DATA PROCESSING ON THE HP-41 C/CV

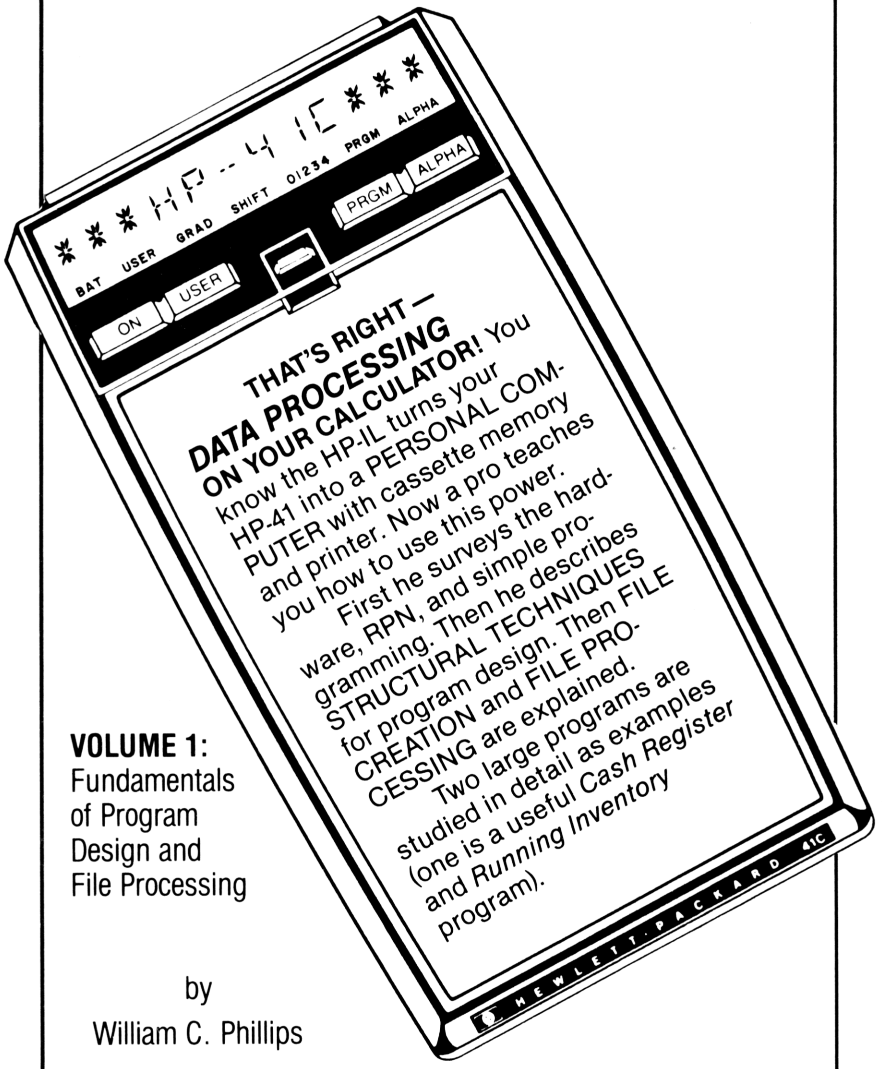
VOLUME 1:
Fundamentals of Program Design and File Processing

by

William C. Phillips

The EduCALC Technical Series

DATA PROCESSING ON THE HP-41 C/CV



THAT'S RIGHT — DATA PROCESSING ON YOUR CALCULATOR! You know the HP-IL turns your HP-41 into a PERSONAL COMPUTER with cassette memory and printer. Now a pro teaches you how to use this power.

First he surveys the hardware, RPN, and simple programming. Then he describes STRUCTURAL TECHNIQUES for program design. Then FILE CREATION and FILE PROCESSING are explained.

Two large programs are studied in detail as examples (one is a useful *Cash Register* and *Running Inventory* program).

VOLUME 1:
Fundamentals of Program Design and File Processing

by
William C. Phillips

The EduCALC Technical Series

EduCALC

EduCALC Publications Box 974, Laguna Beach, California 92652

DATA PROCESSING ON THE HP-41 C/CV

by William C. Phillips

ISBN 0-936356-02-2
Library of Congress Catalogue Card Number 83-81097

NOTE for the EduCALC Technical Series:

This volume was printed directly from printout prepared by the author. The publisher has not performed his usual functions of reviewing, editing, typesetting, and proof-reading the material prior to publication. The publisher fully endorses this rapid and informal method of bringing you technical notes at a moderate price, and he wishes to thank the author for his preparation of this material for publication!

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

Contents

Preface v

Chapter 1: The HP-41C/CV Personal Computer 1

Introduction 1

The HP-41C/CV System Configuration 1

The HP-41C/CV Calculator 2

Internal System Architecture 7

The HP-41C/CV Addressing Scheme 11

HP 82161A Digital Cassette Drive 12

HP 82162A Thermal Printer 14

HP 82162A HP-IL Module 16

HP 82180A Extended Functions/Memory Module 18

HP 82181A Extended Memory Module 19

HP 82182A Time Module 20

Port-X-Tender 20

The Manuals 21

Notes and References 21

Chapter 2: RPN and the Stack 22

Introduction 22

History of RPN 22

Algebraic vs RPN Notation 23

Translating Algebraic Expressions into RPN 26

The Stack 27

Stack Operations 29

RPN and the Stack 29

Conclusion 33

References 33

Chapter 3: Simple Programming Summary 34

Introduction 34

Definition of a Program 34

Structure of a Program 35

Keying a Program into Program Memory 37

Program Verification 41

Editing Programs 41

Running Programs 43

Chapter 3: Simple Programming Summary(continued)

Compiling Programs 44

Conclusion 44

Chapter 4: Program Design Methodology **45**

Introduction 45

Historical Perspective 45

Structured Design 47

Structured Programming 48

Conclusion 52

References 52

Chapter 5: Program Control **54**

Introduction 54

Unconditional Transfers 54

Conditional Transfers 56

User and System Flags 61

The IF-THEN-ELSE and CASE Control Structure 62

Loops 67

The DO-UNTIL and DO-WHILE Control Structures 69

Conclusion 70

Chapter 6: Subroutines **72**

Introduction 72

Calling Sequences 72

Types of Subroutines 73

How Subroutines Work 74

Constructing Subroutines 75

Conclusion 77

Notes 77

Chapter 7: Simple Ordinary Annuities Program Example **78**

Introduction 78

Simple Annuities 78

FV 78

PV 81

PV/FV Equivalency 82

PMT 82

n and Final Payment 82

i_c 83

Chapter 7: Simple Ordinary Annuities Program Example(continued)

Annuities Program Design	85
Techniques Used	86
ANNUITY Routine	89
Internal Subroutine N	89
Internal Subroutine INT	89
Internal Subroutine PV	89
Internal Subroutine PMT	90
Internal Subroutine FV	90
CALC-N Subroutine	90
CALC-I Subroutine	92
CALC-PV Subroutine	94
CALC-FV Subroutine	94
CALC-PY Subroutine	95
SINK Subroutine	95
PRS-VAL Subroutine	96
FIN-PMT Subroutine	96
Conclusion	96
Chapter 8: Internal Files	**98**
Introduction	98
Notation	98
Extended Memory	98
File Headers	99
File Pointers	99
Working Files	107
File Creation and Management	108
Recording Data into Internal Data Files	109
Retrieving Data from Internal Data Files	111
Recording Data into Internal ASCII Files	112
Retrieving Data from Internal ASCII Files	114
Conclusion	115
Chapter 9: I/O Operations	**116**
Introduction	116
Notation	116
Initialization Procedures	117
File Pointers	117

Chapter 9: I/O Operations (continued)

File Creation and Management	118
Recording Data into External Data Files	120
Checking Files	121
Protecting Files	121
Retrieving Data from External Data Files	121
External ASCII Files	122
Transferring Internal ASCII Files to External ASCII Files	123
Transferring External ASCII Files to Internal ASCII Files	123
Conclusion	124
Chapter 10: Sales Program Example	**125**
Introduction	125
SALES Program Background	125
Program Files	128
SALES Program Design	130
SALES Routine	131
GETSAL Subroutine	133
GETPRIC Subroutine	134
TOTAL Subroutine	136
SUBTOTL Subroutine	136
TAX Subroutine	138
RECEIPT Subroutine	139
RECSAL Subroutine	139
Conclusion	142
Index	**143**

Preface

Since its introduction in 1974, the handheld, programmable calculator has opened up a whole new area of interest and study for individuals who probably would not have gotten involved in personal computing. Later, when the first personal computers became available, another area of interest opened up for a different set of individuals. The path taken by the programmable calculator owner was a divergent one to that taken by the personal computer owner. In the early days of these machines, however, this is as it should have been. The personal computer was a quantum leap above the programmable calculator in both speed and capabilities. The personal computer owner was interested in interfacing his machine to the outside world, in playing games and in implementing all kinds of programs. The programmable calculator owner, on the other hand, was interested in reducing the number of keystrokes necessary to solve a problem. In 1979, the HP-41C/CV was introduced and was destined to cause the two paths to begin to converge. The owners of these new machines began to study the same kinds of things about programming as the owners of the personal computers had been studying for years.

In 1981, Hewlett-Packard introduced the HP-IL for the HP-41C/CV. This interface was intended to allow the HP-41C/CV to "talk" to any compatable device attached to the loop whether it be a printer a tape drive or another computer. The HP-IL was the final step in removing the barrier between personal computer and programmable calculator.

This is the first volume in a series concerned with data processing on the HP-41C/CV personal computer. The intent is to teach programming skills and techniques which the programmers of the larger machines have been using for years. This in turn should lead into an understanding of how to use the machine as a computer rather than just another fancy calculator. It is assumed that you are familiar with the basic commands for the HP-41C/CV and how to program it.

This volume serves two purposes. First, it takes the reader through a somewhat high level summary of the hardware and of programming the "basic" machine. Secondly, this volume introduces structured techniques and file creation and processing.

The first three chapters are summary material. This material will serve to place readers on a common plane of understanding. The next four chapters introduce and discuss structured techniques. This material is strictly intended to whet the appetite of the reader. The second volume of this series will be devoted entirely to this and other program design considerations. The final three chapters introduce and explain file creation and file processing. This material will be covered in greater detail in volume three of this series.

It is my hope that after reading this first volume, you will begin to understand how your HP-41C/CV can be used for its intended purposes and that you will want to further your understanding of the HP-41C/CV by learning as much about your machine as possible.

I would like to solicit your help in making this volume as useful as possible. To that end, please send any comments or suggestions to the publisher.

Chapter 1

The HP-41C/CV Personal Computer

INTRODUCTION

The HP-41C/CV represents a totally new concept in programmable calculators. By itself, the HP-41C/CV is a powerful programmable scientific calculator with more than 130 built-in functions, a continuous memory which may be partitioned between program and data storage and an alphanumeric display. Various application modules which contain pre-written programs may be inserted into the I/O ports to further increase the capabilities of the basic machine.

The newness in concept of this machine, and the reason it is termed a personal computer, arises from the fact that the HP-41C/CV may be attached to and drive up to 31 devices such as printers, tape drives and test equipment. The basic HP-41C/CV now becomes a true computer capable of storing and retrieving vast amounts of data and capable of executing large, complex programs.

This chapter will discuss the HP-41C/CV personal computer in detail. The topics to be discussed are:

HP-41C/CV system configuration
HP-41C/CV calculator
Internal system architecture
HP-41C/CV addressing scheme
HP 82161A Digital Cassette Drive
HP 82162A Thermal Printer
HP 82160A HP-IL Module
HP 82180A Extended Functions/Memory Module
HP 82181A Extended Memory Module
HP 82182A Time Module
AME Port-X-Tender

THE HP-41C/CV SYSTEM CONFIGURATION

This volume, and the volumes that follow, treat the HP-41C/CV as more than a mere calculator. The HP-41C/CV is treated as a personal computer, and as such, a certain minimum system configuration is assumed. This configuration is listed next.

HP-41C calculator with full memory or an HP-41CV calculator
HP 82180A Extended Functions/Memory Module
two HP 82181A Extended Memory Modules
HP 82161A Digital Cassette Drive
HP 82160A HP-IL Module
HP 82162A Thermal Printer
HP 82182A Time Module
Port-X-Tender

THE HP-41C/CV CALCULATOR

The HP-41C/CV is a scientific, alphanumeric, programmable calculator. Externally, it consists of a liquid crystal display, two operating keys, a 35 key keyboard and four I/O ports. The **keyboard** is the primary method of inputting data and commands. It consists of 35 keys arranged in 8 rows as illustrated in Fig 1.1. The first three rows of keys contain five columns of keys and the remaining five rows contain four columns of keys.

Most of the keys on the keyboard serve three functions and some serve four functions. On the top face of each key is printed the normal unshifted function of that key. Above the key and printed on the overlay is the normal shifted function of that key. The character printed in blue on the front face of many of the keys is the alpha character that the key will represent when pressed with the machine in ALPHA mode. And finally, the picture of the keyboard on the back of the HP-41C/CV shows the shifted alpha representation of each key.

Each key of the keyboard is assigned a **keycode.** These keycodes are used by the machine to represent and locate keys on the keyboard. If the keyboard is thought of as a matrix with five columns and eight rows, then any key may be represented by its row/column number.

The keycode for an unshifted key is simply its row/column number. For example, the keycode for the SIN key which is in row 2 and column 3 is 23 and the keycode for the CHS key which is in row 4 column 2 is 42. Unshifted keys will

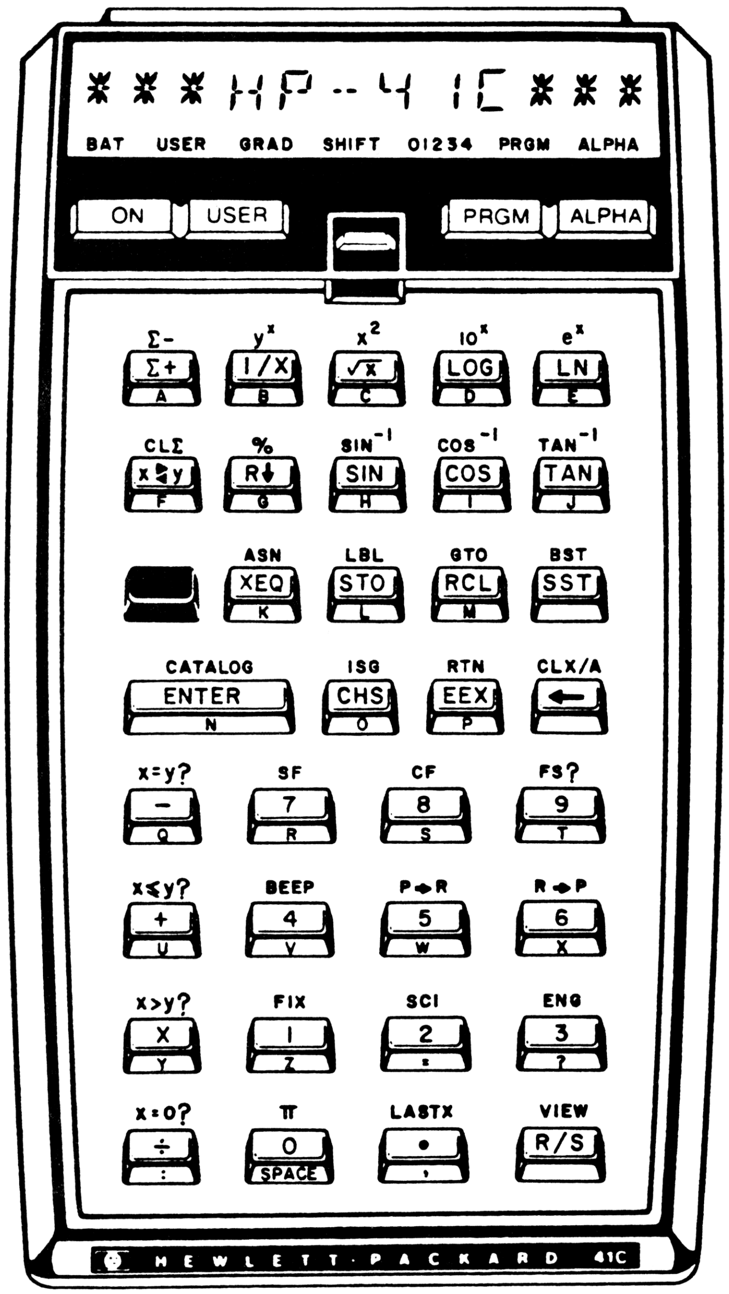


Fig. 1.1 HP-41C/CV keyboard

execute the function printed on the top of the key.

The keycode for a shifted key is the row/column number of the key preceded by a minus sign. For example, the keycode for the LBL key is -33. Shifted keys execute the function printed on the overlay above the key. In order to execute a shifted function, the gold shift key must be pressed before pressing the appropriate key.

There are no keycodes assigned to the unshifted or shifted alpha keys.

The operating system uses keycodes instead of the name of the function associated with the keys to reduce operating space and time. Every key has a preprogrammed, or normal function associated with it. These functions are stored in a key assignment table located within the operating system ROM's. Whenever a key is pressed and the machine is not in USER mode, the operating system determines the function to be performed by looking up the keycode in the table. If a key is pressed and the machine is in USER mode, the system first checks the appropriate key assignment bit map in the status registers to determine if the normal function of the key is to be located or if a reassigned function is to be executed.

When keys are reassigned, the new functions associated with the keys are stored in low user memory beginning at address 0C0. Through reassignment, keys may now serve up to 6 functions - 4 in NORMAL mode and two more in USER mode. Key reassignment also causes the corresponding bit to be set in one of the appropriate key assignment bit maps located in the status registers. In USER mode, the system will first scan the bit map to see if the bit corresponding to the key pressed has been set indicating that the key was reassigned. If the bit is set, the function of the key is obtained from the key assignment area.

