimal (511 decimal). Thus if you set SIZE 100 on an HP-41CV then you have 100 (decimal) data registers, and data register 00 is at absolute register 19C hexadecimal (412 decimal). Nybbles 3, 4 and 5 will contain 19C and the position of data register 10 will be calculated relative to the address of data register 00, at absolute register 1A6 hexadecimal (422 decimal). If you change the SIZE then the pointer in bytes 3, 4 and 5 will

change. If you add memory modules to an HP-41C then the position of data register 00 will not change, but the total number of data registers available will change. To avoid confusion here I shall give absolute register numbers as three-digit hexadecimal numbers and relative registers as two-digit decimal numbers.

The address of register 00 is called the curtain (named by Bill Wickes, the originator of synthetic programming, who first suggested manipulating it), because it separates data registers from the top of the program area. The address of the .END. marks the other limit of the program area. Just as the curtain address is used to find any data register, so the .END. address is used to find any program in CAT 1. As was explained in Chapter 8, the .END. points to the preceding global label or END and this chain extends back to the first END or global label in program memory. All addresses below the .END. down to absolute register 0C0 hexadecimal (192 decimal) are available for storage of buffers, alarms and key assignments. The HP-41 sometimes checks this area starting at the .END. and going down. This is called Method 1 (or just M1, see the article by Bruce Bailey in PPCCJ V9N7P10-11), it is used to count the number of free registers below the .END. whenever this number is displayed together with the .END. (or as 00 REG NNN). Method 2 (or M2) starts at absolute register 0C0 and goes up towards the .END.; this is used by PACK to remove unused key assignment registers (KARs) or to delete unwanted buffers. M2 stops as soon as it comes to an empty unused register, so PACKing will ignore any registers placed by synthetic means in the middle of the free area. Since M1 stops at the first non-zero register below the .END., storing anything in the free area will also prevent the program area from expanding right down to the key assignments. There is a third problem; a non-zero register in the free area might be treated as a zero-length buffer. The HP-41 checks buffers one after another by finding the length of each buffer and using this to calculate the position of the next buffer. A zero length buffer is therefore checked time after time without any stop, and the HP-41 hangs up. These problems are described in the article by Bruce Bailey and in the first paragraph of page 282 in the PPC ROM User's Manual. The conclusion is simple; avoid storing anything in the free area below the .END. unless

you know what you are doing.

The third pointer in c gives the absolute address of the first of the six statistics registers. When you do ΣREG nn, the HP-41 takes the absolute address of the curtain, adds nn to this, and stores the new address in nybbles 13, 12 and 11 of register c. When you change the SIZE, the curtain address and the statistics pointer change by the same number of registers.

Nybbles 9 and 10 are a scratch area. When a copy of the contents of register c is put into the CPU then these bytes may be used by the printer, but in register c itself they are never changed. You can therefore use them to store a byte of data. The CCD Module uses these nybbles to set modes for hexadecimal arithmetic and for its permanent Autostart function.

That leaves the 169 in nybbles 6, 7 and 8. The HP-41 checks this number when it turns on, after each function executed from the keyboard, and whenever a program stops or pauses. It does <u>not</u> check this number while a program is running; if a program starts running when the HP-41 is turned on then the number is not checked till the program stops or pauses. If the contents of these nybbles are ever found to be other than 169 then the HP-41 operating system assumes something has gone badly wrong, most likely the batteries have gone completely flat and the contents of Continuous Memory have been disrupted. To prevent the user from working with corrupted data, the operating system clears everything from memory and restarts from a MEMORY LOST. An operation like this is called a **cold start** on computers and the 169 is usually called a **cold start constant**. Matters are confused by the fact that Hewlett-Packard calls this a **warm start constant**, since a cold start is made only if the constant is lost, otherwise a normal warm start is made. When the HP-41 checks the 169, it also checks whether the first register below the curtain address actually exists. This should be the top of program memory and if it does not exist then something has gone wrong. Perhaps a memory module has been removed from an HP-41C. In any case the absence of this register means that CAT 1 is not complete, and again the HP-41 does a MEMORY LOST.

As you can see, register c is indeed a vital register. It tells how the memory is partitioned at any given time, and it controls whether you will lose everything from your HP-41. So long as an "HP-41" was an HP-41C with 64 registers (and possibly one extra memory module), clearing all of memory in case of trouble could be considered reasonable. Now that many users have 41CVs or 41CXs with Extended Memory, a MEMORY LOST can mean up to 923 data registers (6K bytes of data) lost. Even the people who wrote the HP-41 operating system were not sure if clearing out all of this memory would be a good idea, but the Hewlett-Packard view is that no data is better than bad data. This could well be right if you are building bridges or landing space shuttles; in such cases you will have stored the data on a Card Reader anyway. (Won't you? Well you will in future!) By the way, some HP-41s on the Space Shuttle have extra loud beepers built into Card Reader shells, but others do have real Card Readers.

In view of all this you must be very careful when altering register c. On the other hand it can be very useful to change its contents. One example is changing the curtain position to hide or uncover some data registers. You can write a synthetic program by using BAB to build up several NNNs, and store them in the data registers immediately above the curtain. Then you can move the curtain up so that these data registers become part of the first program in memory. To move the curtain you have to do a RCL c, copy this to the ALPHA register, then use ALPHA operations and BAB to alter the register. Finally you recall the new version of c from Alpha and use STO c to put it back.

Many synthetic programs change register c temporarily. In these cases, the instruction X<>c is most useful; it lets the program change c, and later put back the original contents. A common example is that one program has stored important values in registers 00 to 10, but another program is also written to use registers 00 to 10. You can move the curtain up by 11 registers and hide the original registers 00 to 10 inside the program area. Then the registers that were previously numbers 11 to 21 become registers 00 to 10, and the second program can use them without affecting the real registers 00 to 10. When the second program has finished, you can move the

curtain down by 11 registers again and thus recover the original registers 00 to 10. Another example is that you can pull the curtain down, turning part of the program area into data registers. Then you can use CLRG to clear these registers and later you can move the curtain back up. This lets you clear part of the program area, rather like a programmable version of DEL or CLP. Figure 14.3 shows the effect of raising the curtain by 10 registers. The contents of the registers do not change, the registers are just renumbered.

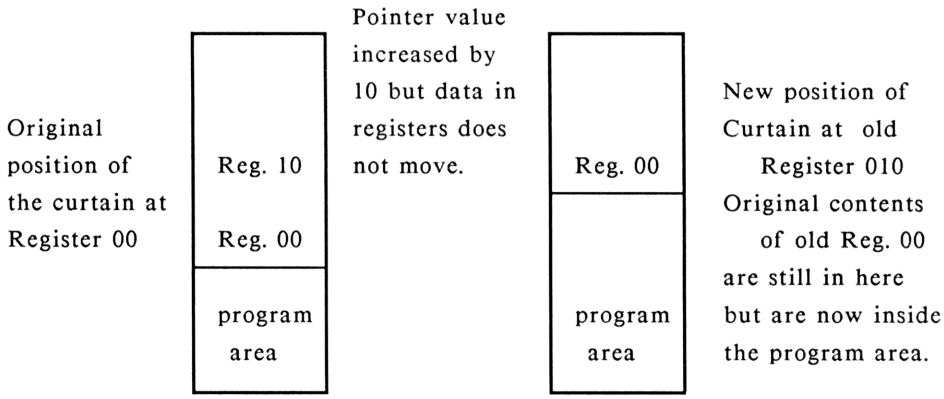


Figure 14.3 Moving the curtain

While the curtain is raised, ordinary data suddenly finds itself in a program. A PACK or GTO.. would treat such data as program bytes and would remove nulls, destroying the data. You must therefore make sure that PACKing will not occur while the curtain is raised. Program memory now starts at the new curtain position, so the first program will look for labels in the data, and may try to execute the data as a program. Thus the first program in memory should not be used while the curtain is raised; or at least it should not execute any uncompiled backward GTOs. You can arrange this by putting an END right at the beginning of memory. To do this, execute CAT 1, stop it at once, GTO.000, turn on PRGM mode, and enter XEQ "END". Then PACK memory and you are protected against problems when you lift the curtain. If you lower the curtain and the first program

in memory does an uncompiled GTO then it will start the LBL search at the first register below the curtain - this can lead to NONEXISTENT if the label is now above the curtain. If you lower the curtain so far that it is below the END of the current program then the label search will begin immediately below the curtain, inside the program where the curtain now is. This lets you execute a local LBL which is not inside the current program. A RTN or END from that program will still return you to the calling program. A curtain-raiser will be introduced soon.

You can use synthetic programming to change the .END. pointer but this is rarely useful as it just loses you the CAT 1 linkage. An example of moving the .END. to get into Extended Memory will be shown in the PPRP program in Chapter 16. Another use is for suspending Time Module alarms, as explained in PPC CJ V10N7P11, by Tapani Tarvainen. Changing ΣREG has a few uses. One is to move the statistics registers to a non-existent area so that statistics functions cannot damage any real data. Another use is to move the statistics registers so that they coincide with part of the stack. If you have Extended Memory, you can locate the statistics registers in an Extended Memory data file. A third use is for curtain raising as below.

If you move the curtain you need to know where it was previously so that you can move it back later. A very clever synthetic operation is to exchange the statistics pointer and the curtain pointer. If you execute ΣREG 010 and then exchange the two pointers you will have hidden data registers 00 to 09 as was described earlier. Since the original curtain position is saved in the statistics pointer, you can recover the curtain just by executing the same program again. The instruction ΣREG 00 will move the curtain to the statistics pointer, but moving the statistics pointer to the curtain demands some byte shifting. The most difficult part is to move nybble 11 to nybble 3 of register c without changing nybble 2. In actual fact, nybble 2 can only be a 0 or a 1 unless the .END. has been moved synthetically, so a flag operation can be used to change the necessary bit.

The program ΣCX (Sigma - Curtain Exchange) will swap the statistics pointer

and the curtain so that you can try out curtain raising and lowering for yourself. The version given here was inspired by the program ΣC in the PPC ROM and by later versions published in the PPC Journal. The idea of ΣC came from Keith Jarett who together with Clifford Stern wrote several versions.

```
01♦LBL "ΣCX"
02 RCL c
03 X<> d
04 CF 15
05 CF 14
06 CF 13
07 CF 12
08 FS? 47
09 SF 15
10 X<> d
11 ΣREG 00
12 X<> c
13 STO [
14 STO \
15 "⊢****"
16 X<> c
17 STO \
18 "⊢**"
19 X<> [
20 X<> \
21 "⊢*"
22 X<> ]
23 X<> c
24 RDN
25 CLA
26 END
```

Lines 02 to 10 use flag operations to clear nybble 10 of register c in case it has been set by the user or by the CCD Module, then they copy a 1 or 0 from nybble 02 to nybble 10. Lines 11 and 12 make a new register c with the statistics address at the curtain address and then put the old c back. Lines 13 to 15 shift the new version of register c by 4 bytes, using two copies of register c. Lines 16 to 21 combine the original version of c with the new version using more shift operations. Lines 22 to 24 put the final version of c into c and roll down the stack, with X, Y, Z unchanged. No numbered data registers or flags are changed either but the ALPHA register contains leftover rubbish, which is cleared by line 25.

You can now try curtain moving. Put 0 in register 00 and 10 in register 10. Execute ΣREG 10, ΣCX, and RCL 00. You will get 10 instead of 0 showing that the original register 10 is now register 00, but if you execute ΣCX a second time then register 00 will again be the one containing a zero. If you have raised the curtain to store some synthetic instructions (codes stored in the data registers before you raised the curtain) in your first program then you will not want to move the curtain down again. In that case you should execute ΣREG nn.

If you do want to restore the original curtain position, be careful not to

lose it. Should you want to do some statistics, do a RCL c first, then do a ΣREG nn, do the statistics, and then replace the previous register c. Never store the contents of c in a numbered data register because they will be normalised when you recall them.

You must be careful if you want to store any non-normalised numbers for conversion to program steps using ΣCX. ΣREG nn normalises the first and last registers of the statistics block - registers nn and nn+5. The ΣCX program includes a ΣREG 00 instruction. This means you cannot store NNNs into registers 00 or 05 if you plan to use ΣCX to raise the curtain over them. Indeed you should always use this program with care, particularly if you have synthetically set the curtain or the statistics pointer to point to the status registers.

14.10 Registers ⊦ and e; making synthetic key assignments.

Whenever you press a key in USER mode, the HP-41 checks an internal flag, part of an assigned key index, to see if something has been assigned to that key. Flags for shifted keys are stored in register e, and those for unshifted keys in register ⊦ (also called the append register or register R). Figure 14.4 shows which bit corresponds to which key. The nine nybbles at the left of each register cover all the keys, so the remaining five are available for other purposes. Bit number 24 corresponds to a non-existent key hidden under the right-hand side of the ENTER key. These bits are not normally used for anything; ZENROM uses them to save the status of the USER flag in ALPHA mode.

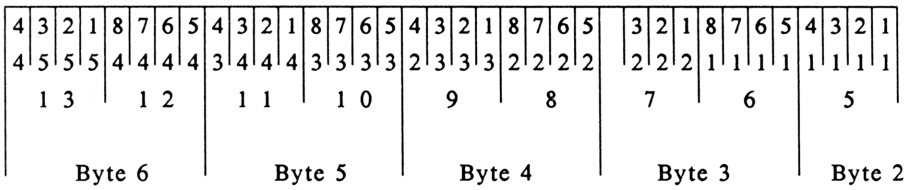


Figure 14.4 The key assignment flags

Each flag is shown as a box - the number in that box is the keycode for the key whose status is stored in that flag. Register ⊦ has the flags for the unshifted keys, and register e has the flags for shifted keys.

The last three nybbles of register e contain the current program line number. During a running program or a CAT 1 this number is set to FFF (which is 4095, higher than any possible valid line number) so as to indicate that the line number will need to be recalculated. Other nybbles are used for scratch purposes. You can change the line number synthetically to any value up to FFF (4095 decimal). This will let you BST from any program to the previous one. Line numbering will be corrected when you SST an END. You can also correct line numbering by setting flag 53 in a running program (use SCF or a similar program).

By synthetic means you can set assignment flags to make new key assignments. If no corresponding key assignment is stored in the assignment area then CAT 1 is searched and, if no assignment is found there either, then bytes left over from the CAT 1 search cause execution of some function, most often ABS or CAT. Details will be given in Example 2 of Section 15.5. You can clear individual flags, or you can clear them all. When the flags are cleared, you can use the keys for their normal functions. This is particularly useful if you need to use the top two rows of keys for local labels in some programs, but you want to use them for assigned functions at other times. The instructions:

CLX, X<>e, 0, X<> ⊦

will temporarily disable assignments, and you can re-enable the assignments by storing the original values back in the registers. You can also recover the assignment flags by reading a program from Extended Memory, from a Card, or from an HP-IL device (using READP or the Extended I/O function INP), even if USER mode is not set.

We have now reached a stage when a synthetic key assignment program will be comprehensible. First of all BAB lets us use Alpha and register d to build

up a key assignment register (KAR) containing a synthetic key assignment. We next change register c, moving the curtain down so that the KAR area can be treated as data registers and the synthetic key assignment can be stored in this area. Then we use SCF to set an assignment flag in register e or register f, and we have a new key assignment.

The writing of key assignment programs has been a major industry in HP-41 user clubs around the world. The most helpful programs check for errors made by the user, and this makes them fairly long unless they use Extended Functions. If you have a lot of HP-41 memory, and the patience to enter a long synthetic program, or if you can afford a PPC ROM or an Extended Functions Module or an HP-41CX, then these programs are what you want. Readers who have an HP-41CX (or an Extended Functions Module and a Time Module) can use the completely non-synthetic program given in Chapter 11 of this book. That program is not long, but it is difficult to understand and will not make assignments to the SHIFT key. The Key Assignment Program (KAP) given here will make assignments even to the SHIFT key; it is reasonably short because it does not check for errors and uses BAB and SCF as subroutines. It is fairly easy to understand, as it follows the outline given above and does not use any loops or GTOs. Experienced synthetic programmers may be horrified at some of the shortcuts taken, but the resulting program will fit even into an HP-41C, will do the required job, and is fairly easy to enter even for a beginner. If you have an Extended Functions Module you can speed the program up and make it shorter by using XTOA instead of BAB in lines 13, 15 and 38.

```
01♦LBL "KAP"      17 R↑             33 LASTX          49 X<> c          65 STO '
02 CF 10          18 -              34 8              50 X<>Y           66 FC?C 10
03 X>0?           19 X<0?           35 FS? 10         51 STO IND Z      67 STO e
04 SF 10          20 CHS            36 CLX            52 X<>Y           68 END
05 ABS            21 DSE X          37 +              53 X<> c
06 INT            22 DSE L          38 XEQ "BAB"      54 "┣ABCDE"
07 10↑X           23 LASTX          39 36             55 X<> [              146 BYTES
08 X<> L          24 10             40 RCL Z          56 "┣FG"
09 10↑X           25 /              41 -              57 R↑
10 X<> L          26 INT            42 176            58 RCL \
11 "XXX"          27 LASTX          43 FS? 10         59 X<> d
12 X<> Z          28 FRC            44 RCL '          60 X<>Y
13 XEQ "BAB"      29 80             45 FC? 10         61 XEQ "SCF"
14 X<>Y           30 *              46 RCL e          62 X<>Y
15 XEQ "BAB"      31 +              47 X<> [          63 X<> d
16 43             32 ST+ L          48 8.00016901     64 FS? 10
```

To enter the program's synthetic instructions, use the methods described in Section 14.5. Lines 44 and 65 recall and store the append register, which is provided by the decimal postfix 122. You can create these lines by writing RCL IND 16, SIGN and RCL IND 17, SIGN, and by Byte Grabbing the first byte in each case. The only synthetic instruction that cannot be recognised is at line 11; it consists of the five bytes F4, F0, 58, 58, 58. This is a text string containing 4 characters (so the first byte is F4), but the second byte (F0) is not recognised by the printer and does not get printed. The remaining three characters are ordinary Xs and are printed as such. There are several ways to make this instruction; here is one.

1) Enter the text line "XXXX", and PACK.
2) BST to the line before this line and press the Byte Grabber (BG).
3) Insert the step RCL IND T.
4) BST and BG, then delete the BG line. This leaves an F0 byte.
5) PACK; this pushes all the bytes together again.
6) BST, BG and backarrow. This releases the F4 text line header.
7) SST and backarrow, then SST again and see a text line with the first "X" replaces by a boxed star (the F0 byte).
8) SST again and delete the E↑X-1. This is the fourth "X" which has got pushed out of the text line.

Entering a program with a lot of synthetic instructions demands a lot of editing, and the messages PACKING, TRY AGAIN might be displayed occasionally by the HP-41. Do not worry about this, just carry on entering the program when TRY AGAIN is displayed.

To use the program you put a function prefix in register Z, a function postfix in register Y, and a function keycode in register X. Then you can execute KAP from the keyboard, or call it from another program. In other words prefix, ENTER, postfix, ENTER, keycode, XEQ "KAP". The prefix and postfix are decimal numbers representing two bytes; the keycode is the same as that used by ASN. If you are assigning a one-byte function then make it the postfix and use a prefix value of 4. This is the same as using other key assignment programs such as GASN in this book or the ones in Wickes' or Jarett's books or in the PPC ROM. There is one major difference; KAP stores the new assignment in the lowest key assignment register, destroying any previous assignments that were there. (Wickes' program and the PPC ROM program store the assignment above the highest assignment register, destroying the alarm buffer instead.) To avoid destroying any important assignments, you should PACK and make at least two (three would be safer) dummy assignments (using ASN or PASN) to two different keys (or the same key shifted and unshifted), then delete the dummy assignments. This will normally produce an empty assignment register which can be replaced by KAP. You must make two dummy assignments and then delete them every time before using KAP. KAP makes only one synthetic assignment per register; this is wasteful but fast. The three programs BAB, SCF and KAP take up a total of 296 bytes, or just under 43 registers, and do not use any numbered data registers. On an HP-41C with no memory modules this leaves a total of 21 registers for other programs, data, alarms, and key assignments. This is sufficient for a few assignments, but not many. KAP uses flag 10 and leaves it clear at the end.

If you are not interested how KAP works then skip this paragraph and the next two, and go to the examples. Lines 02 to 06 set flag 10 if the keycode is positive, and clear it if the keycode is negative (a shifted

key). Then they obtain the positive keycode and remove any fractional part. Lines 07 to 10 check if the keycode is larger than 99; if so it cannot be a valid keycode and the program stops. The test is made twice in case flag 25 has been set. If you don't care about preserving the status of flag 25, use CF 25 instead. Line 11 puts the bytes F0, 58, 58, 58 into register M. The F0 is a key assignment register (KAR) header and the next three bytes put a nonexistent key assignment in the first half of the KAR. The rest of the program builds a real key assignment in the second half of the KAR; although this is wasteful it is much faster than putting two assignments in one KAR. Lines 12 to 15 append the prefix and then the postfix to the KAR.

The program must now append a keycode to the KAR. This means that the keycode RC (row R, column C) must be turned into the different code used in a KAR. This is given as a decimal number by the formula 16*(C-1)+R. If the key is shifted then the formula is 16*(C-1)+R+8. The program must also work out which flag to set in the key assignment flags. The flag number is given in decimal by 36 - 8*(C-1)-R. Flags for shifted and unshifted keys are stored in two different registers, so the formula does not change for a shifted key. If the key is 42, 43 or 44 (shifted or unshifted) then C must be increased by 1 in all the above formulae because the ENTER key is treated as two keys. Lines 16 to 22 take the code RC from register Z and turn it into R and C-1 (unless the keycode is 42, 43 or 44 in which case C is left unchanged). Note that lines 14 to 16 produce a value in X that is 1 or 0 for keys 42, 43 and 44. Line 21 then uses DSE X to skip line 22 for any of these values. For integer values, DSE X acts like an X<=1? instruction. If line 22 is executed then it turns the RC (actually 10R+C) in register L into the code 10R+C-1. Line 23 will be executed unless the value RC was 0 or 1 which would not be a legal keycode. Lines 23 to 28 put R into Y and the new C divided by 10 into X. Lines 29 to 37 put the KAR code into X and 8*C+R into Y (using the new C). Line 38 appends the KAR code to the ALPHA register. At this stage the new KAR value is ready in register M. The keycode is not fully checked to see if it is legal; keycodes with rows 0 or 9 and with columns 0 or over 5 will produce unusable assignments or cancel assignments but will not stop the

program.

Lines 39 to 41 calculate the assignment flag for the selected key. Line 42 puts the number 176 onto the stack for later use. Lines 43 to 47 put the assignment flags into register M and the KAR into the stack. Lines 48 and 49 put a temporary value into register c. This value puts the cold start constant in the right place and puts register 00 at absolute address 010 (=16 decimal) so that MEMORY LOST cannot occur. A total of 9 bytes could be saved here and later by the use of a synthetic text string instead of a number, but this would add extra difficulty to the task of entering the program. Lines 50 and 51 store the KAR value into absolute register 0C0, using the 176 entered at line 42 as an indirect address, then lines 52 and 53 restore the original register c value. Lines 54 to 56 replace the last two bytes of the key flag register with the first two bytes of the former temporary register c. These two bytes are 08, 00 so no crucial flags are affected and the program can be SSTed; the digits 800 were chosen as the first three digits at line 48 for this reason. The last two bytes need to be changed because the last byte of register e normally contains FFF in a running program and this would clear flag 30 when copied into register d with a Time module (or CX) present. (Indeed this causes the PPC ROM programs MK, 1K and +K to suspend assignments made to key -61.) Line 57 recovers the number of the status flag to be set, lines 58 to 61 set this flag, and lines 62 to 67 replace the altered status flag register while also restoring the original register d.

KAP can be used to assign any two-byte function to any key except the toggle keys. It can also be used to assign any ordinary one-byte function to any key, including the SHIFT key. As a simple example, use KAP to assign PACK to the shifted SHIFT key. First make two dummy key assignments. Use ASN to assign + to both ENTER and SHIFT, ENTER. Then use ASN to cancel these two assignments. Put the prefix 04 (any number from row 0 of the Byte Table will work, but 4 is the default) into X. Press ENTER and put in the postfix 10. This is the byte for PACK; see the Byte Table again. Press ENTER and put in the keycode -31; the shifted SHIFT key - row 3, column 1, negative for a shifted key. Now execute KAP. Various

annunciators will turn on as register d is used. When the program finishes, the stack and the ALPHA register will contain non-normalised numbers which you can remove with CLA and CLST. You can add these steps to the program if you like. If you now enter USER mode and press SHIFT twice, you will get the PACK function. If you keep the SHIFT key pressed down the second time then you will cancel PACK, and you will cancel SHIFT too, which is what should happen when you press SHIFT twice. This is a very useful assignment for synthetic programming because it lets you PACK by pressing a single key twice, and PACK is used a lot in combination with the byte grabber.

The program has been written to avoid the difficulties that some synthetic programs have. It can be the first program in memory and you can SST through it to see what it does. (In this particular case you can SST through BAB and SCF because the contents of register P are not used.) You can stop it at any time by pressing R/S and you can SST through part of it and press R/S again to restart it. **However, you must always let KAP run to completion if you don't want to risk MEMORY LOST.** You can even call KAP as a subroutine from another program if that program puts the required numbers into the stack. KAP is also short and fairly easy to enter into memory; this is partly due to the fact that it does not check carefully for incorrect inputs. Any text strings will stop KAP at lines 03 or 05 or within BAB. The stack will be mixed up but registers c and d will not be affected. Byte values over 255 will be turned into a number modulo 256 by BAB. The keycode in register X is only checked to see if it is smaller than 100.

No other tests are made by KAP. This means that it will overwrite whatever is in absolute register 0C0. If you have not made a pair of dummy assignments then this could be a pair of useful assignments, or the first register of a buffer, or even the .END.. If an assignment has already been made to a key then KAP will replace that assignment with a new one. So long as you are careful with KAP it is a very useful program. Some of its uses will be described in the next section and in the next chapter.

14.11 Registers a and b - the current address and RTN stack.

The rightmost two bytes of register b contain the absolute address of the program step being performed. The program step displayed if you are in PRGM mode is the next complete step following this address. This is called the **current address**. When you XEQ a subroutine the next program step is not performed until the subroutine does a RTN. Until then the current address is pushed two bytes to the left in register b, and the first address of the subroutine becomes the current address. When the routine does a RTN or END then register b is pushed two bytes to the right so that the address of the XEQ becomes the current address and the program carries on from the address after that. A GTO replaces the current address with the address of the label to which it goes. The previous current address is lost, meaning that a program cannot return to the line after a GTO.

One subroutine can XEQ (or call) another subroutine; this is called **nested** calling to distinguish it from the case when one subroutine is called and returns, then another subroutine is called separately. If you XEQ a subroutine and that subroutine does another XEQ then the contents of register b are pushed two more bytes to the left and a second return address is saved. The first RTN or END will return to this address, another RTN or END will be needed to bring you back to the original program. If the second subroutine executes a third subroutine then b is pushed another two bytes to the left. This make 8 bytes in all so the leftmost byte can no longer fit in register b. The eighth byte is put at the right of register a instead. Register a contains six more bytes, so another 3 subroutine calls can be made, giving a total of 6. If a seventh XEQ is made then the oldest return is lost from the left-hand end of register a. Figure 14.5 shows how this scheme fits into the two registers.

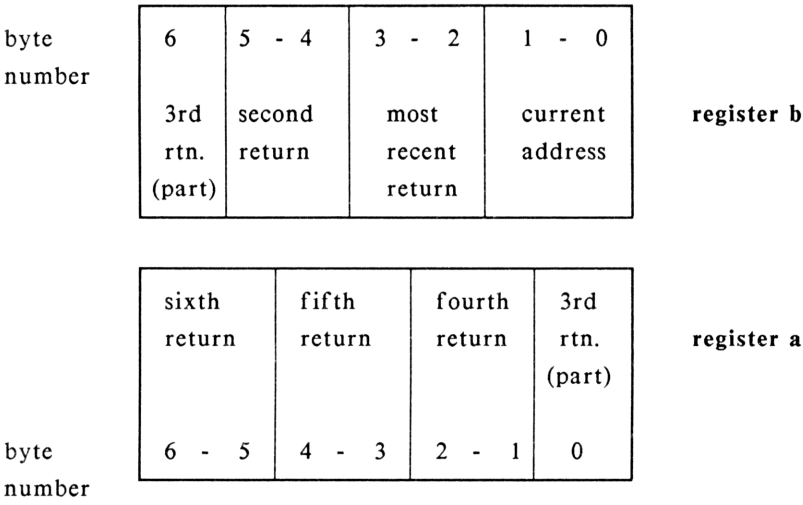


Figure 14.5 Registers a and b

Registers a and b are both pushed two bytes to the left whenever you perform an XEQ. They are both pushed two bytes to the right whenever a RTN or END is executed and two null bytes are pushed in at the left of register a. When you have performed as many returns (RTN or END) as XEQs, then the first return address contains two nulls; the next RTN or END has nowhere to return to, so it stops the program. An END also clears the return stack and sets the program pointer to the top of the program containing the END. A RTN does not clear the return stack, so if you have put something in register a and the HP-41 finds a zero return address in b then a RTN will leave register a alone whereas an END will clear it before stopping the program.

Details of how addresses are stored in these registers will be given in the next chapter. For the present let us see what can be done with them. Use an assignment program (KAP from the previous section, or GASN from Chapter 11, or some other program) to assign RCL b and STO b to two keys. The prefix and postfix for RCL b are 144, 124; for STO b they are 145, 124. If you use KAP, remember to make two dummy key assignments and cancel them each time before you execute KAP. Now go to line 001 of any program; it

does not matter which program you are in. Go out of PRGM mode and press RCL b. Register X now contains the address of line 001. Go back into PRGM mode and SST a few steps, say to line 010. Go out of PRGM mode, press the STO b key and go back to PRGM mode. You will see the line number 010; this has not changed because register e is unaltered, but you will actually be positioned to line 001 of the program because you have replaced the previous contents of register b.

The instructions RCL b and STO b let you save an address and then go back to it at any time. One way of using them is to replace a LBL and GTO. Look at the two programs below:

```
01♦LBL "VREGS"
02 CLX
03♦LBL 50
04 VIEW IND X
05 PSE
06 ISG X
07 ADV
08 GTO 50
09 END       24 BYTES
```

```
01♦LBL "VREGS"
02 0
03 RCL b
04 VIEW IND Y
05 PSE
06 ISG Y
07 ADV
08 STO b
09 END       23 BYTES
```

Both programs display the contents of all your registers until they stop at a nonexistent register on line 04. The program on the left uses GTO 50 at line 08 to go back to line 03. The one on the right uses STO b at line 08 to go straight back to line 04. This makes it a little faster, and one byte shorter, than the program on the left. If the one on the left used a short-form label instead of LBL 50 then it would become one byte shorter than the program on the right, and a little faster than before but still slower than the one on the right. The method on the right is therefore worth using if you are short of labels in a long program or if you want to squeeze every last millisecond out of your programs. It also uses no labels so the flying goose sits motionlessly. This can be used to confuse other people or yourself!

A more important use of synthetic access to registers a and b is to increase the depth to which subroutines can call other subroutines. Normally a maximum of six returns can be pending, but if you save registers a and b then you can make another six calls, return from them and restore the original six returns in registers a and b. Consider the following example.

A program to integrate a mathematical function FUNC(X) uses a depth of four subroutines. The fourth level subroutine calls the program FUNC to calculate the mathematical function. This leaves only one subroutine level for use in the function calculation. If the function calculation routine FUNC needs to call more than one subroutine level then it has to be rewritten to use a set of GTO IND instructions in place of its subroutine calls. The integration program would look as follows.

```
01  *LBL "INTEG"
      .
      .
100   LBL 04            Fourth subroutine level
101   RCL IND 10        Obtain the value at which FUNC is to be calculated
102   XEQ "FUNC"        FUNC can only use one depth of subroutine call
103   STO IND 11        Store the calculated function value
104   RTN
      .
      .
200   END
```

The synthetic way would be to save registers a and b before executing FUNC. Then FUNC would be able to use as many as five subroutine levels itself without being rewritten. The new integration would look as follows.

```
01 *LBL "INTEG"

100 LBL 04          Fourth subroutine level
101 SF 09           Set flag 09 to mark the start of the routine
102 RCL a           Recall a and store it in M
103 STO M
104 RCL b           Recall b and store it in N
105 STO N
106 FC?C 09         Do not RTN the first time this step is executed,
107 RTN             but clear flag 09 so you will RTN the 2nd time
108 RCL IND 10
109 XEQ "FUNC"      Now execute FUNC as before, but with
110 STO IND 11      5 subroutine levels available to it
111 RCL M           When FUNC returns, restore the original a
112 STO a
113 RCL N           Now restore the original b - this takes you back
114 STO b           immediately to line 105, and the RTN there is
                    executed.  You do not need a RTN after STO b
210 END             because STO b takes you straight back to line 105.
```

Registers a and b contain non-normalised numbers so their values must be stored in status registers such M and N, not in numbered data registers. Your FUNC program should not disturb M, N, or flag 09.

In this way ten extra lines and one flag allow you to use a maximum of twelve nested subroutine calls instead of six. The PPC ROM provides two subroutines to extend the subroutine stack even further.

Register a is not used at all if subroutines are called to a depth of less than 3, and some people are tempted to use it as an extra storage register. This temptation should be avoided because a later change to the program can use more levels of subroutines or may shift the contents of register a to the right or to the left. A few extra returns might even cause part of the value put into register a to be used as a return address.

On the other hand register a can be used to shift a register two bytes to the right or left. You would normally do this by using the ALPHA register, but if that is required for something else then use the subroutines SL2 and SR2 to shift the contents of X left or right by two bytes. In each case the program uses XEQ and RTN to shift the value after it has been put in register a. SL2 always restores the original value of a and leave b unchanged. SR2 restores register a unchanged unless it is executed with five returns already pending: in that case it loses the first (earliest XEQ) return. SR2 pushes two null bytes onto the left of the original value in X. SL2 pushes two random bytes from register b onto the right of the original value in X. These will only be nulls if SL2 is executed as a first level subroutine.

```
01 *LBL "SL2"          01 *LBL "SR2"
02  X<>a               02  XEQ 01
03  XEQ 01             03  RCL a
04  STO a              04  X<>Y
05  RDN                05  XEQ 01
06  GTO 02             06  RDN
07  LBL 01             07  GTO 02
08  RCL a              08  LBL 01
09  X<>Y               09  X<>a
10  LBL 02             10  LBL 02
11  END                11  END
        25 BYTES               26 BYTES
```

These two routines are entirely self-contained and do not change any of the registers Y, Z, T, L, M, N, O, P. The only problem will occur if a timer alarm interrupts them; this affects many SP programs and a way round it was described at the end of Section 9.4.

This brings us to the end of a long chapter. Have a rest or do some of the exercises before going on to the next one.

Exercises

Exercise 14.A Try the second half of exercise 4.D again, but use SP now. Hint: the routine will be easier to enter if you make key assignments of the functions STO O, RCL O, STO N and RCL N.

Exercise 14.B Write a program which sets flag 45, then executes a number entry step (such as 1.2345) and stops. Press a number key and you will find that you are adding digits to the number which the program put into X. This is one of the flag operations described in Wickes' book; it simplifies the entry of numbers whose first digits are known.

Exercise 14.C Assign STO b to a key, then execute CLA, 192, XEQ "BAB", RCL M, STO b. Turn on PRGM mode and you will be in the key assignment area (as address 192 is at the bottom of the KAR area) and will be able to study the KARs. (This assumes you have made at least one key assignment.) You can examine or alter assignments now, but read Chapter 15 before being too adventurous. An insertion where there is no space gives MEMORY LOST. STO b can be used to position you anywhere in memory.

CHAPTER 15 - USING SYNTHETIC PROGRAMMING

15.1 When should Synthetic Programming be used?

The examples of Synthetic Programming in the previous chapter let you do things that would otherwise be difficult or impossible. Are they worth the extra effort involved in Synthetic Programming? Can you do without extra tones, additional characters or more than six nested subroutines? When should you go to the trouble of making a synthetic key assignment or writing a program containing synthetic instructions?

Similar questions can be asked about ordinary calculator functions. Addition is necessary but the COS function is not really necessary on a calculator with the functions SIN and SQRT. Nevertheless COS makes life easier and is therefore used on scientific calculators and computers. Less frequently used functions such as the Gamma function are not provided on the HP-41 but can be added in the form of programs. If a problem requires a Gamma function you write a program to provide it, or else you find an alternative solution which does not use it.

Synthetic instructions are rather like the Gamma function; either you create them or you find a way to do without them. If you want an HP-41 program to display a result with quotes around it then you use Synthetic Programming or you make do without the quotes (or you buy an extra module to provide quotes). If an HP-41 program is to be operated by touch alone then you need more than ten tones to tell which key has been pushed - either you use synthetic tones or you make do with combinations of tones. If you want a single key assignment that replaces half a dozen separate key-pushes then either you make a synthetic assignment, or you write a special program, or you carry on pushing six keys each time. It is up to you in each case to balance the extra trouble of creating the required synthetic instructions against the value of having those functions. Of course if you have a ZENROM then this is no extra trouble, and in any case you will speed up with practice and use SP as quickly as normal functions.

In most cases Synthetic Programming extends the features of the HP-41; additional stack registers, better control of the Auto Start Module, and faster execution of programs. That is why SP is described in this book; it lets you "Extend your HP-41". On the other hand a few synthetic instructions let you do things that are otherwise completely impossible on an ordinary HP-41. In particular they let you create additional synthetic instructions and they allow you to study how the HP-41 works. This chapter will show how additional synthetic instructions can be made, it will explain how some HP-41 operations are carried out, and it will help you decide when SP is worth using in your work.

15.2 More key assignments.

A selection of synthetic key assignments has already been described in Chapters 10 and 14, but it is worth going through the whole Byte Table systematically. If you want to test any of the assignments described here you can now make them using the programs described earlier; of course you can use any other assignment programs. The bytes will be given here in hexadecimal, remember to use the Byte Table to convert them into decimal values for the assignment program. All assignments take up two bytes, but if the first byte is from row 0 of the Byte Table then it is ignored and only the second byte is used. These are called one-byte assignments. The HP-41 functions ASN and PASN always use 04 as the dummy first byte, but it makes no difference which byte is used; the HP-41 assumes a one-byte assignment if the leftmost nybble is a zero. Let us examine the one-byte assignments first; the best way to study an assignment is to press the key in PRGM mode where the result will be recorded for further study.

Assignments from row 0 were described in Section 8.5. Five of these are synthetic assignments, the most useful is byte 0C. This lets you reassign the toggle key functions in case a toggle key gets damaged. The function depends on which keyboard row is used; assignments to rows 1 or 5 produce ALPHA, rows 2 or 6 give PRGM, the others provide USER. The ALPHA assignment only works into ALPHA mode, use the ON key to get out of ALPHA

mode, or set the program line to an AOFF step and press SST. The USER assignment also works in only one direction but you can use SF 27 to set USER mode. An amusing anomaly; if you assign byte 0C to a shifted key in rows 3, 4, 7, or 8 and press SHIFT followed by the key then USER mode will be cancelled but the SHIFT mode will not be cancelled.

Byte assignments from row 1 are sufficiently interesting and complicated that they are described separately in Section 15.6. Assignments from rows 2 and 3 result in some very interesting function previews and several of them prompt for a parameter. When the key is released, and a parameter is provided if necessary, nearly all of these assignments turn into the expected RCL and STO functions. A few assignments do behave unusually. Byte 30 displays as ASCI; this is not an ASCII character function but merely the name of the SCI function with an extra A. If you fill in the prompt with an IND or Alpha parameter it turns into an XEQ instruction. The four bytes 20, 33, 37 and 3C are all immediate execution functions (like BST and SST), their exact action depends on the modules plugged into ports 2 and 4, they may have no effect at all or they may set PRIVATE mode (do a CAT 1 to get out of this). I gave a more detailed description of byte 33 in PPCCJ V11N6P13b.

Rows 4 to 9 produce ordinary non-synthetic one-byte assignments. In row A, bytes A8 to AD behave normally. AE acts as a GTO IND 00. AF prompts for an input but does nothing in run mode, because AF is a spare do-nothing byte. In PRGM mode an input of nn produces GTO IND nn and an input of IND nn produces XEQ IND nn (as if byte AF had taken on the role of byte AE). The bytes A0 to A7 provide interesting and, in some cases, very useful results. For example byte A0 is the first byte of all the two-byte XROM functions from XROM 00,00 to XROM 03,63; the second byte decides which XROM function is selected (see Section 8.5 for more details). If you assign 04,A0 to a key you get a one-byte function which produces the prefix A0; it displays as ↑__ and accepts a postfix. The parameters 00 to 99 produce the functions XROM 00,00 to XROM 01,35, and the postfixes IND 00 to IND 99 produce the functions XROM 02,00 to XROM 03,35. You can also use the postfixes IND T to IND L to obtain XROM 03,48 to XROM 03,52.

Consider how useful this assignment is. XROM 01 produces the Mathematics Module functions, XROM 02 produces the Statistics Module functions, and XROM 03 produces the Surveying Module functions. The one assignment 04,A0 allows you to execute all of these except the ten Mathematics Module functions 01,36 to 01,45. (If you have a ZENROM then you can produce postfixes greater than 99 or IND 99, so you can create all the Mathematics Module functions.) Even if you do not have one of these modules plugged in you can put its functions into a program by using the key. XROM 00 functions are less useful unless someone takes up the idea of using XROM 00,nn instructions in a program to provide an argument nn for a preceding XROM function (see Section 8.5). XROM 00,00 gives NONEXISTENT on an HP-41CX, otherwise it produces an immediate crash unless you have a Time Module plugged in. You can work out the correspondence between programs in a module and XROM numbers by counting through a CAT 2 list of functions (the module name is XROM nn,00), or you can get a Synthetic Quick Reference Guide and use the lists in it.

The assignment 04,A1 works in the same way. It displays as ␣__ and accepts the parameters 00 to 99, T, Z, Y, X and L, as well as the corresponding indirect parameters. This gives you access to XROM 04 (the whole of the Financial Decisions Module), XROM 05 (most of the Standard Applications, or all of ZENROM or most of the lower 4k of PANAME), XROM 06 (all of Circuit Analysis or PANAME upper 4k) and XROM 07 (all of Structural Analysis A). If you have a ZENROM then you can create all the XROM functions, otherwise XROM 05,36 to XROM 05,47 and XROM 05,32 to XROM 05,63 will be unavailable as well as the same XROM 07 numbers. Never use ␣65 or ␣66 if you have a ZENROM - can you see why not? The assignments of A2 to A6 are unhelpful; either they give no prompt or they only take an Alpha parameter and then ignore it. Incidentally, if you have a Standard Application Module (XROM 05) present, synthetic STO key assignments will preview as module functions rather than the usual XROM 05,xx. However the correct synthetic STO will still be executed.