The two **operating keys** are located above the keyboard and below the display. These two keys control the power to the machine and the mode or modes the machine is operating in. The operating keys are rocker switches which toggle the named function. The **ON** switch will turn the machine on if it is off or it will turn the machine off if it is on. When the machine is turned on, it wakes up in NORMAL or USER mode depending on which mode was active at the time the machine was turned off. Also when the machine is turned on, PRGM and ALPHA modes will be cleared if either happened to have been on prior to turning the machine off.

The **USER** switch will turn USER mode on or off. When in USER mode, keys which have previously been reassigned to different functions will assume their reassigned functions. In NORMAL mode, the keys assume their preprogrammed functions.

The **PRGM** switch will turn PRGM mode on or off. When in PRGM mode, all keystrokes are recorded as program lines in the current program file. The display shows the current program line and the instruction in that line. The machine will not remain in PRGM mode once the power is turned off.

The **ALPHA** switch will turn ALPHA mode on or off. ALPHA mode controls what is going into the ALPHA register. When in ALPHA mode, the characters appearing on the front face of each key become active. Each shifted or unshifted key pressed in this mode will cause the corresponding character to be recorded in the ALPHA register. The machine will not remain in ALPHA mode once the power is turned off.

The liquid crystal **display** serves two functions. First, it displays data located in one of the stack registers, one of the main memory registers or in the ALPHA register. Second, it displays the status of the machine.

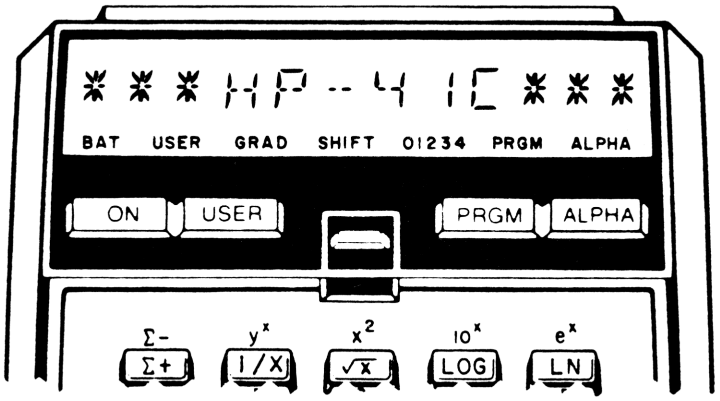


Fig 1.2 HP-41C/CV display

The display is capable of displaying up to 12 characters at one time. This will allow a signed 10 digit number with a signed two digit exponent to be displayed or 12 characters from the ALPHA register to be displayed. As a string of ALPHA characters is placed in the display either from the keyboard or from the ALPHA register, the display scrolls to the left dropping one character off the left end of the string and picking up one character on the right end of the string. In this manner, up to 24 characters may be displayed.

The status of the machine is displayed to the user through 7 annunciators. Each annunciator tells the user something about how the machine is operating and which mode the machine is in. The **BAT** annunciator signals the user that the batteries are weak. If throw-away alkaline batteries are being used, then there is about 20 days of operating time left. If rechargable NiCad batteries are being used, then there is about one minute of operating time left.

The **USER** mode annunciator, when on, signifies that the machine has been placed in USER mode. USER mode, you will recall, is set when the USER operating key is pressed. This mode may also be set by setting user flag F27. This mode may be turned off by again pressing the USER operating key or by clearing user flag F27. When in USER mode, any previously reassigned keys become active.

The **GRAD-RAD** annunciators signal the user that the machine is in GRAD or RAD trigonometric mode. If neither annunciator is on, then the machine is in DEG mode.

The **01234** annunciators signal the user that the corresponding user flag is set on. If the 0 annunciator is on then F00 is on, if the 1 annunciator is on then F01 is on, and so on.

The **SHIFT** annunciator indicates that the SHIFT key has been pressed. This warns the user that the next keystroke will invoke a shifted function. The annunciator will turn off if either the SHIFT key is pressed again or if a function key is pressed.

The **PRGM** mode annunciator indicates either one of two things. First, it may indicate that the machine has been placed in PRGM mode by pressing the PRGM operating key. Or the PRGM annunciator may indicate that a program is executing. PRGM mode may be turned off by pressing the PRGM operating switch or by turning the machine off. If a program is executing, PRGM mode will turn off when the program finishes.

The **ALPHA** mode annunciator, when on, indicates that the machine is in ALPHA mode. ALPHA mode is controlled by the ALPHA operating switch. When the machine is in ALPHA mode, the blue characters on the front face of the keys or the shifted characters corresponding to the chart on the back of the machine will be placed into the ALPHA register. ALPHA mode is turned off by again pressing the ALPHA operating switch or by turning the machine off.

The **I/O ports** allow the user to expand the capabilities of the basic HP-41C/CV. These ports are located on the top end of the machine and are numbered as shown in Fig. 1.3.

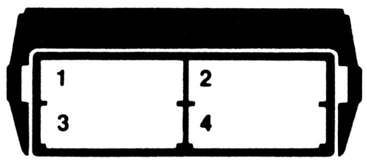


Fig. 1.3 The I/O ports

The HP-41C/CV system bus is accessible at the I/O ports. This enables the user to plug in any device capable of interfacing to the system bus. Plug-in ROM (read only memory) can increase the basic function set of the machine with programs prewritten in either the machines low-level language (assembly language) or as a sequence of user written instructions. Plug-in RAM (random access read/write memory) allows the user to expand the available memory of the basic HP-41C to equal that of the HP-41CV. Peripheral devices can be added to the system to further expand the machines capabilities. The functions for each peripheral are contained in the ROM associated with the device. Available peripherals include a magnetic card reader, a plug-in printer, an optical bar code reader and an interface loop capable of supporting up to 31 devices such as mass storage devices, printers and video interfaces.

INTERNAL SYSTEM ARCHITECTURE

Internally, the HP-41C contains eleven CMOS circuits and the HP-41CV contains 27 CMOS circuits. A CMOS circuit is a very high density integrated circuit which requires low power. A CMOS chip can retain data even when its operating voltage is drastically reduced. It is this feature of CMOS that makes the HP-41C/CV a continuous memory machine. The chip set of the HP-41C/CV consists of one CPU, up to 21 data storage chips, three ROM's and two display drivers. Fig. 1.4 is a block diagram of the system.

The **power supply circuits** supply power to the machine and monitor the battery. The hardware can exist in any of three power modes. In **SLEEP** mode, when the machine has been turned off, all of the circuits are inactive and are supplied with only enough battery current to sustain the continuous memory. When the machine is turned on, it enters **RUN** mode. If the machine remains inactive after being turned on, it enters **STANDBY** mode. In this mode, only enough power is supplied by the batteries to power the display. When a key is pressed, the machine once again enters RUN mode (the full power mode). If, after 10 minutes in STANDBY mode, the machine will turn itself off. A low battery circuit will signal the CPU to turn on system flag F49 and the BAT annunciator.

The **CPU** (central processing unit) is the "heart" of the machine. It is the circuit which interprets user commands and keystrokes and which controls the rest of the machine. Inside the CPU are five working registers (A, B, C, M and N). These registers are not to be confused with the status registers to be discussed later. Also within the CPU are an 8-bit register and a 14-bit status register, two pointers and four subroutine return registers. The CPU obtains ROM data using a 16-bit address. Data is passed to or from data registers using a 10-bit address. Internal to the CPU is a keyboard interface where the keys of the keyboard are decoded.

There are five **data storage** circuits in the basic HP-41C and 21 data storage circuits in the HP-41CV each consisting of 16 registers. One of these circuits is dedicated to a block of 16 registers known as the status registers. The status registers, shown in Fig. 1.5, contain, among other things, the program pointer, the subroutine return stack, the stack registers, the ALPHA register and the system flags. The other data storage chips provide the user with between 64 and 319 registers that can be used to store data and/or program files.

There are three system ROM circuits each containing 4K words of system read-only memory. These system ROM's occupy the lowest three addresses in the system "address space". Contained within these three ROM's is the operating system. The operating system is nothing more that a huge program written by the manufacturer and which makes the HP-41C/CV behave more like an advanced, alphanumeric programmable calculator rather than a computer. The operating system controls the function of the keyboard, display and I/O ports.

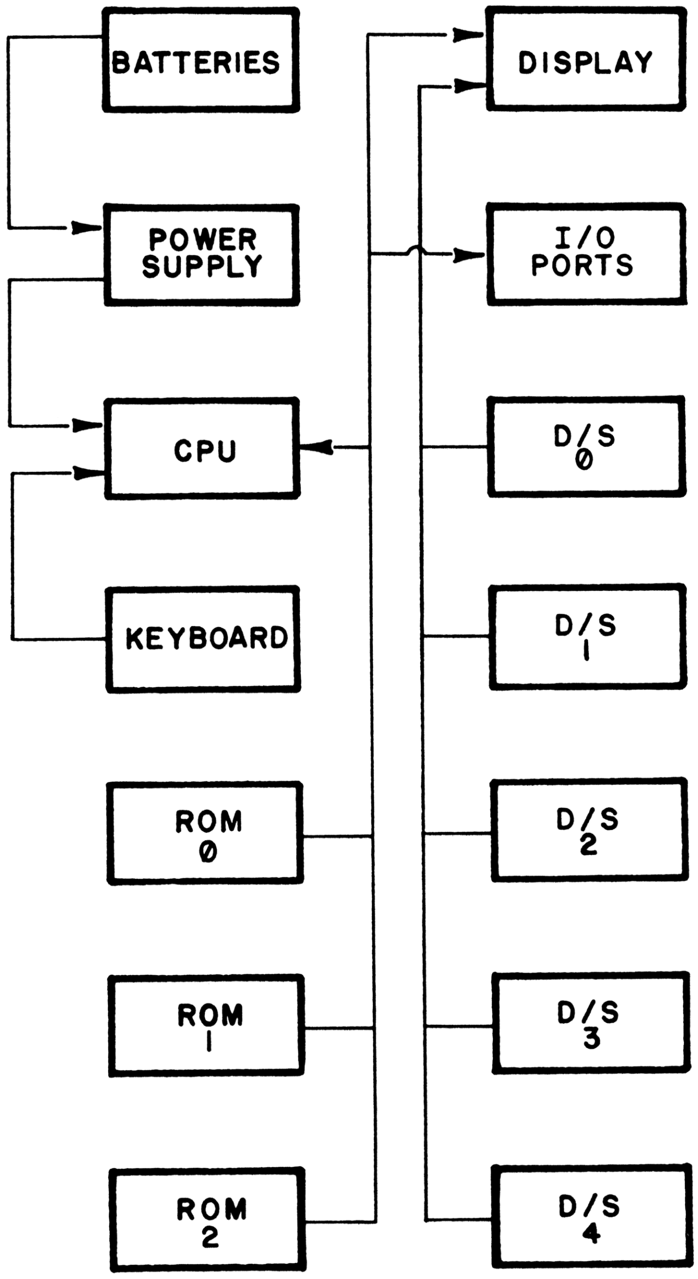


Fig. 1.4 Block diagram of HP-41C/CV

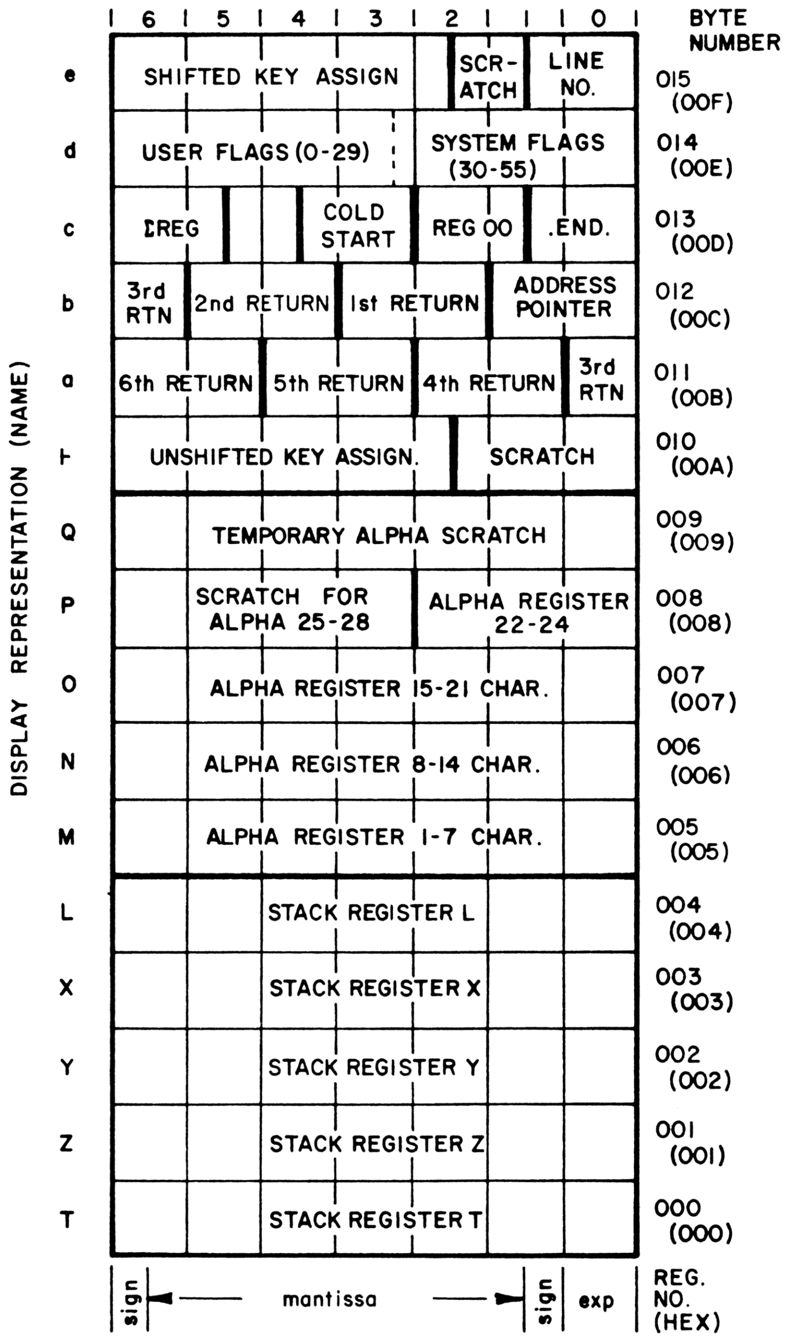


Fig. 1.5 The status register block

THE HP-41C/CV ADDRESSING SCHEME

The HP-41C/CV is capable of addressing (accessing) up to 16 pages of ROM each page being 4K words in length. The pages are numbered from bottom to top of memory beginning with page 0 at the bottom and going through page 15 at the top. The operating system occupies the first three pages numbered 0, 1 and 2. The next five pages are reserved for plug-in extension ROM's such as the HP-IL module, the Time module and the Extended Functions/Memory module. Also, pages 3-7 may be used for plug-in peripherals and modules which are addressed as actually residing in the I/O port ROM area (pages 8-15). These modules and peripherals are refered to as "hard addressed" since they will always occupy the same page location no matter which I/O port they are plugged into.

The ROM area for the four I/O ports begins with page 8. Each I/O port will accomodate up to 8K words of ROM. The first 4K words will occupy the lower of the page number pair for the I/O port. For example, a 4K single density module plugged into I/O port 1 would occupy page 8. An 8K dual density module plugged into I/O port 1 would occupy pages 8 and 9. Pages 8 and 9 are reserved for I/O port 1, pages 10 and 11 for port 2, pages 12 and 13 for port 3 and pages 14 and 15 for port 4. The modules occupying pages 8 to 15 are refered to as "port addressed" since the actual ROM address depends on which I/O port the module is occupying. Fig. 1.6 illustrates the system address space.

The HP-41C/CV is capable of accessing up to 8 pages of RAM each page being 4K words in length. The pages are numbered in a fashion similar to that for ROM from bottom to top beginning with page 0 and ending with page 7 (see Fig. 1.6). The first 16 registers of page 0 are occupied by the status register block. The remainder of page 0 and all of pages 1 through 4 are occupied by main memory. In the HP-41C, pages 1 through 4 are located in four single memory modules (HP 82106A) or one quad memory module (HP 82170A) plugged into the I/O ports. In the HP-41CV, pages 1 through 4 are built into the machine. Main memory is filled from the bottom up with no allowable gaps.

The other three pages numbered 5, 6 and 7 are occupied by the HP 82180A Extended Functions/Memory module (page 5) and two HP 82181A Extended Memory modules (pages 6 and 7). Each of these modules are "hard-addressed" to the page. The rule that there be no gaps in memory applies to extended memory also.

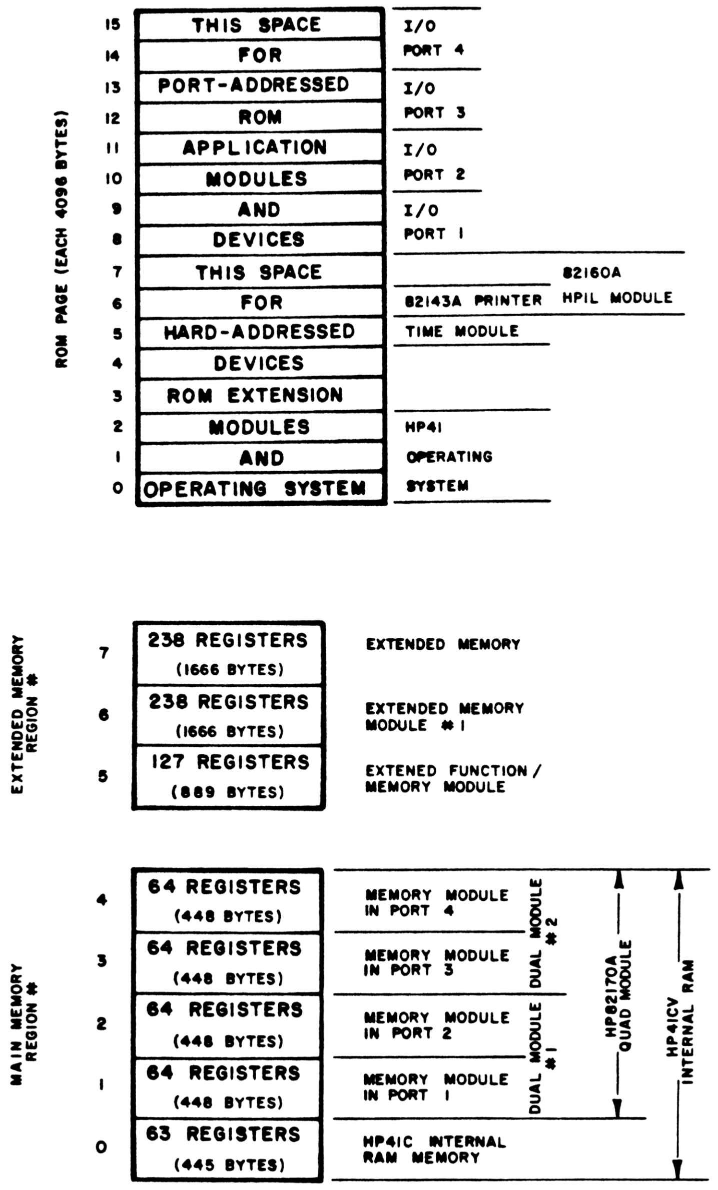


Fig. 1.6 System ROM/RAM address space

HP 82161A DIGITAL CASSETTE DRIVE

While the combined size of main and extended memory is large, the need exists to store several orders of magnitude more data. Secondary storage devices capable of storing enormous amounts of data meet this need even though access to the data is slow compared to internal memory access.

The movement of data between secondary memory and the rest of the computer is via the internal memory (main and extended memory). That is, data may be transfered from the internal memory to a secondary device, or the process may occur in reverse.

There is a class of secondary storage devices called mass storage devices which are capable of storing on the order of 131,000 bytes of information. The HP 82161A Digital Cassette Drive uses tapes which can store up to 131,000 bytes (nearly 19,000 registers) or 512 records of from 32 to 37 registers each.

Digital information is stored on magnetic mediums such as tape by lining up the magnetic domains in different directions to represent 0's and 1's. The surface of the medium passing under a read/write head has its magnetic domains lined up in one of two directions to represent 0 and 1. To read the information on the medium, the polarity of magnetism is sensed by the read/write head.

The front panel of the tape drive contains three switches and buttons and three indicator lights. The **ON-STANDBY-OFF** switch controls power to the machine. In the OFF position, the device is turned off and may only be turned on by manually moving this switch to the ON position. In the STANDBY position, the device may be turned on and off by using HP-IL commands sent from the controller which in this case is the HP-41C/CV. If the drive is operated in the STANDBY position, it will require more power than if operated in the ON position. In the ON position, the drive is continuously powered up and cannot be powered down except manually by placing the switch in the OFF position.

The **REWIND** button, when pressed will rewind the tape to the beginning of the clear tape leader. With the tape rewound it is less likely to be damaged. If the tape drive is busy, this button will not function.

The **OPEN** button will cause the tape access door to open allowing access to the tape. After pressing this button, the drive status is set to "new tape".

The **POWER** light indicates that the device is powered up. This light will turn off if the unit is powered down either manually or in the standby mode. This light will also turn off if battery power is low.

The **BAT** light indicates the condition of the internal batteries. If the light is on, then battery power is too low to operate the machine for much longer. The batteries should be recharged at this point to avoid damage to them. Recharging may occur simutaneously with operation of the device. If the light is off, then battery power is sufficient to run the device.

The **BUSY** light is on whenever the device is performing some requested operation. Conversly, the light is off if the device is idle.

The digital cassette has three internal features which are important for understanding its operation. **Buffer 0** which is 256 bytes in length is used for storing the information being transferred between the tape and the interface loop. **Buffer 1** which is also 256 bytes long is used to store the information going to or coming from the interface loop. The **byte pointer** directs the transfer of information to and from the two buffers by specifying which byte (numbered 0 to 255) to transfer.

There are two modes available for recording data onto the tape. In **continuous recording mode,** each time Buffer 0 is filled, its contents replace one tape record. This is useful for storing entire records. In **partial recording mode,** the current tape record is first copied into Buffer 0, then all or part of this copy is altered by data from the interface loop. Whenever the buffer is filled or the tape record update is complete, the revised contents of Buffer 0 replace the original record on tape. This is useful for changing part of a record without affecting the entire record.

HP 82162A THERMAL PRINTER

The HP-41C/CV display is very limited when it comes to outputting large amounts of data. The display is slow because of the scrolling which must occur when move than 12 characters are to be displayed at one time. Also, a hardcopy of data is not possible when using the display alone.

Output devices such as the HP 82162A Thermal Printer solves both of these problems. The printer attaches to the HP-IL and is controlled by the HP-41C/CV through HP-IL commands.

The front panel of the printer contains five switches and two indicator lights. The **OFF-STANDBY-ON** switch controls the power to the machine. In the OFF position, the device is powered down and may only be powered up by manually moving the switch to the ON position. In the STANDBY position, the device may be powered up or down through HP-IL commands sent by the controller which in this case is the HP-41C/CV. If the machine is operated in the standby mode, it will require more battery power. In the ON position, the unit is powered up and will remain on until manually turned off.

The **INTENSITY** switch controls the darkness of the print. There are five darknesses ranging from very light to very dark.

The **MODE** switch sets the printer to one of three operating modes. In **MAN** (manual) mode, the printer is idle until a print function is given by a running program or by the user. If a program listing is produced in this mode, the listing will be left-justified.

In **NORM** (normal) mode, all function names, numbers and alpha characters are printed as they are keyed in by the user. If a running program issues a print function or a PROMPT command, the data or the prompt will be printed. Program listings produced in this mode will be right justified with the LBL statments marked so as to stand out better.

In **TRACE** mode, all numbers, function names and alpha characters are printed as they are keyed in by the user. In a running program, all function names, intermediate results and final answers print in addition to data printed by a print function. Program listings are produced in a special packed format.

The **PRINT** button, when pressed, will insert print functions into a program or it will print the X or ALPHA registers. When the HP-41C/CV is in PRGM mode, pressing the PRINT button will insert a PRX command as the next program step. If the HP-41C/CV is in both PRGM mode and ALPHA mode, pressing the PRINT button will insert a PRA command as the next program step. If the HP-41C/CV is not in PRGM mode and not in ALPHA mode, pressing the PRINT button will cause the contents of the X register to print. If the HP-41C/CV is in ALPHA mode, pressing the PRINT button will cause the contents of the ALPHA register to print.

The **PAPER ADVANCE** button will insert ADV commands into a program if the HP-41C/CV is in PRGM mode. If the HP-41C/CV is not in PRGM mode, the PAPER ADVANCE button, when pressed, will advance the paper on line.

The **POWER** light is on whenever the printer is powered up. The light will go off if the machine is powered down either manually by placing the OFF-STANDBY-ON switch in the OFF position or through program control when in STANDBY mode. This light will also go off if battery power is insufficient.

The **BAT** light indicates the condition of the batteries. If the light is off, then there is sufficient power to operate the machine. If the light is on, then battery power is too low to operate the machine. At this point the batteries should be recharged. This may be done while the machine continues to operate.

There is an internal feature which is important to the understanding of the printer operation. The **PRINT BUFFER**, which is 101 bytes in length, accumulates data to be printed. In each byte, a character to be printed or a control character may be stored. The printer will print the contents of the print buffer when one of two things happen. First, if the buffer is full it will be printed. Second, if the user sends a control character such as a carriage return (CR) the buffer will be printed and cleared.

The printer may operate in one of several modes. In **Uppercase/Lowercase** mode, the print buffer will accumulate alphabetic characters in either upper or lower case. Other characters will not be affected.

In **Single Wide/Double Wide** mode, characters will be accumulated and will print either as normal width characters or as double width characters.

In **Character/Column** mode, the print buffer will accumulate either character or dot patterns.

In **Left/Right Justify** mode, the printer is set to print the print buffer either left-justified (lined up with the left margin) or right-justified (lined up with the right margin).

In **Parse/Nonparse** mode, the printer will output the print buffer and break a line only at a space (parse mode) or when 24 characters have been printed on a line (nonparse mode).

The printer, in addition to outputting regular lines of print, may also output graphics or barcode. When outputting graphics, the print buffer accumulates columns of dots to be printed. By building each column in the proper fashion, special graphics characters may be output. Lines of barcode may also be accumulated in the print buffer and then output. This barcode may be produced either for program lines or for data lines.

HP 82160A HP-IL MODULE

The HP-IL (Hewlett-Packard Interface Loop) is a general purpose interface between a controller (the HP-41C/CV in this case) and one or more HP-IL devices. The controller and all devices attached to the loop are connected together in series forming a closed communication circuit. Fig 1.7 illustrates this circuit with the HP-41C/CV and two devices attached. The HP-IL is "hard-addressed" to ROM pages 6 and 7 (see Fig. 1.6) and occupies 8K words.

Information is passed from one device to another around the loop. When the information reaches the intended device, that device will respond accordingly. All the other devices which receive the information simply ignore it and pass it on. The device which originally sent the information will eventually receive it again.

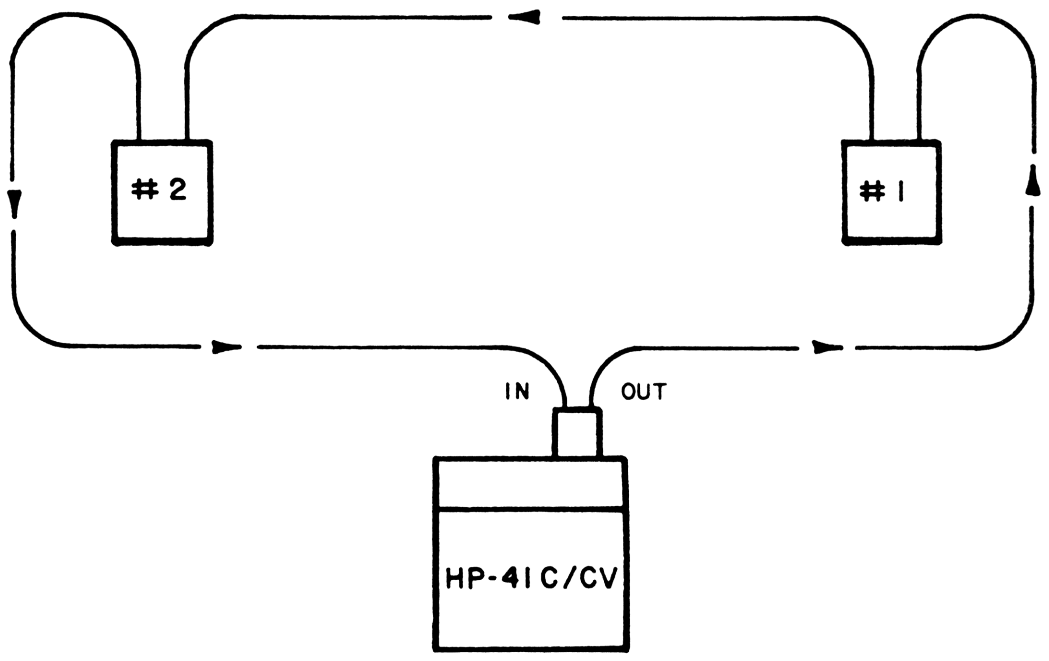


Fig. 1.7 The HP-IL communications circuit

Once the information is received back at the sender, it is compared with what was sent out on the loop. If the information sent does not agree with the information received, an error message will be issued. If the information sent and received agree, the next set of information will be sent.

Each device attached to the loop is assigned a role by the **system controller.** Once a device is assigned this role, the device must operate within the limits of its role. For example, a printer cannot magnetically record information. The HP-41C/CV is both the system controller and the active controller. Commands are initiated by the controller and are monitored by all devices on the loop.

A **listener** is a device which receives information sent around the loop. There may be multiple listeners assigned by the controller. A mass storage device is a listener when it is receiving and recording data. A printer is a listener when it is receiving and printing data. The controller is a listener when it is receiving data from another device.

A **talker** is a device which sends information out over the loop. There may be only one talker assigned by the controller at any one time. A mass storage device is a talker when it is reading a recorded file and sending out the data read. The controller is a talker when it is sending data out over the loop.

Any device which is not a talker or a listener is inactive.

The HP-IL is a ROM which contains three sets or classes of functions. There is a set of **interface control functions** which give the user more complete control of interface activity for any device attached to the loop.

Printer functions give the user almost complete control of the printer or printer-like devices. The printer functions can be classified as:

> **Standard Printing Operations** - instruct the printer to print data immediately without waiting for the internal buffer to fill first
>
> **Print Buffer Operations** - data is accumulated in the print buffer along with control and formatting information. The buffer prints when it is full or when it is explicitly instructed to do so
>
> **Graphics Operations** - specially built characters are accumulated column by column and then printed
>
> **Plotting Operations** - allow single-valued mathematical functions to be plotted on a predefined set of axes

Mass Storage functions give the user almost complete control over the operation of mass storage devices.

HP 82180A EXTENDED FUNCTIONS/MEMORY MODULE

The Extended Functions/Memory module adds two new features to the basic

HP-41C/CV. First it adds 47 new functions. Second, it adds 127 extended memory registers which may be used to store data and programs. This module is "hard-addressed" to RAM page 5 (see Fig. 1.6).

Extended memory registers are similar to main memory registers in that they may be used to store data and programs. However, there is one important difference between main memory and extended memory. Extended memory registers may not be accessed directly with STO or RCL functions. The data contained within extended memory must first be moved to main memory using extended functions.

Extended functions may be grouped into four categories. **Programmable functions** are functions which may be coded directly into a program. When used in this manner, these functions operate exactly as if they had been executed manually. Some of the standard HP-41C/CV functions such as ASN, SIZE and CLP which may not be included in programs have extended function analogues which may be used in programs.

Register and **Flag functions** are a new class of functions not available on the basic HP-41C/CV. Register functions operate on entire blocks of main memory registers by moving or swapping data between one block and another. Flag functions give the user access to the status of flags 0 through 43 and allow the user to change the status or to store the status of these flags.

ALPHA Register functions give the user the ability to shift characters between the X register and the ALPHA register, determine the length of a string in the ALPHA register, search the ALPHA register for a specific string and rotate the string in the ALPHA register a specific number of positions left or right.

Extended Memory functions give the user the ability to define a new class of data called internal files. Data which is stored into internal files is considered to have a logical relationship with all other data in that internal file. These functions also enable the user to edit data within one class of internal file called ASCII files.

HP 82181A EXTENDED MEMORY MODULE

The HP 82180A Extended Function/Memory module contains 127 extended memory registers. Up to 476 more extended memory registers may be added for a total of 603 extended memory registers.

Each HP 82181A Extended Memory module will expand the available extended memory by 238 regsiters. The HP-41C/CV will address up to two of these modules. If more are added, the registers in the excess modules will not be accessed due to the fact that extended memory is "hard-addressed" to the last two RAM pages (pages 6 and 7).

HP 82182A TIME MODULE

The HP 82182A Time module expands the basic HP-41C/CV even further with new functions and capabilities. There are 29 new functions supplied by the TIME module. These functions may be grouped into five categories. **Date** and **Time functions** give the user the ability to format the date and time, set the date and time, specify the clock display contents, display the clock, recall the clock and date and append the date or time into the ALPHA register.

Calendar functions give the user the ability to perform date arithmetic. A specified number of days may be added to the date, the number of days between two dates may be calculated and the day of the week may be determined.

Stopwatch functions turn the HP-41C/CV into a manual or programmable stopwatch. Splits may be stored and recalled, the stopwatch may be set, started, stopped and the stopwatch time may be recalled.

Alarm functions give the HP-41C/CV the ability to start and stop programs, turn itself on and off and audibly signal the user of some event.

Time Adjustment functions give the user or the HP-41C/CV the ability to correct the current time or date and monitor the accuracy of the internal clocks.

The Time module is "hard-addressed" to ROM page 5 and may be installed in any I/O port above HP 82106A or HP 82170A Memory modules.

PORT-X-TENDER

The AME PORT-X-TENDER is a device produced by AME Design[1]. The function of this device is to increase the number of available I/O ports from four to 10. This device plugs in to I/O port 3 and attaches to the bottom of the HP-41C/CV by means of Velcro^{TM} strips. Due to the large number of plug-in modules (between 5 and 9), the standard number of I/O ports is quickly used up. To overcome this difficulty the Port-X-Tender is used.

THE MANUALS

This book is not intended to be a replacement for the owners manuals which accompany the HP-41C/CV and the various peripherals and modules. Rather, this book is intended to serve as a guide for anyone interested in using their machine for its intended purpose - to process data. For this reason, it is highly recommended that the user read and master the various owners manuals as far as the installation and operation of the peripherals and modules is concerned.

NOTES AND REFERENCES

^{1} AME Design
Box 373
13450 Maxella, G185
Marina Del Rey, California 90291

Figure 1.6 is copied from the PORT-X-TENDER owners manual

Johnson, N. L. and Marathe, V. V.; "Bulk CMOS Technology Forn The HP-41C"; **Hewlett-Packard Journal**, March, 1980.