The most useful and famous of these assignments is A7. This was discovered

by Robert W. Edelen just six months after the first HP-41Cs became available, and while Synthetic Programming was still in its infancy. The assignment produces the amazing display eG0BEEP__ and accepts the parameters 00 to 99. These cover all the printer functions and all the other HP-IL module functions too. If you have a list of function numbers then this single key assignment lets you execute all 64 IL and printer functions, or include them in a program, without spelling the names. (You can also create the module "titles" such as "-PRINTER 2E".) eG0BEEP lets you write programs for a printer even if you do not have one attached to your HP-41, without spelling out the function names. The "non-programmable" functions PRP and LIST can be included in a program, but this is not much use; a programmable PRP will be described in Chapter 16. Out of PRGM mode you can GTO "name" (or use GTO."name" in PRGM mode) then press eG0BEEP 77 and the program called "name" will be printed.

If you provide an Alpha parameter for eG0BEEP it produces a label. If you have a ZENROM then you can provide eG0BEEP with parameters from 128 to 164 which will give you all the Card Reader functions as well. (This was described in my article in DATAFILE V3 N6 pp16-17.) A list of eG0BEEP results is given in the Synthetic Quick Reference Guide. This makes the important point that eG0BEEP 003 wipes out HP-IL mass media - never press 003 (nor the SQRT button) after eG0BEEP.

The bytes in row B are rather like the bytes in rows 2 and 3. They produce strange displays and some of them prompt for parameters, but they mostly turn into the default GTO instructions. Byte B0 corresponds to a spare byte; it prompts for an Alpha input, then turns into RCL 00. Byte B1 becomes STO 00. Byte B2 prompts with a flying goose which is pretty, but not much use in PRGM mode. Byte BA becomes XEQ IND if the prompt is filled with IND nn or IND ST n. Bytes BB and BD can take Alpha parameters which turn into labels. Some of these bytes behave like compiled GTOs if they are pressed in run mode, and they can take you past the .END. into the free area, so be careful. In particular, byte B2 is often used as a selectable "byte jumper" in run mode. If you use the value 16b+r to fill in the prompt, you will jump r registers and b bytes forward in program memory.

Use IND to jump backwards. To get back into the program area, execute CAT 1.

Row C mostly produces ENDs, which behave like RTN when pressed in run mode. Byte C0 is the normal END assignment. The others produce some more odd displays. Byte C4 can take Alpha prompts and become a label. Byte CB becomes GTO. if . is pressed, GTO IND if SHIFT if pressed, and GTO "Alpha" if Alpha is pressed. Byte CD can be used to produce labels with non-keyable characters, it will be described in Section 15.6. Bytes CE and CF are ordinary non-synthetic assignments.

Row D mostly produces GTO 00. Byte D0 is the normal GTO assignment, the others again produce odd displays. Bytes D1 and DB can take stack register names as parameters and produce instructions such as GTO X which will go to the synthetic LBL X (bytes D5 and DE do the same given an IND ST parameter). In run mode bytes D1, DB and DE take IND nn parameters and go to the corresponding synthetic indirect labels if nn is greater than 14. For example you can assign 04,DB (decimal 004,219) to a key, press that key in run mode, and fill the prompt with IND 83. This will take you to a synthetic LBL IND 83, not to the label whose number is in register 83. You can synthetically create LBL IND 83 by using the bytes CF,D3. If you are writing a long program and editing it a lot then it can be useful to put a few LBL INDs in at important places. You can then GTO these labels in run mode to get to these places instead of having to check line numbers and using the GTO. instruction. Since these labels cannot be reached by any instruction in a running program you can delete them once the program is working, without danger of removing an important label. The discovery of these synthetic GTOs was first described by G. McCurdy in PPCCJ V9N7P9-10.

Another use of the assignments D1, D5, DB and DE is described in the same article. Filling in the prompt to any of these with IND 00 to IND 14 produces three-byte GTO 00 to GTO 14 instructions. Normally GTO 00 to GTO 14 are two-byte long instructions from row B. These save one byte but can only store a compiled distance of 112 bytes or less within a program. Using three-byte GTOs with the labels 00 to 14 lets you save a byte because

these are short-form labels, yet you can have compiled GTOs which will go directly to these labels from any distance. In other words you can use a long-form GTO with a short-form label.

Row E assignments resemble row D; E0 is the normal XEQ assignment and most of the others produce XEQ 00. E2, E4 and EF prompt for a number and produce the corresponding XEQ. In run mode they will XEQ the synthetic indirect labels just as bytes D1, DB and DE will GTO these labels. Bytes E5, E7, E9 and EB prompt for a three-digit label number which can be turned into four digits if you press the EEX key. Filling in the prompt will produce a number modulo 128, so that you can make any label with a postfix in the first half of the Byte Table. You may consider this a useful device for doing arithmetic modulo 128; fill the prompt with 1283 and you will see XEQ 03, which tells you that the remainder of 1283 divided by 128 is 3.

Oddly enough most of the bytes from row F produce GTO 00 just like the bytes from row D. Bytes F8, F9, FC, FE and FF behave differently. They prompt for a two-digit number; if you provide a number below 15 they turn into the corresponding short-form GTO . You can also fill in the prompt with a number over 14, a stack register name, an indirect number or an indirect stack register. In all these cases a text string is produced, with a null character first, and a second character corresponding to the parameter you have given. The total length of the character string is given by the second nybble of the assignment. For example assign 004,248 to a key, press the key in PRGM mode and give the parameter 38. You will see a string eight characters long because 8 is the second nybble of F8 (decimal 248). The second character of this string will be & which is character number 38. All five of these assignments act as Byte Grabbers, but they are not as clean as the F7 Byte Grabber because they grab more than one byte. FF grabs nine bytes! Be careful not to grab the .END. with any of these assignments.

You can use these assignments instead of BAB or the Extended Functions to produce non-keyable characters. If you went through the example above you will have produced a program character string containing an ampersand. SST

this string in run mode to put the character into the Alpha register. There will now be seven characters after &. How can the & be isolated? Normally you could press SHIFT K (APPEND) and delete the unwanted characters, but this does not work if there are any nulls in ALPHA.

An alternative is to use the synthetic key assignment ASTO M. Make this assignment (154,117) and press the key in run mode. Go into ALPHA mode and see that the & has moved one place to the right. What has happened? The first six characters of ALPHA have been put at the right of M, with a text identifier (10 hexadecimal) in front of them. This has moved everything left one byte. Go out of ALPHA mode and press the key five more times. The & is at the right, preceded by six hexadecimal 10 bytes which display as boxed stars. Execute ASHF to delete these six rightmost characters and you have & on its own in ALPHA. You can ASTO it in a register for later use. This use of ASTO M to shift the Alpha register to the right by one character at a time was described by W. Cheeseman in PPCCJ V9N4P13. A list of one-byte key assignments was given in PPCCJ V9N4P67-68 by M. Katz. You will notice that I have made a lot of references to user club journals on the last few pages. User clubs are an ideal forum for the study of subjects such as synthetic key assignments; the journals allow people to exchange discoveries and suggest uses for them.

Now let us turn to two-byte assignments and go through the Byte Table again, looking at combinations of prefixes and postfixes. Prefixes from row 0 are ignored and provide the one-byte assignments already discussed. Prefixes from all other rows produce two-byte assignments which display as XROM mm,nn or as names of XROM functions. An assignment displays as the name of a function on a plugged-in device if the first nybble is not a zero and the remaining three nybbles correspond to a function on a plugged-in device. Take the Time Module function TIME as an example: its hexadecimal code is A6,9C. The synthetic assignments 96,9C (ISG IND 28) and D6,9C (GTO 28) would also be displayed as TIME if they were pressed on an HP-41 with a Time Module. If the function displayed prompts for a parameter then any corresponding two-byte assignment will also prompt for a parameter, but will ignore it. Otherwise no prompts are produced.

Two-byte assignments with a prefix from rows 1 to 8 act as if only the prefix existed. They ignore the second byte, do not prompt, and execute the first byte or store it in a program, ignoring the second byte entirely. Two-byte assignments from row 1 will be mentioned again in Section 15.6. They are the Q-loaders.

Assignments with a prefix from row 9 provide genuine two-byte functions as was described in the previous chapter. The functions of most of these assignments are obvious; assigning ST/ IND 17 to a key can save you four keystrokes and is worth doing if you use this particular instruction often. Assignments affecting the status registers, especially b, c or d, can be dangerous; they should be made and used with care.

ASTO and ARCL have many uses when combined with the status registers; they let you put synthetic text strings into ALPHA then transfer them directly to status registers. For example ASTO d lets you set your favourite display format, angle mode, and time display mode all at once. ARCL and ASTO to the ALPHA registers give you additional control of ALPHA and byte shifting, as was shown above using ASTO M. To shift ALPHA left by n bytes store the number n-2 in register qq and use PI, FIX IND qq, ARCL X. This lets you calculate n in a program instead of using an Append instruction with a predetermined number of bytes. FIX n, ARCL M can also be used to shift the ALPHA register if you know what the contents of M are.

Synthetic TONEs provide 16 frequencies and a range of tone lengths from .025 to 4.8 seconds. It is less obvious that synthetic FIX, SCI and ENG functions can be useful. FIX 10 lets you see the whole 10-digit mantissa of numbers greater than 10↑10, unfortunately exponents that are 1, 2, 3 or 4 less than an exact multiple of 14 may display strange characters at the right. A better way to show all the digits is to use FIX 9, ARCL X or to use the program APX from the next chapter. If your favourite display mode is FIX n then try assigning SCI 112+n to a key and pressing that key four times in USER and run mode. You will cycle through the possible angle modes in the following combinations:

	Display mode	Angle mode
First:	ENG n+1	RAD
Second:	FIX n	GRAD
Third:	FIX n	RAD
Fourth:	FIX n	DEG

This lets you cycle through the three possible angular modes while keeping your preferred FIX n display mode. For a more complete explanation of the behaviour of synthetic display setting instructions, see PPC CJ V9N1P10d by Randy Cooper and V11N8P27-28 by Mark Gessner.

Two-byte assignments from the first half of row A are the real XROM functions. The next six can provide synthetic flag instructions. Only indirect synthetic commands are useful; you can create the instructions SF X and FC? I, but these do nothing as there are only 56 flags. Byte AE followed by a postfix provides a GTO IND or XEQ IND assignment. Byte AF followed by the same byte when pressed in PRGM mode produces an instruction that looks like the AE instruction but does nothing.

Byte B0 displays as GTO 15 and treats the next byte as its postfix, but it actually does nothing since it too is a "spare" byte. The remaining bytes from row B act as the two-byte GTOs with the second byte providing a compiled jump distance, unless it is a zero. By selecting the second byte and pressing the assigned key in run mode you can jump forward or back in program memory by any distance up to 112 bytes. The structure of the second byte will be described in Section 15.4. Byte B1 is anomalous; if it is followed by a postfix smaller than 16 (decimal) then in PRGM mode it becomes a one-byte STO nn with nn given by the postfix. In run mode it acts as a compiled GTO.

Bytes C0 to CC produce an END in PRGM mode. The second byte is changed into the byte required by an END in the position given. The third byte is always 0F indicating an unpacked END in a program which should be

decompiled. Byte CD produces a global label and will be described in Section 15.6.

The two-byte assignment C7 85, discovered by Roger Hill, has a special use. When a card reader is present it previews as VER, which is non-programmable. The assignment therefore performs a RTN as if you had pressed it in run mode, even if you are in PRGM mode. This is a handy way to GTO.000 in PRGM mode.

Byte CF makes local labels and byte CE is a displaced person - it would have been more at home in rows 9 or A. Byte CE can be followed by any postfix to give X<> any register or any IND register. The postfix is preceded by a space so you can see that X<> Y is not the same as X<>Y. Byte CF can be followed by any byte from the first half of the Byte Table to produce the numeric labels 00 to 101 and the local labels A to J, T to append, and a to e. If the postfix byte is 00 to 14 then the corresponding one-byte label is put into a program. Postfixes from the second half of the Byte Table produce synthetic labels such as LBL IND 01 or LBL IND d. The postfix FF behaves differently; pressing a key with CF,FF assigned to it in PRGM mode puts a completely null register into the program if the program pointer is at an END, otherwise it executes a SST. In run mode all these assignments act like a LBL, so they do nothing.

Any byte from row D followed by any postfix produces a GTO instruction. In PRGM mode the postfix gives the GTO label, for example D0,10 produces GTO 16. Assignments with postfixes 00 to 0E turn into two-byte GTOs, all other postfixes except FF give three-byte GTOs, with a null byte between the Dn and the postfix. The FF postfix turns into a two-byte B0 instruction which looks like GTO 15 but actually does nothing. Other postfixes from the second half of the Byte Table do not produce GTO IND instructions, but rather direct GTO instructions. The first bit of the second byte indicates the GTO direction; this will be explained in Section 15.4. If pressed in run mode, the assignment will search for a label; this will be a LBL IND nn if the postfix is greater than 7F.

There are no short-form XEQ instructions so all Em,nn assignments turn into 3-byte XEQs in PRGM mode. The first byte is Em, the second is 00, the third is nn. In PRGM mode the first bit of nn is again treated as a jump direction and the remaining bits give a label number from 00 to 127. In run mode, the XEQ will execute a LBL IND if the postfix is from the second half of the Byte Table.

Fm,nn assignments behave like Dm,nn, they put the three bytes Fm, 00, nn into a program. If nn is 00 to 0E they produce two-byte GTOs, and if nn is FF they produce the do-nothing GTO 15. F7,3F provides the standard Byte Grabber but Fn assignments can be used for other purposes, particularly the study of byte patterns in the HP-41. This will be described next.

15.3 Byte Grabbers, Byte Jumpers and program analysis.

The F7 Byte Grabber works by forcing the HP-41 to make a space seven bytes long, but actually taking eight bytes. Since any Fm,nn assignment turns into a three-byte instruction in program memory, it will open up seven bytes unless three or more empty bytes are already available. If an F7 Grabber is used and one null is available it will grab that null. A set of bytes such as:

```
                COS, null,  174
        bytes:  5A,  00,  11,17,14
```

will become the following if you press the Byte Grabber in PRGM mode at the position of the COS:

```
                COS,        ┬─?─────,          174
        bytes:  5A,   F7,00,3F,00,00,00,00,00,  11,17,14
```

If there are three or more bytes already empty then the Grabber will fill them and can grab more than one following byte. The set of bytes

```
        COS, null, null, null, null, null, null,        174
        5A,  00,   00,   00,   00,   00,   00,          11,17,14
```

would become

```
        COS,        ┬─?───▒▒,          4
        5A,  F7,00,3F,00,00,00,11,17,    14
```

This may not be what you want and is the reason why you should normally PACK before using the Byte Grabber.

You may need to grab a byte without moving everything that lies below the byte. If you use the Byte Grabber below the .END. then everything including register c will be moved down by one register and you will get MEMORY LOST. Use the F3 Byte Grabber in these cases; make sure there are exactly three nulls before the byte to be grabbed, then press a key which has F3,nn assigned to it. This will fill the three nulls and grab the next byte without moving anything. If there are more than three nulls ahead of the byte to grab then fill in the unnecessary nulls with bytes that you can delete later. The method can be taken further; use an F4 Grabber or an F8 Grabber to remove two bytes for instance.

Using F0,nn or F1,nn lets you put the byte nn directly into a program (unless nn is 00 to 0E or is FF). To create synthetic X<> instructions you could assign F1,CE to a key. Then you would create X<>d as follows:

1) Put +, +, +, AVIEW into a program and PACK
2) BST once and delete the three pluses, using the backarrow key
3) Press the F1,CE key
4) Backarrow the text string and SST to see X<> d.

The Fm assignments behave quite differently when the keys are pressed in run mode instead of PRGM mode. The F nybble is recognised as the beginning of a text string, and the HP-41 looks at the second nybble of the previous instruction in program memory to see how long the text string is. If a running program has just stopped at a RTN, or if you have just single-stepped past a RTN, then the previous instruction is RTN, or byte 85. Should you press an Fm,nn key next, the HP-41 quite sensibly assumes you have just executed a text instruction, and equally sensibly assumes that the text string length will be given by the second nybble of the instruction it has just gone past. The HP-41 therefore checks the second nybble of the latest program instruction (a 5 in the case of RTN) and

copies that many bytes from the program into the ALPHA register, then puts the program pointer at the next byte. Try the following:

1) Put the steps RTN, +, -, *, GTO 93 into a program and PACK.
2) Leave the program positioned at the + and go out of PRGM mode.
3) Press the Byte Grabber key (obviously you must be in USER mode).
4) Press ALPHA and see the text string @AB▩¯. You have copied the next five program bytes into the ALPHA register.
5) Press PRGM and see ACOS. You have stepped over five bytes and the sixth one, which was the 93 of GTO 93, now appears on its own as ACOS.

This operation is called Byte Jumping. It was discovered and analyzed by Bill Wickes in the relatively early days of synthetic programming. You have jumped over five bytes into the middle of an instruction and you can edit that instruction by inserting or deleting bytes. Byte Jumping is not quite as easy as Byte Grabbing because it requires a preceding byte to determine the jump length, but it was discovered before Byte Grabbing and made the creation of synthetic instructions much easier than it had been before.

Step 4 above has shown you that Byte Jumping also copies program bytes directly into Alpha where you can study them without altering the original bytes. The last character of the text string is a null, showing that the middle byte of the uncompiled GTO 93 is a null. In fact the middle byte of any uncompiled three-byte GTO is a null. This is why synthetic assignments of Dm,nn produce the bytes Dm,00,nn in a program. Actually all two-byte assignments behave in this way if the leftmost two bits are 11, unless the assignment can be turned into a two-byte form.

The Byte Jumper can be used to analyse any collection of bytes in an HP-41. Just position the program pointer before the bytes you wish to study, and immediately after a byte whose second nybble is large enough to let you copy all you want. Press the Byte Jumper in run mode and analyse the contents of the Alpha register. If the Fm,nn byte has the same values of m,nn as an XROM function Am,nn plugged into your HP-41, and if this is a

non-programmable function then you can even use the Byte Jumper in PRGM mode. With a Card Reader plugged in you have the non-programmable function VER (A7,85). The text assignment F7,85 will therefore execute as a Byte Jumper even in PRGM mode. To analyse data or the status registers you can use STO b to move the program pointer where you wish, then Byte Jump to copy into the ALPHA register. You can even analyse programs in ROM modules this way. If the first byte you copy is 7F it will not be copied but the bytes after it will be appended to the ALPHA register, since 7F is the Append byte.

Once you have copied a set of bytes to the ALPHA register you need to analyse them. The program RAB (Read a Byte) will let you do this. It takes a string of bytes from ALPHA and converts them into decimal numbers one at a time; in effect it does the opposite to BAB. An Fm Byte Jumper cannot copy more than 15 bytes into ALPHA, so RAB starts at the fifteenth byte from the right end of ALPHA and ignores any bytes to the left of that.

When you XEQ "RAB" it first displays the byte number it is examining. This is number 15 to begin with. The number is shifted across the display from left to right, then the fifteenth byte value is displayed and the program stops to let you write down this number. If byte 15 is a zero then RAB assumes it is a leading null and goes on to check byte 14 without stopping. All leading nulls are treated in this way until a non-null byte is found or all 15 bytes have been checked.

At the moment when RAB stops, the leftmost byte has been removed from the ALPHA register, all the other bytes have been shifted one place to the left, and the byte value is in register X. Register Y contains the original value of X, and register T contains the position of the byte in the ALPHA register, counting from the right and treating the rightmost character as number 1. If you want to carry on analysing the contents of Alpha, press R/S and the number of the next byte will be scrolled across the display, then the byte value will be displayed. You can repeat this until byte 1 has been displayed, then press R/S to see the display "DONE"; at this stage Alpha will be cleared and the original X will be in X again.

Alternatively you can quit whenever the program has stopped and is displaying a byte value. The display is left set in FIX 0, and flag 09 is cleared unless the string was all nulls.

The program will work if there are more than 15 bytes in the ALPHA register, but it will ignore characters to the left of the fifteenth one. Just as BAB is similar to the Extended function XTOA, so RAB is similar to ATOX; RAB is much slower but it allows for null bytes within a text string whereas ATOX ignores them.

```
01♦LBL "RAB"
02 15
03 RDN
04 FIX 0
05 SF 09
06♦LBL 01
07 VIEW T
08 R↑
09 X<> I
10 XEQ "SL2"
11 X<> d
12 39.031
13 ENTER↑
14 CLX
15 SIGN
16♦LBL 02
17 FS? IND Y
18 ST+ L
19 ST+ X
20 DSE Y
21 GTO 02
22 RDN
23 RDN
24 X<> d
25 X<> L
26 .
27 X<> I
28 RDN
29 "⊢-"
30 CLD
31 X≠0?
32 CF 09
33 FC? 09
34 STOP
35 X<>Y
36 DSE T
37 GTO 01
38 "DONE"
39 AVIEW
40 CLA
41 END
                82 BYTES
```

Lines 02 to 05 set up the byte counter, set flag 09 to mark the start of the program and set FIX 0 to display bytes as integers. The main loop starting at LBL 01 analyses one byte at a time. The byte number is shown and exchanged with register O which contains the byte to analyse. This byte is now at the right end of register X and at line 10 the routine SL2 (see Section 14.11) is used to shift this two bytes to the left; SL2 avoids the need to append bytes to the Alpha register. The byte to analyse is now in flag positions 32 to 39, where it cannot be affected by problems with flags 53 or 30. The loop counter 39.031 is put into register Y by lines 12 and 13. Lines 14 and 15 store a zero in L and a 1 in X (this is much faster than 1, STO L, RDN, 0). The loop at LBL 02 adds X to register L if the corresponding flag is set, then it doubles X without altering L. This lets you add 1 for the lowest bit, 2 for the next bit, 4 for the next, and so on up to 128.

The use of SL2 at line 10 produces a flag register with flag 50 clear unless RAB is called from a program that is itself called from a program in RAM. When this flag set is stored in d at line 11 the HP-41 finds that flag 50, the message flag is clear. This normally means that there is no message in the display, only a goose, so the program shifts the display one place to the right each time it comes across LBL 02. In fact flag 50 has been cleared synthetically, and there is a message in the display, so this message (the byte number) steps across the display. This helps to distinguish it from the byte value which is displayed later.

When the loop finishes, lines 22 to 28 restore register d, put the byte number back in register T, put the byte value in register X and the original X value in register Y. Line 26 is a decimal point which puts a zero in register X but is faster than a zero. This is exchanged with register O to clear O and recover the byte number. Line 24 appends a minus to Alpha, shifting the next character to analyse into register O. The byte value in X is ready to be viewed, but if it is a zero and flag 09 is set then it is a leading null and need not be displayed, so line 34 is skipped. If the value is not a null then flag 09 is cleared at line 32 to show that any later zero bytes are not leading nulls. Line 30 clears the display so that the byte value will be seen if the program stops at line 34.

If the user pushes R/S after the program has stopped then line 35 recovers the original value of X. Lines 36 and 37 continue the main loop if necessary. If the value in T is zero then all the bytes have been analysed, so the program displays a message and stops.

As is the case with the other synthetic programs, RAB does not use any numbered data registers. If you want to analyse all 24 characters in Alpha then you will have to replace lines 09 and 27 with X<>P. The leftmost character byte is already in the third position in register P so XEQ "SL2" will be unnecessary, but the byte count at line 02 has to be increased to 24. The numeric entry at line 12 alters register d, so you will have to replace it with a RCL nn, and put the following three steps after line 01:

39.031, STO nn, RDN

RAB is not as sophisticated as some other byte analysis programs but it provides several examples of synthetic programming techniques, and can be changed to analyse all 24 characters as shown above. Try using it by putting a text string "ABCD" in Alpha and executing RAB. Bytes 15 to 5 are nulls, then byte 4 will be displayed as the number 65. If you press R/S you will see 66, then 67, then 68 and finally DONE.

15.4 Addresses and multi-byte instructions.

When Hewlett-Packard announced the HP-41 they provided PPC with three articles about the HP-41 internal operations. These were printed in PPCCJ (V6N4P11, V6N5P20, V6N6P19) and described the Byte Table, the structure of multi-byte instructions, and the layout of memory and status registers. With the aid of techniques such as Byte Jumpers and byte analysis programs, further details were studied. An understanding of program addresses and of multi-byte instructions that use these addresses allows synthetic programmers to move around HP-41 memory without the normal limitations.

Chapter 8 described the division of memory into registers and bytes; addresses are stored as byte and register numbers. The status register b holds the address of the last byte of the last non-null program step before the next one to be executed (in run mode) or displayed (in PRGM mode). The address is contained in the rightmost four nybbles of b as a byte number n (numbered from 0 to 6 as in Figure 14.1) followed by an absolute register number rrr.

The structure of this address in nybbles is: **nrrr**

If this is split up into bits then a RAM address becomes

0nnn 000r rrrr rrrr

The three bits nnn are sufficient to give any byte number from 0 to 6, and the nine bits marked r can describe any RAM register up to 1FF, which is

the highest address in main memory. One exception: with SIZE 000, at line 01 of the first program in CAT 1, register b contains the address 0200 (hexadecimal).

When a program is running, the length of each instruction is checked while the instruction is being executed and this length is subtracted from the address in register b. This way register b automatically points to the next instruction. Any GTO or XEQ must provide the information to change register b so that it points to the target label, not the next instruction. We shall soon see how this is done.

When a subroutine is called by an XEQ then the current RAM address (the address of the last byte of the XEQ instruction) is pushed two bytes to the left and is rearranged as below

0000 nnnr rrrr rrrr

This means that the first nybble of a RAM return address is always a zero.

ROM addresses are stored in a different form:

pppp bbbb bbbb bbbb

The first four bits give the ROM page. ROM page numbers were explained in Figure 8.6; the HP-41 system resides in pages 0, 1, 2, and system extensions are in pages 3 to 7. Pages 8 to F address ordinary plug-in modules. The remaining twelve bits of the ROM address give a byte number between 000 and FFF within the page. Programs in ROM run from lower bytes to higher bytes so the byte number is incremented as a program executes. When a subroutine is called by a ROM, the return ROM address is pushed to the left without any change.

On execution of a RTN or END the HP-41 checks the first return address. If this is all zeroes then the program stops. If the first nybble is a zero then the return could only be to page 0 of ROM, or to a RAM address. Since

page 0 is part of the operating system it never calls subroutines, therefore a zero means the return has to be to RAM and the address is rearranged before a return is made to the RAM program. If the first nybble is not zero then the return has to be to a ROM module.

You can see that the current program address looks much the same for programs in ROM and RAM. For example, address 6108 could be byte 6 of register 108 in RAM, or it could be byte 108 of ROM page 6 (LBL"PRAXIS" in the printer module). The HP-41 has an internal flag (not one in register d) to tell if the current program is in ROM or in RAM. You can therefore use STO b to move from place to place in ROM, or from place to place in RAM, but not to go from one to the other. More will be said about this in the next section when the routine XRO is described.

The first time an XEQ or GTO local label is executed in a RAM program it calculates and stores the distance to the label. After this the XEQ or GTO can jump directly to the label; it is said to be **compiled**. In a two-byte GTO the compiled distance is stored as a direction bit d, a byte number nnn, and a register number rrrr

Two-byte GTO: 1011 mmmm dnnn rrrr

The four bits at the left identify this as a row B function. The next four bits, mmmm, identify the label number. Just as with the labels in row 0 the number stored is one more than the label number. A value of 1111 is interpreted as the number 15 but refers to label 14. Note that a value 0000 has a 1 subtracted and therefore looks like a GTO 15 instruction, but actually does nothing, as there is no corresponding one-byte label in row 0.

The second byte contains a null when the instruction is created. This tells the HP-41 that the distance to the label is not known; the GTO has yet to be compiled. The byte should also be set to zero whenever the program containing it is edited, since the distance to the label may have changed. One of the important techniques in synthetic programming is to

prevent decompilation, either to save time or to create synthetic jump distances. (There is more about this as well in the next section.)

When an uncompiled GTO is executed the jump distance byte is filled in, assuming the label has been found. The bit d is set to a 0 if the jump is forwards (towards the END) or a 1 if the jump is backwards. The four bits rrrr give the number of complete registers to be jumped (a maximum of 15), and the three bits bbb give the additional number of bytes. This allows a maximum jump distance of 112 bytes to be stored; if the distance is greater than this, the GTO is left uncompiled.

As soon as a running program comes to a two-byte GTO it changes the address in b by two bytes to point to the next instruction (this is where the label search begins if it is required). Then if the GTO is compiled a number of bytes has to be added to or subtracted from the address so that it will point to the label instead. The jump distance therefore includes all the bytes from the end of the GTO to the target label. Two examples of this were given in Section 8.5.

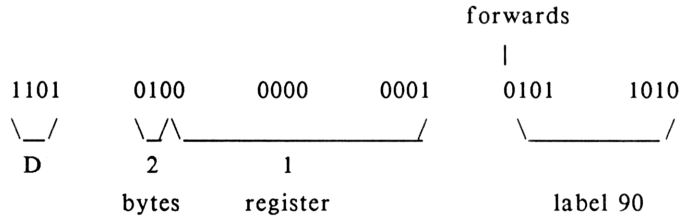
Three-byte GTOs have the bits arranged like this:

1101 nnnr rrrr rrrr dmmm mmmm

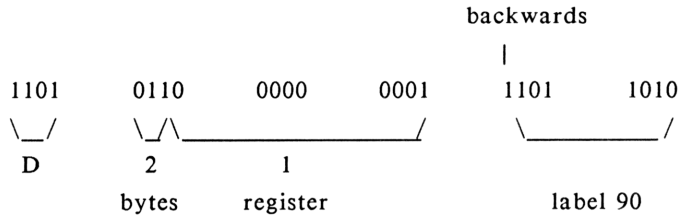
The nine bits used for the number of registers together with the three bits nnn used for the byte number allow a jump distance of up to 512 (decimal) registers. This allows any jump within main memory. It even allows synthetically compiled jumps between the status registers and program memory. The bit d gives the direction again and the remaining seven bits can hold any label number from the first half of the Byte Table. During execution of a three-byte GTO, the program pointer is advanced at first by only one byte, so a forward jump distance has to include the second and third bytes of the GTO. A reverse jump distance must include the first byte of the GTO and both bytes of the LBL. A pair of examples will help again:

steps	bytes to jump	steps	bytes to jump
X=0?		LBL 90	2
GTO 90	2	"X'="	4
"X'="	4	ARCL X	2
ARCL X	2	PROMPT	1
PROMPT	1	GTO 90	1
LBL 90	0		
	9 bytes		10 bytes

The example on the left displays a value unless it is zero, in which case it jumps forwards to label 90. The forward jump includes 2 bytes of the GTO, making a total of 1 register and 2 bytes. The compiled GTO label looks like this:



The example on the right displays any value, and repeats the display if the user accidentally presses R/S. The backward jump includes 1 byte of the GTO and both bytes of the label, making a total of 1 register and 3 bytes. The compiled GTO looks like this:



Note that the text string in these examples contains an apostrophe which has been included synthetically to display the value "X prime". This is a typical example of the use of synthetic text strings. One way of creating this string is to use the Q-loaders to be described in Section 15.6.

A compiled XEQ has exactly the same layout as a compiled 3-byte GTO, but it begins with the nybble E instead of D.

The global GTO and XEQ instructions contain just a name, not a compiled distance. The label is always found by a search up the CAT 1 chain. To speed up this search each element of the CAT 1 chain stores the distance to the previous element. As with three-byte GTOs, the distance is stored in three bits n for a byte count and nine bits r for a register count. The direction is known to be backwards, and the distance is calculated as for three-byte GTOs. Global labels and ENDs have the form:

END 1100 nnnr rrrr rrrr 0zzz apda

LBL 1100 nnnr rrrr rrrr 1111 nnnn kkkk kkkk tttt tttt...

The first nybble is always a C. In an END the third byte gives the type of END; zzz is 010 for the .END. and 000 for a normal END. The bit p is set if the program before the END should be packed, and the bit d is set if it should be decompiled. The two bits marked a can be 0 or 1; when you do a GTO.. the last byte of the .END. is set to 20 but when you begin to put in a program this becomes 2F.

Bits p and d are set every time any steps in a program are moved or deleted, so the last nybble of the program's END becomes an F. As soon as you leave PRGM mode all the local GTOs and XEQs are decompiled, and d is cleared so the last nybble becomes a D. When you do a GTO.. or a PACK the program is packed (and decompiled again) if the bit p is set, and the last nybble becomes a 9. If you create an END by using a normal or synthetic assignment from row C the last byte is always created as 0F.

Global labels contain Fn as their third byte (shown above as 1111 nnnn). The fourth byte (shown as kkkk kkkk) is the code for the key to which this global label has been assigned. This keycode is obtained from the same formula as for function assignments (see Section 14.10). If the label is

not assigned to a key then the code is zero. The remaining bytes of a global label contain the label name, coded as in a normal text string (shown as tttt tttt ...). The n of the Fn byte includes the keycode, so there are n-1 characters in the label name. Ordinary global label names have a maximum length of seven characters so n should have a maximum value of 8.

The first item in the CAT 1 chain stores zero as the distance to the previous item. This means there are no global labels or ENDs before this one; but there could be some other bytes between it and the curtain.

15.5 Four examples.

This section gives examples of how the information presented so far can be exploited. The examples are; labels more than 7 characters long, the results of setting a key assignment bit without storing an assignment, how to avoid decompiling programs, and how to execute selected pieces of programs from a ROM. If you are bored with examples you can skip these four.

Example 1. If you want more meaningful labels in your programs and have enough room in memory then you can make labels up to 14 characters long. Such labels will not be found by GTO or XEQ but they can be assigned to keys. They will also be displayed correctly by CAT 1, CAT 6 (on a 41CX or with a CCD module), by the printer function PRKEYS and the PANAME function VKEYS.

First of all go into PRGM mode and enter the instructions RCL IND 64, LBL 00. Follow these with a text string containing the label name you want. The first character of this string will be a keycode and will not be displayed as part of the name. If you want to assign this label to a key then choose a character which corresponds to a real keycode. You can use a synthetically entered character, or you can use any keyable character whose decimal byte value is 76 or less, except 69 to 72.

Now PACK, BST three times and BG. Delete the BG string and SST to see your new label. Since this label has been created synthetically it is not part of the CAT 1 chain. You must PACK or GTO.. to get it included in the CAT 1 chain. If that doesn't work, try deleting and replacing some steps near your new LBL first.

If the label is to be assigned to a key then the corresponding key assignment flag must be set. You can recall the register from the keyboard, put it in register d, set the required flag, then copy it back from register d to the assignment register. Use the key assignments RCL e, X<>d, and STO e to do this, and use the program SCF to set the flag if it is flag 30 or above.

To take a specific example - you have a rainfall prediction program which you want to call RAINFALL and you want to assign it to the shifted TAN key. The keycode for this key is calculated from the formula for a shifted key:

$$16*(C-1) + R + 8$$

The keycode for TAN is 25 so the code is 74. This corresponds to the character J (see Byte Table). The key flag given by the formula

$$36 - 8*(C-1) - R$$

is 2. As this is a shifted key, the flag is in register e.

1) Make sure that RCL e, STO e, X<>d. and BG are assigned. Cancel any assignments to keys −25 and −12.
2) GTO.000 and RCL e. The GTO.000 makes sure that the line number in register e is 000 so it is safe to put the contents of register e into register d.
3) Execute X<>d, SF 02, set USER mode if necessary, and X<>d again.
4) Press the STO e assignment. The flag for a −25 key assignment is now set. The flag for key −12 is also set because the USER mode flag had to be set at step 3. This will be discussed in the second example.

5) Go into PRGM mode. You should still be at line 000 of your program. If you are not then the operations in register d have reset the line number and you must GTO.000 again.
6) Put in the program lines RCL IND 64, LBL 00, "JRAINFALL". The letter J will become the keycode; the LBL 00 can be any one-byte step.
7) PACK, GTO.000 and BG. Backarrow the BG string and SST to see the eight-character long LBL "RAINFALL".
8) PACK again, press PRGM, and then press the shifted TAN key in USER mode to see the key previewed as "RAINFALL".
9) If you release the key soon enough then the RAINFALL program will be executed. If you press the key in PRGM mode then the program step XEQ "AINFALL" will be recorded. Only the last seven letters can fit into a non-synthetic XEQ.
10) You can cancel the assignment in the usual way, by assigning nothing to the key. The best use for long labels is to provide a meaningful title at the start of a program.

Example 2. At step 4 above you set the assignment flags for keys −25 and −12. You then created an assignment for key −25, but none for key −12. If you press key −12 (SHIFT, 1/X) in USER mode then the HP-41 finds the key is assigned but can find no assigned function or program. Something has to be executed though!

The HP-41 actually searches all the key assignment registers first, then it examines the key assignment byte in each global label, ending at the first global label of CAT 1. At the end of this fruitless search the HP-41 is left with the address of the first key assignment byte. This address is stored in two bytes:

nnnr rrrr rrrr rrrr

When the search is completed the HP-41 assumes a valid assignment has been found and executes this address as if it was a key-assigned function.

Suppose you have a 41C with full memory, or a 41CV or a 41CX, with SIZE set to 100. Suppose also that the first instruction in program memory is the

global label at the start of the first program. Then this label will be aligned in register 19B as below:

Byte	6	5	4	3	2	1	0
	C0	00	Fn	kc			

The keycode (kc) is in byte 3 of register 19B, so the address will be stored in two bytes as

61 9B

You can tell from Section 15.2 that this will execute like a one-byte assignment of the function 61, the ABS function. To check this out PACK, do a CAT 1, stop it immediately, GTO.000, set PRGM mode and press SHIFT, 1/X five times. You will have entered five ABS steps. The function previews the first time as XROM 06,27 which is to be expected for an assignment of 61,9B, but it becomes ABS when the key is released. After the first ABS has been entered, the rest of program memory is shifted down by one register, the keycode address becomes 61 9A, and the function preview is XROM 06,26.

Now PACK again and the first keycode will be five bytes below its original position, at byte 5 of register 19A. Its address will be stored as A1,9A. When treated as a function this displays as XROM 06,26, and it actually is XROM 06,26. If you have a Circuit Analysis Module plugged in it becomes "RS=". Change the SIZE to 90 and the keycode address will be A1, A4 which is the function XROM 06,36 or "LP=".

If the keycode is in byte 0 of a program register then its address will always be 01,XX or 00,XX and the function executed will always be XX. The whole business may be interesting but no one has found much use for it; it is simpler to assign different functions to a key than to keep changing SIZE so as to get various functions from one key. The reasons for this behaviour were explained (with one error - he did not point out that the byte position number is doubled, because nnn is stored in the leftmost 3 bits of a nybble) by T. Cadwallader in an excellent article in PPCCJ

V9N7P20-21. To cancel the "non-assignment" to key −12 use ASN and assign nothing to key −12.

Example 3. Synthetic ENDs can be created as well as synthetic global labels, but they have different uses. The most important is to prevent decompiling of programs. Each local XEQ and GTO in a program is compiled the first time it is executed. This greatly speeds up future execution of the program. There are three occasions when a compiled program can be decompiled (and thus slowed down) although you do not want this to happen.

Firstly, all the compiled information is recorded whenever a copy of the program is made, and this information is still there when the program is read back from a Card Reader, from Mass Storage or from Extended Memory. (There is one exception; when the Card Reader function MRG merges a program the compilation information has to be removed.) This compiled information makes the program run rapidly when it has been read back, but it is lost the next time you do a GTO.. - this can seriously slow down a long program.

Secondly compiled distances will be lost if you enter PRGM mode and accidentally press a key which records a new line. In this case you have to delete the unwanted line, and the ideal thing to do would be to PACK without decompiling.

Thirdly GTOs and XEQs can be created synthetically with the jump distance already stored in them. This makes labels unnecessary, since a compiled jump will be made without a check of the label, and the absence of labels saves space and speeds up program execution. Pre-compiled steps like this can even be used to jump from one program to another, or from a program in main memory to one in Extended Memory or in the status registers. The precompiled information will be lost if you leave PRGM mode by pressing the PRGM switch; at this moment the HP-41 will check if bit d in the END is set and will decompile the program.

If you have an HP-41C which was made before the middle of 1980 and has not been repaired since then you can use Bug 8 (see Appendix C) to avoid

decompiling. Simply switch the HP-41C off without leaving PRGM mode, then switch back on and the program is still compiled but you are out of PRGM mode. You can use some tricks to make a program run while you are still in PRGM mode, and let this program clear flag 52, but this is unsafe as any number entry in the program will start the HP-41 programming itself. The best way is to create a synthetic END with the pack and decompile bits both clear.

The following method to do this is given in the HP-41 Synthetic Quick Reference Guide. (Described by R. Hill in the Proceedings of PPC Conference 5, and later in PPCCJ V11N7P22, it was invented by C. Stern.)

1. Go to the END (or .END.) of the program which is not to be decompiled and insert the steps:

   STO IND 77, SF 09

2. BST twice and press the Byte Grabber

3. Press backarrow to delete the BG string. You now have a synthetic packed and compiled END, but it is not yet part of the CAT 1 chain.

4. GTO.. or PACK. The HP-41 will see the old END and will pack the program (removing bytes left over by accidental keystrokes and by steps 1 to 3.) until it reaches the synthetic END. Then it will include this END in the CAT 1 chain. After this it will check the decompile bit, find it is clear, and will <u>not</u> decompile the program.

5. To save space you should delete the extra END which was originally the ordinary END of the program. This is only necessary if the program finished at a normal END, not if it finished at the .END. . The unwanted END can be found by going through CAT 1 and stopping at the second of two ENDs with no LBL between them.

Note: to prevent accidental alterations in future you can make the program private (and therefore difficult to alter accidentally) by replacing SF 09 with SF 73 in step 01.

Example 4. You may find that you need to use two modules with the same XROM number, as was described in Section 12.5. As was explained it is usually impossible to use two such modules at the same time, but you can do it by jumping directly to an address in the ROM in the higher-numbered port. At other times it is useful to execute part of a program in a ROM module, not the whole program. This is particularly true of some Math module programs which always prompt for data. Each data prompt stops the program so it is impossible to store all the data beforehand and let another program do everything by calling the Math module program as a subroutine. Some programs in the Standard Applications and Statistics modules have the same problem. Most other modules have routines which prompt for data and separate routines to do calculations. This lets you write programs which prepare the data themselves and just call the calculation subroutines.

You could copy the whole program from a module into RAM, then edit it so as to keep only the parts you need, but the point of buying a ROM is to get a program which was written by someone else and takes up none of your valuable RAM.

The routine XRO is a compromise: it takes up 46 bytes of RAM but lets you jump to any ROM address and continue execution there. When the ROM routine reaches a RTN or an END it returns directly to your program at the line after XEQ "XRO". Since the ROM program may need any combination of values in the stack and numbered registers, XRO leaves these (and all flags) unchanged. It takes the ROM address from register M and uses the Alpha register to store values. One subroutine call is required to call XRO and another is used internally, so there are four return addresses still available, or 5 once the program in ROM is executing.

The following is a specific example of how XRO can be used, you can alter it to suit your own purposes. Chapter 9 pointed out that the Time module does not provide a function to put days of the week into the Alpha register. You can write your own program to do this but it will take at least 51 bytes even if you use 3-letter abbreviations for the days. If you

have a Standard Applications module then you can use a part of the CLNDR program to provide names of the days of the week instead. Use the Time module function DOW to compute a day number, add 2 to this, jump into the CLNDR program at line 143 and you will get the day of the week in Alpha. The RTN at line 149 of CLNDR will send you back directly to your own program. (If you do not have a Time module or if you want the name of a month then you can jump into CLNDR at different places to calculate day numbers or to get the name of a month.)

1. Assign RCL b and STO M to two keys. Put the Standard Applications module into the port where you want to have it when you want to run the program. If you want to use ZENROM as well then put the Standard Applications module into a higher-numbered port and remove the ZENROM while you carry out steps 2 and 3.

2. GTO "CLNDR" and then GTO.143. Check if you are at the right place by pressing PRGM and seeing the program step:

   143 7

   (If you do not see a 7 on line 143 then you might have a version of the Standard Application module with the program steps numbered differently. SST and BST a few steps until you find this one.) Press PRGM again.

3. Press CLA, RCL b and STO M. The ROM address is now in register M.

4. You must make sure that this same address will be in the rightmost two nybbles of register M before you execute XRO. One way is to use ASTO nn now and ARCL nn just before XEQ "XRO". Another way is to include the two bytes as synthetic text string in your program just before you execute XRO. A simple way to do this will be explained in the next Section. For the time being just leave register M as it is and go on to the next step.

5. When you want to use XRO to access this piece of the ROM, write a program that will put these two bytes at the right of Alpha and then do XEQ "XRO". Make sure the Standard Applications module is in the same port as it was when you carried out step 3. Make sure the stack contains the required values (in this case X should contain the day number + 2) then run the program. If you have already entered XRO and carried out steps 1 to 4 then try putting a number into X and executing XRO from the keyboard.

The idea for programs such as XRO came from Wickes (see his book, described in Appendix A). Originally they were designed to eliminate the shortcomings of the Math module, but they can be used to avoid XROM conflicts too. If you have a ZENROM in port 3 and a Standard Applications module in port 4 then you will be unable to execute the Standard Applications programs, as was explained in Section 12.5. However XRO will let you execute any program from the Standard Applications module, from the very beginning of a program, or from any other point. You can choose the place by going to it instead of carrying out step 2 above. The address used by XRO will be the ROM address in port 4 and will not be checked as an XROM number, so there will be no conflict. The same method can be used to execute a routine (but not a function) from any other ROM module.

XRO works by storing the ROM address as a return address in register b, then executing a RTN. The RTN moves the current program pointer to this address, even though there was no XEQ at the previous step. The HP-41 uses register b without asking how the addresses got into it! The trick is to rearrange register b without losing the previous return addresses in it, and without losing the values in the stack.

```
01*LBL "XRO"
02 XEQ 14
03*LBL 14
04 STO ]
05 RDN
06 RCL b
07 STO \
08 "┣'β*"
09 RDN
10 RCL ]
11 "┣υ"
12 STO \
13 "┣*"
14 X<> ↑
15 "┣**"
16 X<> \
17 STO b
18 END
```

Lines 02 and 03 push the return stack up. The RTN to line 03 will be replaced by a RTN into the ROM module. This means that XRO will be able to RTN to the ROM address, and the ROM program will return to the program that called XRO. Line 04 saves X in register O. Lines 05 to 08 copy register b to register N and then push the leftmost 3 bytes into register O. Line 10 puts these bytes into register X while line 11 pushes register M another two bytes to the left (only one of these bytes is printed in the listing). Line 12 puts the three leftmost bytes of register b back into N, and lines 13 and 15 push Alpha four more bytes to the left so that a new value of b is now in N. Line 14 recovers the original X value from register P and line 16 puts it in N. This use of register P means that the program can only be single-stepped in ALPHA mode.

Line 17 puts a new value in register b. The current program address becomes 60,05 hexadecimal; these two bytes were put in by the synthetic text line at 08. This address is at byte 6 of register M - in the Alpha register! The next two instructions executed are X<>N, RTN which were put into register M by lines 11 and 13. X<>N restores register X and RTN causes a return to the ROM address that was originally in M. Lines 05 and 09 were there to make sure nothing was lost from the stack. The STO b at line 17 jumps to another address so there is no real need for an END after it. You can put a global label at line 18 and carry on with a new program. The use of Alpha to execute X<>N, RTN makes this program 12 bytes shorter than the PPC ROM equivalent (which could not use this trick because it is in ROM) but leaves Alpha uncleared.

The text lines 08, 11 and 13 have to be made synthetically. Line 08 is made up of the bytes F4, 7F, 60, 05, 2A. You can create it by making a normal text line such as " ├ ***" then grabbing the F4 and replacing the first two stars (which become RCL 10 steps) with STO 12 and LBL 04, then grabbing the BG string. 60,05 is the address in Alpha where the program jumps to. The star could have been any character. Line 11 consists of the bytes F3, 7F, CE, 76. It can be made in a similar manner but involves more byte grabbing since CE,76 is the X<>N instruction and has to be made synthetically too. Line 13 is F3, 7F, 85, 2A. The 85 is the RTN executed after X<>N and has to be put in synthetically. The 2A is a star but again could be any character, as could the stars in line 15 which is an ordinary text line with no synthetic characters.

Putting in all these synthetic characters is a chore and can be simplified by the use of a Q-loader as will be described next.

15.6 Synthetic text and Q-loaders

One way the HP-41 uses register Q is to build up a number when one is entered from the keyboard or from a number entry in a program. This has an unusual effect if one of the number entry keys from row 1 of the Byte Table is assigned to a key and this key is pressed in PRGM mode. The previous contents of register Q are reversed and copied into program memory as a text string immediately after the digit assigned to the key, then register Q is cleared.

Use a key assignment program to assign the bytes 1B,04 to a key. These bytes represent the assignment of the EEX function to a key. Press GTO.. then GTO "XYZ" and wait till you see NONEXISTENT. If you have a LBL "XYZ" somewhere then use a different label name which does not exist. Now set USER and PRGM mode and press the key you have just assigned. You will see an E on its own; if you SST you will see the text string "XYZ". When you executed the GTO, the label name was reversed and stored in Q during the search for the label. When you pressed the E assignment, the contents of Q were reversed again and copied into your program.

Press the key twice more, then SST twice. You will see two empty text strings. These are the byte F0, used as a NOP. Register Q was cleared after it had been copied into the program so a null text string was copied the second and third times. Backarrow twice and you will see two Es in one line. Assigning a digit to a key prevents the entry of a null before a new number, so these two EEX instructions have merged into one line. BST three times and you will come to the END of the previous program in memory. The two Es were numbered as separate lines when they were entered but have become one single line, so the line numbering has changed and you can get into the previous program in memory.

This one assignment lets you:

1. Create the F0 NOP by pressing the key twice.
2. Create the step E which is like an entry of the number 1 but faster.
3. Create an E E numeric string which will mystify people and let you BST into the previous program.
4. Load text strings from register Q into a program. You can create any string in Alpha containing up to seven bytes by building it up using the program BAB or the Extended Function XTOA. Then you can put this string into Q by executing RCL M, STO Q. Finally you can copy it into a program by using this assignment. That is why it is called a Q-loader. Remember that the string will be reversed; a way round this is described below.

If you want to create a combination of bytes from the last quarter of the Byte Table you may not be able to do it with the Byte Grabber alone. You can store the bytes in register Q, then use a Q-loader to copy them as a text string into a program. Then you can byte grab the leading Fn text string header and leave the required bytes where you want them. A Q-loader cannot create a text string with nulls at the end, but nulls can be put in the middle or at the beginning.

For strings of six bytes or less you can avoid the need to enter the bytes in reverse order. As an example you can make line 08 of the program XRO in

the previous section by doing the following:

1. CLA
2. 127, XEQ "BAB"
3. 96, XEQ "BAB"
4. 5, XEQ "BAB"
5. ALPHA, Append * (use SHIFT, K, SHIFT, *)
6. ASTO . X, ALPHA. This puts the text string into register X.
7. GTO IND X. This reverses the text string and copies it into Q.
8. Wait for NONEXISTENT, or for the label to be found. Use CAT 1 and GTO.nnn to get to the place in program memory where you want to put the text string. This will not affect register Q unless you have a CCD module whose CAT' function alters register Q. In that case press CAT, then 1 before you release the CAT key, then release CAT and then the 1 key.
9. Set PRGM mode, then press the Q-loader key.
10. Delete the E, then SST to see the text line.

Most one-byte or two-byte assignments which execute bytes 10 to 1C will act as Q-loaders in the same way. Some of them provide prompts or function previews and many people like these since the prompt or preview identifies them as unusual functions and allow the user to cancel the function. I prefer the two-byte assignment 1B,nn (decimal values 27,xx), as it loads the E and saves time because it does not prompt. One-byte assignments of byte 17 prompt for an Alpha parameter and create a text string; those of byte 1B can take an Alpha parameter and create a global label. All the other one-byte assignments ask for no parameter, or ignore it. These assignments can be used to produce very long or odd-looking numeric strings but the normal keyboard number entry rules given in Chapter 4 still apply during program execution: only the first ten digits are used, only the first EEX is used, multiple sign changes alternate between positive and negative values. Two or more consecutive copies of byte 1A display as a single comma, regardless of the setting of flag 28, but all after the first one are ignored during numeric entry. Byte 1C on its own should be avoided because it looks like a subtraction step in a program but actually puts a zero in the stack.

Why do these assignments act like this? Because only bytes 2D and 2E of row 2 were meant to be assigned to keys, any row 2 assignment is treated like GTO or XEQ; it is put directly into a program, a null byte is not inserted before it, and if any byte leaves ALPHA mode clear then register Q is copied into the program at once as a text string (just the same as when you finish enetering an alpha GTO or XEQ and go out of Alpha mode). Assignments of bytes 1D and 1E act as Q-loaders too (because they leave Alpha mode clear) but they make the instructions GTO and XEQ followed by the contents of register Q. They can therefore be used to create instructions such as GTO "1.5" or XEQ "A&B" which contain characters that cannot normally be put into a global label. All assignments of 1D and 1E act as non-prompting Q-loaders. Two-byte assignments of byte 1F create non-prompting W Q-loaders. One-byte assignments prompt for a number (they prompt 2__ just like assignments of byte 0D), but they too produce the instruction W followed by a text string. The "spare" function W has been described in Section 8.5, and all three bytes 1D, 1E and 1F will come up again in the next section.

There is also a LBL Q-loader. Any assignment with byte CD as the first byte can be pressed in PRGM mode to produce a global label made up of the contents of Q. One-byte assignments (byte CD preceded by a byte from row 0) prompt for a number but then also produce a global label.

Assign the LBL Q-loader (bytes CD,00) to a key and try this:

1. ALPHA, 1, ., 5, ASTO X, ALPHA
2. GTO IND . X
3. PRGM, press the LBL Q-loader
4. See LBL "1.5", which cannot be made directly from the keyboard

The ordinary Q-loaders can be called "text Q-loaders" to distinguish them from the others. A text Q-loader followed by a BG to remove the leading Fn can load a maximum of seven arbitrary bytes into a program. Special synthetic programs to load more than seven bytes have been written - not surprisingly they are called "Byte Loaders".

The GTO, XEQ and LBL Q-loaders provide a very easy way of producing and using the synthetic global labels "A" to "J" and "a" to "e". Simply put the one letter into Alpha, ASTO X, GTO IND X and then press the appropriate Q-loader. These synthetic global labels are different from the local labels LBL A to LBL e and can be used in programs that need to put a single letter into a register then GTO IND to use that letter. An example is a program that prints large letters and needs to use the labels "A" to "Z" to specify each letter to be printed.

15.7 Other SP bits and pieces

Some synthetic instructions that are worth mentioning did not fit well into other sections. They have all been gathered here. If your favourite SP trick is not here - sorry. Maybe you should write it up and publish it in one of the user group journals!

1. Decimal values are not well-suited for conversion to or from binary numbers in the flag register. Instead of repeatedly multiplying or dividing by 2 as in the programs BAB and RAB you can use OCT and DEC to convert decimal numbers to or from octal. For example consider how the decimal number 110 can be turned into the hexadecimal byte 6E. The decimal number 110 can first be converted into the octal number 156 by the function OCT. This can then be stored in register d where it will appear as the bit pattern shown below:

0000 0001 0101 0110 - 156 stored at the left of register d

A set of flag tests can be used to shift all the bits into the second byte of register d, from where the byte can be appended to the Alpha register.

0000 0000 0110 1110 - 156 converted to byte 6E

Since all octal numbers are made up of the digits 0 to 7, each digit uses only the lowest three bits of a nybble and conversion to a hexadecimal byte

is carried out by pushing bits to the right, removing the empty bit at the front of each nybble. This is done by a set of flag tests which produce the lower of the two bit patterns shown. (In this case the first test would be FS?C 11, SF 12.) There are several excellent examples of this technique, invented by Roger Hill, in the PPC ROM. The method is explained in the PPC ROM manual write-up for the routine "DC". (The author was apparently so enthusiastic about it that the paragraph was printed twice.)

2. The above method can be used in reverse to convert hexadecimal vlaues to decimal numbers into register X. These numbers, and others created by synthetic techniques, might not be normalised. Operations such as RND or HR or adding zero can be used to normalise these numbers. If you see such an instruction in a program for no obvious reason then it is probably being used for normalisation.

3. Until a number is normalised it may contain the nybbles A to F in places where only the decimal digits 0 to 9 are expected. The HP-41 uses the remaining characters of row 3 in the Byte Table to display these digits, but it also tries to "carry" from one digit greater than 9 to the next, so you will not always see them.

To study what happens, create the number 1.2ABCDEF00 E00 in register M. To do this set FIX 9 mode and CLA. Then put the bytes 01, 2A, BC, DE, F0, 00, 00 into M by using the program BAB or the Extended Function XTOA. (1, XTOA, 42, XTOA, 188, XTOA, 222, XTOA, 240, XTOA, 0, XTOA, XTOA). Use RCL M and see the number displayed as:

1 . 2 ▒ ; ∠ = \ ? 0 0 Display in X

1 2 A B C D E F 0 0 Digit values

1 . 2 : ; ∠ = \ ? 0 0 Obtained by ALPHA, CLA, ARCL X

The second line shows the value of each digit, and the third line shows how the number will look if recalled to the Alpha register (it will be followed

by a prompt). The character 3A (representing the hexadecimal digit A) shows up in different guises in X and in Alpha. The other digits do not change. This has been called the HP-41 **natural notation** because it is the way the HP-41 displays hexadecimal digits unless told otherwise. It is also called **Australian notation** as it was first studied by the user group in Melbourne.

Now go out of ALPHA mode and set FIX 8. You will see:

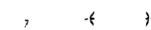
1	.	3	1	2	3	4	5	5	0	Display in X
1		2	A	B	C	D	E	F	0	Digits
1		2	10	11	12	13	14	15	0	Decimal digit values

Each digit is displayed now as a number below ten plus a "carry" from the previous digit. The bottom row shows what each digit value is and you can check that the digits are displayed with the "carry" from the previous one. The display shows the effects of carrying except in FIX 9 mode and in the synthetic FIX 10, SCI 10 and ENG 10 modes. These can be set by adjusting register d directly, by using synthetic assignments, or by using the Extended Functions RCLFLAG and STOFLAG (FIX 08, RCLFLAG, FIX 2, 36, STOFLAG). You can also create the NNN 0A,00,00,00,00,00,00 then put it in X and execute FIX IND X (or SCI or ENG). The easiest way is to use ZENROM and set FIX 10 directly from the keyboard. You can use these display modes to examine nybbles 12 to 3 of NNNs so long as they display as a number. **The standard numbering of nybbles is 13 for the leftmost and 0 for the rightmost position in a register.**

4. Numbers with non-zero exponents and particularly with non-normalised exponents behave oddly too. Try 1.23456789E13, FIX 10 with flags 28 and 29 set. Clear flag 29 and the funny exponent will vanish. Clear flag 28 and all the digits except the first one will display as characters from row 2 instead of row 3. If you try ARCL X now the HP-41 will give up and only recall the 1. Set flag 29 and you will see a mixture of characters from

rows 2 and 3 with an exponent containing only a decimal point.

Clear flag 29 again, create the byte string 0B,00,0C,00,0E,00,13 and RCL M. You will see a miniature 7 (the semicolon actually) and two flying geese:



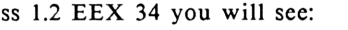
Wickes' book "Synthetic Programming on the HP-41" describes more of this. You can even get a "goose" in the exponent. Try entering and running the program:

```
01 *LBL "GSE"
02  45
03  XEQ "SCF"                    See Note 1
04  F7,AF,FF,FF,FF,FF,FC,FF      See Note 2
05  RCL M
06  STO Q
07  CLX
08  END
```

Note 1. You can use the PPC ROM routine IF or the ZENROM function TOGF instead of SCF on this line.

Note 2. This is a way of writing "enter a text line made up of the text string header F7 followed by the bytes AF,FF,FF,FF,FF,FC,FF". On a program listing this line would actually show up as 04 "" , which does not reveal what the text string contains. In synthetic program listings this method is often used instead to show that a synthetic text string (or some other synthetically made set of bytes) is to be entered.

Run this program, then enter a number with an exponent from the keyboard. The exponent will be entered as a negative exponent, not a positive one, but you will see a backwards goose instead of the minus sign! If you XEQ "GSE" then press 1.2 EEX 34 you will see:



Press ENTER and you will see this is just 1.2E-34. Do not press EEX on its own (nor after just a zero or a decimal point) after running GSE as this

will force the creation of a 1 to precede the exponent and that will replace the special value in register Q which allows the strange exponent to be displayed. The program GSE sets the "system data entry" flag to tell the HP-41 a new number is in the process of being entered. Then it puts an NNN into Q to show that no digits have yet been entered and that the exponent sign is to be displayed as byte 2C instead of byte 20. The CLX resets register P to accept the new number from the keyboard.

Numbers with a non-normalised mantissa can slow down the execution of arithmetic functions. The arithmetic and mathematical operations expect the most significant digit to be in nybble 12, and are designed to work efficiently on this assumption. If you create the byte string

00,00,00,00,01,00,00

then put it into register X and press 1/X you will have to wait about 13 hours for the result (the time taken increases by a factor of about 10 for each additional zero to the left of the most significant digit).

Numbers with an exponent sign other than 0 or 9 will slow down the display, particularly in FIX 10 mode. Set FIX 10 mode and put the NNN

00,00,00,00,00,03,00

into register X. Go into PRGM mode and turn off your HP-41. Turn it back on and you will see the program step first, while the HP-41 is working out how to display the contents of X. Press ENTER and the function will preview for a longer time than normal while the display again tries to work out how to display X. It is only the display that is slow as you can tell by running a program and seeing it runs as usual.

5. During the entry of a number or text string flag 45 is set. It becomes clear when number entry or text entry is terminated. In run mode Alpha entry is restarted and flag 45 is reset when Append (SHIFT,K) or ARCL are executed. Number entry in register X can be restarted by a short program

in Chapter 16. You can restart text entry in a program by setting flags 45, 48 (ALPHA) and 52 (PRGM). Register Q must be reset too, much as in the example given above. In the case of text entry, nybble 13 of Q (the leftmost nybble) must be cleared.

One way to do all this and so alter a text string that is already part of a program is to use an NNN with nybble 13 clear and nybbles 0, 1 and 2 containing 488. Make this NNN, go to the program line to be edited, set run mode and USER mode, then store the NNN in register Q and in register d. Pressing a character key now will not add new characters to the text string but will make it swallow up bytes that are already in the program after the text string. Deleting characters does not delete bytes but just removes them from the text string and leaves them in memory; this can be used as a different way of freeing bytes put into a program by a Q-loader. Pressing the backarrow key before any characters creates a character string which incorporates the next fifteen program bytes. This can be used instead of the Q-loader to make text strings.

The exact details of what happens can vary and can cause a crash depending on the contents of Q and on the date an HP-41 was made, so you should test exactly what this process does on your HP-41. The synthetic lines produced can be made with the Byte Grabber, so this technique is not used much. You may find it to your liking though; it was invented by W. Wickes and called **text enabling**. It is described in his book "Synthetic Programming on the HP-41".

6. The global GTO and XEQ instructions (bytes 1D and 1E) expect to be followed by a text string no more than seven bytes long. Other bytes can be made to follow them by synthetic means. Text strings of up to 15 characters can be used, producing GTOs and XEQs with longer names, but these will not go to corresponding longer labels, because register Q can hold a maximum of seven characters for a comparison. Instead they will go to labels which just contain the last seven characters of the long name.

If the first byte after a 1D or 1E is not an Fn then the second nybble is

still used to give the length of a text string displayed after the GTO or XEQ. When you SST past the first byte (and its postfix bytes, if any), the HP-41 realises that there is no text string following. It treats the next byte as part of the GTO or XEQ, but displays the following bytes in the normal way. Imagine you have the following set of bytes at the beginning of a program:

1D , F5 , 41 , 42 , 43 , 44 , 45 , 46

This will be displayed as **GTO "ABCDE", X<=Y?**. If you replace the second byte with 75 (RDN) you will have:

1D , 75 , 41 , 42 , 43 , 44 , 45 , 46

which displays in a program as **GTO "ABCDE", -, *, /, X<Y?, X>Y?, X<=Y?**. The byte 75 is treated as a text string header while the GTO is displayed. When you SST past the GTO, the following byte is not a text header so the HP-41 does not skip a string of text but just displays the bytes in their normal guise.

I find this **non-text GTO** useful for program analysis. Try this to create a seven-byte non-text GTO and see some examples. I assume you have BG, RCL Q, and STO M assigned to keys.

i. Enter the program
LBL 38, RCL 29, CLX, LN, X↑2, SQRT, Y↑X, CHS, E↑X, LOG

ii. PACK and BST until you see the LBL 38.

iii. BG and delete the text line.

iv. SST and see GTO "PQRSTUV"
The 29 of the RCL 29 has become a GTO, and the CLX (byte 77) is being treated as the header of a seven character name.

v. SST twice and see LN, then X↑2

vi. Insert the line XEQ 38.

vii. BST three times and see **GTO "PQ▩¯&¯¯"**
You have put an XEQ after the Q, so this takes up the next three bytes. Inserting a new instruction produced seven bytes of which three now contain the XEQ, but the other four are still empty - you can see the first two after the &. This & is the 38 of XEQ 38. The XEQ has not yet been compiled so its middle byte is a null.

viii. SST three times, go out of PRGM mode and press SST two more times. This executes the XEQ and compiles it.

ix. Go back into PRGM mode and you will now see **GTO "PQ▩▩▩¯¯"**
The XEQ is now compiled so its last two bytes have changed.

x. PACK in PRGM mode and the GTO is now **GTO "PQ▩¯▩RS"**
The null bytes at the end have been removed and the bytes RS have moved back into the string. The XEQ has been decompiled so its middle bit is null again but the jump direction has not been changed so the & is still displayed as ▩.

xi. Go to run mode, press SST and wait for NONEXISTENT. Go back to PRGM mode and see the line Y↑X. All the bytes in the pseudo-text string have been skipped, and the last one is displayed when you go into PRGM mode. You can set flag 25 in a running program, then execute a test immediately before a non-text GTO. If the test is false then only the GTO and the next byte (plus any postfix bytes) are skipped. If the test is true then the HP-41 tries to execute the GTO, finds it nonexistent, and skips all the bytes of the GTO. This can be used as a conditional test to jump over n bytes instead of just one byte. Unfortunately, the time needed to complete the label search makes it rather slow.

xii. Go out of PRGM mode, press RCL Q and STO M. Go to ALPHA mode and see

SR▩¯▩QP

These are just the bytes in the GTO string, copied to Q by the GTO, then from Q to M, with their ordering reversed. You can use the program RAB, or some other technique, to see what the bytes were. If you do this you will see that the last byte of the GTO 38 (now the third byte from the left in M) was indeed 168 decimal, in other words 38 with a backward jump bit (value 128 decimal) added to it.

You could use a non-text XEQ instead of a GTO for all this. The point is that you have a relatively simple and harmless program analysis tool. If you forget to delete the GTO before running the program you were analysing then you will get NONEXISTENT - it is most unlikely that you will have a global label with the same name as the text in the GTO. You can also use the byte 1F for program analysis with similar results. However W will not copy the text into register Q and may cause a crash (especially if the byte after it is from row 0 of the byte table). It may even cause execution of a program or function from the upper 4K of a ROM in port 2, or FL NOT FOUND if you have a 41CX with nothing in the upper 4K of port 2.

When you SST past a text GTO or XEQ in PRGM mode the HP-41 skips past the instruction immediately after the byte 1D or 1E. Normally this should be a text string, so skipping the next instruction would correctly skip over the string. If the next instruction is replaced synthetically by a one-byte function then that one byte will be skipped as we have seen above. If it is replaced by the first byte of a multi-byte function then interesting things can happen. If the byte is from row C of the byte table then the HP-41 will try to skip over the apparent global label prefix and will try to use the jump information in the bytes after the "global label". After a long wait you will find yourself at some random point in program or data memory, unless you have put a chosen jump distance after the Cn byte on purpose.

Should the GTO or XEQ be followed by one or more null bytes the HP-41 will look for the next non-null instruction and will skip that. An amusing thing happens if a global GTO or XEQ is followed by null bytes right up to the END of the program. If you SST in PRGM mode the END is treated as the step to be skipped, but it is still recognised, so the next step displayed is line 01 of your program. How's that for ENDless Synthetic Programming?

Even more amusing is Clifford Stern's exploration of the recursive properties of following the Alpha GTO prefix with another Alpha GTO prefix. He has written a program to fill program memory with hex 1D bytes -- a 2240-byte single-line program that has no visible END!

7. As you get to know more details of what the HP-41 operating system does you may think of new things to do. Among these is writing functions that you would like but which the operating system does not provide. Some of these are obvious extensions of the mathematical functions, such as the hyperbolic functions in Chapter 7. Others are extensions of the functions used to control the HP-41 itself; this is the realm of SP. Synthetic key assigning and system flag altering are two such system extensions, achieved by SP. Re-enabling the editing of register X is another case; this can be done by the program APX in the next chapter. Programming with non-programmable functions is yet another example; the next chapter will show how a programmable PACK can be achieved. It will also show how the non-programmable printer function PRP can be used in programs. A programmable PRP for use in documentation will be given in Section 16.8 and a simple programmable PRP which does not use any Extended Functions will be shown in the last exercise of Chapter 16.

15.8 Do's and Don'ts of Synthetic Programming

Following some serious uses of SP, and some frivolous ones, here is a list of warnings and advice.

While writing HP-41 programs do bear in mind the possibility of making programs better, shorter, or faster by using synthetic instructions or key

assignments. Do not use SP just for the sake of it though; at times it may be equally easy to use ordinary HP-41 functions.

Do read about SP occasionally, but do not expect to remember everything. If you have the time then try to keep a notebook (as suggested in Exercise 1.C) or an index of tips and ideas as you read of them.

If you decide to use SP in a program do make notes of what you are trying to do. Do not begin to push keys at once unless the program is trivially simple. See if someone has written a program to do the same thing, maybe you can find such a program in this book or in one of the books or journals in Appendix A.

Once you have decided to enter a synthetic program do make a complete record of any important programs or data you have in your HP-41. Be aware that MEMORY LOST can happen - do not think "I'll be careful so it won't happen to me". It will.

While entering a synthetic program do be careful not to disrupt the CAT 1 linkage by byte grabbing ENDs or global labels. In particular avoid grabbing the .END. - if you want to create some new globals synthetically do not try to alter existing labels or ENDs. It is safer to create them from other instructions. If you have created a new label or END, then do GTO.. to rebuild the CAT 1 linkage and include the new globals in it. (If you have edited the program in such a way that the pack and decompile bits are clear then the program will not be packed, and the new global will not be added to the CAT 1 chain. Normal editing, such as deleting and reinserting an instruction, will set the decompile bit so the new global will be added to the chain.) If you have lost the .END. you may be able to make a new one; change the last three nybbles of register c so they point to a completely empty register below the position of the previous END, then execute GTO.. - null bytes at the position of the .END. are treated as an .END. that needs rebuilding and are turned into a real .END. which points to the previous global, if one can be found.

Do be very careful when writing programs that alter register c. If there is any chance that the program will be interrupted or single-stepped by someone, or will stop at an error, then put acceptable values into register c. Acceptable values are ones which:

1. Have 169 in the three cold start constant nybbles.

2. Have a register 00 pointer (curtain) for which the register immediately below the curtain address exists. Do not put register 00 directly above a gap in memory.

There may be occasions when you are writing a very short routine and you do not want to spend time and space on these precautions. If the routine is sufficiently short and fast, and if it is impossible for it to stop at an error then you might be able to put other values in c, but warn anyone who is going to use it (including yourself). Make a note that the routine must not be single-stepped. In such routines it is a good idea to set one of the flags 0-4 while register c contains an unsafe value, and remind the user not to stop or single-step while this flag is set. Do not take risks with register c!

Do be careful not confuse data register 00 with status register O. Even more important, do not confuse synthetic access to register 104, which shows up as a capital **C**, with status register **c**.

Do avoid assigning STO c or X<>c to keys and leaving them assigned; you may forget and accidentally press one of these keys later. The safest key for these assignments is ISG; it is not used for keyboard operations - even if you press it without realising USER mode is set then you are only likely to do so in PRGM mode, and will not get an immediate MEMORY LOST. If you have a ZENROM or a CCD module then do not press STO . c or X<> . c just for the fun of it. It's not fun. Do not press STO 125 or X<>125 either if you have a ZENROM, as they have the same effect.

If the first program in CAT 1 has any uncompiled GTOs or XEQs then the

label search goes down to the END, then restarts at the register below the curtain. If you have raised the curtain then the search will begin in the data registers. Do avoid this; make sure there is at least one END above programs that raise the curtain and have any local GTOs or XEQs. If you have lowered the curtain then no label search will be made above the curtain - this might mean that labels near the top of your program are not examined or it may even mean that the search begins in a program below the one that is executing. These restrictions do not apply if the jumps are already compiled. Do not run uncompiled programs with the curtain lowered unless you are intentionally trying to jump from one program to another. You can compile a program before running it by taking the next seven steps:

1. Assign RCL b and STO b. This is not necessary but speeds things up.
2. PACK, GTO.001 and set PRGM mode.
3. SST until you find a local GTO or XEQ.
4. Go out of PRGM mode. Press RCL b to save the present address.
5. SST - this compiles the GTO or XEQ.
6. Press STO b to get back to the GTO or XEQ, then go back to PRGM mode. You can use GTO.nnn instead of RCL b and STO b to get back where you were.
7. Repeat steps 3 to 6 until you reach the END.

This compiles the steps without editing the program, so the steps will not be decompiled. After you use STO b at step 6 the program line numbers will be incorrect. Do not worry about this, as it does not affect the compilation. Obviously this method cannot compile GTO IND, XEQ IND, nor short-form GTOs to labels more than 112 bytes away, as none of these can ever be compiled.

Do take care with register d as well. The setting of most flags in a running program has little effect until the program stops, pauses or prompts, but setting flag 52 and then executing a number entry makes the HP-41 start to program its own memory. If you set flag 54 then the final RTN or END will only execute a PSE, then the program will carry on running. If you leave flag 25 set, serious errors could be ignored. If you leave

flag 24 set, all arithmetic overflow errors will be ignored until you clear it. If you leave flag 47 set then pressing a key will execute its shifted function, and executing OFF will start the clock running if you have a Time module. Leaving flag 44 set will prevent the HP-41 from turning off to save the batteries. Leaving flag 33 set will disable all HP-IL operations. In summary, do not leave rubbish in register d.

Do be careful with the other status registers too. Do not risk losing return addresses by playing with registers a and b, nor losing the key assignment maps in registers Append and e.

Do remember that inserting instructions in a program moves everything down by a register until the .END. is reached. If you insert instructions below the .END. then everything is moved down, including register c, so a MEMORY LOST is the result. Do not add bytes into areas such as the key assignments until you have deleted enough bytes to make room for the new ones. If you want to use a Byte Grabber in these areas then use the F3 grabber which does not insert a new register, but be careful with that too; it needs three empty bytes. To make things a little safer, PACK and set the maximum SIZE first, so that a mistake will give PACKING, TRY AGAIN, rather than the less welcome MEMORY LOST.

Do remember that several operations normalise NNNs, so you must take care if you are working with NNNs such as the contents of registers c or d. STO and ASTO do not normalise so you can put an NNN anywhere but RCL, ARCL, VIEW and X<> all normalise an NNN unless it is in a status register. These functions normalise the value that is being recalled or viewed, both at its original position and the copy brought to X. X<>nn does <u>not</u> normalise the value that is being stored in nn. Printer functions can also normalise; be careful if a printer is attached and you run programs that deal with NNNs. Even if you do not execute any print functions X can be normalised when it is printed if program execution is being traced. Operations on blocks of data often normalise one or both of the registers at the ends of the block; they do this while checking if the block exists. Σ REG nn normalises both ends of the statistics block (registers nn and nn+5), even if it uses SP to

put the block in the status registers. Normalisation by Extended Functions is described in Section 16.1. Articles on normalisation have been published by C. Close in PPCCJ V10N1P64 and by C. Harris in the PPC Conference notes for the seventh conference (Aug. 27 and 28 1983). The Synthetic Quick Reference Guide also explains how numbers are changed when they are normalised.

Do be very careful if you use the free area below the .END. to store anything, and clear this area again as soon as possible. Do not forget that the .END. cannot move down further when it meets a register with something in it. Remember too that a register with something in the leftmost two nybbles might be treated as a buffer header, and that a zero in the next three nybbles will cause a lock-up when you pack or turn on. An HP-41 made or repaired since the Summer of 1985 (yes 1985 - this is a recent change in the operating system) can be taken out of this lock-up if you turn it on while holding down the backarrow key. This produces a MEMORY LOST, but on earlier HP-41s the buffers were cleared before the backarrow key was checked, so you could not even do a MEMORY LOST to get out of this lock-up.

Once you have written your synthetic program do make a record of it on a sheet of paper, on a magnetic card or on Mass Storage. If the program does go wrong you will be able to check for possible errors, and if you get MEMORY LOST you will be able to put it back and change the part that is wrong instead of having to start all over again. If you experiment with SP then keep a record of what you are doing. Do not finish up with a result which you cannot repeat because you have no record of how you got there.

If you are planning to let other people use your synthetic programs then do describe them very carefully and give warnings of what they do. If you are planning to use someone else's synthetic programs then check what they do. A few synthetic instructions behave differently on various HP-41 models (W is an example of this). Do not submit badly documented programs to the Users' Library; follow the guidelines in their Reference Guide. When the Library began to accept programs with synthetic instructions they published

some helpful guidelines in PPCCJ V8N6P14-15. One point that has come across very clearly is that the Library refuses to accept programs with synthetic tones, no matter how strongly you feel a program needs them.

Do consider joining a users' group. Better still, go ahead and join one. The journals themselves are worth the money, they provide programs, advice, and information about new products, as well as new ideas on SP. Do not imagine that the groups are for children who swap games programs, or for HP fanatics, or an outlet for HP sales personnel. The groups tolerate all these people, but they are really for serious users who are pleased with good products, critical of poor products, and keen to learn and help.

Exercises.

15.A One way to understand a synthetic program is to rewrite it. SST the original version (unless the author says this is dangerous), make notes, and try to write a new version. Try to change RAB in Section 15.3 so it will analyse up to 24 characters. Hints on this were given in Section 15.3.

15.B Instead of the non-text GTO you can make a global label with C0,00,F0 in its first three bytes to analyse instructions in program memory. The next byte is treated as a key assignment and is not shown, the following fifteen bytes are displayed as a label name. Make such a set of bytes and study their effects. Avoid executing this instruction - it causes a crash.

CHAPTER 16 - SYNTHETIC PROGRAMMING WITH EXTENDED FUNCTIONS

16.1 New tricks for old

The Extended Functions made many SP tricks obsolete (at least for those who have Extended Functions), but provided scope for new ones. Byte-building and analysing programs can be replaced by XTOA and ATOX. Curtain raising to hide data registers can be replaced by REGMOVE, REGSWAP, and data files in Extended Memory. On the other hand, new tricks can be played with register d by using X<>F, STOFLAG and RCLFLAG, and there are many ways to use Extended Memory. Additional Extended Functions available in the CX and in modules such as the Extended I/O module can be used in SP. Of course if you do not have the Extended Functions then you need SP to do what they do.

Non-normalised numbers (NNN's) remain important, for example they are used to address Extended Memory directly. Several of the Extended Functions allow greater control of NNNs - they will be described in the next section. Other Extended Functions let you store NNNs in Extended Memory and recall them without normalisation. These will be described in Sections 16.5 and 16.9 but it is worth noting at once that SAVEX, GETX, SAVER, GETR and GETRX do not normalise any data. SAVERX normalises the first and last registers moved, both in their original positions and in the data file to which they are moved. For example 5.001, SAVERX and 1.005, SAVERX both normalise registers 01 and 05. REGMOVE and REGSWAP normalise the top registers of the source block and of the destination block. With a control value of sss.dddnnn both the registers sss+nnn-1 and ddd+nnn-1 are normalised.

The indirect comparisons on the HP-41CX (X=NN? etc.) normalise any value in a numbered data register before making the comparison. They will compare NNNs safely with X if these are in registers Z, T, or L.

16.2 Alpha register operations

XTOA lets you convert any byte number in X to a byte which it adds at the right of the Alpha register. It can therefore replace the routine BAB from Chapter 14 to build up NNNs in Alpha. The Extended I/O module (called the XIO as well) has two functions XTOAL and XTOAR which add a byte at the left or the right of the characters already in Alpha. Evidently XTOAR is the same as XTOA. XTOAL can be replaced by XTOA, -1, AROT, RDN if you do not have the XIO module. The HP-IL Development module (the Devel) has the functions X-AL and X-AR which are the same as XTOAL and XTOAR.

ATOX lets you analyse any string of bytes in Alpha, starting with the leftmost character. It is similar to RAB in Chapter 15 but it does not tell you which position the character was at, and it cannot handle null characters. However it can handle up to 24 characters. A null in the middle of a text string is lost as soon as the characters to its left have been removed. To check for a null you need to use ALENG. Consider the string of hexadecimal bytes:

28,00,29

If you use ATOX once, this string will turn into 00,29 which is just the same as 29 alone. To check for a null you need a subroutine that will do something like this:

CF 00, ALENG, 1, -, ATOX, X<>Y, ALENG, -, X≠0?, SF 00, X<>Y, RTN

This puts the ATOX value into X, but also sets flag 0 if any nulls followed the character. Register Y contains the number of nulls. For keyboard use you could assign this routine to a key and execute it instead of ATOX. The XIO and Devel modules have functions called ALENGIO and ASIZE? respectively; both are equivalent to ALENG. The XIO provides the functions ATOXR and ATOXL. The first is equivalent to -1, AROT, RDN, ATOX unless the character at the righthand end is a null; the second is the same as ATOX. The Devel has the same functions called A-XR and A-XL.

At times it is useful to check one byte in an NNN at a given position, without removing that byte. This can be achieved by a combination of AROT to put the byte at the lefthand end, followed by ATOX, then XTOA to replace the byte, and another AROT. If null bytes are to be checked for then this becomes much more difficult. The XIO function ATOXX lets you put a number in X to specify which character you want, and it puts the corresponding byte value in X without altering the value in the Alpha register. The same function in the Devel is called A-XX. A corresponding function YTOAX (character in Y to Alpha position X; Y-AX in the Devel) lets you insert a byte into the Alpha register, replacing the byte at that position.

ATOXX and YTOAX are particularly useful when you are dealing with strings which might contain nulls. Both functions use the number in X in the same way to define a byte position in the Alpha register:

i. If X is positive or zero then it gives the character position counting from the first non-null byte at the left. The leftmost character is counted as 0. Positions larger than or equal to the number of characters in Alpha give DATA ERROR. The largest valid positive value in X is therefore 23 (not 24 as stated in the manuals for both modules).

ii. If X is negative then it gives the character position counting from the righthand end of the Alpha register. The rightmost position is -1 and the leftmost is -24, numbers below -24 give DATA ERROR.

Only negative values of X can be used to put a character to the left of the first character currently in the Alpha register. An example of the use of YTOAX will be given in the next section.

POSA will let you find a selected byte in the Alpha register and will return its position using the same numbering scheme as in i. above. You can follow this immediately with nnn, X<>Y, YTOAX to replace the selected byte with a byte value nnn, thereby altering an NNN.

ANUM lets you convert an ASCII number (a text string) in Alpha back into a BCD value (number that you can do arithmetic with) in X. ANUM is rarely used in SP, but see APX in the next section. The XIO function ANUMDEL extracts a number from Alpha, then deletes it. This lets you take several numbers from the Alpha register which you can therefore use as a temporary store for numbers separated from each other by a non-numeric character. This can be quicker than data packing, specially if the numbers only have a few digits each. ANUMDEL has no equivalent in the Devel.

The Paname module (see Section 12.8) provides all the Alpha handling functions of the XIO and Devel modules, including ANUMDEL. If you do not need other functions from the XIO or Devel then you may prefer the Paname.

16.3 Flags and numbers

X<>F is useful not only for providing more flags, but also for synthetic operations. For a start, CLX, X<>F rapidly clears flags 00 to 07 so you can then use them to build up a byte. X<>F, X<>F will normalise a number in X, remove its sign and fractional part, without changing L - but the absolute value of the number must be smaller than 256.

If you have used Alpha operations as above to extract a byte from an NNN then you can use X<>F to put this byte into register d and study or change the individual bits of that byte. One example is that of turning upper case letters into lower case or the other way round. Upper case letters begin with a nybble 4 or 5, lower case begin with 6 or 7, so it is only a matter of setting or clearing bit 2. To turn a string of upper case letters in Alpha (assuming there are only letters in Alpha) into lower case you can use the short routine:

```
ALENG, LBL 01, ATOX, X<>F, SF 05, X<>F,
XTOA, RDN, DSE X, GTO 01, RDN, RTN
```

The routine finds the number of characters in Alpha, then uses a loop to change each character and put it back. At the end the registers X, Y, and

L are left unchanged. An improved version of this routine can be used to make printed text more legible than all capitals, or it can be used to make text illegible on an HP-41 display since lower case letters are mostly illegible in the display.