Kane, G., Harper, S., Ushijima, D.; **The HP-IL System: An Introductory Guide to the Hewlett-Packard Interface Loop.**; Osborne/McGraw-Hill, Berkeley, California, 1982

Musch, B. E., Wong, J. J. and Conklin, D. R.; "Powerful Personal Calculator System Sets New Standards"; **Hewlett-Packard Journal**, March, 1980.

Chapter 2

RPN and the Stack

INTRODUCTION

This chapter will discuss the stack and the language of the HP-41C/CV. These two topics are so closely tied together that it is useless to discuss one without also discussing the other. The importance of a good understanding of these concepts cannot be overstated. The HP-41C/CV expects the user to communicate with it in its own language which is termed RPN.

This chapter will guide you to an understanding of RPN and the stack by discussing the following topics:

History of RPN
Algebraic vs RPN notation
Translating algebraic expressions into RPN
The stack
Stack operations
RPN and the stack

HISTORY OF RPN

In his book **Elements of Mathematical Logic** (1929), and in other of his publications, the Polish logician and mathematician Jan Lukasiewicz defines a parentheses-free mathematical notation in which the functors (operations) precede their parameters. For example, rather than writting (8+3)*2, one would write + 8 3 * 2. The reason for this strange notation is compactness:

parentheses are eliminated
the number of symbols which must be written is reduced

Since each symbol in an expression such as this represents a keystroke on the HP-41C/CV, this Polish notation system has its advantages. By placing the functor after its parameters, reverse Polish notation (RPN) is obtained. The above example now becomes 8 3 + 2 *.

ALGEBRAIC VS RPN NOTATION

Most of us learned that the use of parentheses is necessary in order to reduce ambiguity in mathematical expressions. The expression **3X4+2,** for example, may be interpreted in one of two ways:

3X4 +2
3X 4+2

In the first case, the answer would be 14 while in the second case the answer would be 18. Written as **3X4+2,** there is no way to know the intended order of evaluation. The use of parentheses alleviates this problem to some extent. If the above expression were written as **(3X4)+2,** then the multiplication would be done first, followed by the addition. If the expression were written as **3X(4+2),** then the addition would be done first, followed by the multiplication. This example, although simple, does illustrate the need for parentheses as **grouping symbols.**

For more complicated expressions, it might be necessary to use several sets of parentheses. But the more sets of parentheses used, the harder it is to read and comprehend the expression. It is a good idea to use only as many sets of parentheses as necessary to completely reduce ambiguity.

Once the expression has been written with parentheses, the question still remains - in what order is the expression to be evaluated? For example, the expression:

(3X(4+2)+(2X3))/(2+(6X8))

could yield two answers depending on how the expression in the numerator is evaluated. The numerator could be evaluated in one of two ways:

3X (4+2)+(2X3)
3X(4+2) +(2X3)

For this reason, certain mathematical conventions have been adopted to solve the order of evaluation problem.

1. evaluate all of the **inner most** sets of **parentheses** and then work out to the next sets of parentheses, and so on
2. evaluate in a **left to right** direction
3. perform **arithmetic operations** in the following **order**

exponentiation
multiplication and division
addition and subtraction

Using these conventions, we can now evaluate the expression properly. The steps involved would be:

(3X(4+2)+(2x3))/(2+(6x8))

1. evaluate the inner most sets of parentheses first

(4+2)=6
(2X3)=6
(6X8)=48

(3X6+6)/(2+48)

2. evaluate the numerator from left to right

3X6=18 +6=24

24/(2+48)

3. evaluate the denominator

(2+48)=50
24/50

4. finish the evaluation of the expression to get **0.48.**

From the above example, it is apparent that "there's got to be a better way" to write and evaluate expressions. Writting expressions without parentheses does not work because the order of evaluation can not be readily determined. On the other hand, parenthesis are difficult to use and key into a computer. In 1928 or earlier, the Polish logician, Jan Lukasiewicz, invented a parenthesis-free notation to describe expressions in symbolic logic. Because his nationality is easier to pronounce than his last name, this system of notation became known as Polish notation.

Polish notation is unique for two reasons. First, it is completely free of any type of grouping symbols. Functors and their parameters are written as they are to be evaluated. Secondly, Polish notation is unique because the functor (mathematical operator) is written first followed by its parameters.

Polish notation is best understood by means of an example. Look again at the expression

3X4+2

You will recall that in algebraic notation this expression may be written in one of two ways depending on the intended order of evaluation.

(3X4)+2
3X(4+2)

In Polish notation, the expression could be written in one of two ways depending on the intended order of evaluation.

X34+2
+42X3

At first this notation may look foreign and unnatural. However, upon closer examination, one will discover that this is written in essentially the same order as one would mentally go through the evaluation. Assume that the order of evaluation is intended to be **(3X4)+2.**

Mentally, the evaluation of this expression would proceed as follows:

1. **multiply** 3 by 4
2. **add** 2

Now look again how this expression would be written in Polish notation.

X34
+2

The similarity between actual mental evaluation and Polish notation should now be clear. Polish notation, whether we realize is or not, is really how we mentally evaluate mathematical expressions.

The calculator market today is occupied by two classes of machines. First, there are those machines which use an algebraic entry system. The expression to be evaluated is entered into the machine exactly as it is written, parentheses and all. The expression is evaluated following the rules of algebra where evaluation begins at inner parentheses, multiplication and division first, etc.

The other end of the market is occupied by those machines which use a varient of Polish notation called Reverse Polish Notation (RPN). Reverse Polish Notation, as the name implies, is a reversed Polish notation. Instead of the functor preceeding its parameters, the functor follows its parameters. In RPN, the expression (3X4)+2 would be written 34X2+.

The natural question at this point might be - how does one translate algebraic expressions into RPN?

TRANSLATING ALGEBRAIC EXPRESSIONS INTO RPN

The translation process is really quite simple. Beginning with the algebraic expression, it is evaluated according to the following conventions and, as the evaluation proceeds, written in RPN.

1. begin with the **inner most** sets of **parentheses**

2. translate in a **left to right** fashion
3. perform **arithmetic operations** in the following **order**

exponentiation
multiplication and division
addition and subtraction

Following these conventions and with a little practice, even the most complex mathematical expression may be translated into RPN. The following example will illustrate.

$$Y_f = 4/3\ h(1 + 2\sqrt{V_f/V_o})$$

1. starting inside the parentheses, evaluate V_f/V_o

   V_fV_o/
2. **SQRT**
3. multiply by 2

   2*
4. add 1

   1+
5. multiply by h

   h*
6. evaluate 4/3

   43/
7. multiply

   *****

Written out in full, the RPN for this expression would be written as follows.

V_fV_0/SQRT2*1+h*43/*

THE STACK

A **stack** is so named because of its similarity to the device in some cafeteria's which holds plates. A stack of plates rests on this spring-loaded mechanism. As more plates are added, the stack is pushed down by the weight of the new plates. As a plate is removed, the stack lifts because some of the weight has been removed. This type of stack is called a LIFO (last-in first-out) stack. As plates are added, they are **pushed onto** the stack. As plates are removed, they are **popped off** the stack.

The HP-41C/CV has a LIFO stack. This push down stack is four registers in length. The last-in first-out register is called the X register and is the register normally displayed. The other three registers in order are the Y, Z and T registers.

The registers in the stack are nearly always pictured upside down so that the X register is on the bottom and the T register is on the top. This arrangement allows the registers to be logically oriented for the operations to be performed. For example, the operation x-y says to subtract y from x. If register X contains x and register Y contains y, the upside down arrangement of the stack will place register Y above regsiter X meaning that y is entered before x. This is more logical than the other way around where register X would appear above regsiter Y. Fig. 2.1 illustrates the upside down stack arrangement.

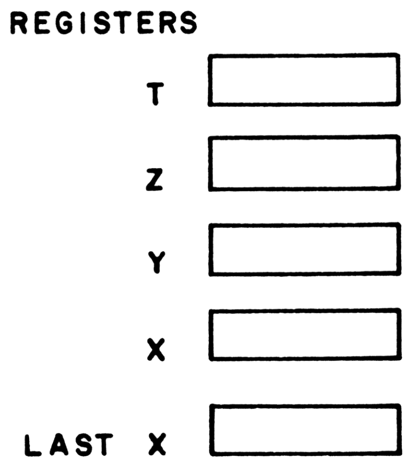


Fig. 2.1 The HP-41C/CV stack

STACK OPERATIONS

Basically, there are two operations which affect the stack - **manipulators** and **functors.** Functors can be **monadic, diadic** or **bifid.** A **monadic functor** is a functor which requires only one parameter in the X register. Examples of monadic functors are the CHS and the SIN operations. One number is popped off the stack, operated upon and the answed placed back into the X register.

Diadic functors require two parameters, one in the X register and the other in the Y register. Examples of diadic functors are the arithmetic operations of +, - , * and /. Two numbers are popped off the stack, combined to give the answer which is then placed into the X register.

Bifid functors require two parameters, one in the X register and the other in the Y register. Examples of bifid bunctors are the R-P and P-R operations. Two numbers are popped off the stack, operated upon to give two answers which are then placed back into the X and Y registers.

Manipulators are operations which affect either the arrangement of or the contents of the stack. Examples of manipulators are the ENTER operation and the X swap Y operation.

The HP-41C/CV is normally in the auto-push state. A new number keyed into the machine in this state pushes into the X register, while the previous contents of the X register goes into the Y register, and so on. The previous contents of the T register are lost.

Some functors and manipulators will disable auto-push. In this state, a number keyed into the machine will overwrite the X register and not disturb the Y, Z and T registers.

RPN AND THE STACK

You may have noticed in the discussion of translating algebraic expressions into RPN, that diadic functors in RPN sometimes appear to have only one parameter. For example, in the RPN expression 34*2+, the parameters for the multiplication are obviously 3 and 4, but what are the two parameters for the addition? One of the parameters is the 2 and the other parameter, as we shall see, is the resulting product of the multiplication of 3 by 4.

When the expression 34*2+ is to be evaluated on the HP-41C/CV, the parameters and functors are entered into the machine starting at the left and proceeding to the right. As each parameter is entered, the stack is affected in some way. In order to understand exactly how the stack is affected it is useful to picture the stack after each operation. Let us walk through this simple example and after each digit entry or mathematical operation, we will picture the stack using the printer function PRSTK. In this manner, we can see how the stack changes after each operation or after each number entry. The assumption is being made here that the stack contains all zeroes.

Begin the evaluation of the RPN expression 34*2+ by keying in the number 3. When the 3 is keyed in, it appears in the display (X register).

```
T=  0.00
Z=  0.00
Y=  0.00
X=  3.00
```

Stack after entry of the number 3

Now, unless we tell the machine otherwise, the next number to be keyed in will be appended onto the 3. In other words, we must tell the machine when one number ends and another number begins. Digit entry disables auto-push. What we need is some operation which will enable auto-push for us. One way to enable auto-push is to press the ENTER key. The ENTER functor operates by enabling auto-push which causes the stack to lift when the next digit key is pressed.

```
T=  0.00
Z=  0.00
Y=  3.00
X=  3.00
```

Stack after ENTER is pressed

The next number can now be entered into the machine.

```
T=  0.00
Z=  3.00
Y=  3.00
X=  4.00
```

Stack after entry of the digit 4

The multiplication can now be performed. This is done by pressing the X (multiplication) key. Most functors will enable auto-push so that a digit entered into the machine after the functor has executed will cause the stack to lift. The X operation is a diadic functor which requires two parameters -one in the X register and the other in the Y regsiter.

```
T=  0.00
Z=  0.00
Y=  3.00
X=  12.00
```

Stack after multiplication

We are now ready to enter the next digit into the machine. Since auto-push was enabled by the X operation, we don't need to worry about digit separation between the 12 in the X register and the 2 we are about to enter.

```
T=  0.00
Z=  3.00
Y=  12.00
X=  2.00
```

Stack after entry of the digit 2

Do you now see the two parameters required by the + operation? The 12 in the Y register is the product from the previous multiplication. The 2 in the X register is the number just entered.

The final operation may now be performed which will leave the final answer in the X register.

```
T=  0.00
Z=  0.00
Y=  3.00
X=  14.00
```

Stack after + operation

This example has illustrated how the stack is affected by data entry and machine operations. Almost every operation which may be performed on the HP-41C/CV will affect the arrangement of the stack. It is important that the user understand how the stack is affected by data entry and machine operations even though he may not wish to use the stack as it was intended to be used.

The user may take one of two approaches when using the stack. One approach is to store the results of all calculations into main memory registers and recall them as needed. This approach is useful when there is no time to carefully design an evaluation which will use the results of previous calculations left in the stack. This approach may be used in a small program where the number of available main memory registers used for storage is essentally unlimited.

The other approach the user may take when using the stack is to carefully design the evaluation algorithm so as to take full advantage of the stack. It is possible to evaluate expressions in such a way as to ensure that the stack registers are set-up for the next operation.

If it is important to take full advantage of the stack, the user must know ahead of time the parameters which will be needed and what the stack looks like at any particular time. One way to know this is to analyse how the stack will be affected after each operation or digit entry.

There is a printer function which will print the stack registers for you. The **PRSTK** command could be executed after each parameter has been entered or after each operation has been performed to show what the stack looks like. In programs, it could be included after each program line so that as the program is running, the contents of the stack will be printed. Notice that the PRSTK command does not print the LASTX register. This command, as will most of the other commands on the HP-41C/CV, will enable auto-push.

CONCLUSION

At this point, you should have a good idea of what RPN is and how it operates with the stack. If you do not feel comfortable with RPN and how to convert algebraic expressions into RPN notation, you should pause now and practice evaluating expressions with pencil, paper and HP-41C/CV.

REFERENCES

Ball, J. A. ; **Algorithms For RPN Calculators.**; John Wiley & Sons, New York, 1978.

Chapter 3

Simple Programming Summary

INTRODUCTION

This chapter will serve as an HP-41C/CV programming summary. This summary will be presented in such a way as to show how programming will save time and labor. The following topics will be discussed.

Definition of a program
Structure of a program
Keying a program into program memory
Program verification
Editing a program
Running programs
Compiling programs

DEFINITION OF A PROGRAM

The best way to define a program is to explain what a program will do. Suppose that you want to do a calculation that will convert gallons to liters. The equation to be used is

liters = gallons * 3.785

Using this equation, you must key in the number of gallons to be converted and then multiply this amount by 3.785. To convert 33.2 gallons to liters, for example, would require the following keystrokes:

```
 33.20 ENTER↑
 3.785      *
125.66    ***
```

After pressing the X key, the number of liters (**125.66**) will be displayed. This problem required 11 keystrokes to arrive at the answer. If this conversion is not to be done often, then this method of converting gallons to liters is adequate. Suppose, however, that is necessary to do this calculation 16 times a day, every day. Not only would the number of keystrokes required become significant, but the chances of an error would greatly increase. Notice that each time the calculaton is to be done, the constant 3.785 must be reentered. If some of the numbers of this constant are reversed, the final answer would be in error. Therefore a slightly better method of doing this conversion does not require that the constant be entered each time. The following soluton is based on the fact that the HP-41C/CV will "remember" data stored in its main memory registers even after the machine is turned off. Suppose that the constant 3.785 is pre-stored in main memory register R00. Then the conversion would require the following keystrokes:

```
33.200 ENTER↑
        RCL 00
             *
125.662    ***
```

This solution requires 9 keystrokes and is virtually error free. When it is necessary to do the conversion several times, the constant does not need to be entered.

There is an even better solution to this problem. This solution is based on the fact that the HP-41C/CV can "remember" a series of keystrokes called a program. Once the keystrokes are entered into the machine, they will remain there until explicitly removed. To make these keystrokes perform their intended function, you simply press one or two keys to start the program. The program to perform the gallons to liters conversion is as shown on the following page.

STRUCTURE OF A PROGRAM

Look carefully at the gallons to liters program. There are several lines of code which do not correspond to any keystroke used in

```
01♦LBL "GALLIT"
02 "HOW MANY GALLON"
03 PROMPT
04 3.785
05 *
06 "LITERS = "
07 ARCL X
08 AVIEW
09 END
```

Program to convert gallons to liters

the manual solution. These extra lines of code are called "overhead" and are necessary in order to name the program, to prompt the user for information and to display the results. In the following discussion, each line of the program will be discussed.

Line 1 of the program is the **program header.** The program header is necessary in order to tag the program with a unique name.

Programs are keyed into and stored in a part of main memory known as **program memory.** Program memory and data storage registers together make up main memory. Program memory may be made as large or as small as desired at the expense of the data storage registers. If program memory is made large enough to contain more than one program, then it is necessary to have some way of telling programs apart. Programs are typically stored as individual program files each with a unique name. Access to a program then is by its name. Another advantage to storing programs as separate files is that the operating system will enter the name of the program in its program catalog. This catalog may be viewed at any time by executing the **CAT 1** command. As this command is executing, the name of each program file will be displayed in the order the programs were stored.

Line 2 is the beginning of the **body** of the program and is a message which will be displayed to notify the user that some information is needed. This is one of the main uses of the alphanumeric capability of the HP-41C/CV. Prompting for data by name when it is needed makes the machine very friendly because the user is told exactly what is needed by the program.

Line 3 of the program is the instruction that actually stops the running program and displays the data prompt. Once the requested data has been keyed in, the program is restarted by pressing the **R/S** key.

Line 4 of the program is the constant which will be multiplied by the number of gallons.

Line 5 is the multiplication instruction itself.

Line 6 is the first part of a message which will be built in lines 6 and 7. The purpose of this message is to label the result of the calculation. The user will not have to guess what the number being displayed is. In this program, annotating the output is trivial but in larger programs with several outputs, annotating the results is essential.

Line 7 of the program is the instruction which will append the contents of the X register onto the end of the string in the ALPHA regsiter. After the ARCL instruction has executed, the ALPHA register will contain the complete message.

Line 8 of the program is the instruction which will cause the contents of the ALPHA register to be displayed thereby showing the user the result of the calculation.