Note that flag 05 had to be set to set bit 2 of the letter. This is because X<>F converts bits to flags starting at the right, whereas ATOX and XTOA begin at the left. For example the byte A3 (decimal 163) is turned into two different bit patterns by XTOA and X<>F.

163 , XTOA gives 1 0 1 0 0 0 1 1 (A3 hexadecimal)

163 , X<>F gives 1 1 0 0 0 1 0 1 (C5 hexadecimal)

You can see that X<>F produces a mirror image of the normal bit order. If you want to reverse the order of flags 00 to 07 you can use one of these routines:

```
-7                    RCL d
ENTER                 X<> M
CLX                   SIGN
SIGN                  X<>F
LBL 01                -7
FS? IND Y             YTOAX
ST+ L                 X<> L
ST+ X                 X<> M
ISG Y                 X<> d
GTO 01                R↑
LASTX                 R↑
X<>F
R↑
```

The nonsynthetic version on the left is 20 bytes long; it leaves Alpha and X unchanged but is rather slow because it uses a loop to do the same flag trick as RAB. Note the use of -7 combined with ISG so that flags 00 to 07 can all be tested; 7 with DSE would only work for 07 to 01 and would skip

flag 00. The version on the right is 19 bytes long; it leaves X, Y and Alpha unchanged, but requires the XIO function YTOAX (you can replace this with the Devel function Y-AX). Other flag mirroring techniques by A. van den Brug, F. de Vries and M. Markov were described in PPCJ V12N1P24.

RCLFLAG produces a text string which contains flags 00 to 43 preceded by the three nybbles 1FF. This makes it impossible to ARCL the string and ASTO it again as the 1FF would become 10F. If flags 00 to 03 are all set then the second byte of the string is FF and (due to a minor display bug described in Appendix C) register X appears to be empty. This does not affect the flag values themselves.

RCLFLAG and STOFLAG can be used for various synthetic purposes. On the simplest level they provide an alternative to RCL d and STO d. Another simple use is:

RCLFLAG, 0, STO d, RDN, STOFLAG, RDN

This leaves flags 44 to 55 cleared - it is a simple way to clear flag 55 and so disable the printer, but you should clear flag 21 as well.

Since RCLFLAG pushes register d 3 nybbles to the right it can be used in cases where an NNN has to be shifted by an odd number of nybbles. One example would be moving register c to produce an alternative Σ CX routine.

You may have wondered why Alpha entry and editing can be restarted from the keyboard by the APPEND function (ALPHA, SHIFT, K), but numeric entry cannot be restarted. An APPENDX function would be useful in several cases:

i. You have entered a long number and made an error in one digit.

e.g. you do: 1.23456789E25 , ENTER

Then you realise you wanted 1.23456709E25

If you could press APPENDX then you could delete the 89E25 and replace it with 09E25. As it is you have to press CLX and start again.

ii. You see a result with the last two digits hidden by an exponent and you want to see those digits.

e.g. a program displays the result 2.7182818 10

Could this be e x 10^{10} ?

If you had APPENDX you could press it, backarrow three times, and see the whole mantissa. It could be:

2.718281895

Not only is this result different from e (2.718281828 to ten places) but also the last two digits are well over 50 which you could not tell as the HP-41 does not round up digits hidden by the exponent.

iii. You are using a polynomial expansion program and want to "tweak" the last digit of a parameter to see what value gives the best fit. (This is also used to overcome rounding errors when writing polynomial expansions.)

e.g. You want to try changing the last digit of PI/2. You would do:

PI, 2, /, APPENDX

and you could change the last digit at once instead of entering the whole number (and easily getting one of the digits wrong).

iv. You want to change the sign of the exponent without altering the mantissa of X.

e.g. Change 1.4793 E12 to 1.4793 E-12 by pressing APPENDX, CHS.

Digit entry can be restarted by having flag 45 set but registers P and Q must contain the right values too. This can be arranged by means of the ANUM function as in the program APX here.

```
01*LBL "APX"
02 VIEW X
03 "DBLI "
04 X<> [
05 X<> \
06 RDN
07 RCLFLAG
08 ASTO d
09 VIEW [
10 ASHF
11 ARCL [
12 STOFLAG
13 RDN
14 7
15 AROT
16 RDN
17 ANUM
18 END
```

The program is fairly easy to enter except for line 03, which is the 6-character text string F6, 44, 42, 4C, 49, 84, 20.

APX does not alter your display mode but finishes with all the digits of X displayed and numeric entry is enabled. None of the stack registers or L are changed, so you can use APX in the middle of a keyboard calculation, then carry on. The Alpha register contains the number in ASCII form and in FIX 9 mode, followed by the original X value in register M. This means that you can edit the number, then view the original value, or recover the original value by doing X<>M. APX works by temporarily storing an NNN in register d, displaying X in FIX 9 mode, then restoring flags 00 to 43. Flag 45 is left set, flag 55 is left clear to prevent a printer from changing register Q. The other operations keep the stack and Alpha in order.

Try assigning APX to a key, then going through examples i. to iv. above. (I assign APX to key -32 since this is the same key as is used in ALPHA mode for APPEND.) If you have anything assigned to the numeric entry keys then you will have to be out of USER mode after using APX. You can press USER after executing APX, or include CF 27 after line 02 of the program. When you use APX do not press any keys until the PRGM annunciator is

cleared: number keys pressed before APX finishes will be ignored, while pressing R/S will stop the program before it has completed its job.

APX is an excellent example of what Synthetic Programming can do. It extends the HP-41 by providing a useful new feature that takes little time and does not use much memory. You can use the HP-41 without this feature but it makes life easier. You are not redesigning the whole HP-41, just making it more helpful.

16.4 Registers, keys and programs

This section will cover synthetic uses of the Extended Functions which control registers and assignments. To begin with consider the use of curtain raising to renumber data registers. This was suggested in Section 14.9 as a way of letting two subroutines both use the same registers, say 00 to 09. REGSWAP does this more easily, by exchanging the register contents instead of renumbering the registers. Be careful not to move blocks which overlap (unless you are doing this on purpose to rotate or shuffle a block) - the results of moving overlapping blocks were covered in Section 10.3.

SIZE? is not likely to have synthetic uses, but it replaces routines that are slow or require synthetic functions. SIZE? is surprisingly slow (for an M-code function) because it uses an internal HP-41 operating system function to check each data register until it comes to a nonexistent one. Fortunately this operation does not normalise anything.

PSIZE was never achieved by Synthetic Programming alone. Note that it is faster to use PSIZE than SIZE from the keyboard, as CAT 2 is searched before CAT 3.

GETKEY (and GETKEYX) can save you the need to make temporary synthetic key assignments. If you need to use RCL d ten times from the keyboard it might be quicker to write a short routine LBL 34, RCL d, RTN by using the Byte Grabber than to find a synthetic assignment program and assign RCL d to a key. You could write other routines that perform synthetic operations like this, and use a single control routine to execute them:

LBL "XKEY", "KEY?", AVIEW, GETKEY, RDN, GTO IND T

Assign XKEY to a key as well and you can push that key, then key 34 to execute RCL d, or another key to execute some other synthetic function you need. A few will not work in a program; for example RCL b will recall its own address. You can still assign other functions to the same keys as well, so GETKEY effectively increases your total number of key assignments. If you want to use a lot of synthetic functions from the keyboard you could assign control programs to several keys, and each control program would redefine the entire keyboard - say one program for synthetic RCLs, and another for synthetic STOs.

One use of GETKEY is to prompt the user for a key to be used by PASN. Many synthetic assignment routines were written before PASN became available, but these are still necessary because PASN does not assign synthetic functions. However PASN can be used to help make such key assignments more quickly and without affecting any buffers. GASN (see Chapter 11) uses PASN, and like PASN it cannot make assignments to the SHIFT key. GASN only uses byte codes to identify the function prefixes and postfixes to be assigned, not function names, but it contains no synthetic functions. Some extremely clever assignment programs have been written to use Alpha names for synthetic functions, just like PASN. The programs by T. Tarvainen and G. Westen in DATAFILE V2N3P11 and in the book "HP-41 Extended Functions Made Easy" are an example. An alternative way to use Extended Functions in key assignment programs is to write very short assignment programs. A good one by J. Franklin in DATAFILE V2N1P12 fits on one side of a magnetic card.

CLKEYS can be used with PASN. It is more dramatic than storing zeroes in the key assignment flags to disable the assignments. However it recovers all the space used by assignments, without affecting any buffers.

PCLPS belongs with this group of functions as it too clears registers and makes more space available. A useful trick is to call a subroutine from a program, PCLPS the program, then return to it. The .END. is moved up by PCLPS so the return is to the free area below the .END. - this is considered to be a Good Thing by many users of synthetics. One example is GASN which uses this method to fall into the alarm buffer. The setup program GASETUP has put a synthetic routine into the alarm buffer, and this is executed to make a key assignment.

If you look at GASETUP you will find that it uses non-synthetic methods to create a text string whose bytes make up the routine:

RDN, X<>c, GETP, /, RCL N, X<>c, RCL M, STO 00

It then sets an alarm with this as its message. The bytes are rearranged by XYZALM so the routine executed by GASN in the buffer is:

RCL N, X<>c, RCL M, STO 00, RDN, X<>c, GETP, /

The GETP at the end retrieves the program GASN from Extended Memory and restarts execution of it from line 01, where GASN stops or returns to the program which called it. If an error occurs and flag 25 is set then GETP will be ignored and the alarm itself will be executed as program steps. To avoid this, GETP is followed by / which attempts to divide Y by the value in X. This results in an ALPHA DATA error and stops the program.

As well as creating this text string which is later executed, GASN creates an alternative register c in register N, and a key assignment register in register M. All of this can be done non-synthetically by using XTOA and appending characters to Alpha. Register O contains the program file name "GASETUP"; this name is used by PCLPS to clear the program and later by

GETP to retrieve it. The file name is seven characters long so that none of the characters in N are used in this name. The contents of register N are:

1D, 1D, 31, 69, 0C, 01, 01

This means that register 00 is at address 0C0, so that STO 00 puts the new key assignment there. 169 is in the right place and Extended Memory lies immediately below register 0C0 (we shall come to this in the next section), so GASN can be safely interrupted or single-stepped.

16.5 Understanding Extended Memory

The normal user can treat Extended Memory as a single block containing 127 or 365 or 603 registers, depending on how many Extended Memory modules are plugged in. At least two registers are used as file headers, and one is used to contain an end-of-data marker, so a maximum of 600 registers can be used to store data in files. Synthetic programmers need more details so as to use these registers for other purposes. These details will be given here before any programs. Extended Memory fits into the HP-41 RAM memory layout as shown in Figure 16.1. If you compare this with Figure 8.5 you will see that Extended Memory has been put below and above the Main Memory. The 127 Extended Memory registers in the Extended Functions module (EFM) lie below Main Memory, the registers in the two Extended Memory (EM) modules fit in above Main Memory.

Each block of EM is connected to the others by a **link register** at address 040, 201, or 301. The total number of Extended Memory registers in the EFM (or the HP-41CX) is 128, which is a multiple of 16. Why do the Extended Memory modules not have 256 registers each?

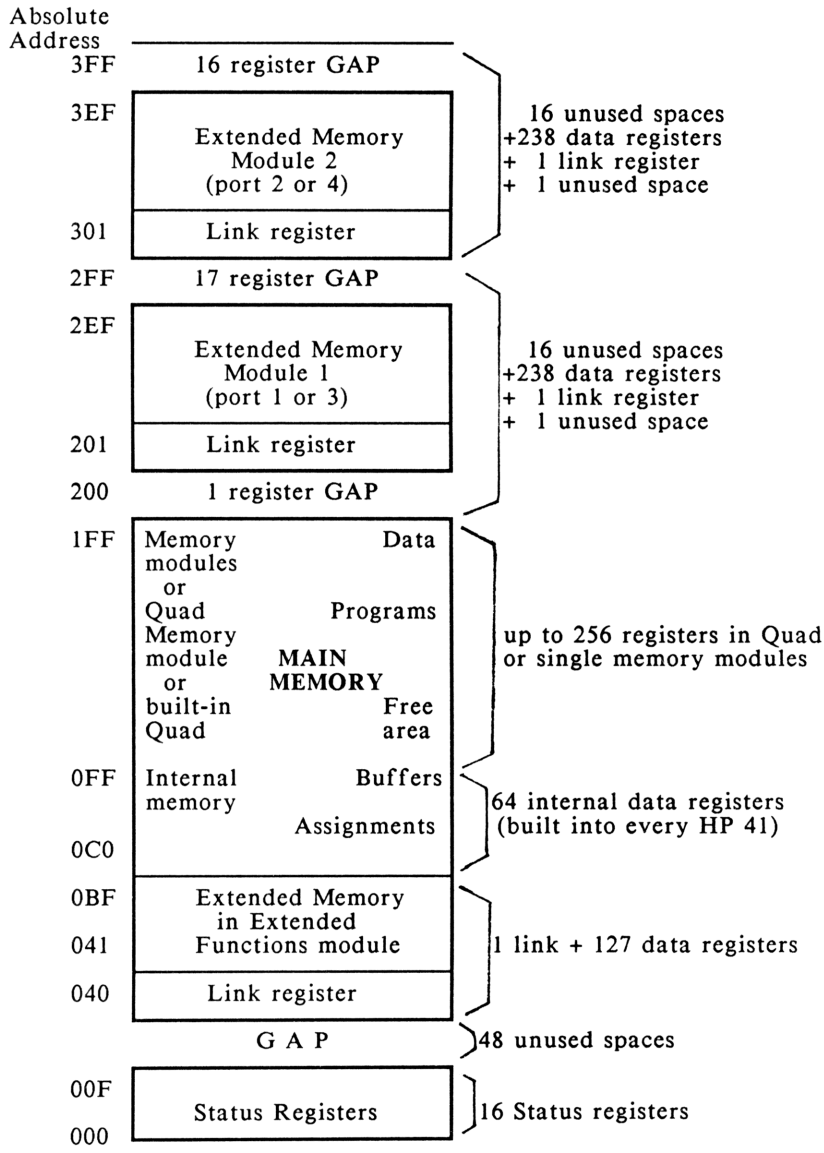


Figure 16.1 HP-41 RAM memory including Extended Memory

Whenever the HP-41 needs to address RAM built into a peripheral device (for instance send messages to the printer or the display) it must **select** the peripheral so that data will be sent to it. This **de-selects** the RAM in all other peripherals so data will not be sent to them. However the RAM in the HP-41 itself can never be de-selected. Anything sent to a peripheral is also sent to the HP-41 RAM. To avoid writing to data registers when it is addressing a peripheral the HP-41 must select an area of RAM that is not used. RAM is selected in blocks of 16 registers, so a single unused register will not do - the block most commonly selected is in the gap above the status registers, but other areas were used sometimes. The Card Reader selects the top 16 registers 3F0 to 3FF, so these cannot be used to hold data, as they would be overwritten by Card Reader operations. (In fact one Card Reader function uses the next lower block of 16 registers, see the VER bug in Appendix C.) Register 200 must also be absent, to provide a break between Main Memory and Extended Memory. All Extended Memory modules have to be the same so that they can be plugged into any port, so they must all be without the bottom one and top sixteen registers. When the display driver synchronization bug (see point xii. in Section 4.1) was fixed, the block of registers 2F0 to 2FF was selected as an unused area, so even if you do not use a Card Reader you must be prevented from using this block.

To allow all of Extended Memory to act as if it were a single block the link registers connect the addresses in each module to the next one. They are laid out as shown in Figure 16.2.

Nybbles	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	U	U	C	C	C	B	B	B	N	N	N	T	T	T

Figure 16.2 Extended Memory Link register contents

The fields T, N, B, C all contain three-digit hexadecimal numbers:

T = Address at top of this module, 0BF or 2EF or 3EF.

N = Address of next module, 2EF or 3EF or 000 if no next module.

B = Bottom address of module before this one, 040 or 201 or 301. The first block (in EFM) has no previous module - B contains either zero or the current file number after EMDIRX, or after EMDIR (or CAT 4) has been executed on an HP-41CX and allowed to finish.

C = Current working file number. If you are using the second file in EM then C contains 002. Only two digits are normally needed, but all three are used. C is only used at address 040, it is left clear at 201 and 301.

U = Unused but useful. The EFM puts 00 in here when Extended Memory is initialised (e.g. after MEMORY LOST) but does not use this field. If you want to recall this register (to check the current file number) you should put the byte 10 here, but you must get the fields N and T right as well. This will avoid normalisation when you recall this link register, but must be done carefully (John Franklin suggested this in DATAFILE V1N3P28d).

The link fields are needed to tell the Extended Memory functions where to go if any operation moves from one block of EM to another. One important use of these fields is to tell which port an Extended Memory module is plugged into if only the EFM and one EM module are plugged in. When you

first plug in any Extended Memory the link registers might contain anything. They are initialised as soon as you do an EMDIR, but not all the fields are set as described above. Only when you create a file that uses part of an Extended Memory module will all its link register fields be set. The field N in register 040 points to 2EF if both EM modules are plugged in when the pointer is initialised. If only one EM module is plugged in then N in register 040 can contain either 2EF or 3EF; 2EF if the module is in port 1 or 3, 3EF otherwise. The field N in the first EM module will contain 000 if there is no second EM module, otherwise it will be initialised to point to the other module when that comes into use.

All the above mean that you should store one of the following strings of nybbles into register 040 if you want to be able to recall it later:

10,001,000,000,0BF if there is no EM module, or if you have not yet used an EM module.

10,001,000,2EF,0BF if there is only one EM module and it is plugged into port 1 or 3, or if there are two EM modules which were plugged in at the same time, or if there are two EM modules and one was plugged into port 1 or 3 first.

10,001,000,3EF,0BF under any other circumstances.

If all the fields are initialised in all three modules, then the middle one is removed for a short time, the fields are re-initialised, and the module which was in the middle is replaced then it will be outside the link between the first and last module, and you will not be able to use it. This is explained in Appendix C.

Each file in Extended Memory has two header registers. The first contains the file name, up to seven characters long. Names of less than seven characters are filled out with blanks (not nulls) at the right. Names with commas are illegal, since commas are used as separators in the Alpha register, but such names can be created synthetically to make a file

inaccessible. This is rather like having a program with no global label - the remedy is the same; use EMDIR or EMDIRX to make this file the current one. You will not be able to PURFL a file with a name containing a comma, except by clearing all of Extended Memory. A name consisting of seven FF bytes is also illegal, and dangerous - see below.

The second header register contains status information arranged as in Figure 16.3.

Nybbles	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	T	U	U	U	U	C	C	C	R	R	R	S	S	S

Figure 16.3 Extended Memory file status header

The fields T, U, C, R and S all contain hexadecimal numbers:

T = file type - 1 for program files, 2 for data and 3 for text files. Other file types are not recognised by the EFM but can be used by different modules. The CCD module uses 4 for matrix files, 5 for buffer files and 6 for key assignment files. The Advantage module also uses 4 for matrix files.

S = file size. This is the number of registers in the file, not including the two header registers. The total file length is obtained by adding 2 to S; if you put FFF in this field then the total file length will be interpreted as 1 register so the second header word of this file will be treated as the name of a file that comes next, and the first data register of this file will be used as if it was the file status header of the next file. Each file contains a whole number of registers so S is an integer.

R = record pointer. In data and text files this gives the number of the current record (the first record in a file is number 0), in program files it contains the program size in bytes.

C = the character pointer in text files. It holds the number of the current character in the record pointed to by R; the first character in a record is number 0. Only two hexadecimal digits are needed because records cannot be over 254 bytes long, but three nybbles are used.

U = four unused nybbles. The absolute address of the register last used is often put in the leftmost three nybbles by operations on text and data files, but it is only left here after the operation has finished so you will not gain anything by putting values in here. You can use this value to check which register has been used if you want to study or alter Extended Memory.

You still need to understand file contents before you can begin to manipulate EM files. Data files are easy; each register contains zero when the file is created or cleared, otherwise it contains a seven byte value copied to it from X, from a data register, or from an external device. Each register in a data file is treated as a separate data record. Program files are easy too; they are copied from program memory byte by byte, including nulls (nulls that come before the first step of a program are copied too), compiled GTOs and XEQs, global label key assignments and even the .END. or END. The first byte of the program is copied to the first byte of the file (byte 6 of the first register after the second header). One extra byte is put in after the end of the program; this contains a checksum (the sum of all the program bytes modulo 256). It is not included in the byte count, but can add a whole register to the file length if the program itself contains a number of bytes which is an integer multiple of seven.

Text files consist of records each of which can contain from one to 254 characters. The first byte of each record gives the number of bytes in

that record. This header byte is not included in the calculation of that number. A text record which contains "FRED" will be made up of the bytes:

04, 46, 52, 45, 44
length=4 F R E D

A text record header byte of 00 could only precede an empty record, so it is not used. A header byte of FF marks the end of a file - the file can expand into any space after it. If you delete one or more characters from a record then the end-of-file mark is copied to its new position but the data after it are not changed. This means that deleting six characters one at a time will make six copies of the FF byte, producing a string of seven FFs. If you delete the last record in a file by using DELREC (or by using DELCHR on a record which contains only one character) then the record header is replaced by FF but nothing else is changed.

Just as a text file is terminated by an end-of-file byte, the last file in EM is followed by an end-of-memory mark, which is a register containing seven FF bytes. If the last file in EM is deleted then this marker is just inserted after the previous file, leaving all other data undisturbed. If any other file is deleted then all the following files, and the end mark, are copied up, but the data past the new end mark are left unchanged. If all files in EM are lost (for example because register 040 is altered) then an end mark is put at the top of EM but nothing else is changed. This means that you can recover the data in EM just by putting a file name into register 0BF. Similarly you can recover the last file, if it was deleted, by putting a file name into the register immediately after the end of the previous file.

A simple way to access all the registers in Extended Memory so that tricks like this can be played is to change the status register of the first file in EM, making it into a data file with a length of FFF. This trick, invented by Clifford Stern, lets you recall any register from EM, just by using SEEKPT, GETX.

The end-of-memory mark is not unique. One way of making a string of seven FF bytes is to delete six characters, one at a time, from a text file as described above. You could also create a text string containing seven or more FF characters by using APPCHR or INSCHR. Synthetic programs with text strings containing seven or more FF bytes can be saved in program files. Any of these operations could produce a register in EM containing seven FF bytes. Furthermore NNNs made of seven FFs can be saved in EM by use of SAVEX. The original version of the EFM would not check if a string of seven FFs was at the end of a file but would stop copying, for example during a PURFL, as soon as it came to any such string. This is a bug, and can lead to further bugs described in Appendix C.

Extended Memory operations may seem to be slow. This is because they check the integrity of Extended Memory every time they access a register. One reason to make this check is that EM is not continuous, so addresses have to be altered if a file is partly in one module and partly in another. Another reason is that an EM module may have been removed since the previous EM operation. Nevertheless, the checking is rather too thorough, involving going through all files and then through the registers in the current file (the "working file") to locate the current record. For most EM operations this could have been replaced by a much faster test, for instance field C in register 040 could contain the current address (field U could contain the current file number), and it could just be checked to see if it is still there. It is too late to change the Extended Functions, but new modules could use methods like this. One part of the search is that the current file is looked for by name, not just by the file number in field C of register 040. The name is held in a CPU register which can be altered by other HP-41 operations, such as STO. If this register contains an NNN which is a file name then the original version of the EFM might stop at the file whose name is stored, and then try to use that file instead of the current file. This rarely happens, and can be gotten around by setting flag 25, and STOring something that is not a file name, then trying again if flag 25 is cleared. The current file number is still saved so a second search should work. Newer EFMs and the Extended Functions in the HP-41CX check both the file name and the file number so this problem cannot arise.

(My thanks to Bruce Bailey who helped me understand these EFM problems.)

16.6 Manipulating Extended Memory

Having explained the organisation of EM I shall now provide a few more examples of manipulating it. The subject is too complicated to cover fully in a chapter, but the rest of this section will serve as an introduction. The example programs show the sort of thing that can be done. The book "HP-41 Extended Functions Made Easy" goes into more detail and provides some important programs for synthetic programmers who want to use EM. Mass Storage will be mentioned under points 3 and 4, but only briefly.

1. Clearing Extended Memory

There is no single function to clear all files from Extended Memory. On a 41CX you can do 1, EMDIRX, PURFL repeatedly until an error occurs, showing there are no more files. The PURFL bug on the older modules (which show up as version 1B in CAT 2) lets you clear all EM by doing "filename", PURFL, SEEKPT. ZENROM provides the function CLXM to do the job.

You can clear EM very quickly by altering the value in register 040:

4, X<>c, CLΣ, X<>c, RDN

This uses just seven bytes, alters only register T and is faster than CLXM. It is sufficiently short that it can use an "unsafe" value in register c, but of course it must not be single-stepped. It leaves the contents of EM unchanged, except for clearing absolute registers 040 to 045. If you have no EM then the CLΣ will halt the program (unless flag 25 is set) and so give MEMORY LOST. CXM below avoids all these problems: it can be single-stepped, stores zeroes in all of EM except the first two registers and the link registers, and it does not give MEMORY LOST if you have no EM.

```
01♦LBL "CXM"
02 SF 25
03 "α♦×iß♦♦"
04 X<> [
05 X<> c
06 CLΣ
07 X<> c
08 X<> \
09 RDN
10 ENDIR
11 CRFLD
12 PURFL
13 X<> [
14 CLA
15 END
                39 BYTES

Line 03 is:-
F7,04,00,01,69,05,00,00
```

This works like the first version, by putting the statistics block at register 040 and using CLΣ to clear 040, but it uses an acceptable value of register c. Then it creates an empty data file which fills the whole of EM, and is automatically filled with zeroes. Finally it purges this file, leaving EM cleared and free. None of the stack registers are altered, and Alpha is cleared. The use of CLΣ to clear EM was suggested by A.M.M. Cañas in PPCJ V11N8p31-32.

2. Using Extended Memory as normal memory for data or programs

Figure 16.1 shows that there is a continuous block of memory from 040 to 1FF. If you have an EFM or HP-41CX you can lower the .END. to register 040 and have a total of 448 registers for use as data and program space. If you want to rewrite a program that used 300 registers to use 400 then it can be easier to do this than to rewrite the program to use EM data files.

This trick is particularly useful when you want to use more than 300 registers with a program in a plug-in module. You cannot rewrite the Maths module to use EM for example, and the normal memory limits the Maths module matrix operations to 14x14 matrices if you have three memory modules, or 16x16 matrices with a quad memory module, a 41CV, or a 41CX. Should you wish to use slightly larger matrices you can lower the curtain to address 061 and have room for matrices up to 19x19. A 19x19 matrix can take almost 20 minutes to invert with the Math matrix programs so it would be unwise to go further even if it were possible.

Some precautions are obviously needed. You need to take great care to protect any EM files, assignments, alarms or other buffers as they can be overwritten if you use their area. It is safest to delete them all. If the .END. is below register 0C0 then the HP-41 will not let you lower it, so you should not PACK or GTO.. until you have done all the program editing you want to. PSIZE will let you move the curtain in both directions though. It is safer to leave register 040 unused and to put the .END. in register 041, so that you will see .END. REG 00 to remind you not to move the .END. . You must never execute EMDIR or CAT 4 as this might put an end-of-memory marker into register 0BF, in the middle of your program.

Here is a step-by-step procedure for setting up 414 data registers (enough for a 19x19 matrix) and 33 program registers (enough for a program on two sides of a magnetic card).

i. Make sure you have STO c available either assigned to a key or inside a program in a ROM. No other synthetics are required.

ii. Clear Extended Memory registers 041 to 060. These registers will be used to store programs so they must be clear. You can use one of the methods described above, or a function from a ROM (PPC ROM, ZENROM, CCD module, or an EPROM set).

iii. Execute EMDIR or CAT 4 to re-initialise register 040.

iv. Put the number 4.100169041E40 into register X, then execute STO c. This makes absolute register 041 into data register 00.

v. Do 1E20, STO 00. This simulates a packed .END. in register 041.

vi. Do 6.200169062E41, STO c. This puts the first statistics register and register 00 at absolute register 061 and it puts the .END. at register 041.

vii. Execute CAT 1. This gets you into the new program area. RCL 00 is displayed instead of the .END. because at present there is just a byte 20 in register 041 (put in at step v.) instead of a real .END. .

viii. GTO.000 to get to the top of the program area. You can now go into PRGM mode and enter a program. Begin with a global label; this will cause the bytes in register 041 to be turned into a proper .END. . You can also read a program from a card or via HP-IL as this too will set up an .END. .

ix. Execute CLRG to clear data memory, and CLKEYS to make sure no data will be interpreted as key assignments.

You now have 414 data registers (and 33 program registers). You can check this with SIZE? and change it with SIZE or PSIZE. You can enter programs into the program area, but if you GTO.. or PACK then you cannot expand this area again later. Now write a program to use the Math module routines for up to 19x19 matrices, or execute them from the keyboard.

Another trick is to run programs that are in program files without copying them back to main memory. To do this you need to change register b so that it points to the required program in EM. This is easy if you know where the program is, for example if it lies at the very beginning of EM. Imagine you have a large program which needs a lot of data registers. You test the program on a few registers, then clear EM (see above) and SAVEP the program. If it is less than 124 registers long then it will all lie in the EFM (the part of EM that begins at absolute register 0BD). You can now delete the program from Main Memory (use CLP or PCLPS) and increase the SIZE to a much larger value. Now go to the program by storing its first address in register b:

CLA, 190, XTOA, RCL M, STO b

Press R/S to run the program. If the program file is not the first one in EM then you must work out the absolute address where the program begins (remember this is two registers below the beginning of the program file)

and store this in register b; ways to do this have been the subject of many articles in user group journals. A program in EM will only run correctly if it all lies in the same block of registers, so you must make sure that it does not get saved partly in one block of EM and partly in another. If you do not have any EM modules then either the program will all be saved in the root area, or it will not be saved at all. If you have one or two EM modules as well then do an EMDIR (or EMROOM) and if there are more than 237 unused registers then create a dummy file to leave only 237 registers.

Uncompiled backward jumps will work incorrectly if the program was the first one in CAT 1 before it was saved, because the HP-41 will search down to the END and will then go to the curtain and restart the label search there. Placing a dummy END at the beginning of program memory before SAVEPing the first program in CAT 1 will avoid this problem, as the program's END will then give the distance back to the beginning of the program, even if it is in EM. If you run a program anywhere in EM with some uncompiled forward jumps then these will work and will become compiled, but the program checksum will be probably be changed so you will not be able to read the program back into Main Memory. The same will happen if you run any program with reverse jumps, unless it was the first program in CAT 1. The program "RPF" in "HP-41 Extended Functions Made Easy" corrects the checksum if something has happened to it. XEQ of global labels in main memory and programs in ROMs will work if your program is in registers 0BD to 041, but any return to an address above register 1FF will fail because of the way that return addresses to RAM are stored (explained in Section 15.4). For the same reason an END in registers above 1FF will terminate program execution but will leave the current address at the wrong place. Use RTN to stop a program at an address above 1FF if you want to be able to restart it. RTN does not try to alter the return stack if the first return is 00; it just stops the program.

3. Saving Extended Memory files.

There is no simple way to save EM files on magnetic cards or on Mass Storage devices, unless you have the Paname module whose functions WRTEM

and READEM let you save all of EM on Mass Storage and read it back (in the same way as the HP-IL module functions WRTA and READA let you save and read all of main memory). Otherwise there is only SAVEAS which copies text files to Mass Storage. The slow and safe method of saving other files is to copy them back to Main Memory and then to save them on cards or Mass Storage, provided you have enough room in Main Memory. This is not too bad for programs, which can be copied to Main Memory with a single instruction, and then to cards or Mass Storage with another. Data files can be copied from a file "name" to Main Memory and then to cards with GETR as follows:

SIZE?, "name" ,FLSIZE ,X>Y? ,PSIZE ,E3 ,/ ,GETR ,WDTAX ,RDN, RDN

You can use the same method for copying to Mass Storage, but create a data file and copy to it with WRTRX. Text files must be copied to the Alpha register 24 characters at a time, then to data registers by means of ASTO IND X, ASHF, ISG X, etc. and each end of record should be recorded in some way too. The data registers used must then be copied to cards or Mass Storage. This is a slow process, and it can lose null characters which is another reason why synthetic methods are sometimes preferred. Reverse the methods described above when you want to read a file back into EM.

The synthetic method is to move the curtain so that the file to be copied appears to be in Main Memory, then use WDTAX or WRTRX to write out the file registers directly (and later RDTAX or READRX to read them back). Some care is needed to prevent the program stopping (for instance because the Card Reader makes the batteries go flat) before the original version of register c is restored. Various programs to do this have been published, one is in DATAFILE V1N3P40, and several are in the book "HP-41 Extended Functions Made Easy". Newer and better versions are published from time to time in user group journals. The Card Reader and Mass Storage operations save and replace registers without normalising them, except that the first and last registers of a block are sometimes normalised. To read back the files you must reverse the process.

A third method is to change the file type. If you make all of EM into one large data file (as mentioned in the description of the end-of-memory marker above), then you can find the file header, and temporarily change a program or text file into a data file. Then you can read the file into the main memory data registers and save it. As the first and last registers may be normalised you could put the file length, (and its type as a fraction added to the length) into them. It does not matter if this is normalised, and you might need the length anyway. Once again reverse the whole process to read the data back, then change the file type back to what it was.

4. Extended Memory and Mass Storage

One more way of saving program and data files is to change their file types to "text" and use SAVEAS to write them directly to Mass Storage (and GETAS to read them back). This is a quick but dirty method; SAVEAS copies text records until it comes to one whose header is an FF byte. You will need to take precautions against stopping too early at an FF byte or going past the end of the file without finding an FF byte. The best thing to do is to create files just long enough to hold the file you are copying to or from EM. You will also have to save and restore the file status header.

The above is one example of Synthetic Programming with Mass Storage. In some ways Mass Storage can be treated as an extension of Extended Memory, and SP can be used to study Mass Storage files just as it was used to study EM files. An important application of this is to recover data and directories on damaged HP-IL cassettes or disks. That is really beyond the scope of this book. Programs to repair cassette directories require an Extended I/O module (one by C. E. Reinstein appeared in PPCCJ V10N10p9-11) or a Devel module (see program by M. Backe in PPCCJ V11N1p59-60). A set of HP-41 Mass Storage Utilities by M. Markov is appearing in the CHHU Chronicle in serial form, beginning with CHHU C V1N3p12-14.

5. Other tricks

Two other synthetic uses of Extended Memory are to save assignments of keys in EM and to store alarms or other buffers in EM files. Once these are stored in EM they can be removed from Main Memory, to save space, or so that key assignments can be edited without affecting any buffers. Programs to do this are in journals and in "HP-41 Extended Functions Made Easy".

You can use SAVEP to save a program in EM and then find its length in bytes by using RCLPT. What can you do if there is not enough room for the program in EM? Even if SAVEP does not work it saves the program length in bytes in nybbles 5, 4 and 3 of register Q, so you can get the length from them. SAVEP also saves the program length in registers in nybbles 2 to 0 of Q. Perhaps the best feature is that it saves the address of where the program begins in Main Memory in nybbles 12 to 9. This lets you alter, analyse, or print the program without using the copy in EM.

Extended Memory and the Extended Functions provide some interesting places for the synthetic programmer to hide data, particularly non-normalised numbers (in addition to registers M,N,O,P and a few other status registers available to every user of SP). Values stored in data files can be recovered without normalisation by GETX, but the file header registers provide extra room for a few bytes, and even the file name can be a NNN; on an HP-41CX this can be recovered by EMDIRX. Similarly values can be put in the message saved with an alarm and can be recovered by RCLALM. Numbers can be stored for a short time in the time and date if you do not use these, but remember that they will be changed continuously. A number between 99.9 and -99.9 can be stored in the clock accuracy factor and recalled to an accuracy of one decimal place, but this should only be used if you do not use the clock. A much better way to store a number like this is to use X<>F to store an integer between 0 and 255 in flags 0 to 7.

16.7 Understanding buffers and a programmable PACK

In computer jargon a **buffer** is an area of memory where data are stored temporarily on their way between a computer and a device like a printer or Mass Storage. This buffer prevents the data from running into the computer's main memory and causing trouble, rather as a railway buffer prevents trains from running onto the platforms. On the HP-41 the buffer area is used by some modules to store temporary data as well. For example the Plotter module uses a buffer to store information being sent to the plotter. The buffers can also be used for any information that belongs to one particular module and which can be deleted when that module is removed. For example the alarms cannot be executed if the Time module is taken out. The alarm buffer is therefore deleted if there is no Time module, so that the buffer area becomes free for other purposes. This is annoying if you take out the Time Module for a short time, and you may want to prevent deletion, for example by storing the alarms in EM.

Each buffer has a header, followed by data (in higher-numbered registers), and some modules put an extra register at the top of their buffers too. The buffer header is arranged as shown in Figure 16.4.

Nybbles	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	X	X	N	N	P	P	P	U	U	U	U	M	U	U

Figure 16.4 Buffer header structure

The two hexadecimal digits XX identify the module which owns the buffer. AA identifies the alarm buffer, BB identifies the Plotter module buffer, CC is the HP-IL Development module buffer, EE is an Advantage module buffer, and 55 identifies a CCD module buffer. Other buffers may be used in future.

NN is the length of the buffer in registers, including the buffer header, and the buffer end register if there is one.

The Alarm buffer does not use any other fields; they all contain zeroes. The HP-IL Development module buffer uses PPP as a pointer to a selected byte, and M as a marker set to 1 if the buffer pointer is incremented automatically every time a byte is examined, or 0 if the pointer is not incremented. Other buffers can use this area in various ways. The areas marked U are not presently used, but some future buffers may use them.

Whenever the HP-41 is turned on or packing is carried out the area beginning at register 0C0 and ending at the .END. is checked. (If you set the .END. pointer below register 0C0 then this search will not be carried out, and you can put what you like in this area, or remove modules without losing any buffers. Time module alarms will not activate either.) Registers with F0 in nybbles 13 and 12 are treated as key assignments and removed if they are unused. Registers with anything else except 00 in these two nybbles have them changed to 0X, and the HP-41 skips to the beginning of the next buffer. A register with zero in it is treated as the end of the buffer area and the checking is finished. (Another way to protect buffers from deletion and to prevent hang-ups is to put a zero register below them.) Then each module plugged into the HP-41 is allowed to check for any buffers that belong to it and to reset the 0X back to XX. Finally any buffers left with 0X at the beginning are deleted. If NN contains 00 then the buffer length is zero, so the next address checked is the same as the one that has just been checked, and the HP-41 hangs up.

Above the header are the buffer contents. These depend on the needs of the module which made the buffer. The Time module puts one or more alarms into the buffer. Each alarm begins with a single register which contains the alarm time and date, stored in nybbles 13 to 3 as a binary number of tenths of seconds since the first of January 1900. Nybble 2 contains a 1 if the alarm is a repeating alarm, nybble 1 contains an F if the alarm is activated past due, and nybble 0 contains the number of registers used to

hold the message, or 0 if the alarm has no message. The repeat time, if one was specified, is stored as a number of seconds in nybbles eleven to five of the next register up. Any message is stored in one or more registers above this, with register M at the top and the others below it. If any of these registers is completely empty, the Time Module puts a byte value 10 in byte 1 in its copy in the buffer. The structure of this buffer and others is shown in the "HP-41 Synthetic Quick Reference Guide".

As an example of synthetic control of buffers, CLB is a program to delete the first buffer in the assignment and buffer area. This is usually the alarm buffer, and the main use of CLB is to replace the CLRALMS function of the HP-41CX for those HP-41 owners who have Extended Functions and a Time module but be careful if you have any other buffers). A RCL will normalise the first byte of a key assignment register to 10 - the program replaces this with F0 and puts back the register. The first byte of all the buffers that are used so far will be normalised from XX to 1X, and (if there are no buffers) the first byte of an empty register above the assignment area will be 00. Therefore if the first byte is not 10 then the program replaces it with 00, puts the register back, and finishes. Since no module uses 00 as its buffer identifier, the buffer will be deleted as soon as you pack or turn the HP-41 off and on again.

To save you even this trouble the second part of the program does a programmable PACK. You can use this routine, PPK, on its own. If you set flag 52 and execute a numeric entry then the HP-41 starts to program itself until there is no room left, then it does a PACK. By putting something into the register immediately below the .END. , PPK forces this PACK to be carried out at once, before any memory is changed. Then the register below the .END. must be cleared again. The only problem with this process will arise if the register immediately below the .END. already contains something - this is extremely unlikely if you enter the program and pack it manually before using it the first time.

The two programs contain several synthetic text strings:
Line 02 is F6,30,31,69,01,00,01. Line 06 is F1,10. Line 13 is F2,7F,F0.
Line 22 is not synthetic, it just appends 6 asterisks, other bytes will do.
Line 38 is not synthetic either, it just appends 5 asterisks.
Line 54 is F7,F0,00,00,00,00,00,00.
Line 56 is F6,30,31,69,01,00,01 - the same as line 02. Line 64 is F1,08.

```
01♦LBL "CLB"
02 "01i=♦x"
03 RCL [
04 X<> c
05 176.6
06 "0"
07 RCL [
08♦LBL 01
09 TONE 1
10 RCL IND Y
11 CLA
12 STO [
13 "⊢"
14 RDN
15 RCL [
16 X<> \
17 X=Y?
18 GTO 02
19 CLX
20 STO \
21♦LBL 02
22 "⊢******"
23 X<> \
24 STO IND Z
25 X<> \
26 X=0?
27 STO Z
28 RDN
29 ISG Y
30 GTO 01
31 R↑
32 R↑
33 STO c
34♦LBL "PPK"
35 RCL c
36 CLA
37 STO [
38 "⊢*****"
39 CLX
40 STO \
41 7
42 XEQ 03
43 16
44 MOD
45 6
46 XEQ 03
47 R↑
48 LASTX
49 X↑2
50 *
51 +
52 17
53 -
54 "♦♦♦♦♦♦"
55 RCL [
56 "01i=♦x"
57 RCL [
58 X<> c
59 X<>Y
60 STO IND Z
61 X<>Y
62 X<> c
63 RCL d
64 "Δ"
65 ASTO d
66 SF 25
67 .
68♦LBL 00
69 STO d
70 RDN
71 X<> c
72 .
73 STO IND T
74 RDN
75 X<> c
76 CLD
77 CLST
78 CLA
79 CF 25
80 RCL 01
81 SIGN
82 X=0?
83 GTO IND 01
84 CLX
85♦LBL 03
86 ALENG
87 X≠Y?
88 .
89 X≠0?
90 ATOX
91 END    187 BYTES
```

```
01♦LBL "TSTCLB"
02 CLST
03 CLA
04 23.3
05 XYZALM
06 ALMCAT
07 "BACK"
08 ASTO 01
09 XEQ "CLB"
10♦LBL "BACK"
11 ALMCAT
12 END
        45 BYTES
```

By now you should have had a lot of practice at entering and using Synthetic programs, so I shall only give an outline of what these do and how they work. CLB first changes register c so that the first data register is at absolute address 010 (this makes sure there is a register just below the curtain). It then recalls registers one at a time, starting at register 176, which is absolute address 0C0, the first of the assignment registers. The byte 10, which shows up as ⊖ in the listing, is compared

with the first byte of each register checked. If the two are equal then CLB skips to LBL 02 where it restores the F0 at the front of the assignment and puts it back where it belongs. (The F0 was put into Alpha at line 13, then Alpha was rearranged to put the F0 at the beginning of register N). If the byte tested is not 10 then lines 19 and 20 replace it with 00, and lines 26 and 27 put the zero into register Z as well so that line 29 will not repeat the loop. Steps 31 to 33 replace register c so that the routine PPK does not need to make any assumptions about the stack if it is called separately.

PPK must first find the address of the .END. - this can be done by the PPC ROM routine E? but some people do not have this module, so lines 35 to 51 do the job. In case one of the bytes of the .END. address is a zero the subroutine at LBL 03 is called twice to provide a zero or the ATOX value, depending on whether or not there are any zero bytes at the beginning of the text in Alpha. Lines 52 and 53 subtract 17 from this address to find the number of the register immediately below the .END. when the curtain is at absolute address 010. Lines 54 to 60 move the curtain (putting the same value into register c as at line 04) and then put the byte string F0,00,00,00,00,00,00 below the .END. . This string is used because it will be treated either as a buffer top (as with the alarm buffer) or as a null key assignment, and should not do any harm. Lines 61 and 62 restore the original register c in case something goes wrong during the packing process and you have to interrupt the program. Line 63 saves register d and lines 64 and 65 set flag 52, the PRGM mode flag. Line 66 sets flag 25 so that the number entry at line 67 (a decimal point on its own) will cause a PACKING but will not stop the program. When the HP-41 tries to program itself in PRGM mode it puts a number at every second line, beginning one line beyond the original number entry, and PPK therefore restarts at the second line after the number entry, which is where it tries to put the first number and fails. Line 68 is therefore a NOP which has to be there just to fill the step which is skipped. Lines 70 to 75 put back an empty register below the .END. and then restore the original register c. Lines 76 to 78 tidy up. If the display were not cleared it would continue to show PACKING.

When memory is packed, the current program step is not lost (otherwise you would lose your place every time you PACKed while entering a program), so PPK will continue after the packing. However packing can move programs in memory, so any return addresses could be wrong. Therefore PACK clears the return stack. PPK can thus only return to the program which called it by doing GTO IND global label. The version shown here looks for the label given by register 01, but steps 80 to 82 check if register 01 contains a text string - if not, then GTO IND 01 could restart execution of part of the program, so GTO IND 01 is not executed. Now for a sting in the tail. I have always said that programs should stop at their END - therefore I do not put a STOP or RTN after line 83 but let the steps down to the end execute too. They have no effect since the stack and Alpha have been cleared, and the program finishes at the END. This may seem a painful waste of time (hence a "sting") but it is insignificant compared to the time taken by packing.

To show how CLB and PPK work I have included a short test program TSTCLB. This sets an alarm, displays it with ALMCAT, then executes CLB to delete the alarm (and any others that may be set). To let execution return to TSTCLB it stores the text "BACK" in register 01, so that PPK goes back to the LBL "BACK" at which the alarm catalogue can be seen to be empty.

When is a programmable PACK likely to be useful? The above case is a good example. Other occassions will arise when you are changing assignments or buffers and want to PACK to tidy up. If you are really keen and have programs which write other programs then you can remove nulls from your new programs by PACKing to make more room once you have finished. Part of it is the sheer fun of doing something that the makers prevented you from doing.

We now have a programmable PACK, and we have seen how the non-programmable peripheral functions can be included in programs (by assigning them to a key, then pressing that key in PRGM and USER mode while the peripheral is not plugged in). Non-programmable functions that have not yet been made

programmable by Extended Functions or Synthetic Programming are CAT, COPY and CLP in the HP-41, and PRP and LIST on the printer. (PCLPS is not an exact alternative to CLP because it deletes not only the named program but also all the programs after it.) DEL, SST and BST could also be considered as candidates but it is not obvious how they should work - various programmable versions of SST and BST have been written in M-code (see the next chapter). Some display one step at a time without user intervention, others display and execute each step (the program WATCH in Section 6.6 did something similar). A programmable DEL should be told how many steps to delete, which one to begin at, and which program to delete steps from. In principle someone could do this with SP if it were necessary.

Programmable versions of CAT, COPY and CLP have also been written in M-code, and they can be written without M-code if you have a CCD module. An understanding of M-code is required though, even if you use the CCD module to write these functions, so I have not put them in here. The next section will describe one more function that can be made programmable by the use of Synthetics and Extended Functions alone.

16.8 A programmable PRP

The printer function PRP (PRint Program) can be included in programs like other non-programmable peripheral functions, but when executed it just displays NONEXISTENT. This is a pity because a programmable PRP could be used in documentation programs; ones that print a listing and a description of a program. Another use for a programmable PRP would be to print a listing of a mathematical function automatically next to a plot of it made by PRPLOT or PRPLOTP. It is surprisingly easy to plot a function which you have called "FUNC" and then to forget just what the function was.

What is it that is NONEXISTENT when PRP stops? It has not prompted for a label name, so it must be taking a label name from somewhere and finding that label to be nonexistent. Like other Alpha parameters the global label name is taken from register Q - but register Q is altered by every step of a running program, and cannot hold a label name. If the label name cannot

stay in register Q, then we have to make a special label name which is the same as the contents of register Q. This name can be put into Extended Memory, with a temporary .END. below it, and with the program to be printed lying above it (in a program file). Ah, but the contents of Q even depend on where in a seven-byte register each program step lies, and the program can move around in memory as it and other programs are edited. OK, so we run the relevant part of the program (the piece which contains PRP) in the Alpha register, whose alignment does not vary. One more problem: PRP stops printing when it comes to the END of the program it is printing, but then it executes that END so as to stop at it. By executing PRP in a subroutine we can make this END return to the program which called the subroutine that executed PRP. Any more problems? The program to be printed must be all within one block of Extended Memory, not partly in one block and partly in another. Oh, and one more thing - a programmable PRP should work like the programmable versions of SIZE, ASN and CLP: the program name should be taken from the Alpha register and the stack should not be changed.

Impossible? No -- merely difficult! I published an article with four programmable PRP programs in the PPC Calculator Journal V11N9P2-8. PPRP is the best, most versatile and longest (a shorter version that does not use Extended Functions is given in Exercise 16.B). It is given here as an example of what can be done with a combination of Extended Functions, Extended Memory, Synthetic Programming and an understanding of the HP-41.

```
*******
*PPRP*
*******
```

```
01♦LBL "PPRP"
02 CF 25
03 FS? 55
04 FC? 21
05 GTO 10
06 SF 25
07 PRBUF
08 FC? 25
09 GTO 10
10 X<> \
11 X≠0?
12 FC? 55
13 GTO 01
14 X<> \
15 GTO 09
16♦LBL 01
17 X<> \
18 SAVEP
19 FC? 25
20 GTO 09
21 STO \
22 RDN
23 8
24 X<>Y
25 STO ]
26 RDN
27 EMDIR
28 X<Y?
29 GTO 07
30 X<>Y
31 RDN
32  E3
33 ST/ Y
34 RDN
35 8
36 ST+ Y
37 CLX
38 X<> ]
39 X<>Y
40 "├..TMP.."
41 X<> [
42 X<> ]
43 X<> [
44 CRFLD
45 FS? 25
46 GTO 02
47 CLX
48 STO ]
49 X<> \
50 X<> [
51 GTO 08
52♦LBL 02
53 X<> [
54 X<> \
55 SAVEX
56 X<> \
57 X<> L
58 SAVEX
59 R↑
60 SAVEX
61 R↑
62 SAVEX
63 R↑
64 SAVEX
65 LASTX
66 SAVEX
67 CLX
68 STO ]
69 X<> \
70 X<> [
71 44
72 POSA
73♦LBL 03
74 X<0?
75 GTO 04
76 ATOX
77 SIGN
78 -
79 GTO 03
80♦LBL 04
81 RDN
82 FLSIZE
83 +
84 RCLPT
85  E6
86 /
87 +
88 ALENG
89 2
90 +
91 ENTER↑
92 ADV
93 FS? 12
94 CF 25
95 SF 12
96 42
97♦LBL 05
98 ACCHR
99 DSE Y
100 GTO 05
101 PRBUF
102 ACCHR
103 ACA
104 ACCHR
105 PRBUF
106♦LBL 06
107 ACCHR
108 DSE Z
109 GTO 06
110 PRBUF
111 FS? 25
112 CF 12
113 SF 25
114 R↑
115 "..TMP.."
116 RCLPTA
117 SIGN
118 +
119 ""
120 XTOA
121 "├♦┘♦)"
122 RCL \
123 SAVEX
124 RCL [
125 SAVEX
126 R↑
127 R↑
128 FRC
129  E3
130 *
131 58
132 +
133 RCL c
134 STO [
135 -2
136 AROT
137 RCL Z
138 R↑
139 256
140 /
141 XTOA
142 X<> L
143 MOD
144 XTOA
145 FRC
146  E3
147 *
148 X<> [
149 X<> \
150 X<> [
151 RCL d
152 STO [
153 -1
154 AROT
155 "├┘"
156 X<> [
157 STO d
158 X<> \
159 X<> c
160  E7
161 "♦┌┘↓↓        M"
162 X<> b
163 RDN
164 STO c
165 ADV
166 FIX 0
167 CF 29
168 CLA
169 ARCL Z
170 "├ BYTES LONG"
171 PRA
172 CLX
173 SEEKPT
174 FS? 25
175 SIGN
176 X<>Y
177 STO d
178 X<>Y
179 GETX
180 "..TMP.."
181 X<> [
182 PURFL
183 STO \
184 RDN
185 X=0?
186 CF 25
187 SIGN
188 SEEKPTA
189 GETX
190 SIGN
191 GETX
192 GETX
193 GETX
194 GETX
195 PURFL
196 X<> \
197 RDN
198 EMDIR
199 CLX
200 X<> \
201 GTO 10
202♦LBL 07
203 CLX
204 X<> ]
205 X<>Y
206 CLX
207 X<> \
208♦LBL 08
209 SF 25
210 PURFL
211 STO ]
212 RDN
213 EMDIR
214 CLX
215 X<> ]
216♦LBL 09
217 "├**ERROR"
218 PRA
219 X<> [
220 CLX
221 STO ↑
222 X<> ]
223 X<> \
224 X<> [
225 CF 25
226♦LBL 10
227 CLD
228 END

446 BYTES LONG
```

To use the program, connect an HP82143A or any HP-IL printer, make sure you have just an EFM module in your HP-41 (details of using PPRP with additional EM modules were given in the original article - some advice is given here with the explanation of lines 131-132 in the description below), put the name of any global label from the program you want to print into the Alpha register, and execute PPRP from another program or from the keyboard. Your program will be copied to EM, a title will be printed, then the program will be listed, then its length in bytes will be printed, and PPRP finally returns with X, Y, Z, T, L and Alpha unchanged. The program file is deleted from EM unless you give the name followed by a comma and a different file name. (You may want to do this if you wish to print the program and save a copy in EM as well.) The string containing both names and the comma must be no more than seven characters long unless you use the feature described under lines 16-20 of the program. The program will fail if there is not enough room in EM, if the global label is not found in RAM, if the printer is off or disabled or flag 21 is clear. In those cases where it is possible the program length will still be printed, and an error message will also appear. Flag 25 is set by the program if it succeeds, and cleared if it fails, but it still returns. If you provide two names separated by a comma then the EM file is not deleted; this is considered an error so flag 25 is cleared even if everything else has worked. No other flags are altered by PPRP. If the BAT flag becomes set while the program is running then it will fail. If the printer stops, while the program is being printed, because it runs out of paper or the batteries go flat then you must reload it with paper or connect up a battery charger. Then press GTO . EEX 999 to get to the END of the program you are printing, and press R/S to let it return to PPRP so that everything can be tidied up. The program listing above was printed by PPRP itself: Here is a detailed explanation.

01	Title "Programmable PRP".
02-05	Return with flag 25 clear if flags 21 or 55 are clear.
06-09	Return with flag 25 clear if the printer is switched off or is not connected; otherwise clear the print buffer.
10-13	Check that only seven or fewer characters are in the Alpha

register. This is not a foolproof test since only register N is checked. However if register O contains something then either the user has made a mistake and the SAVEP at line 18 should fail, or the user is saving a program with a different file name - see under steps 16-20 below. Lines 11 and 12 check for a non-zero value in N; they cannot be X=0? because a non-numeric value with that test would clear flag 25 but execute the step after it even though the value in X is not zero. X≠0? followed by a test which is always false forces line 13 to be skipped if N contains anything other than zero.

14-15 If N contains anything other than nulls then retore it and go to the error completion part of PPRP.

16-20 N is clear, so assume O and P are clear too. (If a label name in register O is followed by nulls in register N, and a comma followed by a file name in register M then you can save a program with a long name in a file with a name of up to six characters.) Restore N, then try to save the program in Extended Memory. Go to error completion if this fails because you do not have an Extended Functions module plugged in (nor an HP-41CX) or because there is not enough room in EM.

21-26 Store X in N and Y in O, put 8 in Y. Use RDN while doing this to avoid lifting the stack and losing the values in Z or T. RDN will be used for this purpose many more times.

27-29 Use EMDIR to get the number of free registers in EM. If this is less than 8 then there is not enough room for the temporary file that will be used later, so go to error completion. The 8 is put in at this point as it will be used later on. EMDIR also provides a list of the files in EM, including the name of the program file just created. Use EMROOM instead for faster execution on an HP-41CX.

30-37 The number of free registers in EM (abbreviated to NFR from now on) is needed later, so put the number 8.NFR in Y without disturbing Z, T or L.

38-43 Now prepare to create a data file called "..TMP.." - this name is 7 characters long so that it will exactly fill one register and

will be easy to move around. The name has to be put in register O, in front of the other values being stored in Alpha, and the file length must be in X, so 8.NFR is put in X. The other registers are rearranged at the same time: Y contains its original value, N contains the name of the program to be printed, M contains the original X value.

44-46 Try to create the ..TMP.. data file. This may fail because a file of this name already exists (made perhaps by an earlier run of PPRP which was interrupted). If no error occurred then skip over the next five steps.

47-51 If an error did occur then restore X and M, clear N and go to the error completion point.

Note: At this point Y,Z,T,L contain their original values, X contains 8.NFR, M contains the original value from X, N contains the name of the program we want to print, and O contains "..TMP..". We can now save these values in the temporary data file.

52-66 Save stack and L, M in the file in the order M, L, T, Z, Y, X and keep 8.NFR in register M.

67-70 Rearrange Alpha and the stack: clear O and N, put 8.NFR in X and restore the file name in M.

Note: The file name in M can be of the form "filename" or "label,filename". To obtain the program size we must use only the file name, so the next lines have to remove all text up to and including the comma if there is one. It does not matter if there is a second comma in M because it and anything after it will be ignored by FLSIZE.

71-72 Check if there is a comma (character 44) in M.

73-79 So long as X contains a positive or zero value keep deleting characters from M. Note the use of ATOX, SIGN, - to decrement X by 1.

80-87 Obtain the program file size and add 8.NFR to it to obtain OFF.NFR, where OFF is the offset of the first register of the

program file from the position at which a "special" label will be put in the ..TMP.. file. Then obtain the program length in bytes (LEN) by using RCLPT and add LEN/E6 to OFF.NFR to produce the packed number OFF.NFRLEN in X.

Note: We have now saved the numbers which will be used later on and can use the stack to print a title. This consists of the program file name surrounded by a box of stars all in double-wide characters.

88-92 Put the length of the title line into Y and X. Advance one line to separate the title from anything that might have been printed by PRBUF earlier on. If there was a comma in register M then the title will consist of everything that came after the comma.

93-95 Save the present status of flag 12 in flag 25. This means that the title can be printed double-wide but the program itself will be printed according to the user's setting of flag 12.

96 Put the code for "*" in X; OFF.NFRLEN is now in T.

97-101 Loop to print top line of box.

102-105 Print file name, with stars on either side.

106-110 Loop to print bottom line of box. The box length was stored in both Y and Z so that it could be used in two loops.

111-113 Restore the setting of flag 12 and set flag 25 to allow for possible errors later on.

Note: Next we build up a "special" global label pointing to the register before the first register of the program in EM, and we build up an .END. pointing to this label. This "special" global label corresponds to what Q will contain when PRP is executed. These two items will be built up in the Alpha register, then copied to the last two registers of the ..TMP.. file.

114-125 Recover OFF.NFRLEN, make ..TMP.. the current file again, use the pointer value (converted to a 1) to increment OFF by 1 so that it now points to the second header register of the program file. This register will look like an END so PRP will begin printing from the step after it, which happens to be the beginning of the

program. OFF+1.NFRLEN is now in X and OFF+1 is equal to [9 plus the size of the program file]. Use this to build in the Alpha register a label pointing to 4 bytes and [9+size] registers above itself. The label is LBL "!" and is followed by an .END. pointing to it. Move these to the ..TMP.. file. Line 119 provides the first byte of the label, it is F1,C4. Line 120 provides the number of registers, line 121 appends the rest of the label and the .END.; it is F7, 7F, F2, 00, 21, CA, 00, 29.

126-130 Recover NFR.LEN

131-132 Make NFR+58.LEN : NFR+ 58 is the absolute address of the new .END. if there are no Extended Memory modules plugged into the HP-41. If there are any EM modules plugged in and the program file is in one of them then line 131 will have to be replaced by 507 or 763, and you will have to make sure the entire program is in one module (by creating a dummy file as was described in the last paragraph of point 2. of Section 16.6).

133-144 Build up an alternative register c with the .END. address given as NFR+58 (or NFR+ 507 or 763). note that this involves obtaining the quotient and remainder of (NFR+58)/256 to put into register c as two separate bytes. This leaves the least significant nybble of the curtain address as 0; this may well be wrong in which case part of the first program in CAT 1 will be lost until register c is restored. This does not affect PPRP; even if it is the first program in CAT 1 we are now way below the part that could be hidden by the curtain, but we must make sure the original contents of c are restored at some stage.

145-147 Recover the program byte length, LEN. This was obtained long ago, when the program file name was available, but is only going to be used now.

148-152 Put the new c into N and put d into M. This allows for fiddling with register d even if it contains null bytes at the front. The register c value should not start with a null byte since that would not be a valid address for the statistics registers (you might have trouble if you run PPRP while the statistics registers are located in the status register area). AROT can therefore be

used safely without fear of losing null bytes.

153-157 Replace the last byte of register d with "!" and copy this new value into register d. (! is equivalent to having only the message and printer flags set).

158-159 Now replace register c with the value pointing to the .END. in EM.

Note: At this point the stack has LEN in Z, the old d in Y, the old c in X.

160 Put the address of the top of register N into register X, ready for the X<>b which will move program execution into the Alpha register.

161 Put into M and N the bytes which will be executed as a piece of the program and will carry out PRP with some associated instructions. The text string is FA, EC, 00, 06, 91, 7C, 07, 07, A7, 4D, 85. This corresponds to the steps: XEQ 06 (compiled), STO b (to return to the main program from Alpha after the RTN takes us back to the step after XEQ - the old register b value is still in X unless PRP fails and the user disturbs the stack), LBL 06(a NOP required to pad the PRP position), LBL 06, PRP, RTN (in case the PRP failed and did not itself return to the STO b).

162 Jump into Alpha to execute the PRP section at a clearly defined position in a register.

163-164 Restore the old value to register c.

165-171 Print the program byte length. The original register d value is in register Y so the display mode can be changed. The program length will be printed even if PRP failed.

172-173 Clear X and set the file pointer to the beginning of ..TMP.. .

174-178 Now restore the original register d value, however the status of flag 25 records the success or failure of PRP. Therefore copy the status of flag 25 into X (0 if clear, 1 if set) before restoring the old register d (which has flag 25 set).

179-183 Get original register M contents from ..TMP.. , put "..TMP.." into X for later use, purge the program file, put "..TMP.." into register N. If the program file is not purged for any reason

then flag 25 will be cleared. (One reason is if the file name was different from the program name because they were specified as "label,filename".)

184-186 Clear flag 25 if the value in X records that flag 25 was clear before line 177. Flag 25 is clear anyhow if the PURFL at line 182 failed, this step does not change flag 25 if it is clear already.

187-188 Re-establish ..TMP.. as the current file. Do this by using SEEKPTA which also sets the pointer to the second register; it was there after line 179 anyway, but SEEKPTA is a quick way of re-establishing the current file.

189-194 Restore register L, then T, Z, Y, X from the values saved in ..TMP.. .

195 Purge the ..TMP.. file.

196-197 Save the X value in N and get rid of the text string "..TMP..".

198 Use EMDIR to confirm that the program file and ..TMP.. have been purged. EMDIR also establishes a current file and therefore prevents the PURFL bug on earlier EFMs. If your EFM does not have the PURFL bug (see Appendix C) then you can delete lines 197 and 198.

199-200 Recover X and clear N.

201 Go to the END of the program. This is safer than a RTN which would stop the program here if it was executed from the keyboard, so that an accidental R/S would restart the program and disrupt the stack contents.

202-207 Reset the stack after finding insufficient room for ..TMP.. (at line 28).

208-210 Purge the program file if it had been created and an error occurred afterwards. Set flag 25 because the purging will not work if if the file name had been specified as "label,filename".

211-215 Use EMDIR to avoid the PURFL bug and show the user what is left in EM. Use register O to store X while this is happening, then restore X. You can delete all these lines if your EFM does not have the PURFL bug.

216-218 Print the program name followed by "**ERROR" to tell the user that something went wrong (not just a printer absent, off or

disabled).

219-224 Rearrange the Alpha register after printing this message (easily done because "**ERROR" is seven characters long): shift O and N into N and M, clear P and O.

225 Make sure flag 25 is clear to show an error was detected (flag 25 may be set as a result of line 209).

226 Come here from line 201 after successful completion.

227-228 Clear the display in case an EMDIR message is still on view after a success or a failure of the program, and END.

This program is long - it just fits into the memory of an HP-41C with no extra memory modules, but in that case it can only print itself. It is best used as a documentation program which does not occupy your HP-41 much of the time. You may prefer the shorter version given in Exercise 16.B.

16.9 Non-normalising recalls and RAM editing

To have complete control over the Random Access Memory of your HP-41 you should be able to examine and alter any RAM register. Altering a register is not very difficult because you can always move the curtain and STO into any register. Just be careful to SF 25 in case the register is NONEXISTENT and to avoid trouble if you put the curtain directly above a nonexistent address. But don't SF 25 if you are using the PPC ROM program SX (this is not a problem with RX).

Often you will want to examine a register before you change it, and you cannot just RCL it as that will normalise the contents. There are several approaches to this problem.

The first method used to examine RAM registers was STO b to put the program pointer to the address followed by a Byte Jump to copy bytes from it to the Alpha register. The copy in Alpha could then be studied at leisure. The method required a suitable byte just before the area to be examined. Remember that the second nybble of the preceding byte gives the number of bytes copied, and that a first byte of 7F is not copied but just appends

the following bytes to Alpha. You therefore have to find a suitable byte before pressing the Byte Jumper.

Any method which allows copying of a selected number of registers under program control would be much preferred. One way is to move the curtain so that the register to be studied is in the data area, then writing the register to a data card, then moving the curtain into the status register area, and reading the data card back into a status register from which it can be recalled without normalisation. A similar operation can be carried out with a Mass Storage device. Both these methods are slow and demand peripheral equipment. Even better would be a plug-in module which lets you perform non-normalised recalls to X, or which lets you study and edit a selected part of RAM. The Extended Functions SAVER and SAVERX do the job fairly well; you can move the curtain, then copy registers to a data file from which they can be recalled by GETX without being normalised. SAVERX does normalise the first and last registers moved though.

If your HP-41C has Bug 2 (see Appendix C), another way to copy data registers into EM is with REGMOVE and REGSWAP. You do not even have to move the curtain: REGMOVE and REGSWAP only check if the top register of the block to be moved exists (and they normalise this one register in each block moved). This means you can specify a block of registers whose bottom address is in Main Memory and whose top address is in EM (in other words the block straddles the gap at address 200), and you can move this block so that registers which were in Main Memory are copied into EM, from where you can recall them with GETX.

All these methods still require considerable effort to move the right registers and to move them to the right place. One of the first functions written by the user club pioneers of M-code use was a non-normalising recall function which could copy data from any absolute address to X, but this was only available for people who had EPROM boxes or other special equipment. Much easier to use is a plug-in module with one or more non-normalising recall functions. Three such modules are available at present: ZENROM, the CCD module, and the HP-IL Development module.

ZENROM and the CCD module have functions specially designed to recall any RAM register without normalising it, and also to store data into any absolute RAM address without moving the curtain. The Devel module function **RG-BUFX** copies data to the buffer without normalising it - the registers to be copied are specified as bbb.eee in register X. Another function **BUF-AX** copies n bytes from the buffer to the Alpha register without normalising them; n is given in X. RG-BUFX has the same register numbering "wrap-around" bug as was described as Bug 2 in the early HP-41Cs (see Appendix C) so it can be used to recall any RAM register. The only restrictions are that register numbers greater than 999 cannot be used (they are described as NONEXISTENT) and that SIZE must not be set to zero. Otherwise you can use RG-BUFX to recall any RAM register, using the short program NNR. If NNR does not recall the register you want then you will have to move the curtain, either by changing SIZE or by synthetic methods. Articles and programs on this subject have appeared in PPCJ V11N2P25-26 and V12N1P28-29.

```
01♦LBL "NNR"
02 SIZE?
03 X<>Y
04 +
05 512
06 ST+ Y
07 RDN
08 7
09 BSIZEX
10 X<>Y
11 RG-BUFX
12 RDN
13 BUF-AX
14 RDN
15 LASTX
16 RCL [
17 END
                    33 BYTES
```

To use NNR put the number of the absolute register you want to recall into X, and XEQ "NNR". If the register is accessible (SIZE is not 000 and register number + SIZE + 512 is below 1000 decimal) its value will be returned to X and the stack will be lifted. X will also be copied to register L, and the required NNN will be copied to M as well, so the Alpha register contents will be moved seven characters to the left. If the register does not exist then a zero will be copied to X. If the register number + SIZE + 512 is greater than 999 then the program will stop at line 11 and display NONEXISTENT, unless you have set flag 25. If you use NNR to recall a register from the buffer area then allow for the fact that NNR creates a buffer itself, so other buffers may be moved or deleted.

The number 512 at line 05 assumes that you have full Main Memory. If you have an HP-41C without a quad memory module then replace the 512 with:

256 + n*64 - where n is the number of Memory Modules plugged in.

Even better than a non-normalising recall function is a program to let you examine RAM and edit it as you go. The ZENROM has a function called RAMED which is particularly helpful if you have an early HP-41 on which some of the ZENROM direct-key synthetics do not work. If you do not have a ZENROM then you can try to write a program that will edit RAM. Two such programs have been published in the PPC Journal. One, by G. Peque, in V12N3P29-31 uses the Devel module. The other, by Bruce Bailey in V11N4P15-16, uses the Extended Functions and the PPC ROM. It provides a good last example of Synthetic Programming with the Extended Functions to finish this chapter, so I am publishing it here, with Bruce's permission, for which many thanks.

ED is fundamentally an editor for program memory, though it can be used for any part of RAM as will be described later. It works by copying the region to be edited to a data file in the EFM without normalisation, supporting a range of editing functions, and replacing the edited region in program memory. The PPC ROM and the EFM are required, as is SIZE 006.
Lines 33 and 168 are hexadecimal F5,01,69,00,80,07.

To use ED, insert LBL "2", XEQ "≠≠≠≠≠≠≠" before the region of memory to be edited and LBL "1", XEQ "≠≠≠≠≠≠≠" after it (the LBLs can be reversed). Purge any file named "≠" and XEQ "ED". It will create a data file named ≠ which will require at most the number of registers to be edited plus four. When execution halts you will be looking at the hex of the first register in the region; the decimal point precedes and marks the "current byte". Your key assignments will be suspended and local labels A to I will be defined. The first byte, byte 0 in ED's numbering, will always be 1D, part of the XEQ "≠≠≠≠≠≠≠" instruction. Now follow the instructions given below.

```
01♦LBL "ED"
02 CF 25
03 SF 10
04 GTO "1"
05♦LBL "########"
06 RCL b
07 XROM "RT"
08 7
09 /
10 INT
11 FS?C 10
12 GTO "2"
13 X<Y?
14 X<>Y
15 STO 05
16 X<>Y
17 STO 03
18 -
19 STO 00
20 2
21 +
22 "≠"
23 CRFLD
24 RCL 03
25 8
26 -
27 RCL 05
28 LASTX
29 -
30  E3
31 /
32 +
33 "≈i♦↓"
34 RCL [
35 X<> c
36 X<>Y
37 SAVEX
38 SAVERX
39 RDN
40 STO c
41 4
42 XROM "SK"
43 CLST
44 SF 10
45 SF 27
46 FIX 0
47 DSE 03
48 ISG 00
49♦LBL 01
50 X<0?
51 CLST
52 RCL 00
53 7
54 *
55 DSE X
56 X>Y?
57 X<>Y
58 STO 02
59 LASTX
60 XROM "QR"
61 X<>Y
62 RCL 00
63 -
64 STO 01
65 SEEKPT
66 GETX
67 XROM "NH"
68 AOFF
69 14
70 R↑
71 ST+ X
72 AROT
73 "├."
74 -
75 AROT
76 CLST
77 CF 23
78 PROMPT
79♦LBL A
80 CHS
81♦LBL B
82 CHS
83 RCL 02
84 +
85♦LBL G
86 GTO 01
87♦LBL C
88 FS? 23
89 XROM "XD"
90 CLA
91 XTOA
92 ASTO X
93 X<>Y
94 X<> 02
95 ST+ 02
96 7
97 MOD
98 LASTX
99 -
100 CHS
101 RCL 01
102 SEEKPT
103 GETX
104 STO [
105 RDN
106 SEEKPT
107 RDN
108 XROM "SU"
109 RCL [
110 SAVEX
111♦LBL 02
112 RCL 02
113 GTO 01
114♦LBL H
115 CLA
116 ARCL 02
117 "├:"
118 RCL 03
119 RCL 01
120 -
121 ARCL X
122 AVIEW
123 GTO 02
124♦LBL I
125 RCL 01
126 SEEKPT
127 XROM "HN"
128 SAVEX
129 GTO 02
130♦LBL F
131 FS? 23
132 XROM "XD"
133 CLA
134 2
135 X=Y?
136 SIGN
137 XTOA
138 X<> [
139 STO \
140 RCL 01
141 X<>Y
142♦LBL 03
143 RDN
144  E
145 +
146 X=0?
147 GTO 02
148 SEEKPT
149 GETX
150 STO [
151 RDN
152 RCL Y
153 POSA
154 X<0?
155 GTO 03
156 X<>Y
157 RCL 00
158 +
159 7
160 *
161  E
162 -
163 +
164 GTO G
165♦LBL D
166 CLX
167 SEEKPT
168 "≈i♦↓"
169 RCL [
170 X<> c
171 GETX
172 GETRX
173 RDN
174 STO c
175♦LBL E
176 "≠"
177 PURFL
178 4
179 XROM "RK"
180 CLST
181 END      299 BYTES
```

Commands available:

n A	(Put n in register X, press the A key) moves the current byte pointer (cbp) ahead n bytes.
n B	moves cbp backwards by n bytes.
n C	changes the current byte to decimal n (0<= n <=255).
"nn" C	changes the current byte to hex nn (use legal hex digits 0-9,A-F, and use exactly two digits).
m "nn" C	first changes the current byte to nn, then advances m bytes.
m E↑ n C	first changes the current byte to decimal n then advances m bytes (E↑ stands for ENTER↑).
D	replaces the region with your edited version, then exits.
E	exits without replacing the edited region.
n F	finds the next occurrence of decimal byte n, starting the search at the next full register.
"nn" F	finds the next occurrence of hex byte nn similarly.
n G	moves cbp to byte n.
H	displays cbp and the decimal RAM address from which the current register came.
"..." I	replaces the whole current register with the hex string "..." - this must be 14 legal hexadecimal digits.
+-n R/S	advances/backs up cbp by n bytes.

The version given here displays bytes using the "natural notation" in which the hex digits A to F are displayed as : ; < = > ? respectively. To use the normal digits **A B C D E F** change line 44 to CF 10 (actually you can remove this line entirely, but changing it will leave the line numbers the same as before). When flag 10 is set, the routine NH displays hex numbers in natural notation to speed up its operation. Clearing flag 10 slows the program down but makes the results more legible.

ED is quite useful when entering synthetic text lines. It can be used to create any sequence of bytes you desire; put place-holders into the program as you are keying it in originally, then EDit them into the required final form. It can also be used to explore the contents of program memory byte

by byte, including ENDs; the edited region can contain any number of ENDs. If you want to edit a program without putting the first and last labels (LBL "1" and LBL "2") into it, so that it will not be decompiled, then put these labels into programs before and after the one to be edited.

Technical note: The synthetic strings at lines 33 and 168 are used to lower the curtain in preference to the PPC ROM routine OM because they set the address of the .END. to 007, thus deactivating all Timer alarms. Unfortunately this device also renders ED non-SSTable until the curtain is restored; also don't use a printer in TRACE mode.

The purpose of the XEQ "≠≠≠≠≠≠" instructions is to supply a locatable register guaranteed to be already normalised. ED uses SAVERX to move the region to EM; SAVERX normalises the block end points as mentioned at the beginning of this chapter.

Exit from ED only via the D or E options on pain of loss of your key assignments. The final EFM function used is a PURFL; you may wish to insert an EMDIR after it to dodge the PURFL bug (as was done in PPRP).

It is possible to encounter another EFM bug in ED; execution can halt at line 106 with a spurious diagnostic (usually END OF FL - this bug was mentioned at the end of Section 16.5). Simply press R/S and carry on. If you prefer you can lower the chances of encountering this bug by inserting a STO X after line 105. The STO X sets the register which is compared with file names to a form most unlikely to be in use as a file name.

Instead of editing programs you can edit other parts of RAM. Lines 4 to 12 determine the decimal addresses of the region to edit. You can supply these directly: for instance pick an area between the .END. and your KARs, say from address 225 to 191. Do 225, ENTER, 191, GTO "ED", GTO.013, R/S. Other areas in RAM can be accessed in like fashion, but not the status registers.

As an exercise use ED to edit itself and change the D in the name into a C, thus resolving the conflict with the HP-41CX function ED. While this use is trivial it will serve to illustrate ED's utility.

Exercises.

16.A Unlike most other synthetic routines presented here PPK uses a numbered data register. Try to rewrite PPK so that it can be called with the global label in the Alpha register or in register X, and will not use any numbered data registers.

16.B The program PPRP is long and involved. Here is a much shorter programmable PRP routine, called PPR. It does not use any Extended Functions and does not change register c. It will print any program from RAM or ROM if that program has a one-letter global label which can be copied to byte 0 of register d leaving flag 53 clear and flag 55 set. This limits the usable global labels to:

"+", "1", "3", "9", "K", "Q", "S", "Y"

and some synthetic global labels such as:

"A", "C", "I", "a", "!", ";"

If the Alpha register is completely empty then PPR will try to print a program containing the label LBL "+". You can put LBL "+" as the last line of a program you are debugging, then write the following short program and assign it to a key:

LBL "PRPD", CLA, XEQ "PPR", END

This program will print a program by default: whenever you press the key to which it is assigned it will print the program which contains LBL "+". In other words, it is a non-prompting version of PRP which can be used while you are developing a new program.

Try using PPR and if you are ambitious try to understand how it works. If you cannot work this out but are still interested, get a copy of the PPC Calculator Journal Volume 11 Number 9 and read the article on pages 2 to 8.

```
01♦LBL "PPR"
02 CF 25
03 FC? 21
04 GTO 03
05 SF 25
06 PRBUF
07 FC? 25
08 GTO 03
09 X<> [
10 X≠0?
11 GTO 01
12 X<> [
13 "+"
14 X<> [
15♦LBL 01
16 X<> [
17 "┣******"
18 STO \
19 RDN
20 RCL d
21 X<> [
22 "┣******"
23 X<> [
24 "┣*"
25 RDN
26 RCL \
27 X<> d
28 XEQ 02
29 FS? 25
30 STO d
31 X<> ]
32 CLA
33 GTO 03
34♦LBL 00
35♦LBL 02
36 PRP
37 X<> \
38♦LBL 00
39♦LBL 00
40 STO d
41 X<> \
42♦LBL 00
43♦LBL 00
44 PRP
45 STO d
46 CF 25
47♦LBL 03
48 END      104 BYTES
```

CHAPTER 17 - WHERE NEXT?

17.1 A better machine?

Most individuals and organisations replace their computers every few years with a bigger and better machine. At what stage should we replace our HP-41s? Are the heroic efforts of the previous three chapters worth the trouble or should we retire our HP-41s and get a better machine?

It really must depend on your uses for an HP-41. Like many HP-41 users I have access to micros, minis, and mainframe computers, but still need a pocket calculator. It would be pretty silly to use a Cray to work out the square root of the speed of light, or the area of my garden. Once I have an HP-41 in my pocket I want to do as much as is reasonable with it; that way I have a portable computer too. The logical replacement for the HP-41 would be another calculator-shaped pocket computer, faster and with more memory, a better display and more functions - but still a calculator. Many people though use their HP-41 exclusively as a computer, not a calculator as well; for them an HP-71 or some other small portable computer may be a good replacement right now.

Those of us who want a calculator that is also a computer and which is similar to earlier such machines are likely to carry on using the 41, at least until HP produce a suitable successor. After all, it has recently become common knowledge that 42 is the answer to everything !

The aim of this book has been to show how the HP-41 itself can become a better machine. This last chapter will briefly describe some more ways of making the HP-41 such a "better machine".

17.2 Personalised software and keyboards

You are yourself the best judge of how you need to use an HP-41. The types of programs you use are determined by your activities and you have to choose or write your programs. This book could not tell you how to write

surveying programs, financial programs, and programs for all other applications - it has concentrated on advice and information that would let you write programs as well suited to your work as possible.

This can be taken a stage further. Programs can be made to address you by name, or to ask for a password in case someone else tries to use them. They can be adjusted to use the units that you prefer to work in, and to use abbreviations that you are familiar with. You can write games to play in your spare time, or programs to do Number Theory. Hewlett-Packard designed the HP-41 so that it could be personalised in such ways.

One particular way to personalise your HP-41 is to design one or more sets of key assignments for your use. Synthetic assignments of functions which you often use extend this; if you use GTO IND X or VIEW IND Y a lot then assign them both to unshifted keys. You can write a single program which asks what you want to do, then goes to a selected program; the CCD module even provides a **menu** function to simplify this. (A menu is a list of possible responses with a short explanation of each.) Each of the programs could then use the local labels A to I, a to e, and XEQ followed by pressing one of the keys in the top two rows to provide XEQ 01 to XEQ 10. In this way the one main menu program assigned to a key could give you access to a large number of further programs, each of which could have up to 25 local labels which do not use any further assignments. The Extended Functions GETKEY and GETKEYX provide another way of personalising the keyboard without making additional assignments.

A question that is sometimes asked is "can I reassign the Alpha keys?" At first sight this seems impossible because the Alpha keyboard is quite separate from the others. You can however write a program with many global labels, each of which is assigned to a key and appends a selected letter to the Alpha register. Say you want to assign the following symbols to the top row of keys:

KEY:	**A**	**B**	**C**	**D**	**E**
ASSIGNED CHARACTER:	"	'	**X**	**Y**	**&**

Not only is this quite a different layout, but also three of the characters are ones which are not available at all from the Alpha keyboard. To do this you would write the following program:

```
01 *LBL """          10 *LBL "Y"
02 "⊦-""             11 "⊦-Y"
03 GTO 01            12 GTO 01
04 *LBL "'"          13 *LBL "&"
05 "⊦-'"             14 "⊦-&"
06 GTO 01            15 LBL 01
07 *LBL "X"          16 AVIEW
08 "⊦-X"             17 END
09 GTO 01
```

Now you could assign each of the five global labels to one of the five top keys. If you press any one of the keys in USER mode then you see a preview of the one character it will enter, and if you release it before it is nulled then the character is appended to the Alpha register and you see the new contents of Alpha. If the Alpha register becomes full then you will hear a tone, just as with the normal Alpha keyboard. To delete a character you just set Alpha mode and press Append and backarrow.

Three of these five key assignments and text strings have to be made by synthetic methods. An easy way is to use the Q-loaders described in Chapter 15. The easiest way is to use the special USER ALPHA keyboard of ZENROM; this lets you make special text strings and labels, and also lets you assign such labels. In any case you might find that the USER ALPHA keyboard does the job you want. The CCD module has a special character keyboard as well, but this cannot be used to make labels or assignments.

17.3 Hardware modifications

The HP-41 is a lovely box of electronic tricks: so lovely that the more curious users cannot wait long before they open it up to see inside. Once

the electronics are understood, maybe they can be modified. Then you will have a truly "personal" HP-41 - maybe it will do something useful such as working faster, or maybe it will do something useless like displaying everything upside-down.

The first hardware modifications were made to the plug-in memory modules. If you had four then you could not use any ports for anything else. Some people opened two modules, took the memory chips out of one and fitted them into the other. By suitable wiring they made it appear that these two modules were in two different ports. Then they had a "double memory module" - a **dule** in one port, and a spare port for a printer or some other plug-in. The next trick was to take the circuits out of the modules and wire them inside the HP-41; there is enough room on each side of the display to hold two circuits taken out of their modules. This provided four free ports and a full memory. When HP produced the Quad memory modules they acknowledged the need for such efforts and introduced the HP-41CV with the quad module officially attached to the main circuit board. Nevertheless other things were still worth wiring up inside, for example the Time modules or Extended Functions. Now that these are built into the HP-41CX, people wire two Extended Memory modules into their 41CXs so as to have a full Extended Memory and four ports. Other modules are also considered fair game for wiring in; the PPC ROM or ZENROM or the Devel module which extend the features of the HP-41 for example. Modules are still doubled or tripled, for example an Extended Functions module and two Extended Memory modules can be fitted into one module case if you know how.

All this is possible because different addressing schemes are used for Main Memory, Extended Memory, system devices like the Time module, and normal ROM modules. As was explained at the end of Section 8.3 they can all communicate through the same port. What is more, the addressing lines on the circuit in a module can be pulled high or low so that its address can be selected independently of which port the module is actually wired to.