Line 9 of the program is the **program trailer** which signals the end of the current program and program file. The operating system considers everything between the program header and the program trailer as a single program file.

KEYING A PROGRAM INTO PROGRAM MEMORY

We have written a small program which will convert gallons to liters. The next step is to enter the program into program memory.

As we have seen, a program is nothing more than a series of keystrokes. The HP-41C/CV has the ability to "remember" a series of keystrokes provided they are placed in program memory.

Main memory consists of 319 registers when all memory is present. These registers may be used to store data or programs or both. If both data and programs are to be stored, the operating system must have some way of knowing what information in main memory is data and what information is program since internally both are stored as strings of bits. The way the operating system knows where data ends and programs begin is by means of a soft "curtain". This curtain is the dividing line between the main memory register R00 and the first program file. The address of this curtain is variable and is stored in status register **c**. If the user wants to re-partition main memory (that is change the location of the curtain) he does so using the **SIZE** or **PSIZE** (an extended function) function. Main memory may be partitioned such that there is no program memory and all data storage registers or such that there are no data storage registers and all program memory or anywhere in between.

Program memory is directly accessed by the user only when the HP-41C/CV is in PRGM mode. To enter or exit this mode, the PRGM operating switch is pressed. When in PRGM mode, the PRGM annunciator will be on and a program line or the .END. REG nn line will be displayed. If the .END. line is showing, nn is the number of available program memory storage registers remaining.

The first thing to do after entering PRGM mode is to position the program pointer to the next available program memory register. To do this the **GTO..** instruction is used.

Executing this instruction will not only position the program pointer, it will also **pack** program memory so that wasted space will be released. Wasted space occurs whenever lines of code are deleted. Normally, instructions are packed so that as many instructions as possible are put into each program memory register. Instructions are not split bewtween registers. If an instruction will not completely fit into a register the remaining bytes of the register are filled with nulls (hexadecimal '00') and the instruction is put into the next register. When an instruction is deleted, the space formerly occupied by the instruction is filled with nulls. Packing program memory removes these nulls and replaces them with instructions. The amount of space released by packing can be considerable if the program had been heavily modified or if many mistakes had to be corrected while keying a program in.

After positioning the program pointer to the next available register in program memory, the program may be keyed in one line at a time starting at the first statement. When a character string must be keyed in, simply remain in PRGM mode and go into ALPHA mode and key in the characters between the " "'s. After the characters have been keyed in, take the machine out of ALPHA mode and continue keying the program.

To key in line 1 of the program, press the LBL key. When the display shows LBL_ _, go into ALPHA mode and key in the characters G A L L I T. Notice that after ALPHA mode is entered, the _ _ becomes a single _. This single dashed line will precede the characters as they are being keyed in. The complete set of keystrokes to enter the first line would be as follows. The **shift** means that the gold shift key must be pressed.

shift LBL ALPHA G A L L I T ALPHA

The next line of the program may now be keyed in. This is done by pressing the following series of keys. The program pointer will automatically go to line 2 when you begin keying this line.

ALPHA H O W b M A N Y b G A L L O N ALPHA

The b is the symbol for a space.

Line 3 of the program is entered by spelling out the name of the function PROMPT. Any HP-41C/CV function may be executed by spelling out its name. Some of the more commonly used functions have been preassigned to the various keys of the keyboard. If a function is not assigned to a key, then it may only be executed by spelling it out. In order to execute a function that has not been assigned to a key, you must first press the XEQ key, put the machine into ALPHA mode, spell out the name of the function then exit ALPHA mode. Once ALPHA mode is exited, the function will be placed into program memory as an instruction. The sequence of keystrokes used to key in line 3 of the program are:

XEQ ALPHA P R O M P T ALPHA

Line 4 of the program is entered simply by pressing the appropriate digit and decimal point keys. The sequence would be:

3 . 7 8 5

Line 5 of the program is entered by pressing the X (multiply) key. Notice that when the X key is pressed, the previously entered number was terminated. The X key acted as a number terminator. Suppose that another number was to be keyed in immediately following the 3.785. If the new number, say 1000, is entered after entering the last number and no keys had been pressed, the new number would be appended onto the end of the number 3.785 making it 3.7851000. To overcome this difficulty, auto-push needs to be enabled even though the two numbers are not going directly into the stack at this time. While in PRGM mode, auto-push is enabled by either toggling the PRGM switch or the ALPHA switch before entering the next number.

Line 6 of the program is entered in a manner similar to line 2. The machine is first put into ALPHA mode, the string is keyed in and ALPHA mode is exited. The keystrokes would be :

ALPHA L I T E R S b = b ALPHA

Line 7 of the program is entered by first putting the machine into ALPHA mode, pressing the VIEW key and then exiting ALPHA mode. The machine is put into ALPHA mode first so that the next key pressed will be interpreted as the ALPHA equivalent of the instruction. This same technique is used for RCL and STO making them ARCL and ASTO respectively. The chart on the back of the HP-41C/CV indicates all of the shifted and unshifted ALPHA functions. Alternately, these functions may be input by spelling out their names. The keystrokes to get the AVIEW instruction into the program are:

ALPHA shift VIEW ALPHA

The END statement may be put into the program in one of two ways. First, the END function may be executed like any other function or the GTO.. instruction may be executed. The first method will insert the END statement as the next line of the program and the program file will be exited so that the program pointer is positioned to the next available program register. However, the operating system will not pack program memory. The second method will also cause the END statement to be inserted as the last program line and the program file to be exited so that the program pointer is positioned to the next available program register. In addition, program memory will be packed. Regardless of the method you use, be sure to switch out of PRGM mode after inserting the END.

Once the program has been keyed into program memory, it should be verified line by line and any mistakes found should be corrected.

PROGRAM VERIFICATION

Program verification involves comparing every line of a program as it exists in program memory with what should have been keyed in. To begin the verification process, the first line of the program must be pointed to by the program pointer. This may be done in one of several ways. The first way involves the use of the CAT 1 function. It was mentioned earlier that the operating system maintains a catalog of all program files in program memory. As the CAT 1 function is executing, each program header and trailer is displayed beginning with the first program in program memory. If the catalog is stopped by pressing the R/S key, the program pointer will be positioned to whatever program was being displayed. If you stop the catalog at the program that you wish to verify and then switch into PRGM mode, then either the program header or trailer will be displayed, depending on how soon the catalog was stopped. If you stop the catalog too late, you may use the BST key to backup to the appropriate entry before switching to PRGM mode.

Another way to get to a particular program in program memory is by using the **GTO** instruction. When the **GTO** instruction is executed with an alpha label and the machine is not in PRGM mode, the operating system will search all of program memory for the requested label. The following sequence of keystrokes will locate the program GALLIT:

shift GTO ALPHA G A L L I T ALPHA

To look at each line in the program, you may use the **SST** key to advance one line or the **BST** key to go back one line. Occasionally, you may want to skip down or up several lines at once. The **GTO.nnn** command will do this for you. Simply execute the command putting the target line number in place of the nnn. For example, to get to line 8, the following GTO would be executed:

GTO.008

EDITING PROGRAMS

If an error is found in a program it must be corrected. Correction may take one of five forms.

1. correcting lines as they are being entered
2. correcting lines after they have already been entered
3. removing entire lines
4. removing enire programs
5. inserting new lines of code

Correcting program lines as they are being entered is probably the most common editing operation that you will perform. If a mistake is discovered while the line is being keyed in, the mistake may be corrected by repeatedly pressing the correction key until the character or command in error has been removed. Once the error is removed, the program line may be completed.

If a mistake was made but was not discovered before the line was terminated, the entire line must be removed and reentered. To do this, the program pointer is placed at the line in error and the correction key pressed to remove the line. The program pointer will now be positioned to the line immediately <u>preceeding</u> the line just removed. The line may now be reentered and it will be in the correct place.

If the first line of a program is in error, the line must first be removed and then reentered. To do this, the program pointer is placed at line 1. The correction key is then pressed to remove the line. Finally, the GTO.000 command is executed to place the program pointer to the beginning of the program and the new line is entered.

Removing several lines of code may be done in one of two ways. First, the correction key may be used repeatedly, once for each line to be removed. If this method is to be used, the program pointer should be placed at the <u>last</u> line of code to be removed. The reason for this is that when a line is removed using the correction key, the program pointer will be left pointing to the line of code which **preceded** the line removed. Another way to remove several lines of code is by using the DEL function. To use this function, the program pointer is placed at the first line of code to be removed. The **DEL** function is then executed. When the display shows **DEL_ _ _** , the number of lines of code to be removed including the current line should be entered.

Entire programs may be removed by using one of two functions. The **CLP** function will delete the program named in the ALPHA register. One or more contiguously stored programs may be removed by using the **PCLPS** (an extended function) function. To use this function, the name of the first program to be deleted is entered into the ALPHA register. When the function is executed, all programs in program memory beginning at the one named in the ALPHA register up to and including the last program in program memory will be removed.

Inserting new lines of code is done first by positioning the program pointer to the line which will immediately **precede** the new lines. The new lines of code are then keyed in. If lines are to be inserted in several locations throughout a program, it is a good idea to begin with the last set of lines (the ones towards the end of the program) first and proceed to the next to last group, and so on up to the top of the program. The reason for this is that as new lines are entered, the operating system will automatically renumber the program lines beginning with the inserted lines and going down. If lines are being inserted using a program listing as the source of line numbers, then beginning at the top and working down will not allow you to use line numbers after the first insertions are made. On the other hand, beginning at the bottom will not cause lines above the insertions to be renumbered.

After the program is edited and ready to go, it is a good idea to get a program listing. In order to do this, the printer should be set to **MAN** or **NORM** mode depending on the format desired for the listing. Executing the **PRP** function with the name of the program in the ALPHA register will list the program.

RUNNING PROGRAMS

Once the program has been entered into program memory and all obvious errors removed, it may be run (executed). The actual method of execution depends on how the particular program was designed. To run the program developed in this chapter, the program pointer must first be placed at the first line of the program. This is done with either the **CAT 1** function or the **GTO** function. After the program pointer is set, the **R/S** key is pressed to start the program. When the program stops with the "NUMBER OF GALLON" prompt in the display, enter the number of gallons to be converted and press **R/S**. The program will stop again with the "LITERS = " display.

COMPILING PROGRAMS

Compiling is a process performed by the operating system which computes and stores jump addresses within the jump instruction itself. The first time a program containing GTO or XEQ instructions is executed, the system must search for the address of the label being jumped to. Once found, the address of the label is stored in the instruction. Now, as long as the program is not modified or PACKed, the jump instructions will operate faster because the system does not have to search for the label, it simply jumps to the label location. If a program is to be stored on some external medium, the compiled version should be stored.

CONCLUSION

This chapter has presented a simplistic summary of programming the HP-41C/CV. The remainder of this book will build upon this foundation by presenting the programming of the extended functions and the HP-IL functions.

Chapter 4

Program Design Methodology

INTRODUCTION

It has been said that "computer programs are never designed, they are created on the coding pad". That this is true is certainly unfortunate. It is unfortunate for the programmer because he must spend more time than necessary getting his program written and running error-free. It is also unfortunate for the person who must later make modifications to the program because the areas to be changed may not be easily located.

This chapter will present program design and coding techniques that has been in use for several years by programmers in the larger computer shops. The use of the methods to be discussed are guaranteed to produce efficient and reliable programs and to make future modifications easier to implement.

HISTORICAL PERSPECTIVE

The basic task of a programmer is to write programs, or sets of computer instructions, which will make a computer translate input data into some predefined output. Historically, the programmer would learn the instruction set (commands) of the particular computer to be used and then set about to devise a solution to the problem to be programmed. Through the use of flowcharts (one of several program design tools available), the programmer would "figure out" what steps are necessary to solve the problem using his own rules in the construction of the logic. Little or no attention was given to how to design a "good" program, that is, a program which is both efficient in terms of computer resources and easy to modify. The resulting programs were built on a trial and error basis, coping with each new condition as it arose. If a similar problem was to be programmed at some future date, it is unlikely that the resulting program would be logically or structurally similar to the previous program. This unstructured approach to programming has led to the development of programs which are unreliable.

This approach has also led to the development of programs which are next to impossible to read and understand. If a person cannot follow the logic flow of a program, there is no possible way that the person can correct a problem with the program or make modifications to the program. Therefore, the lack of proper program design has also produced programs which are difficult to modify and maintain.

There are at least two reasons why, historically, little emphasis has been placed on the design of "good" programs. First, most of the emphasis has been placed on program testing and debugging to obtain reliable programs. The program development cycle consists of the following four phases:

analyse requirements
design the steps necessary to program the solution
write the program instructions
test the program and correct errors and problems

A very small percentage of the programmers time (about 20%) was spent in program design and a very large amount of the programmers time (about 50%) was spent in the testing and debugging of the program. It is difficult to produce good programs when so little time is spent in the design phase.

The second reason why, historically, little emphasis has been placed on program design is that there has been little or no consideration given to the person who must eventually modify the program. Except possibly for small "one-timers", it is unreasonable to say that a program will never need to be changed. Programs which are poorly designed and coded are difficult and time-consuming to understand and modify or correct. Frequent errors which must be corrected include:

incorrect processing of all data
incorrect calculations
premature program termination due to incorrect instructions or data

When one or more of these errors occur, they must be corrected immediately. If the program is written such that it is difficult to understand, then these corrections will be difficult to make.

It has been suggested that if more time were spent in the design of a program and if the other person were kept in mind during the design and coding of the program, then less time would be required to test and debug the program and the difficulty in maintaining the program would be all but eliminated.

Considerable research has been conducted aimed at changing computer programming from an art (a non-disciplined, ego-building expression of ones own ideas) to a science (a disciplined approach to problem solving). This research indicates that a method of program design and coding known as **Structured Techniques** (structured design plus structured programming) can solve many, if not all, of the problems described above. The resulting structured program is more reliable and easier to read, understand and maintain.

STRUCTURED DESIGN

Structured design is a method of program design which forces the programmer or designer to isolate each separate operation that the program is to perform. This is accomplished in four steps. The first step involves defining the problem to be solved. This step is very rarely done because it seems so obvious. The final product will greatly suffer, however, if problem definition is omitted or assumed. To carry out this step, the outputs of the program are carefully defined. In other words, "What information is the program supposed to give to the user?". After the outputs are defined, the inputs to the program must be defined -"What input does the program need in order to generate the output?". Only after the outputs and inputs have been defined, the processing which must take place in order to generate the outputs from the inputs must be defined <u>at a very high level</u>. In other words, the programmer or designer must make some very <u>general</u> statements about the processing that is to take place. Once step 1 is complete, every aspect of the problem to be solved should be thouroughly understood. If a complete understanding is not there, step 1 must be done again.

The second step in program design involves designing one or more processes to perform the translation of the inputs into the outputs. This will require thinking about <u>how</u> the program will do what it is supposed to do. After an algorithm has been developed, it must be checked by determining if it will handle all the inputs necessary and if it will generate all the outputs necessary. Also, it is a good idea to determine if it will be easy for the user to use. If not, then the algorithm must be designed more carefully.

Once an algorithm has been decided upon, the third step, that of refining the algorithm, must be done. This involves breaking the algorithm down into finer and finer parts until each part will perform exactly one function. It is at this point that the programmer or designer will decide when each process or function will be performed.

Finally, the last step may be performed. This step also is rarely done but very important. Once the algorithm is defined and each separate function determined, it is necessary to "walk thru" the logic using accurate data. This will insure that logic errors are caught before the actual coding of the program takes place. If a logic error is found, it is easier to correct it during the design stage than it is after the program has been written.

STRUCTURED PROGRAMMING

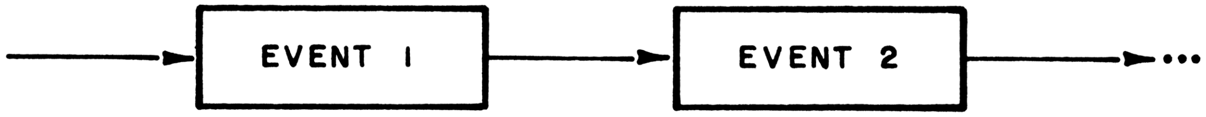
Structured programming is a method of programming which involves the design and coding of programs using a limited number of control statements to form a highly structured and easily read program.

A "proper" program has all of the following characteristics:

one and only **one entry point**
one and only **one exit point**
no infinite loops or unreachable code
uses only the SEQUENCE, IF-THEN-ELSE and DO-WHILE **control structures**
can be **read** from **top to bottom**

An **entry point** is the place in the program where control passes from the caller and execution begins. The **exit point** is the place in the program where execution terminates and control passes back to the caller. An **infinite loop** is a section of code which is executed indefinitely. **Unreachable code** is code that is never executed because control cannot pass to it.

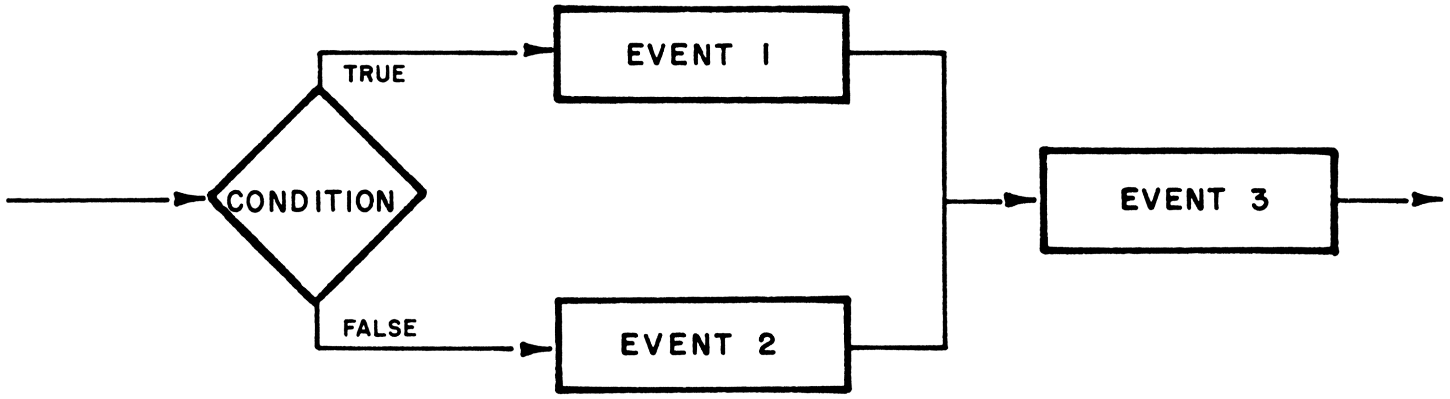
The **SEQUENCE** control structure enables the programmer to sequence actions, one action, then the next, and so on. The following block diagram will illustrate the SEQUENCE control structure.