The other main hardware activity has been speeding-up of HP-41s. The speed at which they run is regulated by a timing circuit. This is controlled by

a capacitor attached to pins 9 and 10 of the CPU chip. The normal value of this capacitor is 150 pF, but lower values will make the HP-41 run faster. If you know enough about electronics you can find this capacitor by counting the pins on the CPU chip and replace it with one of about 50 pF (but read the dire warning at the end of this section). The position of this capacitor has changed as the internal layout has been redesigned from time to time, so it is best to identify it by looking to see which pins it is connected to, not where it is on the board (its latest position has been described from time to time in updates published in PPCJ, and an HP-41C circuit diagram has been given in DATAFILE V2N5P20-21). Most HP-41s can be speeded up by a factor of at least two but odd things can begin to happen beyond that. Some registers become intermittently NONEXISTENT, or data is read or stored incorrectly. A good article on checking this appeared in PPCJ V11N4P16.

A speed-up can affect HP-41 peripherals as well. Early Card Readers can not read cards when the HP-41 is working at high speed. Wands seem to actually like the higher speed and to read barcode more easily. The HP-IL cassette drive takes some time to rewind a cassette if a file is partly at the end of the first track and partly at the beginning of the second track. On a speeded-up HP-41 the HP-IL circuit considers that the rewind operation is taking too long and gives a DRIVE ERR message. (A file which spans the two tracks will always slow things down; it is best to make a second copy which will be on the second track only.) One way to overcome such problems is to use a variable capacitor, but this has to be wired so that it can be controlled from outside. A variable capacitor can be amusing though; the frequency of the TONEs depends on the clock, so a speed-up makes the HP-41 give shorter TONEs at higher frequencies - a variable capacitor lets you change TONE frequencies at will. A simpler solution is to use a switch so the speed-up capacitor can be disabled when a plug-in device is in use.

One more hardware modification is the use of alternative power sources. Some people redesign their battery holders so they will hold Lithium batteries which have a long life. Others connect large batteries through the adaptor socket (see Section 2.4). This is fairly easy if you have an

old HP-41C with the gold balls still in the socket. The socket does not have protective diodes like the internal battery connectors, so it is up to you to avoid blowing the CPU. An article about external power supplies that is worth reading appeared in PPCCJ V6N7P33-35.

DIRE WARNING

Hewlett Packard neither supports nor approves hardware alterations to the HP-41 equipment. If you open an HP-41 or any plug-in device then you invalidate any warranty, as HP takes pains to point out in the Appendix on Service Information (they refer to misuse, or service or modification by other than an authorised HP repair centre). The HP-41 contains CMOS circuits which are far more easily damaged by stray charges than normal TTL circuits. This section is provided for information only, if you want to do something then make sure you know what you are doing, get someone to supervise if possible, and make sure you have enough money in your bank to pay for any repairs that may become necessary. HP repair centres tend to throw away anything inside an HP-41 that was not put there by HP, but they might let you have it back if you ask nicely. If you do not know how to open an HP-41 then you should certainly not try until you have some expert advice: join a user group or apply for a job at HP.

17.4 Black boxes and M-code

The programs that you buy in a plug-in Applications Module have been written to a Read Only Memory chip which is inside the module. As its name suggests, Read Only Memory cannot be written to; it is written once and is permanent. A less permanent type of memory chip is EPROM (Electronically Programmable Read Only Memory) - in other words you <u>can</u> write a program to it if you have the right instruments. If you want to try out a program that will eventually go onto a ROM you can use the HP-41 to write that program onto an EPROM first, then plug the EPROM into a box which connects to an HP-41 port like any other plug-in peripheral. As far as the HP-41 is concerned this **EPROM box** is just another plug-in ROM module. The user can see it is not because the box is so much bigger. An EPROM programmer

(which writes the program onto EPROM) is fairly expensive, but an EPROM box is no more expensive than other HP-41 peripherals.

Therefore if you want to try out some new programs you can start by writing them on a few EPROMs, and give these to people with EPROM boxes. These people will try out your software, then you can clear the EPROMs, and write corrected versions of the software onto them for further testing. Eventually the tests will be complete, and you may want to ask HP to put your software on a real ROM module. This can be expensive; if there is a limited market for your software then you may prefer to sell it only as EPROMs, and the people who need it will buy EPROM boxes to use it.

As with most computer equipment, EPROM boxes for the HP-41 have been getting smaller. The standard size is a little bigger than an HP-41, but you can buy thinner ones. One make is built into a Card Reader shell and plugs into the top of the HP-41 just like a Card Reader. One difficulty with EPROMs is that they store 8 bits per "word" whereas the HP-41 uses 10-bit "words" in its ROMS. The usual solution is to use one chip for 8 bits and a second chip for the other 2 bits of each word, so EPROMs for the HP-41 usually come in "sets" of two chips - this makes it harder to fit software that is on EPROM into a really small space. The ultimate EPROM box is one that looks just like a plug-in module, some companies are beginning to make these. Small numbers of modules with specialist software can be sold on such EPROM modules instead of normal ROM modules which are cheaper only for large numbers.

Before you start copying your programs to EPROMs you must test them out somehow. Programs written in FOCAL can be tested in RAM, except for the different behaviour of a few instructions like STO b, but even the smallest plug-in ROM contains 4K words of program, which is more than the whole of the HP-41 RAM Main Memory. You need a Software Development System which is another box that plugs into an HP-41. This contains RAM to which you can write programs, but this RAM is then read by the HP-41 as if it was on a plug-in ROM module. RAM used like this with an HP-41 is often called Quasi-ROM or Q-ROM.

The original Software Development System boxes were made by Hewlett-Packard themselves. Several enterprising people later designed different Q-ROM boxes with various names: ProtoCODER, MLDL (Machine Language Development Laboratory) and MLI in particular. The RAM inside such a box does not need to be different from ordinary RAM, but the box must allow the HP-41 to write programs to this box, yet later to read programs from the same box as if it was a plug-in ROM module. The box has to contain its own ROM which contains instructions that control it, just like any other plug-in device. These instructions are used to copy programs to the Q-ROM in the box, and to alter or analyse them. Once the programs have been stored in Q-ROM then the HP-41 can be told to execute them as if the Q-ROM was itself a plug-in ROM module.

When you have a Q-ROM device you can write and test programs destined for inclusion in ROMs. This includes programs written in the language used by the HP-41 CPU. The language has been called M-code; it is quite different from the ordinary user language FOCAL. It lets you control the HP-41 completely. Such things as non-normalised recall, any tones you like, double precision arithmetic at the same speed as normal arithmetic are just a sample of what can be done.

M-code programs can be up to 100 times faster than normal ones. Every normal HP-41 operation is followed by a check of the flags, a test whether any module wants to do something, and various other tidying-up. Every arithmetic operation involves turning all numbers into binary and separation of mantissa and exponent. Then the operation is performed, then the result is turned back into a ten-digit mantissa with a two-digit exponent, the value is normalised and stored in a register. In M-code all these tests and operations can be left till the end of a complete operation. For example the HP-41 goes through all these tests after every step of calculating a hyperbolic sine in user language. In M-code all this would only be done once, after the whole calculation had been completed. All the fundamental operations required to run the HP-41 are already available in the internal ROMs, so many operations can be carried out very

efficiently in M-code just by calling a set of internal routines one after another.

Now for the disadvantages. An M-code program is a string of 10-bit numbers, and you have to know these numbers or some symbols for them; such symbols are usually called **mnemonics**. HP writes HP-41 M-code programs, using their own mnemonics, on larger computers which translate them into the final numbers and turn label names into addresses. Only a few HP-41 users have had the chance to write such **assemblers**. Most have to do their M-code programming from an HP-41 keyboard. To make this simpler various users have devised their own mnemonics, and each user has to understand the mnemonics used by others. Before the mnemonics could even be invented the users had to work out what each instruction actually did.

To begin with, this was done by systematic (and brilliant) guesswork, supported by information from articles about the HP-41 published in some HP technical journals. When HP realised that the users were succeeding in this venture they decided to help, perhaps because it is to their long-term financial advantage, as was the case with Synthetic Programming. Internal M-code listings (called VASM listings by HP) were provided on a NOMAS basis (see Section 14.2), and a few companies began to develop M-code software with HP approval.

There is finally a book available to help you get started with M-code. It is called HP-41 M-code for Beginners, by Ken Emery (published by Synthetix). M-code is also described in the ProtoCODER manuals, in the ZENROM handbook, and (briefly) in the book "Inside the HP-41", which is a translation of the French book "Au Fond de la HP-41". The ZENROM handbook is probably the most complete of these additional sources, as it was published later than the ProtoCODER manual, and is designed for use with the ZENROM to enable M-code programming on a Q-ROM device. Check the CCD ROM manual as well.

Another difficulty with M-code is the need to have programs that help with writing M-code programs. You need programs to translate mnemonics to M-

code numbers and to store these numbers in Q-ROM, programs to move or alter the M-code in a Q-ROM, programs to copy numbers out of a Q-ROM and save them on Mass Storage, and programs to turn these numbers back into a legible listing (disassemblers). Such programs are available on several EPROM sets, and on the ZENROM. If you want to avoid such difficulties you may prefer to use an intermediate language which has many of the features provided by M-code but looks more like FOCAL. Such a language is available on EPROMs from the Melbourne user group; it is called I-code (Intermediate code, what else?). In Europe it is available from PPC-T in France. I-code has been described in PPCTN since issue 14, and summaries have appeared in the PPCJ.

If you do not want to buy M-code equipment then you can use the CCD module to execute sections of M-code from ROMs that are inside your HP-41 or are plugged into it. This endeavour requires listings of the ROMs so that you can work out where to go and why.

Finally, if you do decide to embark on M-code programming you must purchase or build a Q-ROM device of some kind. Circuit diagrams and printed circuit boards are available, but even so the equipment will cost about as much as an HP-41 and will be about the same size as an HP-41. The ultimate M-code device would be an MLI inside a plug-in module case, at the same price as a module. Maybe we shall see one on the market yet!

17.5 Missing functions?

We are never satisfied with just what we have; there could always be a little something extra. Every user would like just one more HP-41 function, but which function? That is why the designers made the HP-41 so versatile; we can write our own programs to replace missing functions. If you did not know before, you certainly know now that many users have taken this process much further, and you can extend your HP-41 as far as you like, with a few extra subroutines, with some peripherals, with system extensions, Synthetic Programming, or even machine language programming. You have seen how some of the non-programmable functions can be made

programmable, and how user groups have written their own modules to provide missing functions. Extra functions have been provided in this book. The tools for more have been described or else the sources of information have been given - now you can go and extend your HP-41 !

Exercises

17.A If you have been keeping a notebook as was suggested in Exercise 1.C then look through it and see how much you have learned about your HP-41, and how much of it you are now using regularly. Would it be worth joining a user group so as to carry on learning useful facts like this?

17.B If you have not yet rushed off to get an M-code device then stop and ask yourself if you need one. Maybe it is just what you need, or maybe you are happy to do what you can with a normal HP-41 and a couple of extra modules. Do the complete control over your HP-41 and the extra speed justify the use of M-code for your work ?

17.C We haven't finished yet! Your final exercise will be to go through the Appendices and look for any more useful information. They list books and journals, sources of equipment, HP-41 bugs (good and bad) and the uses of the HP-41 flags. Finally there is an Appendix with recent HP-41 news.

PART V

Appendices

APPENDIX A - Books and Journals for the HP-41

This Appendix is a list of books, journals and other publications referred to in the text. Some additional books and journals are included but the list does not claim to be complete. For addresses of suppliers see Appendix B. I have not given prices here as they vary, check with the shop.

Manuals

The HP-41CX Owner's Manual (volumes 1 and 2) should be read by all users. If you buy a second-hand HP-41 without a manual then you can order one on its own from HP. Even if you have a 41C or 41CV the 41CX manuals provide useful extra information. The HP-41CV is now sold with only one manual; a second more detailed one has to be purchased separately, the CX manuals may be worth buying instead.

Obviously you should read the manuals for any other equipment you use. The manuals for HP-IL devices only explain how they work; the HP-IL module manual for the HP-41 explains how to use HP-IL to control these devices.

The PPC ROM Manual can be bought separately from the PPC ROM. It contains a wealth of information and advice, plus barcodes and listings for all the programs in the PPC ROM and additional programs. It can be purchased from PPC or EduCALC. A Pocket Guide to the PPC ROM is also available.

The ZENROM manual contains excellent introductions to Synthetic Programming and to M-code. Its purchase price can be credited against the ZENROM module if you buy the module later.

The CCD Module manual (English language version) may be available separately by the time you read this.

The Protosystem manual for the ProtoCODER2 describes how this device works and explains how M-code can be used on it. Available from Prototech, Inc.

HP-41 books

The following books were written specifically for the HP-41 rather than for calculators in general, though some of them contain information that applies to other calculators/computers as well.

"Synthetic Programming on the HP-41C" by Dr. W.C. Wickes (1980). Despite its title it applies to the newer HP-41 models too. It was the first book on SP, some techniques have been improved since, but it is still a valuable and in-depth treatment of Synthetic Programming. Available directly from Larken Publications, 4517 N.W. Queens Avenue, Corvallis, Oregon 97330, USA as well as from the sources listed in Appendix B. It is one of the few HP-41 books on sale in the UK; Dillon's in London and Blackwell's in Oxford sell it. A German edition "Synthetische Programmierung am HP-41" is available from Heldermann Verlag Berlin, see Appendix B.

"HP-41 Synthetic Programming Made Easy" (SPME) by Dr. K. Jarett (1982) is a good beginner's guide and contains some more up-to-date information than Wickes' book but does not go into as much detail on some subjects. It is published by SYNTHETIX and is also available from the main bookshops listed in Appendix B. A German edition "Synthetische Progammierung - leicht gemacht" is available from Heldermann Verlag Berlin.

"HP-41 Extended Functions Made Easy" by Dr. K. Jarett (1983) explains the Extended Functions and Extended Memory, including the HP-41CX functions. It provides Synthetic Programs designed to use and control Extended Memory in addition to those in SPME. Published by SYNTHETIX and available from them or from bookshops. A German edition of the above will be available from Heldermann Verlag.

"Calculator Tips and Routines - Especially for the HP-41C/CV" edited by John Dearing (1981) published by Corvallis Software, Inc., P.O. Box 1412, Corvallis, Oregon 97339-1412, USA and available from bookshops. An extensive compilation of tips and utility routines, including some

synthetic routines. An expanded German edition "Tricks, Tips und Routinen fur Taschenrechner der Serie HP-41" is available from Heldermann Verlag Berlin.

"HP-41/HP-IL System Dictionary" by Cary E. Reinstein (1982) 3rd. printing available from PPC and bookshops. This really is a dictionary - it lists and briefly explains the functions in the HP-41 and the following devices and modules: Time, Extended Functions, HP-IL (including printer functions), Card Reader, Wand, and the PPC ROM. It also explains commonly used terms, including a few terms that HP-67 and HP-97 users have carried over to the HP-41. The Appendices include some useful programs and a list of modules.

Another dictionary-type book is in preparation for release in mid-1986. Its title is to be "Catalog 2 -- a guide to HP-41 System Software", by Sharyle Ann Price. This will contain one-line descriptions of over 4000 functions on HP-41 devices, plug-in ROMs, and EPROMs, sorted by name, source, and XROM number. Kelly Publications, P.O. Box 28023, Santa Ana, CA 92799-8023.

"Curve Fitting for Programmable Calculators" by William M. Kolb 2nd. edition (1983) from IMTEC, P.O. Box 1402, Bowie, Maryland 20716, USA or bookshops. Describes 40 types of curves that can be fitted to data. Includes an HP-41 program to fit 19 curves, together with barcode for this program. Contains programs for other calculators too.

"Data Processing on the HP-41C/CV" Volume 1, by William C. Phillips, published by EduCALC and available from them. This briefly explains the HP-41 system, then gives some good programming advice and concentrates on uses of Extended Functions with Extended Memory files and Mass Storage for data processing. Two main sets of example programs are given; Simple Ordinary Annuities and a Sales program including tax routines. The book is equally well suited for the HP-41CX.

"Inside the HP-41" by Jean-Daniel Dodin (1985) is an English translation of the French book described below, published by SYNTHETIX and available from

them or through bookshops. Briefly describes the electrical layout of the HP-41, then concentrates on the HP-41 operating system as seen by the ordinary programmer, the programmer who uses synthetics, and the M-code programmer. Somewhat useful for M-code beginners, but the translation is directly from the French, including the idioms, and some sentences may need to be read twice. Several useful Appendices contain material from other publications, some of these Appendices were added during translation. One is a straight copy from an April fool column in DATAFILE - it is wise to ask authors for permission to copy articles, otherwise you may not realise what you are copying! The French language original is "Au Fond de la HP-41C" by Jean-Daniel Dodin (1982) published by the author, who is leader of PPC-Toulouse, available from Editions du Cagire or bookshops. It contains few programs but the detailed explanations of the HP-41 are very useful for those who take an interest in such things. "Het Onderste uit de HP-41C" is a Dutch translation of this book available from Editions du Cagire and some European bookshops.

"IND 41" Ed. R. Pulluard (1984), published by Editions du Cagire, is an index of HP-41 programs published in professional, computing and user group journals. The editorial material is in French and English, side by side. Each program description gives the program's title, where it was published, the program size, and an abstract in the original language.

"An Easy Course in Programming the HP-41" by Ted Wadman and Chris Coffin (1983) published by Grapevine Publications, Inc., P.O. Box 118, Corvallis, Oregon 97339, USA, available from the publishers and bookshops. This is just what it says - an easy course for new users of the HP-41. It has lots of pictures and is very friendly, but it occasionally assumes that the reader has already used some sort of simple calculator. You may have noticed how many of the books mentioned are published in Corvallis, home of the HP-41; in this case the authors are former Hewlett-Packard employees.

"HP-41 MCODE for Beginners" by Ken Emery (1985), published by SYNTHETIX, is the first book specifically about M-code. The title is too modest as a lot of the contents are useful for experts, not just beginners.

For example it gives details of the display functions, including those of the newest 41s.

"Optimales Programmieren mit dem HP-41" by Gerhard Kruse (1985), published by Friedr. Vieweg & Sohn, Braunschweig. Available from the publishers (write to Friedr. Vieweg & Sohn, Verlagsgesellschaft mbH, Postfach 5829, D-6200 Wiesbaden 1, FRG.), German bookshops and others that specialise in the HP-41. Contains brief advice on HP-41 programming in general, then covers Synthetic Programming, Extended Functions and the Time module. Gives many short example programs, with barcode. The last two chapters cover uses of the HP-IL Development module, particularly in Synthetic Programming, and provide a financial program. 100 pages long and a very nice book if you read German. Vieweg are publishing two more HP-41 books.

"OS-41, An Operating System for the HP-41 Handheld Computer and Peripherals" by Thomas W. and Ted W. Beers (1984), available from 1808 Summit Drive, West Lafayette, IN 47906, USA. A 472 page reference book with programs to automate complicated HP-41 tasks, particularly on HP-IL devices. Also contains tips and summaries of the usual data transfer functions.

"The HP-41 Synthetic Quick Reference Guide" by Jeremy Smith (1983), published by CodeSmith, 2056 Maple Avenue, Costa Mesa, CA 92627, USA. This is a 40-page pocket-sized reference book particularly for Synthetic Programmers, but with useful information for any HP-41 user. It is printed on tough, paper-like plastic and fits into the HP-41 case if you trim the edges. It is available from the publisher, from bookshops which specialise in the HP-41, and from some user groups (many of whose members have one or more copies because it is such a useful reference).

The "HP-41C Quick Reference Card for Synthetic Programming" is a plastic card with the Byte Table, flag functions, and other information (Table 14.1 is a copy of part of this card). Published by SYNTHETIX and available from them, some user groups, and bookshops, it fits into an HP-41 case easily. As it has hexadecimal codes and their equivalents in decimal and binary,

and the corresponding ASCII characters, it is useful for work on other computers.

General Books

"Algorithms for RPN Calculators" by John A. Ball (1978), published by John Wiley & Sons, Inc. This book is available from bookshops which sell Wiley books, from EduCALC and other specialist bookshops, and you can probably find it in a college library (but having your own copy is better). It was published before the HP-41 was on sale, but provides a wealth of advice and information on programming RPN calculators. If you feel that "Extend your HP-41" has not provided as much mathematical advice as you need then you should get "Algorithms for RPN Calculators". It has an excellent Appendix of references and books.

"Scientific Analysis on the Pocket Calculator" by Jon M. Smith (1977) is also published by Wiley and available from the same sources as Ball's book. It gives detailed instructions for a few important operations, including difference tables, Fourier analysis, Linear Systems simulation, Chebyshev and Rational Polynomial approximations, statistics and Financial analysis. It describes and compares several types of calculator, and has appendices with calculator tricks, matrix and complex number operations, and common formulae. (If you have this book, try reading the cover upside-down.)

"The Art of Computer Programming", Vol. 1 "Fundamental Algorithms" (1973) and Vol. 2 "Seminumerical Algorithms", (try to get the second edition) by Donald E. Knuth, published by Addison-Wesley. Available from scientific bookshops, and in many libraries.

"Sourcebook for Programmable Calculators" published by Texas Instruments contains programs and algorithms for the TI-58 and TI-59 calculators. Many HP users consider their machines superior to TI's, but this book contains some very useful calculator algorithms which can be translated directly to the HP-41. Available from EduCALC and other calculator bookshops.

Many other books on computers and calculators are worth studying; order a catalogue from EduCALC or have a look around a technical bookshop.

HP publications

In addition to the manuals and Solution Books, some other HP publications are worth examining. The Users' Library Catalogue and the Users' Library "Programmer's Reference Guide" are received by everyone who subscribes to the Library (see Appendix B). The Users' Library Solutions books described in Chapter 13 can be bought from the Library or from HP dealers. The "Creating your own HP-41 Bar Code" manual contains useful information.

Journals

I have referred to user group journals at many points in this book. Most larger groups publish a journal, or at least a newsletter, and these contain bright ideas and up-to-date information. A year's membership includes the group's journal, usually at a lower cost than a single plug-in module. For that price you get many times more programs than in a single module, and much else besides.

In general I have referred to articles in journals by journal name, followed by Volume number, issue number, and page number. If an article covers several pages I have given the first and the last. If a particular item of information needs to be located on part of a page, I have used the letters **a** for the top left quarter, **b** for the bottom left quarter, **c** for the top right, and **d** for the bottom right.

The PPC Journal has gone through several name changes. Originally it was 65 Notes, from Vol. 5 to Vol. 6 it was the PPC Journal, from Vol. 7 to Vol. 11 it was the PPC Calculator Journal (and three volumes of the PPC Computer Journal were published), and from Vol. 12 it again became the PPC Journal (with the Computer and Calculator Journals combined). Volumes 6 through 9 contain many useful articles on synthetic programming, and are worth ordering if you are interested in learning more about that subject.

Recent topics include M-code and the HP-71. Write to PPC at their new address: P.O. Box 90579, Long Beach, CA 90809.

The Club of HP Handheld Users (CHHU) was started in August 1984 by Richard Nelson, founder of PPC and editor of the PPC Journals from 1974 to 1984. Since October 1984, CHHU has published the CHHU Chronicle, which is an excellent source of information on the HP-41 and HP-71 systems. Part of this is due to the support that Richard has in his new effort from many of the prolific long-time PPC members. CHHU has grown to 1300 members in its first year. CHHU has ceased operation, and back issues are available from HPX, P.O. Box 566727, Atlanta GA 30356.

The first two issues of the UK journal had no special name, then it became known as DATAFILE. The PPC-T journal became a magazine for public sale in 1984 and has since been known as Micro-Revue. The Melbourne journal, PPC Technical Notes (PPCTN, sometimes referred to as just TN) initially had a volume number and an issue number, but after 8 issues of Vol. 1 they gave up using volume numbers, and all issues of TN are referred to by the issue number alone; it is now called PPMTN. Journals can be ordered from the clubs that publish them; back issues are worth buying or borrowing from a club library. The addresses of the user groups and details of their journals were given in Section 13.6.

APPENDIX B - Sources of Information and Equipment

This Appendix lists companies that sell equipment and publications mentioned in this book. It is not a complete list; if you want more addresses then a good place to start is PPCJ V12N4P20-24. That is an index of products offerred and reviewed in PPCJ from V6N3 (a month before the HP-41 was announced) to V11N9 (the end of 1984), and a splendid piece of work it is. From there you will have to go to the relevant issue and find who supplies the item described.

EduCALC Mail Store, 27953 Cabot Road, Laguna Niguel, CA 92677, USA. They stock HP equipment made by HP and independent suppliers, and a very wide selection of accessories and books. They accept payment in U.S. cheques drawn on U.S. banks or on VISA and Master Charge (ACCESS in the UK) cards. They regularly publish free catalogues with a lot of useful information.

A shop that is similar to EduCALC but is in Europe is TH Boekhandel Prins, Binnenwatersloot 30, NL-2611 BK, Delft, The Netherlands. They too specialise in HP equipment and books and take mail orders.

No British shop specialises in HP equipment and publications to the same extent as these two, but several shops do carry a large stock; one of these is Metyclean Ltd., 92 Victoria St., London SW1, UK.

An HP software specialist in the UK is Zengrange Ltd., Greenfield Road, Leeds LS9 8DB, UK. They make and sell the ZENROM, and they are an HP ICC (Independent Custom Consultant) which means they can help you in the design and manufacture of software on ROMs for the HP-41 and other HP equipment.

The *Oilwell module mentioned in Chapter 12 is designed and sold by Daly Drilling Enterprises, 146 Hamilton Place, Aberdeen AB2 4BB, UK.

Heldermann Verlag Berlin, Nassauische Str. 26, 1000 Berlin 31, Germany publish German translations of the books by Wickes, Dearing, and Jarett.

The CCD module is sold by W & W Software Products GmbH, Odenthaler Str. 214, Postfach 200970, D-5080, Bergisch Gladbach 2, Federal Republic of Germany. W & W has a range of products and services, including two exciting new products for the HP-41: a 32K EPROM box and a 32K Quasi-ROM box with a 4K operating system, each of which are inside a Card Reader shell.

Apart from selling memberships, journals, and controlling the HP-67/97 library, PPC sells some hardware, software and books. Among the interesting software related items they have are the HP VASM listings of the HP-41 M-code ROMs. PPC, P.O. Box 90579, Long Beach,, CA 90809, USA.

HP-41 VASM Micro-Code listings are now available only from PPC and W&W.

User Program Libraries: Some language sections of the HP User Program Library in Europe have been transferred to user clubs, HPCC now has the English section and can provide copies of programs to members. The US library has been transferred to an independent organisation: Solve and Integrate Corporation, 460 S.W. Madison, Suite 5, P.O. Box 1928, Corvallis, OR 97339, USA.

SYNTHETIX, P.O. Box 1080, Berkeley, CA 94701, USA publish books and the plastic Byte Table. The address printed on the Byte Table card is out of date.

Below is a list of companies that sell specialist plug-in equipment; most of the names and addresses were taken from announcements in journals. Some companies may have changed their names, addresses, and the products they make -- write and ask first.

Port Extenders can be bought from EduCALC or TH Boekhandel Prins.

A European maker of M-code equipment is ERAMCO Systems. Their present address is Kruiszwin 2102, 1788 RL Den Helder, The Netherlands. Letters sent to their earlier addresses can go astray; if you have written there and not received a reply then write again to the above address. They also sell other equipment and the David EPROM set which is one of the best for M-code users. The Danish user group (see Chapter 13) also make such equipment. ERAMCO products are sold in the UK by: SOFTWORD, Astage, Rectory Lane, Windlesham GU20 6BW, England.

American makers of M-code and EPROM equipment are given below, the equipment they sell is on the first line, followed by the company name and address:

ProtoCODER (Quasi-ROM), ProtoEPROM (an EPROM box), and other products, Prototech, Inc., P.O. Box 12104, Boulder, CO 80303, USA.

CMT-110 16K or 32K EPROM box (Application Module Simulator),
Corvallis MicroTechnology, Inc., Dept. 100A, 895 NW Grant Ave, Corvallis, OR 97333, USA.
This company also sell the following:
CMT-200 Digital data acquisition and control system for the HP-41.
CMT-300 Programmable measurement system (digital multimeter for the HP-41).
CMT-10 **plug-in module** which at last allows the use of up to 16K of software on EPROM in a standard size HP-41 plug-in module. It is a module-sized equivalent of the EPROM box, suitable for limited production custom ROM modules.

HHP Portable EPROM unit (EPROM unit one in a Card Reader case), and other products, especially HP-IL related items,
Hand Held Products,Inc., 6401 Carmel Road., Suite 110, Charlotte, NC 28226, USA.

APPENDIX C - System Bugs, Nasty Surprises, and ROM Revisions

A great deal of information about "bugs" in the HP-41 system is available, but it is difficult to put it all together in one place. This Appendix is an attempt to put together as much information as is reasonably possible. It should help new users, old hands who can no longer quite remember what a bug such as Bug 9R really is, and it should also help bug-hunters by providing a rough map of a territory that is never fully explored. I have included a reference to the original discovery whenever I could find it but have not always given details of the discoveries or discoverers as these are in the original articles. I have not given references if I have been unable to find them, or if the bug is one I have discovered myself and not previously published.

First of all, just what is a **bug**? In computer terms this word describes an error in a program. Program bugs have to be hunted vigorously and exterminated if the programmer is not to suffer professional and financial embarrassment. (An alternative strategy is to claim that the error is not a bug at all; it is a **feature** which produces odd results because the programmer had a reason for wanting such results. Of course the reason is entirely unconnected with the programmer having written the program at three o'clock in the morning, a month after the original deadline.) On pocket calculators and computers the word "bug" is used to describe behaviour that differs from what the manual says should happen. This is fairly apt because the pocket calculator/computer contains programs that make it behave like a calculating device not a doorstop. The people who write these programs are only human, and on something as complicated as an HP-41 they are bound to leave some bugs. Most users do not know how the programs inside the HP-41 work, but they do know when the programs fail to work, and they call these failures "bugs".

If the manual does not describe the results of some particular action then the user has to predict these results by commonsense. In cases where the actual results are seriously at odds with the commonsense prediction, it is also reasonable to cry "bug!". Where the action is very unusual, or no

clear prediction can be made by commonsense alone, unusual results cannot really be called bugs. They can however have disastrous results, and deserve to be called **nasty surprises** at the very least.

This Appendix contains HP-41 bugs arranged according to the source of the bug, and a separate list of "nasty surprises". Many bugs were first discovered and classified by members of PPC or other user groups, so the numbering here follows the PPC classification wherever possible.

Some bugs are very trivial, since they do not affect any results obtained from the HP-41. Others are produced by deliberate maltreatment of the HP-41. These two types are generally ignored in this list, except for a mention of a couple of examples here. Two trivial bugs concerning the CAT functions were discovered very early on, and most users have discovered these bugs indpendently. CAT n where "n" is any single digit other than 1, 2 or 3 (and 4, 5, 6 on the HP-41CX) produces CAT 3 instead of an error message. CAT IND a where "a" is any stack register (or any other status register) also produces CAT 3 regardless of the contents of the register. These bugs might be slightly annoying, but they do not lead you to design variable geometry aircraft with their wings attached backwards. Indeed CAT IND seems a rather unnecessary luxury since CAT is not programmable anyway. Rather than correct this bug on the HP-41CX (where they had to rewrite part of the CAT function anyway) HP simply left all mention of CAT IND out of the manual.

A second example; if you hold down a key while a program is running, you can expect to get some odd results if flag 25 is set as well and an error occurs. The key is ignored while the program runs, but can make the program restart if flag 25 is cleared, the program stops, and you keep your finger on the key. This does not necessarily deserve to be treated as a bug, it just shows that you should not lean on the HP-41 keyboard while a program is running.

Classical bugs

The first class of bug to consider is that of the bugs known to the PPC community since classical times. They were discovered in the first months and years of the HP-41 product life, and were given official designations: **Bug 1** to **Bug 9**. At about that point the numbering system seemed to break down, additional bugs similar to Bug 9 were not clearly numbered or were given numbers and letters, and further bug numbers were only tentative. I have taken the liberty of giving numbers to two more of these well-recognised bugs, but I have also given the names suggested by the people who reported them.

Bug 1 Early HP-41Cs did not save X in register L when the functions Σ+ and Σ- were performed. A simple fix for this bug is to do STO L before Σ+ or Σ- . Reference: PPCCJ V6N5P27.

Bug 2 The early HP-41Cs allowed RCL IND nn and STO IND nn even if the value in nn came outside the limits of the HP-41 memory (nn could be a data register or a stack register). The contents of these registers could be key assignments and programs as well as normal data. This allowed for fiddling with programs and key assignments and led to many of the discoveries that gave us Synthetic Programming. If you have a Time Module plugged into an HP-41C with this bug then you can store splits into registers 999 to 704 as well. Reference: PPC CJ V6N5P28.

The following explanation assumes you are somewhat familiar with the HP-41 RAM addressing scheme described in Chapter 8. The HP-41 adds the number in register nn to the absolute address of the curtain and uses this result, modulo 1024, as the absolute address of the register to be recalled. The result of the addition should be checked to see if it is less than 1024 (decimal), but HP-41Cs with bug 2 do not make this check. A reference to register 999 therefore accesses register 999-1024 = -25, twenty-five registers below the curtain. The modulo 1024 nature of Bug 2 register access is due to the fact that the HP-41 uses only 10 bits to store register addresses. Early HP-41's did not check the higher bits of the

register address. You can use Bug 2 to recall the status registers, key assignment registers, and Extended Memory registers, but it always normalises them (use REGMOVE or REGSWAP to minimize the extent of this normalisation). You should Master Clear afterwards before you PACK or turn off the machine to avoid a hang-up due to a zero length buffer.

Bug 3 The early HP-41Cs also allowed you to SF IND nn and CF IND nn with any number between 00 and 55 in data register nn. (FS?C IND nn and FC?C IND nn will also work.) Unlike bug 2 this did not work with nn in the stack registers. Bug 3 provides a great deal of control over the HP-41C and was useful in the early development of Synthetic Programming but the meaning of many of the flags was not immediately recognised and HP removed this bug as fast as they could. You can find if your HP-41C has this bug by doing: 49, STO 01, SF IND 01. If the BAT annunciator comes on then you have Bug 3. Reference: PPC CJ V6N5P28.

Bug 3 produces a fascinating display if you do: 46, STO 01, SF IND 01, and then press any numerical key, but only if you enter the 01 by pressing the top left key, not by pressing 0 and 1. The display scrolls various characters, mostly nulls for about 5 seconds. Interesting displays can be obtained if you use any of the top two rows of keys to SF IND nn. If you supply the parameter nn by using the ordinary number keys, then the display will rotate to the left when you press the next key. When the bug was removed the code that produced these displays was also changed, so you cannot get them by using 46, XEQ "SCF" or 46, TOGF.

Bug 4 Early HP-41Cs would compute the SIN of small angles incorrectly. The problem is that the HP-41 first converts an angle from degrees or gradians to radians, and then calculates the SIN of the angle in radians. If the angle is so small that it converts to less than 10↑-100 (ten to the power -100) radians, then the leading 1 of the -100 is ignored. Thus the SIN calculated is that of an angle 10↑100 bigger than that intended. This error cannot occur in radians mode (since no conversion is necessary). It can be avoided by using radians mode when working with small angles, or by calculating the TAN instead of the SIN of angles smaller than 10↑-10 (since

the TAN and the SIN are equal to the accuracy of the HP-41 at such small angles and TAN does not have this bug). Bug 4 was not fixed till later than bugs 1, 2 and 3. Reference: PPC CJ V6N6P30.

Bug 5 CLP would only clear 1089 program lines on early HP-41s. With a printer attached, turned on, and set to NORM or TRACE mode the number of lines cleared would be 233. Get around this by using CLP several times or using DEL 1999 instead, but the latter does not delete the END of the program. Reference: PPC CJ V6N6P30. This reference also gave an update on Bugs 1,2 and 3.

Bug 6 HP-67 and HP-97 programs which contain combinations of number entry, EEX and CHS instructions are not correctly translated to HP-41 programs. For example the HP-67/97 program:

EEX, CHS, 7, CHS, 5

should be translated by the Card Reader into the HP-41 program:

1 E-7, -5

but in fact it is translated as:

E-7-5

This is actually a Card Reader bug (see below). It allows you to create numeric entry lines without a 1 before the exponent, and to create all sorts of other odd-looking numbers. If you are translating a real HP-67/97 program it can be a nuisance. The bug is still uncorrected at the time of writing; maybe HP thinks that anyone who buys an HP-41 nowadays will not have an HP-67 or 97. Reference: PPC CJ V6N6P23.

Bug 7 The second nybble of the seventh byte in the Alpha register is copied along with the first six bytes by an ASTO. If Alpha contains ABCDEFGH then ASTO X should copy ABCDEF to register X, leaving the hexadecimal value:

10,41,42,43,44,45,46

in X. This bug causes register X to contain the following instead:

17,41,42,43,44,45,46

The second nybble of the letter G has been attached to the 1 at the start of the string. Try doing the above, then CLA,ARCL X,ASTO Y,X=Y? If your HP-41 has bug 7 then the answer will be NO. Reference: PPC CJ V6N6P23.

Bug 8 Programs are not decompiled on early HP-41s if the HP-41 is turned off while still in PRGM mode. Program compilation was explained in Chapter 6; if a program is altered but not decompiled then local GTOs and XEQs jump to the old address of the label. After an alteration something else may be at the address of the label, and the program will execute steps in an unexpected order. Write the short program:

LBL "BUG8", CF 25, GTO 01, "1≠ZERO??", LBL 01

Now GTO.. and XEQ "BUG8". This will compile the GTO 01 step. Turn PRGM mode on, SST till you see 03 GTO 01, press X<>Y and ON. This will insert seven new bytes after the GTO 01, so the "≠" is where the LBL 01 was - without decompiling the GTO 01 if you have Bug 8. Now turn the HP-41 back on, press RTN to get back to the beginning of the program, and press R/S. If your HP-41 has Bug 8 you will see NONEXISTENT - why? The compiled GTO 01 jumps to the "≠" and treats it as a global GTO. The Z after the ≠ makes the HP-41 think there is a 10-character long label after the GTO; this label is not found, so you see NONEXISTENT. Turn PRGM mode back on and you will see the .END. - the 10 characters of the assumed label include the END of the program and the HP-41 jumps out of the program, to the .END. .

If you have Bug 8 then programs will not be decompiled if the HP-41 is left in PRGM mode and allowed to turn off automatically after 10 minutes. You can decompile a program by going out of PRGM mode before turning off, by pressing PRGM twice after you have turned back on, or by doing GTO.. or PACK. Reference: PPC CJ V6N8P23. This reference contains another update of previously discovered bugs.

Bug 9 Executing CAT 1 while in PRGM mode and then interrupting it with R/S can result in various oddities related to incorrect step numbering. The original bug and several later versions were discovered by Mark MacLean.

i. If you stop at a LBL and put in a new step then that step remains unseen. Press SST and you will see this step, with the step number 01; you will not be able to BST to the LBL, and all the line numbers will be wrong until you SST past the END or use GTO.nnn.

ii. If you stop at any LBL or END and do DEL 001 then you see the previous step of the program, with the step number 4094. If the LBL you delete is the first step of the program then you get back to the END of the program unless it is the first program in CAT 1. Starting at this large step number you can BST through your program, past the END, into the previous program, and from there into earlier programs.

iii. If you interrupt CAT 1 at the first LBL of the first program, and this LBL is the first step in the program, then things are a little different. Deleting this LBL gets you to the bottom of main memory, which allows you to get to the Key Assignment Registers. (Depending on the SIZE and number of memory modules, this sometimes just gets you back to the .END. -- instead of executing DEL 001 you can press ALPHA, backarrow, ALPHA; this method will get into the assignment registers on all HP-41s, so far as is known.) If you start from MEMORY LOST status, do CAT 1, and R/S then you will be at the .END. following which this method will get you into the key assignment registers too. If you have an ordinary END as the very first item in CAT 1 then deleting that will also get you into the assignment area, but there is a chance that the HP-41 will lock up if you use this method.

Point iii. is the most interesting as it lets you edit the key assignments and create synthetic assignments, including the Byte Grabber as described in Chapter 14. It is this that is usually referred to as Bug 9. The step before the deleted one appears as 4094 because the step number is set to

hexadecimal FFF, decimal 4095, during CAT 1 (and during a running program). The first reference to Bug 9 is in PPC CJ V7N9P25, but a great deal has been written about it in connection with Synthetic Programming. The second reference, in PPC TN N5P35-38 describes an independent discovery of bug 9 and gives the first description of two other CAT interruption bugs. This article was reprinted in PPC CJ V8N4P8.

iv. Interrupt any one of CAT 1, 2 or 3 then XEQ "BST" and press backarrow. The display will scroll backwards, put a G in the leftmost display position (except on an HP-41CX) and lock up. Pressing ENTER or ON gets you back to normal (on an HP-41CX press ENTER and ON at the same time). Try spelling out "SST" instead of "BST", and pressing other keys than backarrow.

v. PACK, execute CAT 1 in PRGM mode, interrupt it, and SST to the .END.. If you have nothing in CAT 1 except the .END. then execute CAT 1 and stop it at once with R/S. Now press ALPHA and enter a string of text. The string will not show up as you enter it, and the bytes are stored further up in program memory. This is similar to i. above, but the positions of the bytes are wrong, not just the line numbering.

The next three bugs are also very similar to Bug 9; they were all described by M. MacLean and K. Jarett in PPCCJ V8N5P8-9. All involve line numbering errors similar to the first three points above, and provide means of entering the assignment area, so they are included here under Bug 9.

vi. Do a MEMORY LOST, then make any two assignments. Press SST, keep it down, press ON and keep that down too. Release SST, then release ON. If NULL appears while you are doing this then start again and do it more quickly. While you had SST down, the HP-41 assumed it was going to the line after the .END. , so it set the line number to 02. As you turned the HP-41 off before the .END. was executed, it did not realise that line number 02 does not exist. Turn the HP-41 on, press PRGM then backarrow, and you are at an imaginary line 01. Press GTO.001 and you get to the top of the "program" containing this step - in fact

this is the first byte of the buffer and key assignment area.

vii. Also known as **Bug 9R**, because it is obtained by going to a program label in a ROM, setting PRGM mode, and pulling out the ROM without turning off the HP-41. If you do this and press the backarrow key then you get into the key assignment and buffer area, and can use GTO.001 to get to the top of it. This is not quite a bug since HP warn users not to pull out modules while the HP-41 is on. Discovered by K. Jarett it is the fastest entry to the key assignment registers.

viii. If you have no room for more program steps, even after packing, then you will see .END. REG 00. If you now SF 25, go into PRGM mode, and press a key then the HP-41 will start running the program you are in. If the program stops at a STOP instruction then the line number will be the number that should have been given to the inserted step (unless the program stops at a PSE or an END). This bug is really quite separate from Bug 9, but it has not been given a number, and it fits in with the other bugs that cause incorrect line numbering. By including a backward GTO in the program, and putting a STOP after the target LBL, you can make your program line numbers too high, and thus get into the assignment area. To check if your HP-41 has this bug:

At the top of Catalog 1, GTO.000 and enter the program

LBL 01, STOP, BEEP, AON, GTO 01, END .

PACK and go to line 000 of the program again. Add the number of registers shown to the present SIZE, and set the SIZE to this value. The display should now show 00 REG 00.

SST to the step 03 BEEP, go out of PRGM mode, SF 25, go back into PRGM mode and press ENTER. See PACKING, then 04 BEEP with ALPHA mode set. Press the backarrow key 3 times and you are at step 01, in the assignment area. Press ALPHA to get out of ALPHA mode. You cannot BST, but you can use GTO.001 to get to the front of the assignment and buffer area. The HP-41 still remembers that you had zero free registers for program editing, so if you accidentally try to put in a

byte too many then instead of moving everything down and getting MEMORY LOST the HP-41 will display PACKING and TRY AGAIN. The second time you may get MEMORY LOST, so be careful.

All the above bugs are related, and most of them exist even on the HP-41CX. They are interesting because they let you create synthetic assignments and edit buffers too. Even if some of them are eventually corrected it seems unlikely that they will all be removed. It is not surprising that having this many bugs under one number caused the bug numbering scheme to go awry.

Bug 10 If flag 25 is set and MEAN or SDEV causes an overflow error then the flag register can be altered. The overflow can only occur if the Σx or Σy value divided by n exceeds 10↑100 and you execute MEAN, or if the Σx or Σy divided by the square root of n exceeds 10↑100 and you execute SDEV. If there is an overflow on y then the original value in register d changes to:

99,99,9x,xx,x9,99,99

where x marks bytes that are unchanged, except that flag 25 is cleared. One serious result is that the HP-41 is in PRGM mode, and will alter program memory if a running program comes across a numeric entry step.

If Σy does not overflow, but there is an overflow on Σx then part of the Σy value is copied to register d. Starting with the bytes in the Σy register represented as:

AA,BB,CC,DD,EE,FF,GG

you finish with register d containing:

DD,EE,FF,xx,AA,BB,CC

The bytes xx are unchanged, except that flag 25 is cleared. The last three nybbles are usually altered in some way because they are in flags 44 to 55 which are system flags that can change, and they can also have odd effects on the HP-41 behaviour if set by this bug.

This bug can only occur if a number smaller than 1 is stored into the n register, so it will not affect normal statistics operations, and HP have been in no hurry to correct it. It will occur only for a limited range of Σx or Σy values; it was used in the example in Section 6.6 to let a running program set the PRGM flag. References: PPCCJ V8N6P45 and V9N4P8.

Bug 11 Originally called the CW1 bug, it is also called the "Return below the .END. bug". If a program at the end of CAT 1 is replaced or deleted then a return to that program will return into the new program, or below the .END. . Originally the Card Reader function RSUB and the Wand functions WNDSUB and WNDLNK were the ones that could replace the last program in memory (the program containing the .END.), so the bug was named after these devices by Bruce Bailey who first described it. Reference: PPC CJ V9N2P40. Since that time new devices which can cause the bug have been brought out, the functions affected are GETP and PCLPS in the Extended Functions module, READP in the HP-IL module, and INP in the Extended I/O module. A list and detailed explanation were given in DATAFILE V2N5P2-6.

A simple example is the following. Do GTO.. and put in a program

CLA, XEQ 01, LBL 01, PCLPS

If you execute this short program then PCLPS deletes the last program in memory, which is this very program. Following PCLPS this program is replaced by the .END. which is executed, and treated as a return. The return is made to the line after XEQ 01, which is now below the .END. and does not exist. The HP-41 drops through to the next instruction it can find which will be a buffer, a timer alarm, or a key assignment. This trick is used by the non-synthetic assignment program GASN in Chapter 11.

One serious bug on early HP-41s did not get a number. LN1+X gave wrong answers for X values from -0.9990234375 to -0.9999999999 . The improved accuracy of LN1+X is not needed at these values, so it is probably best to check for values below -.9 and use 1, +, LN . This will not save X in register L though.

Display Bugs

At about the time when the classical bug numbering scheme broke down, several bugs connected with HP-41 display were discovered. I shall list these bugs, together with ones that were discovered earlier, numbering them roughly in the order in which they were described in PPCCJ.

1. If a text string containing an FF byte is put into any stack or data register and the register is viewed then the FF byte and all bytes after it will be invisible. For example the string 10,41,42,FF,43,44,45 can be put into register X and will be seen as "AB", not as "AB▩CDE". This can be used in word games but not for much else. Reference: PPCCJ V6N5P31a.

2. If you VIEW or AVIEW something and execute an instruction that clears flag 25 then the contents of the display will move around just as if they were the "flying goose". Enter and run the following short program:

LBL "DSPL", PI, SF 25, "?", AVIEW, SF 60, LBL 01, ATAN, GTO 01, END

SF 60 generates an error and clears flag 25. Flag 50, the message flag is cleared at the same time, but the display is not cleared, so the HP-41 acts as if the display had been cleared and the "flying goose" was in it, but actually moves whatever is in the display. The ATAN is just a delaying tactic, without it the display would move too quickly in the LBL 01 loop. You can use this bug to display your favourite message while a program is running, but the message must not be more than 12 display positions long (it can contain extra punctuation though). You can experiment with interesting displays, for example try the string " . , . , . , . , . , ," instead of the "?" in the program above. Reference: PPCCJ V6N8P24.

3. Enter a string of twelve spaces each followed by a comma into the Alpha register, then press the backarrow key repeatedly. Instead of removing the commas and the spaces, the display deletes only commas and ignores the spaces; this is display bug 3. The Alpha register itself acts correctly, commas and spaces are deleted alternatively. If you delete twelve commas

and carry on pressing the backarrow key, then the prompt will vanish, and come back to the right of the display. This bug affects dots, commas and colons with spaces between them. If you turn ALPHA mode off and on again you will see what is really left in Alpha. Reference: PPCJ V7N3P28b.

4. ASTO a text string beginning with a dot, comma or colon in X, stay in ALPHA mode and put twelve characters into Alpha, then switch out of ALPHA mode. The display will show the last Alpha character ahead of the dot (or comma or colon) at the left. This bug occurs if any text string in the X register begins with a punctuation mark and is viewed following any display which had a non-punctuation character in the rightmost display position. After the example above, enter 1E15, X<>Y, VIEW Y, backarrow and you will see 5: -- the HP-41 pushes the display twelve places to the left before showing the contents of register X, but this is not enough if the first character in X is a punctuation mark and is appended to the previous rightmost character. Reference: PPCCJ V9NP6a.

5. If you try to GTO IND or XEQ IND a global label which consists entirely of one or more @ characters, and the label does not exist, then the HP-41 will hang up instead of displaying NONEXISTENT. The label would have to be synthetically created, but the bug can be caused by non-synthetic means using the Extended Functions, for example: CLA, 64, XTOA, ASTO X, GTO IND X will cause a hang-up. This is a display bug because the character code for "@" is zero (when the character is in the display, not in memory), and the HP-41 gets stuck in a loop trying to read back the display before showing NONEXISTENT.

6. The second Card Reader bug described below affects the display.

Card Reader bugs

1. Classical Bug 6 above is really a Card Reader bug - the Card Reader does not know where to put nulls to separate one numeric entry from another. This applies when reading a program card and when using MRG at the line following a numeric entry.

2. Following corrections to early HP-41 bugs, early Card Readers did not treat the display correctly when a 3-digit prompt had one or two digits entered, then backarrowed. If you have a Card Reader which identifies itself as "CARD READER" in Cat 2 and an HP-41 without the early classical bugs then try the following: SIZE 12 (not 012, just 12), backarrow, backarrow and you will see the 1 at the left of the display with SIZE at the right. Pressing backarrow will not clear this display, but most other keys will. DEL, LIST and NEWM do the same thing. Reference: PPCCJ V9N4P5b and P6b.

3. You have this bug unless your Card Reader shows up as CARD RDR 1G or a higher revision in CAT 2. Using VER will cause a register from the card being checked to be written into address 3EF, the top register of an Extended Memory module in port 2 (or port 4 via a Port Extender, or built into a dual or triple Extended Memory module). This register may be somewhere in a data file in which case you will have a bad data value, or in a text file in which case the record lengths may go wrong as well as the text being wrong, or a program will become unusable, or a file header will be altered. Either refrain from executing VER while you are using an Extended Memory module like this, or get your Card Reader ROMs updated by HP. Alternatively take evasive action by using SP; if you copy this register somewhere else while executing VER and then copy it back you will be safe. "Extended Functions Made Easy" has a program to do this. Reference: PPCCJ V9N7P19-20.

4. Up to now all Card Readers have a 7CLREG bug. If you have a complete Main Memory (HP-41C with 4 memory modules or a quad, or a CV or CX) and a SIZE less than 25 then 7CLREG can clear the top of Main Memory and the bottom of the Extended Memory module in port 1 or 3 (or in a Port Extender etc. as above). This will destroy the Extended Memory linkage as well as some data. The solution is simple: set a SIZE of at least 25 before using 7CLREG, indeed the Card Reader manual advises you to do this before using the 67/97 compatibility functions, so maybe this should be classed as a "nasty surprise", not a bug. Still, the response should be NONEXISTENT as

with other 67/97 compatibility functions which try to use nonexistent data registers. Reference: PPCCJ V10N4P6b-c.

5. 7RCL Σ does not save X in register L as the corresponding 67/97 function does. This is pretty trivial unless you are using a translated 67/97 program which uses statistics and the L register.

6. One bug mentioned in HP reports, but never reported in PPCCJ is that reading a program which fits on more than one card can sometimes put rubbish into register T and/or some other unspecified register. This sounds quite horrifying if the "other register" contains data or part of a program, but if it has never been reported in PPCCJ then it is likely to be a rare occurrence. Just in case, if you are running very important programs, load data registers and the stack after reading a long program. The bug was fixed at the end of 1981, so Card Readers made or repaired since then should not have it.

Extended Memory, Extended Functions, and the HP-41CX

Two Card Reader bugs which affect EM have been described above. If your HP-41C has Bug 2 (see above), REGMOVE and REGSWAP will not notice the break between Main Memory and Extended Memory. This can be used for non-normalised recall as was described in Section 16.9, but is more likely to destroy data and the EM linkage.

If you have two EM modules and the link registers are initialised because you have created data files that use them all then avoid taking out the middle module for a short time and then replacing it. If you have a single large file which spans all three modules then you could delete it by turning off the HP-41, removing the module that is in the middle of the file, turning the HP-41 back on, and creating a new file. Now turn the HP-41 back off, replace the other EM module, and turn back on. If you did this quickly then the linkage register in the module you removed still points to the second EM module. However the base module now also points to the second module. If you do an EMDIR then the module you replaced cannot

be detected; its EM is unusable. If you leave the EM module disconnected for a few minutes then the contents of the link register will discharge and you will be able to plug it back in safely.

So much for EM itself, now for the Extended Functions. The original version (-EXT FCN 1B) contained several nasty bugs. Some modules which show up in CAT 2 as -EXT FCN 1B have had a few of these bugs removed; HP apparently made some corrections without using a new version number; this has happened with some revisions of the HP-41 internal ROMs too. Pre-production models of the HP-41CX had most of these bugs, and users who had access to these spread the rumour that many of the bugs were in the CX (I did this myself in an article in DATAFILE). In fact -EXT FCN 1C had some bugs corrected, and -EXT FCN 2D (in production models of the HP-41CX) had more corrections. Some of the information here was provided by Frank Wales, to whom many thanks. First of all, here are the version 1B bugs.

PURFL purges a file and so there is no working file. Until you establish a working file by using EMDIR, or by using a function such as SEEKPTA, or FLSIZE, with a new name in Alpha, you must not do anything to the working file. If you do, the Extended Functions see there is no working file, assume EM is empty, display DIR EMPTY instead of FL NOT FOUND, and clear all EM. I gave a full list of which functions do this, and when, in DATAFILE V2N5P5. In fact EM is not cleared completely, there is just an end-of-memory mark put in register 0BF. You can recover from this by storing a file name synthetically in absolute register 0BF. (For example do "FIRSTFL", RCL M and then store this in register 0BF by moving the curtain, using the PPC ROM function SX, or using the ZENROM or CCD module absolute storage functions.)

If you run a ROM program or go to a ROM program and stay at it, then put the name of a CAT 1 label in Alpha and execute PCLPS from the keyboard, then instead of clearing the required program and those after it, PCLPS alters the contents of some of the status registers (which can lead to loss of CAT 1 linkage or MEMORY LOST). This takes a few seconds, and you may be able to avoid amnesia by removing the batteries quickly.

Much the same happens with SAVEP. If the program pointer is in a ROM and you put a RAM label name in Alpha then SAVEP can destroy some EM files or even do MEMORY LOST. A simple way around both these bugs is to do a CAT 1 before executing either function from the keyboard.

POSFL changes the stack in different ways depending on whether or not the text is found. This problem was described under POSFL in Section 11.5 and a way around it was suggested there.

Two more bugs were described in Sections 16.5 and 16.6. One is that a string of seven FF bytes in one register is considered to be the end-of-memory marker even if it is not at the end of a file. During the re-packing of Extended Memory that follows a PURFL, the Extended Functions module will copy files into the area left by the deleted file, but only up to the first register of FFs found, not to the true end-of-memory. Since EMDIR knows what the file sizes really are, it ignores the string of FFs, so after a PURFL it can end up looking for the next file header in the middle of the next file.

If the files that were moved up in memory after the PURFL include the one that contained the string of FFs, then a copy of that file will still lie further down in EM. Now EMDIR (and other functions that search EM) do not search the directory in one pass through Extended Memory. Instead, they ask an internal subroutine for the name of file number 1, then they ask it for the name of the first file after the one just found, then the first file name after that, and so on. This works fine as long as names are not duplicated. But following a PURFL it is possible for a name to be duplicated. Thus the file after file "X" is not uniquely defined, and the EMDIR can get into a loop. As is pointed out in Section 16.6, other functions can also be affected by this unnecessary name check. For example it sometimes stops the program ED in Section 16.9. The updated Extended Functions still check the name, but they also check the file number, so this bug does not affect CXs.

The following bugs or nasty surprises have survived in the updated Extended Functions, so you will find them in old and new versions. ANUM does not read numbers cleanly from Alpha but follows the convoluted logic outlined under ANUM in Section 10.2; this is not really a bug as was explained there, but you should be aware of it. REGSWAP and REGMOVE can move overlapping blocks, resulting in the interesting results described in Section 10.3; this too is not really a bug, except that the CX manual on page 201 says "the blocks can't overlap". This suggests an error message will be displayed if they do; in fact it means they shouldn't overlap. At the bottom of the same page it says under REGSWAP that which block is the source and which is the destination "is immaterial" - in fact it is immaterial unless they overlap.

GETP and PCLPS can put you into an unexpected position if you return to a location which has been altered by either of them. This is really a special case of Classical Bug 11, the CW1 bug.

STOFLAG does not change the display annunciators if it is used to copy the status of all the flags 00-43 back from the X register. Try the following program:

SF 01, RAD, RCLFLAG, CF 01, DEG, STOFLAG, 99, GETKEYX

The annunciators for 1 and RAD will not be set in this example until you have pressed a key or waited for the 99 seconds. The annunciators will be reset only when you alter the status of a flag which causes the display to be changed.

Once I am on the subject of flags you may like to know that the function X<>F has the same effect when executed by a control alarm; the function is executed by the alarm, but neither the number in the display nor the flag settings displayed are changed. I cannot think why anyone should want to use X<>F in a control alarm, but it is nice to be warned.

The Extended Functions in the CX have two more nasty surprises. ED puts a text marker into an empty record, and does not remove this marker if you

are in insert mode; this was mentioned in Section 11.9. The indirect comparison functions normalise anything that is in a numbered data register before they compare it, this was mentioned in Section 10.6.

The HP-41CX also has two bugs that are more serious and another nasty surprise. The first bug concerns EMDIRX and file names. It is possible for a file to have its first header register (the one with the name) at the bottom of one block of EM and its second header register (the one with the file type) at the top of another block of EM. If this happens then EMDIRX takes the file type correctly from the second header register, but it then takes the file name from the immediately preceding register which in this case does not exist. In other words, if a file name is stored in absolute registers 041, 202 or 302 (hex) then EMDIRX returns a null string to the Alpha register instead of giving the file name. Reference: PPCCJ V11N1p33b.

The second CX bug ocurs if you have an XROM NN,25 instruction in a program. If a module with the identifier NN is plugged in and contains a function or program with the number 25 then this is correctly displayed and executed. For example if you plug in the Clinical Lab module and enter the step XEQ "*9" then this is recorded as XROM 19,25 and is displayed as XROM ᵀ*9. If no module with the identifier NN is plugged in then such a step appears as XROM NN,25 and generates NONEXISTENT if it is executed, for example if the Clinical Lab module is removed then the step described above is displayed as XROM 19,25. All this is normal behaviour so far.

Here is the bug: If another module with the same identifier NN but with fewer than 25 functions is plugged in then this step is incorrectly interpreted by the CX as INSREC. For example the Aviation module contains instructions from XROM 19,00 to XROM 19,20. If this module is plugged into an HP-41C or CV which contains an XROM 19,25 step then this step does not change. However on an HP-41CX it turns into the step INSREC.

From Table 12.2 you can see that this may happen with several other combinations of modules; another example is replacing a Standard Pack

which has functions XROM 05,00 to XROM 05,63 with a ZENROM which has functions up to XROM 05,12. The most common way this can happen is if you have an HP-IL printer and enter the printer function FMT (XROM 29,25), then remove the IL module or set it to "Disable" and plug in an HP82143A printer which has no FMT function. Instead of replacing FMT with XROM 29,25 an HP-41CX will replace it with INSREC and will execute INSREC, with possibly disastrous results. (On an HP-41C or CV the XROM 29,25 produces a quite different bug; see "Bugs due to more than one device" below.)

A rather trivial nasty surprise on the HP-41CX is that CAT 2 will only show module names if they are over seven characters long, as was mentioned in Section 10.5. You can overcome this by pressing ENTER to force every function and program name to be displayed; in any case it only affects the MATH 1B module (so it can really be considered to be a Math module bug).

Wand

Some of the Alpha barcode functions are treated differently by different Wands; this can be a nasty surprise if you change Wands. If you execute a numeric function from the paper keyboard while the HP-41 is in Alpha mode and not in PRGM mode then the Wand should clear Alpha mode and execute the function. However if you scan the % function in Alpha mode it puts a % into the Alpha register instead. There are several other anomalies in the opposite direction too; if you scan some of the Alpha barcodes they can execute as functions. The results depend on the Wand revision, and whether Alpha and PRGM mode are set or clear. A list of these anomalies is given in the middle of the Synthetic Quick Reference Guide. Another nasty surprise is that revision 1E wands cannot read sequenced data barcode; see Section 12.3.

HP-IL Development Module

This module has several bugs, some good, some bad, some indifferent. The best bug is that RG-BUFX has the same register wrap-round feature as Bug 2 on the old HP-41Cs. This means that register numbers up to 999 can be used

to recall RAM registers to the buffer, without normalisation. The program NNR in Section 16.9 shows how this can be exploited. The wrap-around does not work if SIZE is 000, since in that case all registers are NONEXISTENT.

An indifferent bug is that an HP-41 used in SCOPE mode can sometimes leave its display blinking at the end of displaying a set of IL messages and printing them on an HP82143A printer. This does not affect the display of further messages when more are sent round the loop being monitored. It is usually a result of the IL module being monitored continuously checking whether the printer PRINT button has been pressed; the messages involved are not printed in SCOPE mode but are displayed and recorded in the buffer.

A bug that can be good or bad is that the hexadecimal entry keys A-F are not disabled during execution of OCTIN and BININ. When one of these keys is pressed before any numeric digits have been entered, the key's default function is displayed as if it was a program step and the input function is terminated; this happens in USER mode too. If you release the key before the function is nulled then the function you are executing (OCTIN or BININ) is stored as a program step at the current position in the current program, even if you are in run mode. If you have entered at least one digit, numeric entry is terminated and the default function, or the assigned function in USER mode, is executed, as should happen. As it alters programs unexpectedly, this is a bad bug unless you are a synthetic programmer who wants to change a program without decompiling it. The OCTIN or BININ step will open up a new register if there are not two free bytes available at the current position. This will change the positions of other bytes in the program, but compiled jumps will still be made, and will move by the originally compiled distance. The program will be decompiled when you next enter and leave PRGM mode, or when you turn the HP-41 off and on (unless you have Bug 8). If the OCTIN or BININ is executed in a running program or single-stepped then the program pointer will move and a byte will be entered into the program at its new position. These OCTIN and BININ bugs are also present in the Advantage module.

BSIZE? does not display the least significant digit if the buffer size is 1000 or more bytes. Thus a buffer length of 1232 will be displayed as 123 - a nuisance, but you will probably remember if you have created a very long buffer.

There are two problems with BIT? - first of all the manual says that when it is executed in a program then the next program step is skipped if the test is true. In fact the test works like all other do-if-true tests; the next step is executed if the test is true - it is not skipped. Secondly the result of the test is not always displayed if BIT? is executed from the keyboard. What happens is that when the test is made the bit string in Y is rotated 7 bits to the right, and the test is repeated. If the result of the second test is true then the result of the first test is not displayed. Two examples:

128, ENTER, 0, BIT?

the result should be NO, but nothing is displayed because bit 0 is set after 128 is rotated 7 bits to the right.

1, ENTER, 25, BIT?

should also display NO but displays nothing. Both the action and the description of BIT? are fixed in the Advantage module.

Bugs on other plug-ins

A list of all bugs in all plug-in modules and devices would take a lot more room and would never be complete so long as new plug-ins are introduced and old ones are updated. This is particularly true of the HP-IL module and system; even things which seemed perfectly reasonable when the IL module was introduced may turn out to cause problems when new devices are made.

For these reasons I have not tried to provide a list of HP-IL bugs or bugs in Application Packs. You should always be aware that a plug-in module or Application Pack might have a bug; never use them uncritically without

running a few tests. This is not a criticism of anyone, just plain common sense. Even if your module has no bugs you should have checked if you had the bad luck to get one that was damaged somewhere before it reached you.

Users of the PPC ROM should by now be aware of its hostility toward Timer Alarms and I/O buffers (neither were known to the public at the time the PPC ROM was developed). You should clear alarms and buffers before using any of the PPC ROM's key assignment programs. Also be aware that when a Time module (or CX) is present, assignments to key -61 are likely to be suspended. Just read in a program from Extended Memory or any other source to reactivate your assignment.

One further bug does deserve a mention because it is not immediately obvious but can lead to very serious problems. If you use the normal Alpha keyboard to enter text lines in a program while a CCD module is plugged in you should never press the SPACE key. Press USER, SPACE, USER if you must enter a space. Pressing the SPACE key in normal Alpha mode can alter a single program byte, either in the program you are editing or in some other program in the HP-41. Since the altered byte need not be in the program you are editing, this may go completely unnoticed, which can be disastrous; it appears to have led to at least one error in a program listing in the first printing of this book. This bug will probably be corrected quickly.

Bugs due to more than one device

Some of the bugs described above have been due to interactions between two devices or modules. Here is one bug that is due to the interaction of the printer with the Time module and the Extended Functions. It occurs if you have a program which contains the HP-IL printer function FMT, or the synthetically entered XROM 29,25 and you do not have an HP-IL printer attached and enabled. On an HP-41C or CV if you do not have a Time module but have an old HP82143A printer attached then this function causes a **fading crash** - the display fades out, then returns after about a minute. If you have a Time module as well then the function displays as TM1C@ and turns the HP-41 off when executed.

On an HP-41CX this function brings about a different bug; the XROM NN,25 bug described above with the other 41CX bugs. FMT displays and executes as INSREC if you have an old printer attached. This could be quite dangerous; it is entirely possible that a CX user will get a program from someone who has an IL printer, and will try to run it with an old printer. A novice would certainly not expect their Extended Memory to be altered by this.

Nasty surprises

This is really a reminder of other nasty surprises not yet mentioned here but described in the main body of the book.

ISG and DSE truncate instead of rounding the low order digits of the control value.

Control alarms can suddenly interrupt a running program or a keyboard calculation. They can also display NONEXISTENT if they go off but cannot find a program or function because a required program has been deleted or a module has been removed.

ALMNOW only executes control alarms; it is easy to forget that it ignores other past-due alarms.

There is an SST function in the Devel ROM; do not Alpha execute SST if you want to single-step while a Devel is plugged-in.

CLOCK and SHIFT, ON (except on the HP-41CX) clear flags 12 to 20.

Setting the time with a negative fraction does not give a P.M. time whereas a negative number less than -1. does give a P.M. time.

X=0? and other comparisons with zero give an error if X contains a text string, and if flag 25 is set then they execute the next step, though you did not expect them to. See the notes on this at the end of Section 6.7.

The routines in the Math module have been renumbered, so a program which was written using a routine from an old Math module might execute a different one when executed with a new Math module.

Finally, you may have some nasty surprises if you come across any misprints in this book; sorry, they are my contribution to the bug list.

ROM Revisions

Many of the bugs described above affect only certain versions of the HP-41 or of a plug-in. As the bugs are in the programs in the Read Only Memory (ROM) of the HP-41 or module, you may like to know which ROM version you have. If you publish a program which uses a particular bug you should certainly include some information to let the readers determine if they have the same ROM version as you do. This last part of Appendix C will cover the subject of ROM versions. It assumes you have read Chapter 8.

The first thing to find out is the date when the HP-41 or plug-in was made. At the bottom of every module there is a four-digit date code stamped into the plastic. This is not the same as the type number on the label under the module, the date code is in the plastic itself and is not so easily seen. The four digits can be read as YYWW, where YY is a year code and WW is a week code. Add YY to 1960 to give the year when the module was made. WW gives the number of the week when the module was made in that year. WW is actually the week of manufacture + 8 which is supposed to allow for an 8-week delay between manufacture and arrival at the retail outlet where the module is to be sold. The year and week numbers are not always corrected for the fact that there are only 52 weeks in a year, so a code of 2055 means the module was made 8 weeks earlier, in week 47 of 1980. Since 1984 some modules have a year code which gives the actual year of manufacture; 84, 85 and so on. Some custom modules have a serial number instead of a date code, but this does not happen to modules made by HP for general sale.

Larger devices like printers or HP-41s themselves have the same date code followed by a letter to give the country of manufacture (A for USA, B for Brazil, J for Japan, Q for United Kingdom and S for Singapore). The letter is followed by a five-digit sequence number within the week. These date codes tell you when modules and devices were made, but if they have been repaired then the ROMs have probably been updated and you need to identify the ROM versions. The simplest way to do this is by looking at the module header in a CAT 2. Some module versions have never been sold; for example you can find Finance modules which show up in CAT 2 as FINANCE 1B or FINANCE 1C or FINANCE 1D, but not FINANCE 1A.

Some modules have undergone more revisions than this version numbering would suggest, and catalog inspection cannot be used to check the ROM revisions of an HP-41, since these do not appear in any CAT. A ROM revision code (as oppsed to the version number) is placed at the end of every 4K module. This revision code can be read from these addresses (FFB to FFE) by the Devel module's function ROMCHKX; for example you can plug in an Extended Functions module which shows up as -EXT FCN 1B and execute 25, ROMCHKX to see the revision code show up as AP-1B. This is the module name and revision, AP-1B stands for Advanced Programming ROM 1B (the Extended Functions module was originally called the Advanced Programming ROM by HP).

You have to own a Devel module to use ROMCHKX, and in any case it does not work for the ROM revisions of the internal ROMs in the HP-41. However you can use Synthetic Programming instead to find revision numbers. This will work as well to find the revision numbers of the internal ROMs; these are stored as a single letter at address FFE in each internal 4K ROM. Here is a method described by Jeremy Smith in PPCCJ V10N10p17-18. Before you start you must have the synthetic functions RCL M and STO b assigned to two keys and you must be in USER mode.

1. Create the non-normalised number 00000000000FFD and put it in X. One way to do this would be to use the program BAB from Section 14.8 of this book by doing: CLA, 15, XEQ "BAB", 253, XEQ "BAB", RCL M

2. Go to any program in any ROM. The important thing is that the program pointer must be pointing to a program step in ROM, not in RAM. If you have any Application Pack then GTO a program label in it (say GTO "SINH" if you have a Math module), if you have a printer GTO "PRPLOT", if you have a Wand then GTO "WNDTST". Do not try to GTO a function, nor to a label in CAT 1. Now go on immediately to step 3 below, without moving the program pointer.

3. Press the key to which you have assigned STO b. This puts you at ROM address 0FFD.

4. Press SST and keep the key down to see the next program step (do not go into PRGM mode, just stay in normal mode, press SST and keep it down). You should see a program step which is a numbered label. LBL 00 corresponds to a revision A, LBL 01 corresponds to revision B, and so on up to LBL 14 which corresponds to revision O.

5. This gives you a revision number for internal ROM 0, at address 0FFD. Now repeat steps 1 to 4 using the address 1FFD for internal ROM 1, and again with address 2FFD for internal ROM 2.

6. Write down the three ROM revisions in the order 0,1,2. This will identify your HP-41 and the bugs it has. The oldest HP-41Cs, which have all the classical bugs 1 to 11 (except bug 6 which is a Card Reader bug) have ROM revisions DDE and have date codes up to 1940. HP-41Cs with date codes 1936 to 1952 have ROM revisions FDE and all the bugs except number 3. Note that there is some overlap in date codes since new ROMs can be used at the same time as stocks of older ones are being finished. The table below gives a list of dates, bugs and ROM revisions. The information in it is mostly taken from an article by Jeremy Smith in PPCJ V11N9p16-17, with some additions. If you ever come across an HP-41 with ROM revisions earlier than DDE then it will be a test version released only within HP, and it will have some additional strange bugs which have not been described above. HP-41s with combinations of ROM versions that are not given here will either be new versions, or (most unlikely) they will have been repaired and

given an unusual combination of ROMs.

ROMs	Date codes	Comments
DDE	1926-1940	All classical bugs
FDE	1936-1952	Bug 3 removed
FEE	1951-2034	Bugs 1,2,4 removed as well
GFF	2035-84??	Bugs 5,7,8 removed too.The first version of the CV
HFF	84??-present	The current version of the HP-41CV
NFL	2329-present	Current version of the HP-41CX

Revision numbers are not always changed when small changes are made to the instructions in a ROM. The HP-41CX revision numbers have not been changed, though a number of changes have been made to the reset code (what happens when you press ON while holding down ENTER and backarrow). Even the change in the turn-on code which makes it possible to do a MEMORY LOST when you have a zero length buffer did not merit a new revision number.

7. If you want to use the same method to identify any other ROM them replace the first digit of the address with any other digit from 3 to F (check Section 8.3, Figure 8.6 to determine the address of any ROM) and replace the last digit F with an A. At step 4 above press SST four times to see four steps, each of which represents a letter or number. Do not keep the SST key down too long or you will NULL the step and will see the same step again next time you press SST. Use the Byte Table (Table 14.1) to translate each step. Only the lower nybble is sure to be correct so you will have to use your judgement to translate the program steps into the appropriate codes, which are usually two letters followed by a number and a letter. Note that the codes are read out in reverse order by this process.

APPENDIX D - HP-41 System Flags

The list below describes each flag and gives its status when the HP-41 is turned on or reset (by a MEMORY LOST). C means that the flag is cleared, S means set, and M means the status is maintained without change. ? means the status is determined by external factors such as the presence or absence of a printer.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
00-10	User Flags	C	M	For user's work, display shows if any of flags 0 - 4 are set. Flags 9 and 10 are used by some functions in the PPC ROM, the Advantage ROM, and some others. Check the manuals before using these flags in your own programs.
11	Automatic Execution	C	C	If this flag is set at turn-off, program execution will restart at the current program line at turn-on and a tone will sound. If the flag is set before a program is saved on card or cassette, that program will begin executing when it is read in.
12	Double-wide	C	C	Set these flags to print in the style indicated by their names.
13	Lower case	C	C	
14	Overwrite	C	C	Set this flag to let the Card Reader write on clipped cards. It is cleared when the operation ends.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
15-16	IL Printer	C,C	C,C	These flags set the mode of the larger IL printers in the same way as the Mode switch on the HP82162A Thermal Printer. The modes are: 15 16 Printer mode CLR CLR MAN CLR SET NORM SET CLR TRACE SET SET TRACE/STACK
17	Record Incomplete	C	C	Set by EFM or IL if a record has not been read completely into the Alpha register because it will not fit. When set it also prevents OUTA adding an end-of-line character, so that larger printers will not go to a new line every 24 characters.
18	Interrupt enable	C	C	Used by the HP-IL Devel. module; if set then an interrupt from an IL device will wake the HP-41 from standby or sleep mode and execute a program with the label INTR. One use for this is to allow execution of programs by pressing PRINT or ADV on an HP82162A printer.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
19-20	General use	C,C	C,C	For general use like flags 0-10 but cleared at turn-on like flags 12-18, and might be used for future peripherals.
21	Printer Enable	?	?	Automatically set with flag 55 if a printer is detected. If it is set AVIEW and VIEW print their results, or stop a program if the printer is off or flag 55 is clear.
22	Numeric Input	C	C	Set if a number is entered into X, even if X is then cleared. Can be tested to check for number entry, but clear it before next use.
23	ALPHA	C	C	Set when characters are entered into ALPHA; use it like flag 22.
24	Range Error Ignore	C	C	If set allows OUT OF RANGE errors to be ignored and have no effect on flag 25. ±9.999999999E99 is used to replace the result and flag 24 remains set.

Note: Flags 22 and 23 are only set by keyboard entry and ANUM (or ANUMDEL), not by entry from a program.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
25	Error Ignore	C	C	Allows one error of any type to be ignored (except Devel ROM I/O errors) and is then cleared. Use FS?C and FC?C to test and clear it; otherwise it might cause a serious and unexpected error to be ignored.
26	Audio Enable	S	S	Clear this to disable TONE/BEEP.
27	User Keyboard	C	M	Set this to enable USER keyboard.
28	Decimal Point	S	M	If set, a decimal point separates the integer and fractional parts of numbers and a comma acts as a digit grouping mark (see below), if clear these roles are reversed.
29	Digit Grouping	S	M	If set, groups of 3 digits are separated by grouping marks, if not this is suppressed as is the mark after numbers displayed in FIX 0.
30	CAT	C	C	Set during CAT execution. Always clear when tested unless set by SP. If set it may produce odd CATs.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
31	Date Format	C	M	Set by DMY to display dates as Day.Month.Year and treat numbers in X as having the day in the integer part. Cleared by MDY to change the order to Month/Day/Year.
32	Manual I/O	C	M	Used by HP-IL; set by MANIO to prevent automatic selection of the device to carry out an operation, cleared by AUTOIO.
33	IL Lock	C	M	Set to prevent sending commands by IL; used by Devel module to let the 41 act only as a loop monitor.
34	Auto	C	M	Set by the Extended I/O function ADROFF to prevent auto-addressing and status changing of IL devices.
35	Disable Auto Start	C	M	When set, this disables the auto start feature of the Auto-start/Dup module. HP provide no functions to control this flag; you must use SP!
36-39	No. of Digits	CSCC	M	The number of digits to display after the decimal point, expressed as a hexadecimal digit.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
40-41	Display Mode	S,C	M	Stores the display mode as below: 40 41 Display mode CLR CLR SCI CLR SET ENG SET CLR FIX SET SET FIX/ENG: can only be set synthetically and acts like FIX but overflows to ENG instead of SCI.
42-43	Angle Mode	C,C	M	Store the angle mode as follows: 42 43 Angle mode CLR CLR DEG CLR SET RAD SET CLR GRAD SET SET RAD
44	Continuous On	C	C	Set by the function ON to prevent automatic turnoff after 10 minutes.
45	System Data Entry	C	C	Set during number entry or Alpha entry or appending. Always clear if tested unless set synthetically, which can be useful as described in Sections 15.7 and 16.3.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
46	Partial Key Sequence	C	C	Set when the 41 is prompting with the underscores for a parameter.
47	SHIFT	C	C	Indicates SHIFT is on, can be tested in a program as shown with the program F47 in Section 6.9.
48	ALPHA	C	C	This indicates that the ALPHA keyboard is active.
49	Low Battery	?	?	Set whenever the 41 detects a low battery condition. Worth checking during long programs to let these turn the HP-41 off if power is low. Cleared if power is OK at turn-on.
50	Message	C	C	Set when a message is displayed by the user with PROMPT, VIEW or AVIEW, or by the HP-41, e.g. after an error.
51	SST	C	C	Set during an SST so it is always clear in run mode but can be tested in a program. FC? 51, STO c can be used to prevent MEMORY LOST if a program is being single-stepped.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
52	PRGM Mode	C	C	Set during program editing, but not during a running program. If it is set synthetically in a running program then numeric entry lines are not executed; they start the HP-41 programming itself.
53	I/O request	C	C	Can be set by peripheral devices to request some action. Tested and cleared by the 41 after every step, so it never appears to be set in a FOCAL program. If set synthetically it is cleared at once and flag 54 is cleared as well. If flags 53 and 45 are both set then flags 30, 45, 47, 50, 53 and 54 are all cleared. As this flag is never set it can be used in tests that need a definite result, for example to skip a step as suggested in Section 6.7.
54	PSE	C	C	Set during a PSE, always clear if tested unless set synthetically. If it is set and a running program reaches the END or RTN which should stop it then the 41 will only PSE and then continue execution. This could be disastrous.

Flag Number	Flag Name	Flag status at: Reset	Turn-on	Notes
55	Printer Existence	?	?	The HP-41 automatically sets flag 55 and flag 21 too when it detects the presence of a printer attached to the 41. In the case of the older 82143A printer this is even if the printer is off. Clearing this flag synthetically disables a printer and speeds up program execution if a printer is attached.

Note: Normal HP-41 flag functions can test all these flags, but can only set or clear flags 0 to 29. All the flags 0 to 55 can be set or cleared by the program SCF in Section 14.8 of this book. The PPC ROM, ZENROM, Paname and the CCD module all provide functions to set and clear flags above number 29. The HP-41 CPU has additional internal flags which can only be affected by M-code instructions.

APPENDIX E - Recent changes to the HP-41 Series and New Products

1. The HP-41C is no longer made. Remaining stocks are still being sold, some at reduced prices. The HP-41CV and HP-41CX have undergone an internal redesign (see below) and will continue to be made.

2. The internal layout and circuits of the HP-41CV and HP-41CX have been redesigned to make manufacture and repair easier; almost everything is now on one chip. As far as sales are concerned there is no change and shops will probably not even know anything has happened. The programming of the new design is the same as before, and Synthetic Programming works on the new HP-41s. They will gradually become available as old stocks sell out.

The display has been redesigned; in fact the only easy way to tell the new design from the old one is that the display has a black "frame" with rounded corners. The display has also been internally redesigned so that it can show more characters, including all lower case characters, and its contrast can be adjusted. However - to avoid confusing the customers - the new characters are not accessible by any means except special M-code programs. This means that an old user who buys a new HP-41 will not see any difference; it also means there will be some frustrated customers.

The changed circuitry uses two extra RAM registers, and some of the CCD module math functions affect these registers, causing the display to behave oddly until another function is executed. This new design (code named the **halfnut** inside HP, because the original HP-41 design was called the **coconut**) has also been used for a new layout of the HP-41C. This new HP-41C will not be sold, it will only be used by Repair Centres, so getting your HP-41C repaired will be the only way to get a halfnut HP-41C.

3. Following the departure of Richard Nelson from PPC and his setting up of CHHU there now exist two user groups which claim international support. Many members of PPC have joined CHHU, and several local and national user clubs have changed their names to eliminate the letters "PPC". These name changes do not imply any serious changes in club policies; Richard Nelson

set the pattern for an effective and independent user group, and most clubs have confirmed that imitation is the sincerest form of flattery.

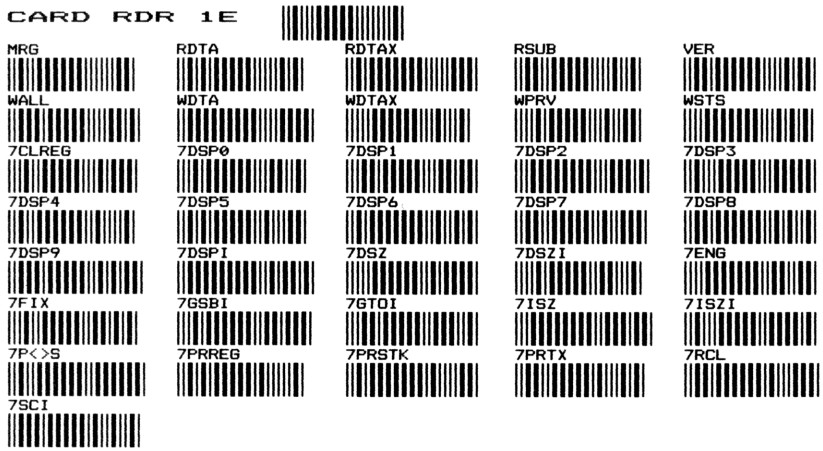
4. The Math and Statistics modules have been combined and been sold as a single plug-in module since late 1985.

APPENDIX F - Barcodes

The following pages contain three types of barcodes. Function barcodes are provided for all the Time and Extended Functions including the HP-41CX functions as these are not on the Paper Keyboard, and also for some plug-in modules and devices. Alpha barcodes for all characters from the first half of the Byte Table are given as well. The uses of these two kinds of barcode were described in Section 12.3. Program barcodes are provided for some longer programs and those with a lot of synthetic instructions.







—MASS ST 1H



CREAT



DIR



NEWM



PURGE



READA



READK



READP



READR



READRX



READS



READSUB



RENAME



SEC



SEEKR



UNSEC



VERIFY



WRTA



WRTK



WRTP



WRTPV



WRTR



WRTRX



WRTS



ZERO



--



—CTL FNS



AUTOIO



FINDID



INA



IND



INSTAT



LISTEN



LOCAL



MANIO



OUTA



PWRDN



PWRUP



REMOTE



SELECT



STOPIO

TRIGGER

—PRINTER 2E

ACA

ACCHR

ACCOL

ACSPEC

ACX

BLDSPEC

LIST

PRA

'PRAXIS

PRBUF

PRFLAGS

PRKEYS

PRP

'PRPLOT

'PRPLOTP

PRREG

PRREGX

PR

PRSTK

PRX

REGPLOT

SKPCHR

SKPCOL

STKPLOT

FMT

--

—CX EXT FCN

ASROOM

CLRGX

ED

EMDIRX

EMROOM

GETKEYX

RESZFL

ΣREG?

X=NN?

X≠NN?

X<NN?

X<=NN?

X>NN?

X>=NN?

-HP-IL DEV



A-BUF


A=BUF?


A=BUFX?