SEQUENCE control structure

The rectangular boxes specify a particular event (instruction or instructions) which must occur. Notice that event 1 occurs before event 2 and that event 1 does not occur again once it has been completed.

The **IF-THEN-ELSE** control structure allows control in a program to branch to segments of code depending on the value of a bivalued condition. The following block diagram will illustrate the IF-THEN-ELSE control structure.

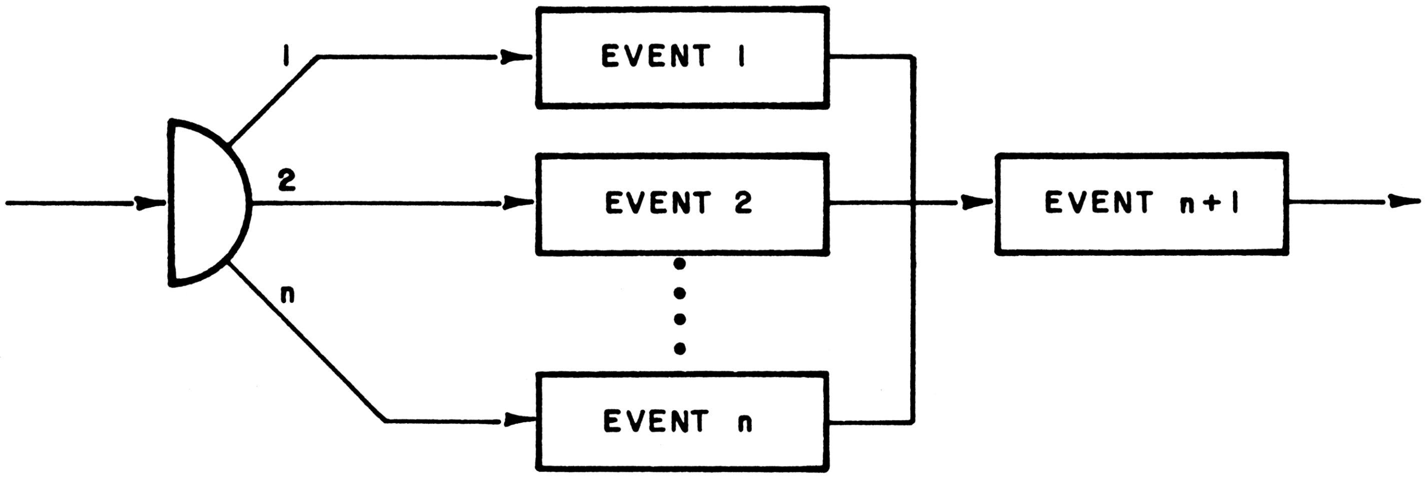


IF-THEN-ELSE control structure

The diamond shaped structure is the condition to be tested. The condition is either TRUE or FALSE. If TRUE, then event 1 is executed. If the condition is

FALSE, then event 2 is executed. Notice that event 1 and event 2 are SEQUENCE control structures and that they both will flow into the same event (event 3). Notice also that event 1 or event 2 but not both could be null operations.

If it is necessary to test a multi-valued condition, a special form of the IF-THEN-ELSE control structure called the **CASE** structure may be used. The following block diagram will illustrate the CASE structure.



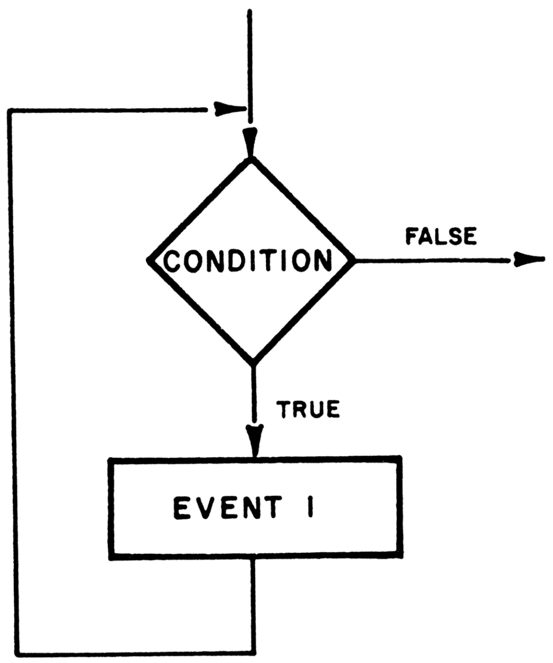
CASE control structure

The semi-circle denotes the multi-valued condition to be tested. If the value of the condition is 1 then event 1 will be executed, if the value of the condition is 2 then event 2 is executed, and so on. Again it is possible to make one or more of the events a SEQUENCE control structure or to make any of the events null events. Also, all events end at the point where the following event (event n+1) will be executed.

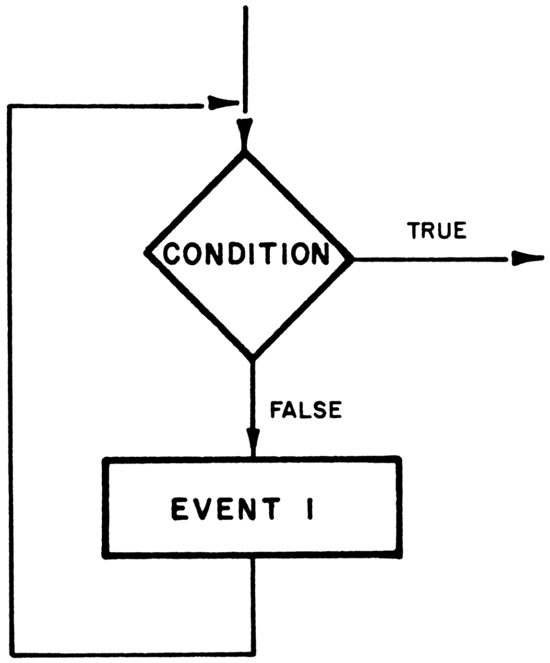
The **DO-WHILE** control structure allows a segment of code to loop (repeatedly execute) as long as the tested condition is TRUE. The block diagram on the following page will illustrate the DO-WHILE control structure.

A special form of the DO-WHILE control structure, the **DO-UNTIL** control structure, allows a program to loop until the tested condition is TRUE. The block diagram on the next page will illustrate the DO-UNTIL control structure.

Look again at the characteristics of a "proper" program. The last characteristic states that a "proper" program can be read from top to bottom. This means that the program will flow beginning at the top (at the entry point) and flow in a downward direction until the end (the exit point) is reached. The diagram on the following page illustrates the flow thru a structured program.



DO-WHILE control structure



DO-UNTIL control structure

The triange represents the execution of a DO-WHILE control structure. The diagram illustrates a basic fact of structured programs. Each segment of a structured program can be determined to be correct independently of other segments of the program. It this basic fact of structured programs that allows one to test a program one module at a time, even before all the other modules have been coded.

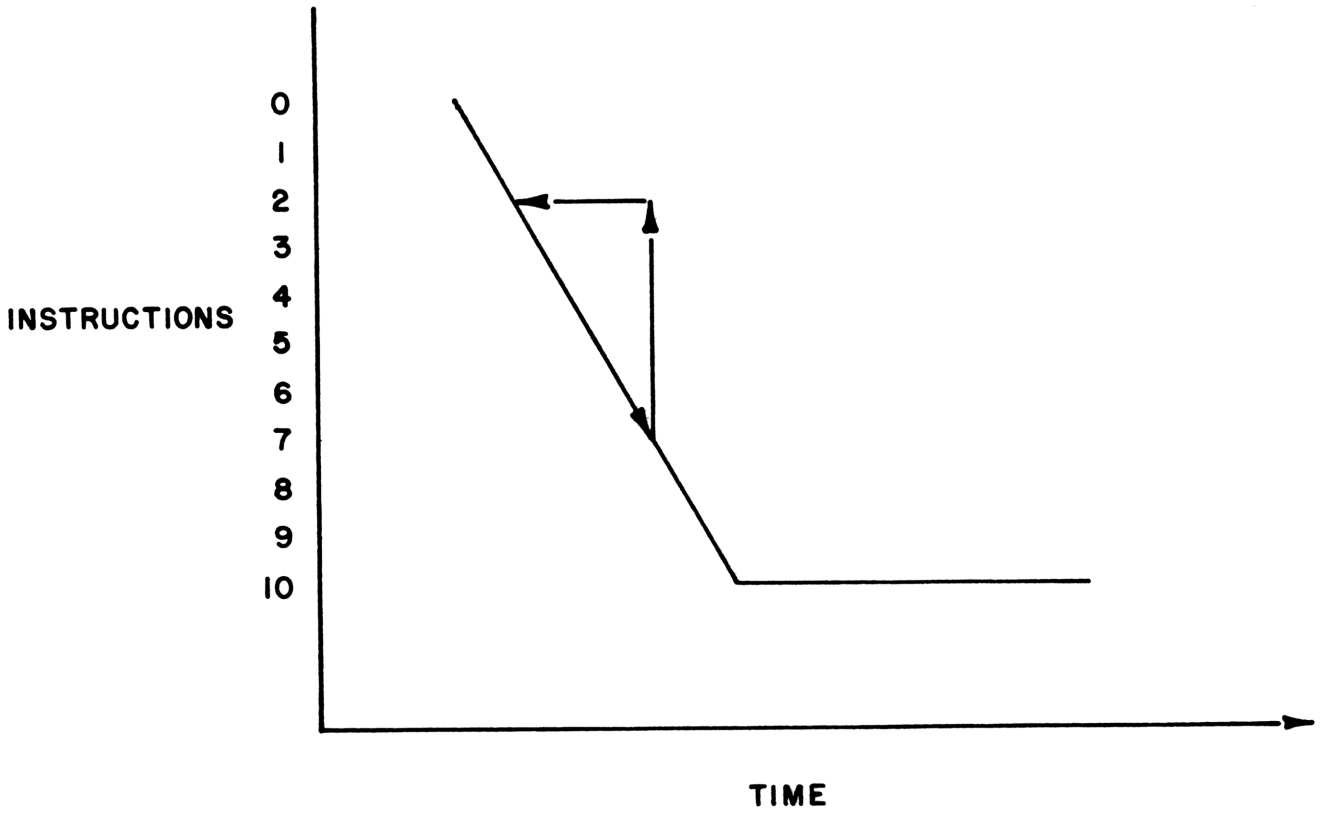


Diagram of structured program flow

CONCLUSION

A structured program is one which will be easier to read and understand. The author of the program will spend much more time designing the program and much less time testing and debugging the program and yet feel confident that a "good" program has been produced.

Chapter 5 will give a more detailed description of the program control structures and Chapter 7 will show how these control structures are implemented on the HP-41C/CV.

REFERENCES

Hearn, A.D.; "Some Words About Program Structure."; **Byte**, September 1978, pages 68 thru 76.

Hearn, A.D.; "Top-Down Modular Programming."; **Byte**, July 1978, pages 32 thru 38.

Howard, J.; "What Is Good Documentation?"; **Byte,** March 1981, pages 132 thru 150.

Weems, C.; "Designing Structured Programs."; **Byte,** August 1978, pages 143 thru 154.

Williams, G.; "Structured Programming And Structured Flowcharts."; **Byte,** March 1981, pages 20 thru 34.

Williams, G.; "Is This Really Necessary? A First Look At Design Techniques."; **Byte,** March 1981, pages 6 thru 214.

Chapter 5

Program Control

INTRODUCTION

This chapter will discuss the program control operations which allow sections of code within a program to be executed over and over again. The great power and flexability of computers rests in their ability to execute instructions and sequences repeatedly in a controlled manner.

This chapter will present the theory and operation of conditional and unconditional transfer instructions, system and user flags and the implementation of the IF-THEN-ELSE and DO-WHILE control structures. To explain this theory and to show the implementation of these control structures, the following topics will be discussed:

Unconditional transfers
Conditional transfers
User and System flags
The IF-THEN-ELSE and CASE control structures
Loops
The DO-WHILE and DO-UNTIL control structures

UNCONDITIONAL TRANSFERS

Unconditional transfers are instructions which, when invoked, cause an immediate branch or jump to the location specified within the instruction. The HP-41C/CV unconditional transfer instruction is the **GTO** command. This command may be coded in any one of several ways depending on where control is to pass.

GTO programname or labelname
where programname is the name of a program file located in program memory and labelname is an alphanumeric local or global label

GTO labelnumber
where labelnumber is a label number between 00 and 99

GTO IND nn
where nn is a main memory register number in the range 00 to 99 which contains either a label number in the range 00 to 99, an alphanumeric local or global label, or a program file name

GTO IND ST
where ST is a stack register or the LASTX register which contains either a label number in the range 00 to 99, an alphanumeric local or global label, or a program file name

The **GTO programname** form of the GTO command will cause the machine to halt and begin searching for the program file whose name is specified in the instruction. The machine will search sequentially through all of program memory until either the program file is found or the end of program memory is encountered. If the program is not found, an error message will be displayed. The program name may be specified in a main memory register or a stack register in which case the GTO IND nn or GTO IND ST forms of the command would be used.

The **GTO labelnumber** or the **GTO labelname** form of the GTO command will cause the machine to halt and begin searching downward for the label specified in the instruction. If the label is not found before the end of the program is encountered, the search begins again at the top of the program and proceeds downward until either the label is found or until the executing GTO command is encountered. If the label is still not found, the HP-41C/CV will display an error message and the program will halt with the program pointer positioned at the GTO instruction which was being executed.

The **GTO IND nn** form of the GTO command will cause the machine to halt and begin searching for the label specified in main memory register Rnn. The machine will search sequentially downward until the label is found or until the end of the program is encountered. If the label is not found before the end of the program is encountered, the search will continue from the top of the program and proceed downward until either the label is found or until the currently executing GTO command is encountered. If the label is still not found, the machine will halt and an error message will be displayed.

The **GTO IND ST** form of the GTO command will cause the machine to halt and begin searching for the label specified in the stack register or the LASTX register. The search will proceed sequentially downward until either the label is found or until the end of the program is encountered. If the label has not been found before the end of the program is encountered, the search will continue from the top of the program and proceed sequentially downward until either the label is found or until the currently executing GTO command is reached. If the label has not been found by the time the entire program has been searched, the machine will halt and display an error message.

If the label is found, the search will stop at the label and the machine will begin executing the instructions following the label. The instructions will be executed sequentially until either another transfer instruction is encountered or until the end of the program is encountered.

The GTO instruction fits into the category of a SEQUENCE control structure - almost. You will recall from Chapter 4 that the SEQUENCE control structure enables actions to be sequenced, one action after another. The SEQUENCE control structure does not allow for previous actions to be repeated - this function is left up to the DO-WHILE or the DO-UNTIL control structures. For this reason, the GTO statement should not be used in a stand alone capacity. We will see later that the GTO is necessary with some other instructions like the ISG or DSE instructions.

CONDITIONAL TRANSFERS

Unconditional transfer instructions, as powerful as they are, do not allow the programmer to control when a branch will occur. Whenever an unconditional transfer instruction is encountered, control will be transfered immediately to the label or program specified by the instruction. **Conditional transfer** instructions, on the other hand, allow complete control over when a branch will occur.

There are three classes of conditional transfer instructions available to the HP-41C/CV programmer. The first class consists of instructions which rely on a control word for their operation and includes the ISG and the DSE instructions. The **Increment and Skip if Greater** command will take one of two actions depending on the value of a control word located in either a main memory register or in a stack register or the LASTX register. The general forms of this command are:

ISG nn
where nn is a main memory register in the range 00 to 99 which contains the control word

ISG IND nn
where nn is a main memory register in the range 00 to 99 which contains the main memory register number in the range 00 to 319 which contains the control word

ISG ST
where ST is a stack register or the LASTX register which contains the control word. The possible values of ST are:
- **X** which designates the X register
- **Y** which designates the Y register
- **Z** which designates the Z register
- **T** which designates the T register
- **L** which designates the LASTX register

ISG IND ST
where ST is a stack register or the LASTX register which contains the main memory register number in the range 00 to 319 which contains the control word

The control word is of the form:

cccc.tttii
where **cccc** is the current **counter** value
ttt is the current **test** value
ii is the **increment** value

The ISG command operates as follows. When the command is encountered, the value of cccc is incremented by ii. If the value of cccc is less than or equal to the value of ttt, the next line of the program is executed. If the value of cccc is greater than the value of ttt, the next line of the program is skipped. Assume, for example, that the value of the control word is 0000.00501 and that it is stored in main memory register R00. The execution of ISG 00 would proceed in the following manner.

1. the value of ii (01) is added to cccc (0000) resulting in a new value for cccc of 0001
2. cccc is compared against tt (005)
3. if cccc is less than or equal to ttt, then the next line of the program is executed
4. if cccc is greater than ttt, then the next line of the program is skipped

After one pass through the loop, the new value of the control word located in main memory register R00 would be 0001.00501. Only when the control word reaches a value of 0006.00501 would the next line of the program be skipped. This would occur after five passes through the loop.

The HP-41C/CV will allow a minimum control word of 0.ttt to be used. If this minimum control word is used, the system will use a default value of 0 for cccc and a default value of 1 for ii. If a control word of 0.00 is used, the ISG can be used to increment the control word by a value of 1. This is useful for incrementing (adding to) a register for the purpose of counting by 1. If it is necessary to count by some other number, then the control word must be set up such that cccc and ttt are both zero and ii is the number to count by. For example, assume main memory register R00 is set to zero at the beginning of a program and a count of number of times an event occurs is required. The following commands placed at the appropriate place in the program will accomplish this task.

```
   .
   .
   .
ISG 00
LBL 00
   .
   .
   .
```

The ISG will function as previously described. The value of ii (set by default to 01) will be added to the value of cccc (initially set to 0000). The new value of cccc will be tested against the value of ttt (initially set to 000). Of course, cccc will be greater than ttt, so the next line of the program will be skipped and the program will proceed. The LBL 00 is necessary because the ISG command is actually a two line command. The first line is the ISG itself and the second line is the instruction to be executed if the value of cccc is not greater than ttt. So, the LBL 00 is a dummy line (a no-operation) to satisfy the requirements of the ISG command.

The next conditional transfer command is the **Decrement and Skip if Equal** command. The DSE command will take one of two actions depending on the value of a control word supplied in either a main memory register or a stack register or the LASTX register. The general forms of this command are:

DSE nn
where nn is a main memory register number in the range 00 to 99 which contains the control word

DSE IND nn
where nn is a main memory register number in the range 00 to 99 which contains the main memory register number in the range 00 to 319 which contains the control word

DSE ST
where ST is the stack register or the LASTX register which contains the control word. The possible values of ST are:

X which designates the X register
Y which designates the Y register
Z which designates the Z register
T which designates the T register
L which designates the LASTX register

DSE IND ST
where ST is the stack register or LASTX register which contains the

main memory register number in the range 00 to 319 which contains the control word

The control word is of the form:

cccc.tttdd

where **cccc** is the current **counter** value

ttt is the current **test** value

dd is the **decrement** value

The DSE command operates as follows. When the command is encountered, the value of cccc is decremented by dd. If the value of cccc is greater than the value of ttt, then the next line of the program is executed. If the value of cccc is less than or equal to the value of ttt, then the next line of the program will be skipped. Assume, for example, that the value of the control word is 0007.00501 and that it is stored in main memory regsiter R00. The execution of the command DSE 00 would proceed in the following manner.

1. the value of dd (01) is subtracted from the value of cccc (0007)
2. cccc is compared against ttt (006)
3. if the value of cccc is greater than the value of ttt, then the next program line is executed
4. if the value of cccc is less than or equal to the value of ttt, then the next line of the program is skipped

After one pass through the loop, the new value of the control word is 0006.00501. The second time the DSE instruction is encountered, the control word becomes 0005.00501. Since cccc is now less than or equal to ttt, the next line of the program is skipped.

The HP-41C/CV will allow a minimum control word of cccc.00000. If a control word of this form is used, the system will use a default value for ttt of 0 and a default value for dd of 01. If a control word of 0.00 is used, the DSE instruction can be used to decrement (subtract from) a register. If it is necessary to decrement by some value other than 1, then dd may be set to the necessary value.

The next class of conditional transfer instructions consists of simple relational tests. A relational test is one in which the relationship of X to 0 or X to Y is determined. These instructions allow programs to make "decisions" about which of two paths in a program to take.

There are 10 relational instructions available on the HP-41C/CV. Each instruction operates in essentially the same manner. The value in the X register is compared against either zero or a value in the Y register. If the result of this comparison is TRUE, then the next instruction is executed. If the result of the comparison is FALSE, then the next instruction is skipped. This is the "do if true" rule.

One final class of conditional transfer instructions consists of instructions which test a flag to see if it is SET (ON) or CLEAR (OFF). These instructions also allow the program to make "decisions" about which path to take in a program.

USER AND SYSTEM FLAGS

A flag is nothing more than a bit in status register **d**. Bits may assume a value of zero or one. There are 56 flags available in the system. The first 30 flags are termed "user" flags. These flags record the status of items which are of interest to the user such as whether the printer is enabled or whether a numeric item has been input. The "user" flags may be both altered (set or cleared) and tested.

The other 26 flags are termed "system" flags. System flags record the status of many of the machines conditions. These flags may not be altered by the user but they may be tested.

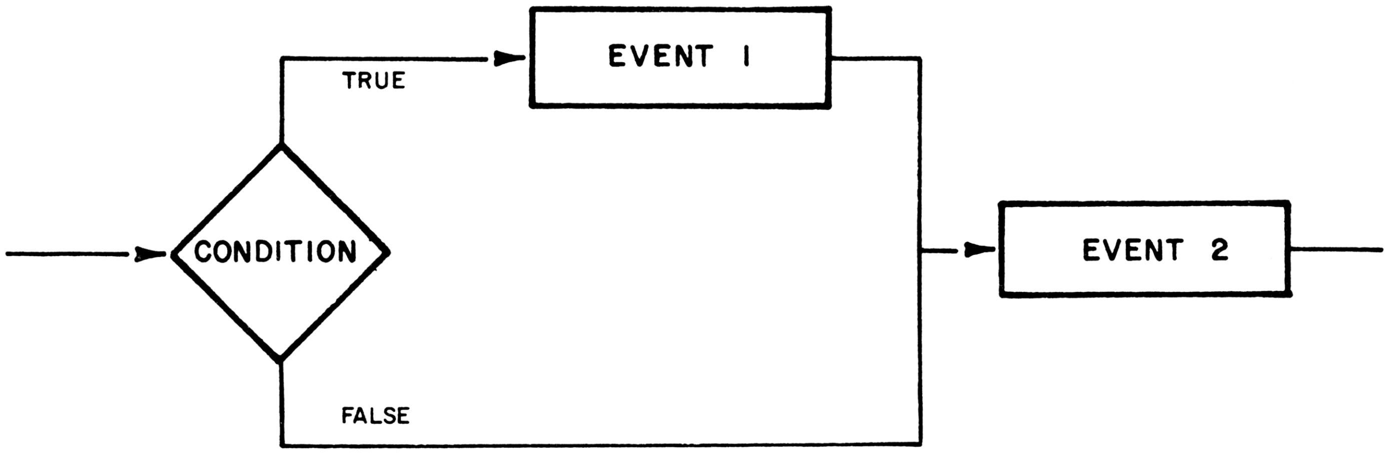
The first 11 "user" flags are termed "general purpose" flags. These flags will always be maintained by the system even if the machine is turned off. Unlike many of the "user" flags, the status of these 11 flags is controlled explicitly by the user. This means that these flags may be set if a certain condition is encountered and later tested to see if the condition occured. In addition to complete control over these 11 flags, the status of the first five flags (F00 thru F04) may be seen in the display annunciators.

THE IF-THEN-ELSE AND CASE CONTROL STRUCTURES

In Chapter 4, the IF-THEN-ELSE control structure and its variant the CASE control structure were discussed. It was mentioned that it is by means of one of these control structures that sections of code may be executed outside the normal sequence of the program.

The IF-THEN-ELSE control structure is used to determine which of two sections of code will be executed next. IF the result of the test is TRUE THEN do some sequence of instructions ELSE do another sequence of instructions. This control structure is normally implemented using either relational test instructions or flag testing instructions. When designing the code for an IF-THEN-ELSE control structure, three cases will become apparent.

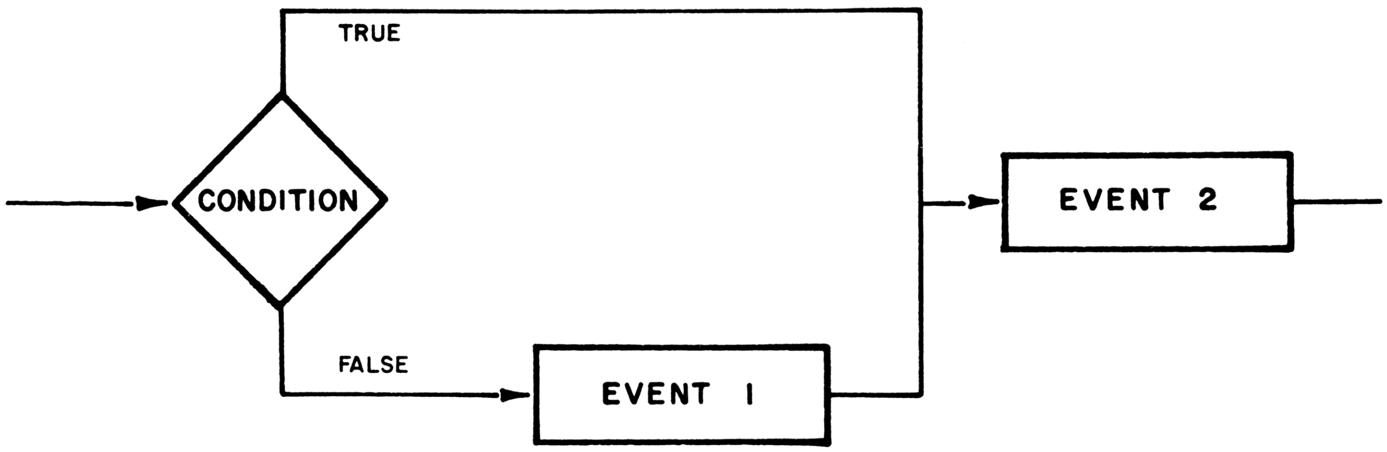
Case 1 This is the case where only TRUE logic is needed. The FALSE logic is a null sequence. This case could be pictured as follows:



Case 1 IF-THEN-ELSE control structure

Notice that event 1 is executed only if the condition is TRUE. Event 2 is executed after event 1 is complete or if the condition is FALSE.

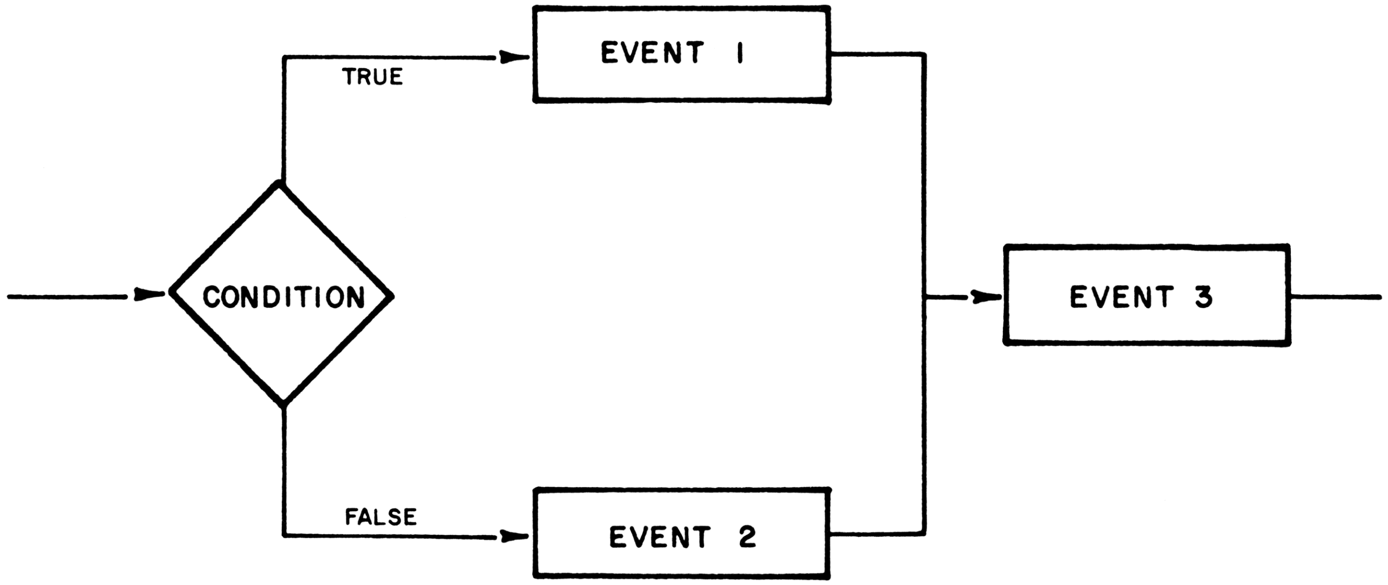
Case 2 This is the case where only FALSE logic is needed. The TRUE logic is a null sequence. This case could be pictured as follows:



Case 2 IF-THEN-ELSE control structure

Notice that event 1 is executed only if the condition is FALSE. Event 2 is executed after event 1 or if the condition is TRUE.

Case 3 This is the case where both TRUE and FALSE logic is needed. This case could be pictured as follows:



Case 3 IF-THEN-ELSE control structure

Notice that event 1 is executed if the condition is TRUE and event 2 is executed if the condition is FALSE. Event 3 is executed after the completion of event 1 or event 2.

Implementing these three cases requires different techniques for each case. The important thing to remember is that the logic that is most frequently executed should be included as soon after the test as possible. For example, if a case 1 IF-THEN-ELSE is being implemented, the logic most commonly executed should follow the test. The wrong way to implement this would be to GTO the most commonly executed logic.

In order to include the proper logic after the test, it may be necessary to reverse the test so that the "do if true" rule will not interfere. If the test is reversed so that the exact opposite of the condition is tested for, then the common logic will not have to be placed away from the test. For example, suppose it is necessary to test for the X register being zero. If X is equal to zero (the TRUE logic) then a six line sequence of instructions must be executed. If the test is FALSE, then the program should continue to flow normally. Since a six line sequence of code is necessary if the test is TRUE, the X=0? command should <u>not</u> be used because a branch to the TRUE logic would have to take place. Rather, the X≠0? instruction should be used. What would have been the TRUE logic for the X=0? test is now the FALSE logic for the X≠0? test. Do you see how test reversal will reduce the number of GTO and LBL instructions in a program and make the program more readable? The following lines of code show the right and wrong way to implement the case 1 IF-THEN-ELSE control structure example discussed above.

```
01◆LBL "RIGHT"
02 "."
03 "."
04 "."
05 X≠0?
06 GTO 01
07 "."
08 "."
09 "X=0 LOGIC"
10 "."
11 "."
12◆LBL 01
13 "."
14 "."
15 "."
16 END
```

Right way

```
01◆LBL "WRONG"
02 "."
03 "."
04 "."
05 X=0?
06 GTO 00
07 GTO 01
08◆LBL 00
09 "."
10 "."
11 "X=0 LOGIC"
12 "."
13 "."
14◆LBL 01
15 "."
16 "."
17 "."
18 END
```

Wrong way

Notice that the wrong way requires 2 GTO statements and 2 LBL statements. The right way requires one of each of these instructions. If the most frequently executed logic can be implemented as a subroutine (subroutines will be discussed in Chapter 6), then test reversal may not necessary because subroutines are invoked using a single XEQ instruction.

Implementing a case 3 IF-THEN-ELSE could be tricky depending on what the TRUE and FALSE logic is to be. Keep in mind the rule that the most frequently executed logic should immediately follow the test. If the case 3 IF-THEN-ELSE can be implemented as a combination of a case 1 and a case 2, then most of the difficulties of a case 3 may be eliminated. An example of how a case 3 IF-THEN-ELSE would be implemented follows.

```
01♦LBL "CASE 3"
02 "."
03 "."
04 "."
05 X≠0?
06 GTO 01
07 "."
08 "."
09 "X=0 LOGIC"
10 "."
11 "."
12 GTO 02
13♦LBL 01
14 "."
15 "."
16 "X≠0 LOGIC"
17 "."
18 "."
19♦LBL 02
20 "."
21 "."
22 "."
23 END
```

Case 3 IF-THEN-ELSE coding example

In all the previous examples, flag testing could have been used rather than relational testing. Both flag testing instructions and relational testing instructions are useful for implementing the IF-THEN-ELSE control structure.

The **CASE** control structure is very straight forward to implement. Implementation depends on a feature of the HP-41C/CV called indirect addressing. This control structure should be implemented using subroutines if at all possible. The reason for this is that after a subroutine is executed, control will pass back to the instruction immediately following the XEQ instruction which called the subroutine. Using the XEQ IND instruction will reduce the number of GTO instructions necessary.

Some preliminary setup work will be required before the CASE is encountered. First, the sections of code to be executed depending on the values expected must have been identified with the appropriate LBL statements. For example, if one of the values that may be encountered is 5, then the section of code to be executed if this value occurs must be labeled with LBL 05. Secondly, some location to receive the values must be agreed upon before hand. The following example shows how the CASE control structure is implemented using a GTO IND instruction. Assume that main memory register R00 will contain the values which may be 0 or 1.

```
01♦LBL "CASE"
02 "."
03 "."
04 GTO IND 00
05♦LBL 00
06 "."
07 "."
08 "00 LOGIC"
09 "."
10 "."
11 GTO 06
12♦LBL 01
13 "."
14 "."
15 "01 LOGIC"
16 "."
17 "."
18♦LBL 06
19 "."
20 "."
21 END
```

CASE control structure coding examples

LOOPS

Before proceeding on to the actual implementation of the DO-WHILE and DO-UNTIL control structures, it is necessary to stop and discuss some theory of loops. Every loop consists of four parts or steps: an initialization step, the body of the loop, the adjustment step and the test for loop exit. Each part or step is related to the other parts or steps in some way and occasionally two steps may be combined.

The **body** of the loop is the area containing the instructions which are the real work to be done by the loop. The body includes the instructions to be done repetitively. The other three steps or parts have only one purpose - to insure that the body will be performed the proper number of times.