AAD


AAU


AIPT


BSIZE?


BSIZEX


BUF-AX


BUF-RGX


BUF-XA


BUF-XB


CF33


CMD


DDL


DDT


FRAV?


FRNS?


GET


GTL


IDY


IFCR?


IFC


INBIN


INBUFX


LAD


LPD


MIPT


MONITOR


NRD


NRE


ORAV?


OUTBIN


OUTBINY


OUTBUFX


PT=


PT?


PRBYTES


PRFRMS


REN

RFRM

RG-BUFX

RG=BUF?

RREG

SAI

SCOPE

SDA

SDC

SDI

SF33

SRQR?

SST

TAD

TCT

UNL

UNT

WFRM

WREG

X-BUF

X=BUF?

X<>FLAG

AND

ASIZE?

A-XL

A-XR

A-XX

BININ

BINVIEW

BIT?

HEXIN

HEXVIEW

NOT

OR

OCTIN

OCTVIEW

ROMCHKX

ROTXY

XOR

X-AL

X-AR

Y-AX

-EXT FCN 2D



ALENG



ANUM



APPCHR



APPREC



ARCLREC



AROT



AIOX



CLFL



CLKEYS



CRFLAS



CRFLD



DELCHR



DELREC



EMDIR



FLSIZE



GETAS



GETKEY



GETP



GETR



GETREC



GETRX



GETSUB



GETX



INSCHR



INSREC



PASN



PCLPS



POSA



POSFL



PSIZE



PURFL



RCLFLAG



RCLPT



RCLPTA



REGMOVE



REGSWAP



SAVEAS



SAVEP



SAVER



SAVERX

SAVEX

SEEKPT

SEEKPTA

SIZE?

STOFLAG

X<>F

XTOA

PANAME

-ADV PRT 3C

AID

ID

FINDAID

OUTAX

OUTCR

OUTLF

OUTLFX

OUTSPX

OUTXB

OUTYBX

OUTa

OUTaX

RCLSEL

-82163 FCNS

CLEAR

CLEARO

CSRDN

CSRHX

CSRL

CSROFF

CSRON

CSRR

CSRVX

CSRUP

CTYPE

HOME

SCRLDN

SCRLUP

SCRLX

XYTAB

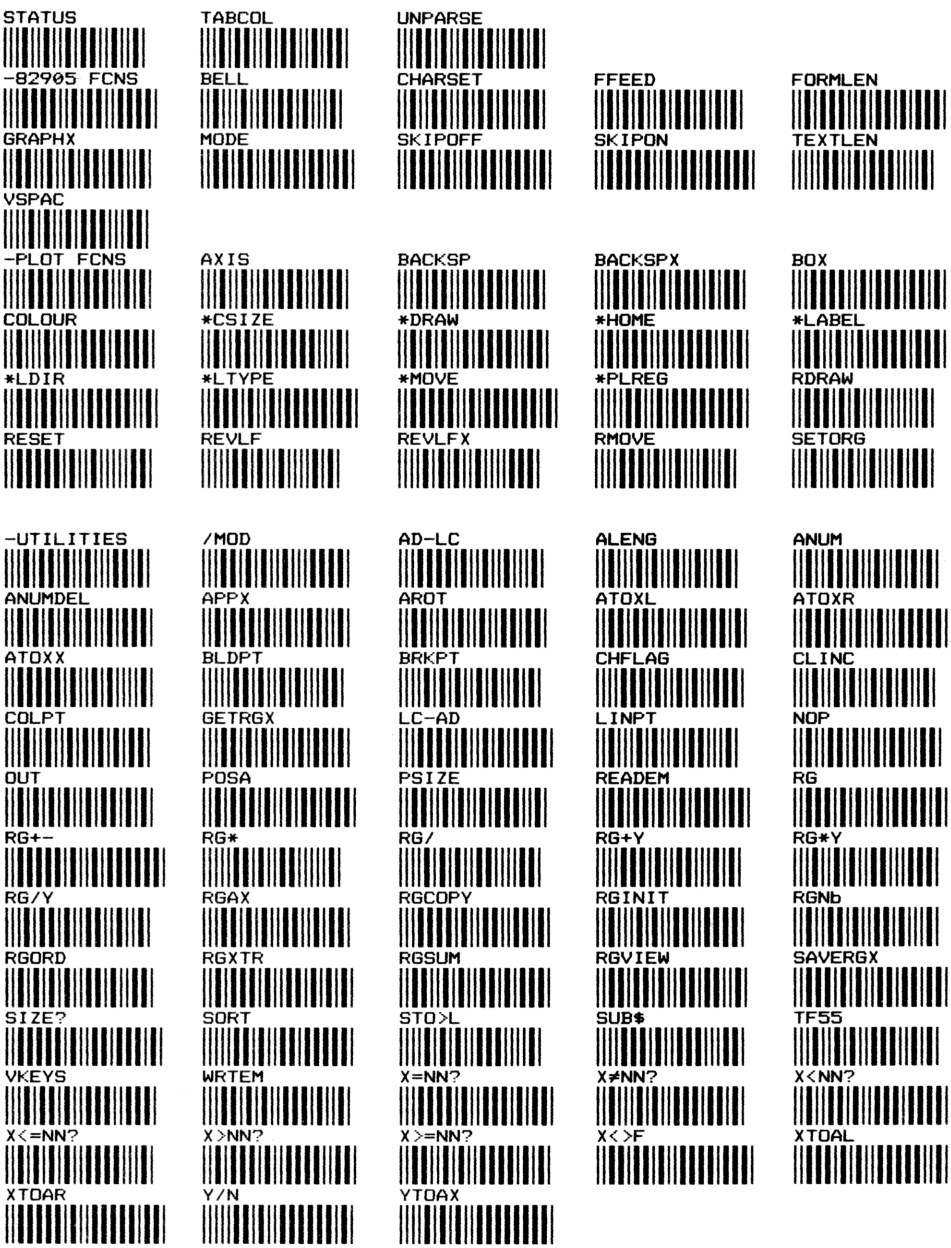
-82162 FCNS

CLBUF

8BIT

ESCAPE

PARSE


STATUS
TABCOL
UNPARSE
-82905 FCNS
BELL
CHARSET
FFEED
FORMLEN
GRAPHX
MODE
SKIPOFF
SKIPON
TEXTLEN
VSPAC
-PLOT FCNS
AXIS
BACKSP
BACKSPX
BOX
COLOUR
*CSIZE
*DRAW
*HOME
*LABEL
*LDIR
*LTYPE
*MOVE
*PLREG
RDRAW
RESET
REVLF
REVLFX
RMOVE
SETORG
-UTILITIES
/MOD
AD-LC
ALENG
ANUM
ANUMDEL
APPX
AROT
ATOXL
ATOXR
ATOXX
BLDPT
BRKPT
CHFLAG
CLINC
COLPT
GETRGX
LC-AD
LINPT
NOP
OUT
POSA
PSIZE
READEM
RG
RG+-
RG*
RG/
RG+Y
RG*Y
RG/Y
RGAX
RGCOPY
RGINIT
RGNb
RGORD
RGXTR
RGSUM
RGVIEW
SAVERGX
SIZE?
SORT
STO>L
SUB$
TF55
VKEYS
WRTEM
X=NN?
X≠NN?
X<NN?
X<=NN?
X>NN?
X>=NN?
X<>F
XTOAL
XTOAR
Y/N
YTOAX

—W&W CCD A



B?



CAS



CLB



RNDM



SAS



SEED



SORT



-ARR FNS



>C+



>R+



?IJ



?1JA



C<>C



C>+



C>-



CMAXAB



CNRM



CSUM



DIM



FNRM



IJ=



IJ=A



M+



M-



M*



M*M



M/



MAX



MAXAB



MDIM



MIN



MOVE



PIV



R-PR



R-QR



R<>R



R>+



R>-



R>R?



RMAXAB

RNRM

RSUM

SUM

SUMAB

SWAP

YC+C

-HEX FNS

1CMP

2CMP

AND

bC?

bS?

Cb

NOT

OR

R<

R>

S<

S>

Sb

UNS

WSIZE

XOR

-I/O FNS

ABSP

ACAXY

ACLX

ARCLE

ARCLH

ARCLI

CLA-

F/E

INPT

PMTA

PMTH

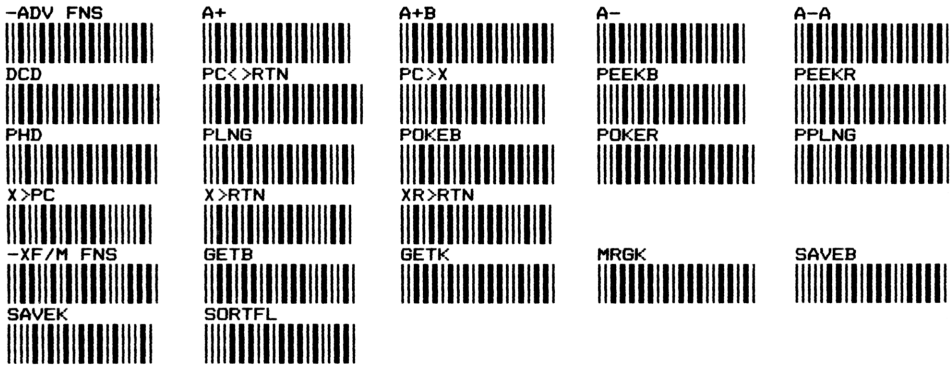
PMTK

PRAXY

PRL

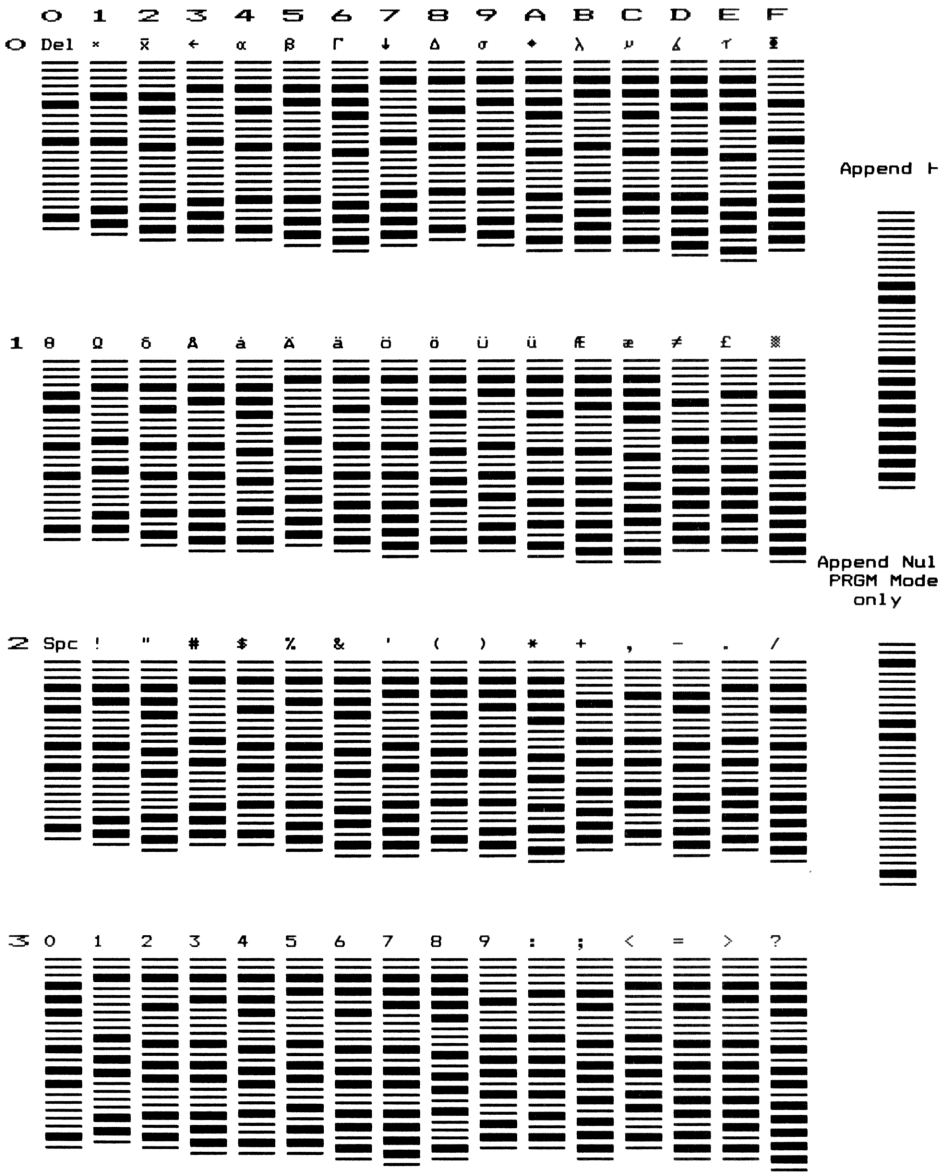
VIEWH

XTOAH

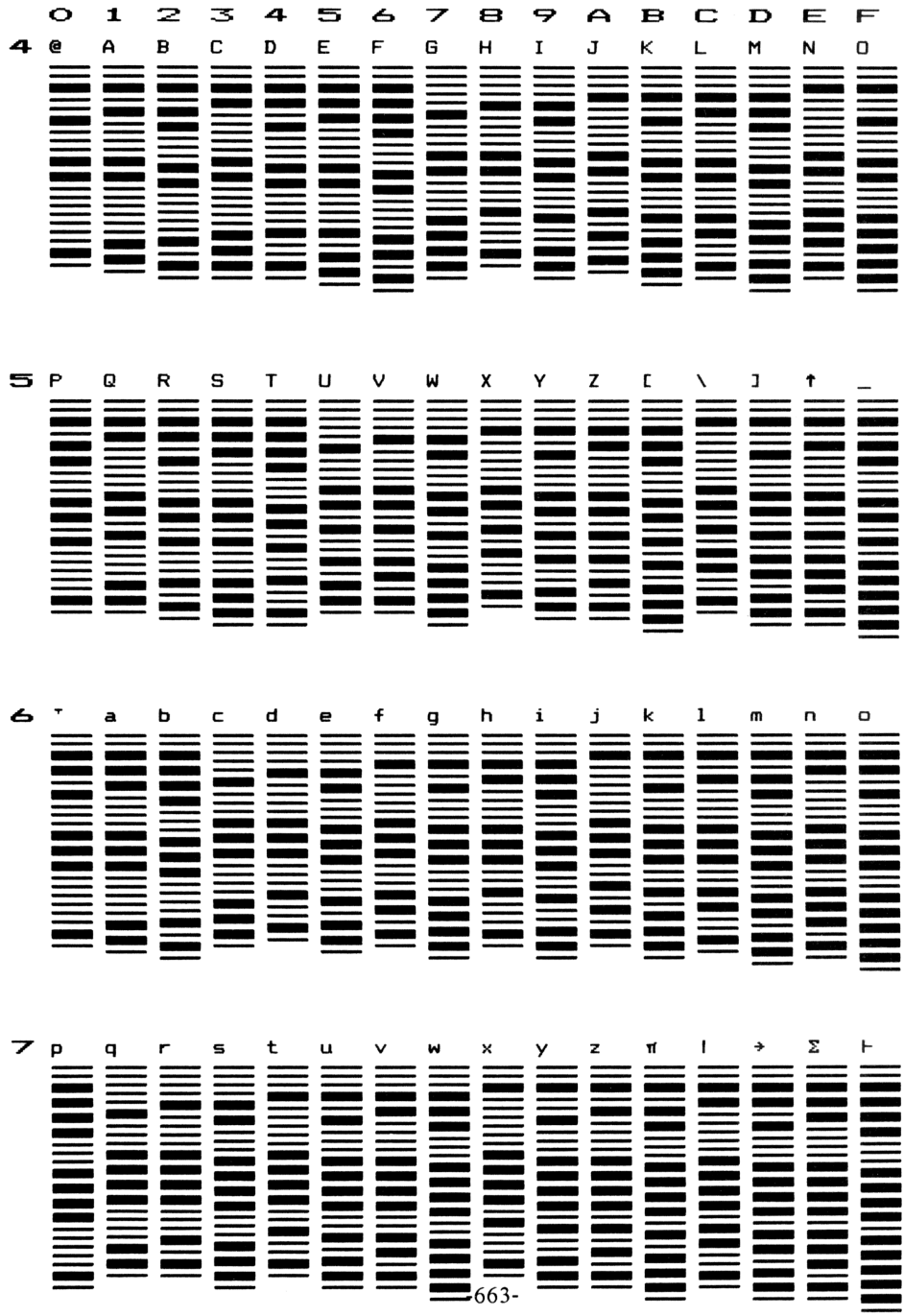

-ADV FNS
A+
A+B
A-
A-A
DCD
PC<>RTN
PC>X
PEEKB
PEEKR
PHD
PLNG
POKEB
POKER
PPLNG
X>PC
X>RTN
XR>RTN
-XF/M FNS
GETB
GETK
MRGK
SAVEB
SAVEK
SORTFL


-ZENROM 3B
CLMM
CLXM
CODE
DECODE
LASTP
MCED
NOP
NRCLM
NRCLX
NSTOM
RAMED
TOGF

Alpha Barcodes 0 to 63



Alpha Barcodes 64 to 127



GASN Row 1 of 17 (1-3)



GASN Row 2 of 17 (4-9)



GASN Row 3 of 17 (9-14)



GASN Row 4 of 17 (15-20)



GASN Row 5 of 17 (21-28)



GASN Row 6 of 17 (29-32)



GASN Row 7 of 17 (32-35)



GASN Row 8 of 17 (35-36)



GASN Row 9 of 17 (36-40)



GASN Row 10 of 17 (41-46)



GASN Row 11 of 17 (46-50)



GASN Row 12 of 17 (50-58)



GASN Row 13 of 17 (59-67)



GASN Row 14 of 17 (67-73)



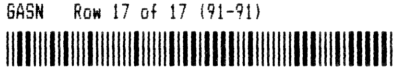
GASN Row 15 of 17 (74-82)



GASN Row 16 of 17 (83-91)



GASN Row 17 of 17 (91-91)



SCF Row 1 of 7 (1-4)



SCF Row 2 of 7 (5-13)



SCF Row 3 of 7 (14-21)



SCF Row 4 of 7 (22-27)



SCF Row 5 of 7 (28-31)



SCF Row 6 of 7 (31-38)



SCF Row 7 of 7 (39-39)



BAB Row 1 of 6 (1-5)



BAB Row 2 of 6 (5-12)



BAB Row 3 of 6 (13-23)



BAB Row 4 of 6 (24-30)



BAB Row 5 of 6 (31-37)



BAB Row 6 of 6 (38-38)



KAP Row 1 of 12 (1-5)



KAP Row 2 of 12 (6-12)



KAP Row 3 of 12 (12-16)



KAP Row 4 of 12 (16-25)



KAP Row 5 of 12 (26-35)



KAP Row 6 of 12 (36-42)



KAP Row 7 of 12 (42-48)



KAP Row 8 of 12 (48-51)



KAP Row 9 of 12 (51-55)



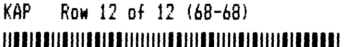
KAP Row 10 of 12 (56-61)



KAP Row 11 of 12 (61-67)



KAP Row 12 of 12 (68-68)



ΣCX Row 1 of 5 (1-4)



ΣCX Row 2 of 5 (5-11)



ΣCX Row 3 of 5 (11-15)



ΣCX Row 4 of 5 (16-21)



ΣCX Row 5 of 5 (21-25)



RAB Row 1 of 7 (1-5)



RAB Row 2 of 7 (5-11)



RAB Row 3 of 7 (11-17)



RAB Row 4 of 7 (18-25)



RAB Row 5 of 7 (25-33)



RAB Row 6 of 7 (33-39)



RAB Row 7 of 7 (40-40)



XRO Row 1 of 4 (1-4)



XRO Row 2 of 4 (5-10)



XRO Row 3 of 4 (11-15)



XRO Row 4 of 4 (15-17)



APX Row 1 of 4 (1-3)



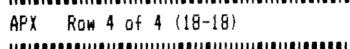
APX Row 2 of 4 (3-9)



APX Row 3 of 4 (9-17)



APX Row 4 of 4 (18-18)



CLB&PPK Row 1 of 15 (1-2)



CLB&PPK Row 2 of 15 (2-7)



CLB&PPK Row 3 of 15 (7-14)



CLB&PPK Row 4 of 15 (15-22)



CLB&PPK Row 5 of 15 (22-26)



CLB&PPK Row 6 of 15 (27-34)



CLB&PPK Row 7 of 15 (34-38)



CLB&PPK Row 8 of 15 (38-43)



CLB&PPK Row 9 of 15 (44-53)



CLB&PPK Row 10 of 15 (54-56)



CLB&PPK Row 11 of 15 (56-62)



CLB&PPK Row 12 of 15 (62-69)



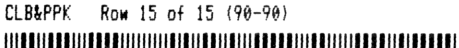
CLB&PPK Row 13 of 15 (70-79)



CLB&PPK Row 14 of 15 (79-89)



CLB&PPK Row 15 of 15 (90-90)



PPRP Row 1 of 35 (1-4)



PPRP Row 2 of 35 (4-10)



PPRP Row 3 of 35 (11-18)



PPRP Row 4 of 35 (18-26)



PPRP Row 5 of 35 (27-34)



PPRP Row 6 of 35 (35-40)



PPRP Row 7 of 35 (40-45)



PPRP Row 8 of 35 (46-53)



PPRP Row 9 of 35 (53-60)



PPRP Row 10 of 35 (60-68)



PPRP Row 11 of 35 (69-76)



PPRP Row 12 of 35 (76-85)



PPRP Row 13 of 35 (85-94)



PPRP Row 14 of 35 (95-101)



PPRP Row 15 of 35 (102-108)



PPRP Row 16 of 35 (109-115)



PPRP Row 17 of 35 (115-120)



PPRP Row 18 of 35 (120-123)



PPRP Row 19 of 35 (124-132)



PPRP Row 20 of 35 (133-139)



PPRP Row 21 of 35 (139-147)



PPRP Row 22 of 35 (148-154)



PPRP Row 23 of 35 (154-160)



PPRP Row 24 of 35 (160-162)



PPRP Row 25 of 35 (162-170)



PPRP Row 26 of 35 (170-171)



PPRP Row 27 of 35 (171-179)



PPRP Row 28 of 35 (180-183)



PPRP Row 29 of 35 (183-191)



PPRP Row 30 of 35 (192-198)



PPRP Row 31 of 35 (199-207)



PPRP Row 32 of 35 (208-215)



PPRP Row 33 of 35 (216-219)



PPRP Row 34 of 35 (219-226)



PPRP Row 35 of 35 (227-227)



ED Row 1 of 23 (1-4)



ED Row 2 of 23 (5-6)



ED Row 3 of 23 (7-15)



ED Row 4 of 23 (16-26)



ED Row 5 of 23 (27-33)



ED Row 6 of 23 (34-41)



ED Row 7 of 23 (42-48)



ED Row 8 of 23 (49-60)



ED Row 9 of 23 (60-69)



ED Row 10 of 23 (69-76)



ED Row 11 of 23 (77-85)



ED Row 12 of 23 (86-92)



ED Row 13 of 23 (93-102)



ED Row 14 of 23 (103-110)



ED Row 15 of 23 (110-117)



ED Row 16 of 23 (118-126)



ED Row 17 of 23 (127-133)



ED Row 18 of 23 (134-143)



ED Row 19 of 23 (144-152)



ED Row 20 of 23 (152-162)



ED Row 21 of 23 (163-168)



ED Row 22 of 23 (168-174)



ED Row 23 of 23 (175-180)



A
absolute addressing, 188
AC adaptor socket, 15
accumulators, 198
accuracy factor, 230
ADOW, 248
ALPHA execution, 41
ALPHA register, 79
alphanumeric, 79
Annunciators, 30
argument, 36
assignments
  cancelling, 336
ATAN2, 172
Australian notation, 520

B
backarrow, 54
backarrow key, 18
bank, 195
Bank-switching, 195
barcode, 23
Batteries, 17
battery compartment, 15
Battery pack, 18
BCD, 183
bender, 199
BG, 427
bifid, 37
binary, 178
Binary Coded Decimal, 183
bit, 178
blocks, 191
Boolean functions, 384
bounce, 48
boxed star, 30
Buffer area, 187
Buffers, 187
Bugs, 59
Byte Loaders, 517
bytes, 179

C
Card Reader, 22
catalogues, 43
Central Processor Unit, 196
characters, 20
Charger, 18
checksum byte, 293
chip, 191
Coconut, 653
cold start constant, 462
compatibility functions, 339
compilation, 121
conditional alarm, 240
Continuous Memory, 19
control functions, 37
control alarm, 238
Conversion functions, 356
Copying Programs, 131
correction key, 54
countdown alarm, 237
CPU, 196
crash, 51
current address, 474
current file, 290

current line, 113
current pointer, 294
current program, 113
current record, 292
curtain, 188

D
data, 20, 181
data packing, 227
data processing, 181
decimal, 178
decompilation, 121
dedicated, 23
deep sleep mode, 35
default, 19
delete key, 54
Devel, 536
digit, 20
disable stack lift, 71
Display, 15, 30
display annunciators, 30
Documentation, 150
dummy, 428
dyadic, 37

E
EBCDIC, 183
EFM, 249
EM, 289
End-of-Memory marker, 289
environment, 150
Execute, 35
execution indicator, 50
exponent, 20
extended storage, 110

F
Fading crash, 638
files, 181, 289
flag, 86
flag 22, 256
flag 53
  special problems, 455
flowchart, 396
flying goose, 50
FOCAL, 13
function preview, 34
functions, 39

G
global labels., 113
goose, 50, 143

H
Halfnut, 653
hang-ups, 51
hangs up, 51
hexadecimal, 178
HHC, 14
HP-41 system, 227
HP-67, 339
HP-97, 339
HP-IL, 20
Hyperbolic functions, 158

I
I/O ports, 15
idle, 35
index sorting, 284
Indirect, 109
Indirect Addressing, 109

Input/Output ports, 15
instruction, 39
interrupting control alarm, 238

K
K, 191
KAR, 215
Key, 15
Key Assignment Register, 215
keyboard, 15
Keyboard efficiency, 9
keyboard mode, 34

L
Łukasiewicz, 36
LCD, 30
leading zeroes, 206
light sleep mode, 35
line, 40
listing, 24
local label search, 126
logical operations, 178, 384
long labels, 125
long-form, 204

M
M-code, 196
M1, 461
M2, 461
mantissa, 20
Mass storage, 22, 26
membrane Touchpad, 16
message alarm, 238
mode, 34
modules, 15
monadic, 36
Monte Carlo methods, 167, 174

N
natural notation, 520
nested subroutine calls, 475
NNN, 184, 413
NOMAS, 414
non-interrupting alarm, 240
non-text GTO, 524
Non Normalised Numbers, 182
NOP, 79
Normal keyboard, 34
normalisation, 184
null, 250
number entry, 75
nybble, 180

O
octal, 178
ON procedure, 66
operating system, 414
operating mode, 35
Optical Wand, 23
overlay, 16

P
Packing, 130
Page-switching, 195
pages, 191
parameter, 37
parameter keyboard, 38
partition register, 289
past due alarm, 238
pending, 118

peripheral devices, 15
plug-ins, 15
pointer, 188, 292
Port Extender, 27
ports, 15
postfix, 202
prefix, 202
primary storage, 110
PRIVATE, 338
Problems, 45
program, 40
program modules, 406
program step, 40
program structure, 153
prompt, 37
pseudo-XROM number, 422

Q
quad, 191

R
radix mark, 182
RAM, 20
Random Number Generator, 167
recall arithmetic, 78
Rechargeable battery pack, 18
recursion, 122
register, 180
register L, 91
registers, 20
Resizing, 130
RNG, 167
rocker switches, 15
ROM, 19
ROMs, 22
routine, 40
RPN, 36
RUN, 35

S
scratch, 440
scrolling, 80
seed, 168
SHIFT, 15
SHIFT, ON, 230
short-form, 122, 203
short-form exponent, 425
SP, 413
spare bytes, 203
split timings, 233
splits, 233
stack, 69
stack analysis form, 71
stack drop, 71
stack lift, 71
standby, 35
statements, 39
Status Registers, 187
step, 40
storage arithmetic, 78
structured programming, 153
subroutine, 39
subroutine return stack, 118

T
terminating numeric entry, 206
tests, 134
text, 25, 79
Text Q-loaders, 517
text string, 20
text enabling, 523

timer alarm, 237
toggle keys, 15
two-key rollover, 57

U
Unit Management, 356
up-arrow, 6
"user language, 14

V
value, 20
View mantissa, 266

W
Wand, 23
warm start constant, 462
word, 180, 191
working file, 290

X
X-register, 36
XIO, 536

Y
Y-register, 37

ADDITIONS - MAY 1988

New Products

The Math and Stat modules are now sold as one combined module, but with the same XROM numbers as before. A new infra-red printer module from HP lets the HP-41 use the (cheaper) new cordless printer introduced for the HP-18C and HP-28C. New modules for the HP-41 continue to be announced by third parties. They include surveying, finance, forestry, navigation, astronomy, racetrack and even a horoscope module. Most are available from Educalc (see Appendix B). Five modules of special interest are: 1. The AEC module which provides new math functions in machine-code, functions to display in metric units or yards, feet and inches, and a routine which lets you type in an equation in algebraic notation, then turns that into a program which you can run with named variables. (Also available from SYNTHETIX.) 2. The Extended IL module which lets an HP-41 exploit the full capabilities of the HP9114 disk drive and the HP2225 ThinkJet printer (for example it allows multi-column listings to be printed on an 80 character wide page). 3. The SKWIDBC module which lets an HP-41 print barcode directly on HP LaserJet or ThinkJet printers. (Available from SYNTHETIX, see order blank. Also available combined with Extended IL module.) 4. The HEPAX module which provides up to 16K memory and functions to use this memory for data storage and machine language programming on the HP-41 with no external accessories. (Referred to in section 17.4). 5. The ZEPROM module which can be used repeatedly to store and run programs that will not be lost by a MEMORY LOST nor when the module is removed from the HP-41. The module can be programmed directly from an HP-41 using an EPROM programmer which is about the same size as a module and requires no external power sources.

The new HP-17, HP-18, HP-19, HP-27 and HP-28 calculators have many new features, but are not replacements for the HP-41. In particular, they provide no means of storing programs or information outside the calculator. They can be used with a printer, but there is no way to put printed data or programs back into the calculator other than keying them again. There is no provision for adding programs or memory on plug-in modules. (The HP-28C

and HP-28S provide an instruction which provides access to a form of synthetic and machine language programming - this is described in a new book "Customize Your HP-28", soon to be available from SYNTHETIX.)

If you want to run HP-41 programs on a larger computer you can use an HP-71B with the HP-41 Translator Pac module. Some software companies provide HP-41 simulation programs that run under MS-DOS. A complete HP-41CX simulator is provided by Corvallis Micro Technology for their hand-held MC-II computer. HP-41 programs can be typed directly into the MC-II or they can be transferred to it via an MS-DOS computer. The MC-II can use RS-232 and HP-IL for interfacing with other devices, it can be extended by means of plug-in modules, and it runs typical HP-41 programs about 10 times faster than an HP-41.

Bugs

-Version B of the CCD module, with the bugs corrected, is now available.
-If you print the Time Module Alarm Catalog, strange things can happen if port 3 contains any plug-in except a printer, HP-IL or the Timer module.
-If the Plotter module function LIMIT gives an unexpeceted PL:RANGE ERR message then SELECT the plotter as the primary HP-IL device. If BCREGX gives NONEXISTENT this can be because two or more consecutive registers are empty: just repeat BCREGX once for each empty register. On an HP-41C without a quad memory module BCREGX should not be used with nonexistent registers, otherwise the Alpha register may get corrupted.
-XROM bug: If you execute an XROM AB,CD instruction and a module with ID number AB is not plugged in then you will get NONEXISTENT. If a module with ID number AB but fewer than CD functions is plugged in, then the HP-41 does not give NONEXISTENT at once. Instead it carries on looking, but for XROM CD,CD. Usually it will not find this and will give the message NONEXISTENT as expected. The bug affects key assignments and keyboard execution as well as programs. A detailed description of the bug is in PPC V14N2p26-29.

NOTES

NOTES

NOTES

p259 line -4 should mention that the Paname module has these functions too

p278 line -9 should mention that the Paname module has these functions too

p358 lines 6 & 7 should have hours: minutes (or degrees): seconds

p396 At the end of paragraph 1 add "The new Advantage ROM provides the HP-41 with many of these features.

p625 add at the end:

"A less serious bug is the following: execute 8E99,ENTER,R-P. On old HP-41s the result in X will be 1.13, not 1.13E100; this is similar to bug 4 which also ignored the leading digit of a three-digit exponent. On later HP-41s the OUT OF RANGE error is shown as it should be, unless flag 25 is set, in which case the operation is ignored but various flags are set (which flags become set depends on the previous contents of register d). This is similar to bug 10. Reference:DATAFILE V4N5/6p35."

INDEX Some pages have been renumbered during printing so the Index usually refers to a page number that is one too high. If the page number seems to be wrong, try looking back a page or two.

ORDER BLANK

	Price per copy	Qty	Amount
For HP-71'S **HP-71 Basic Made Easy**, by Joseph Horn	$18.95	____	_____
For HP-71'S & HP-41'S **Control the World with HP-IL**, by Gary Friedman	$24.95	____	_____
For HP-41'S **HP-41 Advanced Programming Tips**, by A. McCornack & K. Jarett	$20.95	____	_____
HP-41 M-Code for Beginners, by Ken Emery	$24.95	____	_____
Inside the HP-41, by Jean-Daniel Dodin	$12.95	____	_____
Extend Your HP-41, by W. Mier-Jędrzejowicz	$29.95	____	_____
HP-41 Extended Functions Made Easy, by Keith Jarett	$16.95	____	_____
HP-41 Synthetic Programming Made Easy, by Keith Jarett (Includes one Quick Reference Card)	$16.95	____	_____
Quick Reference Card for Synthetic Programming	$2.00	____	_____
Synthetic Quick Reference Guide (SQRG)	$5.95	____	_____
For HP-10C, 11C, 15C, AND 16C **ENTER (Reverse Polish Notation Made Easy)**, by J.Dodin	$5.95	____	_____
Humor **It's Amazing How These Things Can Simplify Your Life: The Harold Guide to Computer Literacy**	$4.95	____	_____
ROM's **Barcode Generating ROM** by Ken Emery	$199.95	____	_____
AECROM by Redshift Software	$ 99.00	____	_____
Sales tax (California orders only, 6 or 7%)			_____

Shipping	1st book	Add'l books
within USA, book rate (4th class)	$1.50	$0.50
USA 48 states, United Parcel Service	$2.50	$1.00
USA, Canada, air mail	$3.00	$1.50
elsewhere, book rate (6 to 8 week wait)	$2.00	$1.00

elsewhere, air mail $12.05 for **Extend Your HP-41**, $6.05 for others

Free shipping for **ENTER** and **It's Amazing...** with purchase of any other book

Free shipping for QRC plastic cards or SQRG (any number)

Free shipping for ROM's

Enter shipping total here $_____

Total due $_____

Checks must be in U.S. funds, and payable through a U.S. bank.

Name_____

Address _____

_____

City_____ State_____ Zipcode_____

Country_____

Mail to:

SYNTHETIX, P.O.Box 1080, Berkeley, CA 94701-1080, USA Phone (415) 339-0601

ORDER BLANK

	Price per copy	Qty	Amount
For HP-71'S **HP-71 Basic Made Easy**, by Joseph Horn	$18.95	____	_____
For HP-71'S & HP-41'S **Control the World with HP-IL**, by Gary Friedman	$24.95	____	_____
For HP-41'S **HP-41 Advanced Programming Tips**, by A. McCornack & K. Jarett	$20.95	____	_____
HP-41 M-Code for Beginners, by Ken Emery	$24.95	____	_____
Inside the HP-41, by Jean-Daniel Dodin	$12.95	____	_____
Extend Your HP-41, by W. Mier-Jędrzejowicz	$29.95	____	_____
HP-41 Extended Functions Made Easy, by Keith Jarett	$16.95	____	_____
HP-41 Synthetic Programming Made Easy, by Keith Jarett (Includes one Quick Reference Card)	$16.95	____	_____
Quick Reference Card for Synthetic Programming	$2.00	____	_____
Synthetic Quick Reference Guide (SQRG)	$5.95	____	_____
For HP-10C, 11C, 15C, AND 16C **ENTER (Reverse Polish Notation Made Easy)**, by J.Dodin	$5.95	____	_____
Humor **It's Amazing How These Things Can Simplify Your Life:** **The Harold Guide to Computer Literacy**	$4.95	____	_____
ROM's **Barcode Generating ROM** by Ken Emery	$199.95	____	_____
AECROM by Redshift Software	$ 99.00	____	_____
Sales tax (California orders only, 6 or 7%)			_____

Shipping	1st book	Add'l books
within USA, book rate (4th class)	$1.50	$0.50
USA 48 states, United Parcel Service	$2.50	$1.00
USA, Canada, air mail	$3.00	$1.50
elsewhere, book rate (6 to 8 week wait)	$2.00	$1.00

elsewhere, air mail $12.05 for **Extend Your HP-41**, $6.05 for others

Free shipping for **ENTER** and **It's Amazing...** with purchase of any other book
Free shipping for QRC plastic cards or SQRG (any number)
Free shipping for ROM's

Enter shipping total here $_____

Total due $_____

Checks must be in U.S. funds, and payable through a U.S. bank.

Name_____
Address _____
_____
City_____ State_____ Zipcode_____
Country_____

Mail to:
SYNTHETIX, P.O.Box 1080, Berkeley, CA 94701-1080, USA Phone (415) 339-0601

PROGRAMS IN THIS BOOK

		Page
LIGHT	: example of a program listing	24
CALC1 and 2	: four examples of calculations	40
CALC3 and 4	: with functions and routines	41
WATCH	: program displays each step before executing it	134
MATRIX?	: prompts for input matrix	145
YN	: asks a Yes/No question	147
F47	: asks a question and uses status of SHIFT flag for reply	148
Hyperbolics	: efficient and accurate hyperbolics and their inverses	158
REGS	: displays values in non-zero registers and shows Size	159
TX	: transforms a function to integrate with infinite limits	163
ATAN2	: complete arc tangent	172
ADDWRD	: inserts a word in a list by indirect sorting	285
PRFL	: prints and displays an Extended Memory text file	308
GASP	: makes synthetic key assignments, does not use S.P.	318
TRYd	: example of saving and replacing flag status register	447
C55	: example of using flags - clears printer flag	449
SCF	: sets or clears any flag specified in register X	452
BAB	: builds a byte value and appends it to Alpha	455
Σ CX	: exchanges the summation register and curtain pointers	464
KAP	: synthetic key assignment program	468
VREGS	: views registers, loops by using RCL b and STO b	475
INTEG	: example of prog. calling more than 7 subroutine levels	476
SL2, SR2	: shifts a reg. 2 bytes left or right,does not use Alpha	478
RAB	: reads a byte from Alpha, allows for 15 chars and nulls	496
XRO	: XEQ from ROM; allows 2 modules with same XROM no.	513
GSE	: displays a left goose instead of - sign in exponents	521
APX	: allows appending to X as APPEND does to Alpha	542
CXM	: clears Extended Memory on any HP-41	556
CLB, PPK	: progs to delete a buffer and do a programmable PACK	566
PPRP	: programmable PRP for use in documentation	571
NNR	: non normalised recall using HP-IL Devel module	581
ED	: editor for RAM - requires PPC ROM	583
PPR	: shorter programmable PRP	587

Short example routines have not been included in this list.

PROGRAMS IN THIS BOOK

		Page
LIGHT	: example of a program listing	24
CALC1 and 2	: four examples of calculations	40
CALC3 and 4	: with functions and routines	41
WATCH	: program displays each step before executing it	134
MATRIX?	: prompts for input matrix	145
YN	: asks a Yes/No question	147
F47	: asks a question and uses status of SHIFT flag for reply	148
Hyperbolics	: efficient and accurate hyperbolics and their inverses	158
REGS	: displays values in non-zero registers and shows Size	159
TX	: transforms a function to integrate with infinite limits	163
ATAN2	: complete arc tangent	172
ADDWRD	: inserts a word in a list by indirect sorting	285
PRFL	: prints and displays an Extended Memory text file	308
GASP	: makes synthetic key assignments, does not use S.P.	318
TRYd	: example of saving and replacing flag status register	447
C55	: example of using flags - clears printer flag	449
SCF	: sets or clears any flag specified in register X	452
BAB	: builds a byte value and appends it to Alpha	455
Σ CX	: exchanges the summation register and curtain pointers	464
KAP	: synthetic key assignment program	468
VREGS	: views registers, loops by using RCL b and STO b	475
INTEG	: example of prog. calling more than 7 subroutine levels	476
SL2, SR2	: shifts a reg. 2 bytes left or right,does not use Alpha	478
RAB	: reads a byte from Alpha, allows for 15 chars and nulls	496
XRO	: XEQ from ROM; allows 2 modules with same XROM no.	513
GSE	: displays a left goose instead of - sign in exponents	521
APX	: allows appending to X as APPEND does to Alpha	542
CXM	: clears Extended Memory on any HP-41	556
CLB, PPK	: progs to delete a buffer and do a programmable PACK	566
PPRP	: programmable PRP for use in documentation	571
NNR	: non normalised recall using HP-IL Devel module	581
ED	: editor for RAM - requires PPC ROM	583
PPR	: shorter programmable PRP	587

Short example routines have not been included in this list.

GET THE MOST OUT OF YOUR HP-41 !

"Extend Your HP-41" is a complete guide to enhancing the performance of your HP-41 system through more efficient keyboard and programming techniques. Designed to supplement rather than replace HP's manuals, this book leads you from the basics of keyboard operation and normal programming, to advanced techniques like synthetic programming. Along the way, a variety of clever tips and tricks are discussed, many of which will be new even to experts.

Containing over 650 pages of information, this is the ultimate reference book for your HP-41 system. Concise and clearly written, it will provide you with many hours of fascinating reading. A few of the many topics covered are hyperbolic functions, efficient use of flags and looping instructions, integration to infinite limits, and random number generation. Also covered are extra hints on using the Time module, Extended Functions, and Extended Memory effectively. The full range of HP peripherals and modules is described, including the exciting new Advantage module. Finally, the last 200 pages provide a sampling of what Synthetic Programming has to offer and how it works. Experts will marvel at the first key assignment program that uses only normal HP-41 functions!

Two appendices list books and third party products for the HP-41. Another appendix lists the "bugs". This will help you avoid losing your valuable programs or Extended Memory files. Barcode is provided for the programs.

"Extend Your HP-41" is by far the largest and most complete collection of useful facts on the HP-41 system. Beginner or expert, no HP-41 owner should be without it.

ISBN 0 9510733 0 3