The **adjustment step** of the loop is the set of instructions which either sets the conditions for exit, or increments or decrements a counter of the number of times the loop has already been performed. The adjustment step may also include moving information around so that it will be in the correct place for the next repetition of the loop.

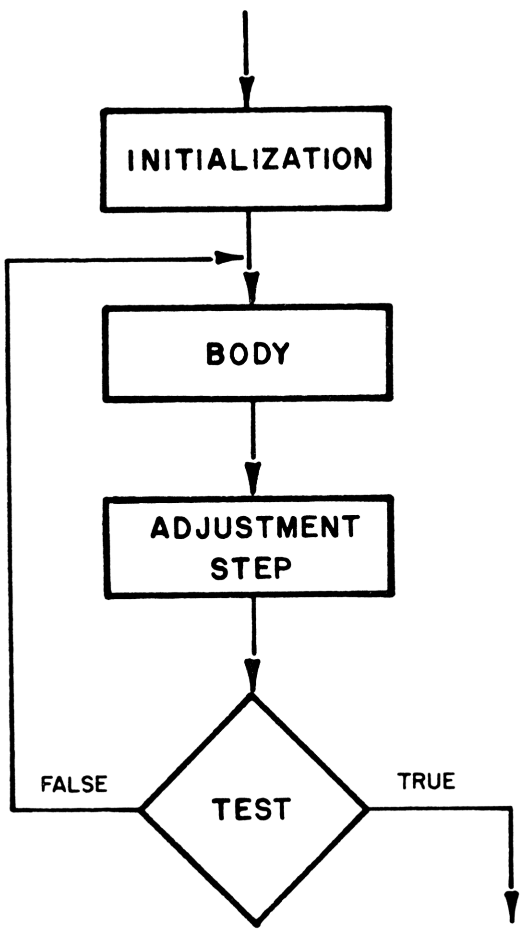
The **test for loop exit** is some sort of conditional transfer that is not always taken. If certain conditions occur then control will leave the loop, if other conditions occur then the loop is continued.

The body, adjustment step and test for exit are repeated over and over again within the loop. The **initialization step** takes place outside the loop, before execution of the loop begins. In the initialization step, all conditions necessary to the proper functioning of the loop are set. A location to be used as a counter will be initialized, a location used to address data items will be set to the address of the first item, etc.

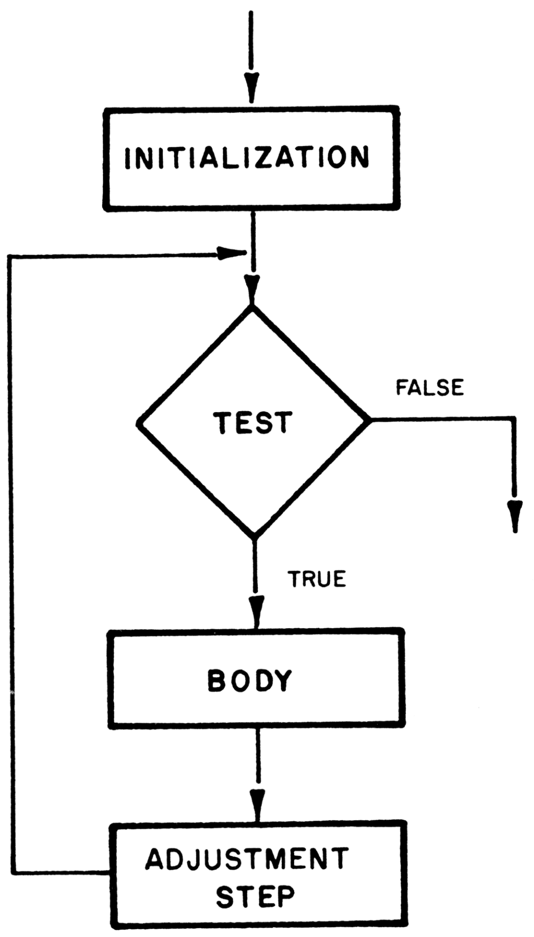
Loops are the physical implementation of the DO-WHILE and DO-UNTIL control structure. If these two control structures are drawn showing the four steps of a loop, they would appear as the diagrams on the next page illustrates.

DO-UNTIL loops are implemented on the HP-41C/CV using the DSE, ISG, flag testing instructions or relational instructions. Notice that this type of loop must be executed at least one time. For some applications, this may present a problem.

The DO-WHILE type of loop solves the problem of the loop having to be executed at least one time. However, this type of loop is not easily implemented using the ISG or DSE instructions. The type of loop used in a particular case depends on the application. The only real guideline is that the number of LBL and GTO statements used should be held to a minimum.



DO-UNTIL loop



DO-WHILE loop

THE DO-UNTIL AND DO-WHILE CONTROL STRUCTURES

The DO-UNTIL control structure is implemented using any of the conditional transfer instructions discussed. Depending on which instruction is used, the adjustment step will be different. If the DSE or ISG instruction is used to implement the loop, the adjustment step will not be a separate entity because it is built into the instruction itself. The adjustment step for the various flag testing instructions is the code which sets or clears the flag being tested. For the relational instructions, the adjustment step is the code which sets the values in the X and maybe the Y registers.

The initialization step is where the control word for the DSE or ISG instruction is built and stored or where the initial flag setting or clearing take place. Segments of code to implement the DO-UNTIL control structure follow on the next page.

In the first example, the initialization step is the CF 00 instruction, the body of the loop is identified, the adjustment step is the X=0?/SF 00 combination, and the test for exit is the FC? 00/GTO 00 combination.

In the second example, an initialization step is not necessary. The body of the loop is identified, the adjustment step is the "ENTER X"/PROMPT combination and the test for exit is the X=0?/GTO 01 combination.

In the third example, the initialization step is the 25.000/STO 00 combination, the body of the loop is identified and the adjustment step/test for exit are both incorporated into the DSE 00/GTO 00 combination.

The **DO-WHILE** control structure is implemented using any of the flag testing or relational instructions. Although the ISG or DSE instructions could be used, their implementation would increase the number of LBL and GTO instructions in the program. The actual implementation of a DO-WHILE control structure is similar to that of the DO-UNTIL structure. The segments of code on the page after next show how to implement the DO-WHILE control structure.

In the first example, the initialization step is the CF 00 instruction, the body of the loop is identified, the adjustment step is the X=0?/SF 00 combination and the test for exit is the FS? 00/GTO 00 combination.

In the second example, an initialization step is not necessary. The body of the loop is identified, the adjustment step is the "ENTER X"/PROMPT combination and the test for exit is the X=0?/GTO 01 combination.

CONCLUSION

The use of loops is a very important programming technique that will save time and memory if the loops are designed and implemented properly. It cannot be stressed too much that careful design of the code and a careful choise of the instructions used to implement loops will pay off in the long run.

```
01◆LBL "EX 1"
02 "."
03 "."
04 "."
05 CF 00
06 "."
07 "."
08 "."
09◆LBL 00
10 "."
11 "."
12 "."
13 "BODY OF LOOP"
14 "."
15 "."
16 "."
17 X=0?
18 SF 00
19 "."
20 "."
21 "."
22 FC? 00
23 GTO 00
24 "."
25 "."
26 "."
27 END
```

```
01◆LBL "EX 2"
02 "."
03 "."
04 "."
05 "."
06 "."
07 "."
08◆LBL 00
09 "."
10 "."
11 "."
12 "ENTER X"
13 PROMPT
14 X=0?
15 GTO 01
16 "."
17 "."
18 "."
19 "BODY OF LOOP"
20 "."
21 "."
22 "."
23 GTO 00
24◆LBL 01
25 "."
26 "."
27 "."
28 END
```

```
01◆LBL "EX 3"
02 "."
03 "."
04 "."
05 25.000
06 "."
07 "."
08 "."
09◆LBL 00
10 "."
11 "."
12 "."
13 "BODY OF LOOP"
14 "."
15 "."
16 "."
17 DSE 00
18 GTO 00
19 "."
20 "."
21 "."
22 END
```

Examples of DO-UNTIL coding

```
01♦LBL "EX 1"
02 "."
03 "."
04 "."
05 CF 00
06 "."
07 "."
08 "."
09♦LBL 00
10 "."
11 "."
12 "."
13 "BODY OF LOOP"
14 "."
15 "."
16 "."
17 X=0?
18 SF 00
19 "."
20 "."
21 "."
22 FS? 00
23 GTO 00
24 "."
25 "."
26 "."
27 END
```

```
01♦LBL "EX 2"
02 "."
03 "."
04 "."
05 "."
06 "."
07 "."
08♦LBL 00
09 "."
10 "."
11 "."
12 "ENTER X"
13 PROMPT
14 X≠0?
15 GTO 01
16 "."
17 "."
18 "."
19 "BODY OF LOOP"
20 "."
21 "."
22 "."
23 GTO 00
24♦LBL 01
25 "."
26 "."
27 "."
28 END
```

Examples of DO-WHILE coding

Chapter 6

Subroutines

INTRODUCTION

Among the concepts that are useful in structuring large programs, the subroutine is especially important. A program, sometimes called a routine, performs a task which may be broken down into smaller tasks, each of which is performed as a subroutine. Often a subroutine will be used, or called, more than once. Sometimes subroutines themselves will have subroutines, so that there may be several levels of subroutines. At the lowest level, there will be subroutines which perform very common functions, and which may be called by more than one subroutine at a higher level, as well as more than once in a routine.

This chapter will discuss subroutines and will present some useful theory and guidelines for the effective use of subroutines. In order to do this the following topics will be discussed:

Calling sequences
Types of subroutines
How subroutines work
Constructing subroutines

CALLING SEQUENCES

The purpose of a subroutine is to perform some task which is part of a larger task performed by the routine which calls the subroutine. In constructing subroutines, data which is in a place where the subroutine may reference it, and which may be changed each time the subroutine is called, is referred to as a **parameter.** Regardless of the number of parameters used, the subroutine must know where to find each parameter. One way to do this is to pass the address of the beginning of a parameter list to the subroutine. Access to the parameters is then via indirect addressing off the parameter list address.

Depending on the purpose of the subroutine, it may be necessary to pass information back to the caller. Information may be passed back to the caller in the form of parameters. Of course, the caller must know where to find this information. One way to pass information back to the caller, is to pass the address of a results parameter list to the caller and let the caller use indirect addressing to obtain each result.

A subroutine is called using the **XEQ** (execute) instruction. On the XEQ instruction is the name or label of the subroutine to be executed. The XEQ instruction itself along with the appropriate parameters or parameter list addresses is termed the **calling sequence.**

TYPES OF SUBROUTINES

Some coding sequences, if implemented as subroutines, would contain so few lines of code that it would be best not to treat the sequence as a subroutine at all. Instead, the sequence would be repeated when and where needed. A section of code which occurs in this way is called an **open subroutine**; an ordinary subroutine is called a **closed subroutine.** The decision as to whether to treat a given task as an open or closed subroutine depends on several factors the most important of which is the size (in bytes) of the subroutine. In general, the larger the subroutine, the more space is saved if it is made a closed subroutine.

The reason for this is best explained by an example. Suppose it is necessary to perform a 3-byte sequence of code 10 times in a program. Should this code be constructed as an open or as a closed subroutine? To answer this question, the following analysis must be performed. If coded as open subroutines, then a total of 30 bytes of code would be required. On the other hand, if coded as a closed subroutine, a total of 35 bytes would be required. The body of the subroutine would require 3 bytes, the header and trailer would require a total of 2 bytes, and the ten XEQ statements required to invoke the subroutine would require a total of 30 bytes. So you see, the closed subroutine in this example would actually require 5 bytes more than the open subroutines. As a matter of fact, this or any 3-byte sequence should never be coded as a closed subroutine no matter how many times it is to be called.

Richard Nelson of the **PPC**[1] has developed a formula which calculates the savings (or loss) in bytes realized by coding a sequence as a closed subroutine. In the formula, if S is positive, a savings of S bytes will occur by coding the sequence as a closed subroutine. If S is negative, a loss of S bytes will occur if the sequence is coded as a closed subroutine.

S = RC - 2 - R - 3C

where **S** is the number of bytes saved or lost

R is the number of bytes in the body of the subroutine

C is the number of times the subroutine is to be called

A similar formula has been developed for determining the savings (or loss) in bytes realized by coding a sequence as a closed subroutine and then calling it indirectly with the XEQ IND instruction. The XEQ IND instruction itself requires 2 bytes and the register used to contain the address requires 7 bytes. Obviously, a sequence of code would have to be fairly large to warrent it being coded as an indirectly executed closed subroutine.

S = RC - 9 - R - 2C

where **S**, **R** and **C** mean the same as in the other formula

HOW SUBROUTINES WORK

In many ways,. the XEQ instruction is similar to the GTO instruction. Both instructions are unconditional transfer instructions which means that control is passed immediately to the label or program specified in the instruction. The difference lies in the fact that the XEQ instruction has a built-in mechanism which allows control to pass back to the instruction following the XEQ instruction (the next sequential instruction or NSI) after the subroutine has completed its task.

This mechanism is called the subroutine return stack. The return stack is a block of six registers located in status registers **b** and **a**. Whenever an XEQ instruction is encountered, the address of the NSI is pushed onto the return stack. The address of the label or program specified in the XEQ instruction is placed in the address pointer and control passes to that address. When an RTN or END instruction is encountered, the address most recently pushed onto the return stack is popped and placed in the address pointer. Control then passes to this new address.

The return stack is large enough to accomodate six return addresses. What this means is that subroutines may be nested (a subroutine or routine calls a subroutine which calls another subroutine and so on) up to six levels deep. If a seventh call is made, the address of the first NSI stored will be lost.

It is important to let the system follow the calling sequence back to the caller by means of the return stack. If control is returned to a calling routine or subroutine using a conditional or unconditional transfer instruction, the integrity of the return stack will be lost because the next NSI on the stack will no longer be the NSI to be used next. Later, when a normal return is to be made, those NSI's popped will belong to a different calling sequence.

CONSTRUCTING SUBROUTINES

Subroutines, whether open or closed, are just sequences of code. Open subroutines are placed at the location within the routine or subroutine where they are needed. Closed subroutines, on the other hand, are placed either at the end of the routine or subroutine but still within the calling program file (internal closed subroutines) or they may be constructed such that each subroutine occupies its own program file (external closed subroutines). Internal and external closed subroutines consist of three parts:

the subroutine **header**
the **body** of the subroutine
the subroutine **trailer**

The subroutine **header** is the statement used to name the subroutine. This statement takes the form of a LBL statement with either a label number or a name on it. If an internal closed subroutine is coded, then the LBL statement must contain either a label number between 00 and 99 or a local alpha label (A thru F or a thru e). If an external closed subroutine is coded, then the LBL statement must contain a global alpha label of up to seven characters.

When an XEQ instruction containing a label number or a local alpha label is encountered in a running program, the program stops and the operating system starts searching the current program file for the label. If the label is not found before the end of the program file is reached, then the search continues from the top of the program file and continues until either the label is found or the XEQ instruction which called the subroutine is encountered. If the label is not found after one complete pass through the program file is made, the program aborts and the system issues an error message. If the label is found, execution begins at the label.

When an XEQ instruction with a global alpha label is encountered in a running program, the program stops and the system searches all of program memory for the subroutine starting at the last program file in program memory. If the subroutine is not found, the program aborts and the system issues an error message. If the subroutine is found, execution begins at the subroutine.

The subroutine **trailer** is the terminating statement of a subroutine. Depending on where the subroutine is coded, the trailer will be either an RTN or an END statement. If an internal closed subroutine is coded and it is not the last section of code within the file, the subroutine must be terminated with an RTN statement. If the internal closed subroutine is the last section of code within the program file, then it may be terminated with an RTN statement or the program file END statement may serve as the terminator. If an external closed subroutine is coded, it may be terminated with either an RTN statment or the program file END statement. Both the RTN and the END statement serve the purpose of telling the operating system that the end of a subroutine has been encountered. The operating system will pop the next NSI off the return stack, load this address into the address pointer and begin execution again at the new address.

The **body** of the subroutine is the sequence of instructions to be executed. Subroutines should be designed and coded with the same amount of care as any program. The subroutine should not use any but the three basic control structures discussed in Chapter 4. In addition, a subroutine should perform one and only one function -that is the subroutine should be **functionally cohesive.**

If a subroutine is to be modified at some later time, then it is easy to determine where the changes must go if the subroutine is functionally cohesive. Also, the change will be a proper change for the function being performed by the subroutine. The changes made to functionally cohesive subroutines are guarenteed to not affect other routines or subroutines improperly.

One good way to determine if a subroutine is functionally cohesive is to write a statement of what the subroutine does. If this statement turns out to be a compound sentence then the subroutine is not functionally cohesive and should be redesigned.

Another point which should be kept in mind when designing and coding subroutines is that the name should be as descriptive as possible within the bounds of a seven character name. Although seven characters is not many, it should be enough to at least logically abbreviate the function performed by the subroutine. For example, if a subroutine is to calculate the monthly payment for a loan, then give the subroutine the name CALCPMT or some similar name. Inappropriate names would be something like CP or CALC because these names do not indicate the function of the subroutine.

CONCLUSION

This chapter has examined subroutines as a method of simplifying programs. Properly used, subroutines will allow programs to be structured and easily read.

NOTES

[1] PPC
The Personal Programming Center founded by Richard Nelson
For a free issue of the PPC Calculator Journal send a 9" by 12" self-addressed envelope with 2 ounces of postage affixed to

PPC
2545 W. Camden Place
Santa Ana, California 92704

Chapter 7

Simple Ordinary Annuities Program Example

INTRODUCTION

This chapter will discuss the design and implementation of a program which will calculate any of the five unknowns related to simple ordinary annuities. The topics to be discussed are:

Simple annuities
Simple ordinary annuity variables to be calculated
Annuities program design
Techniques used
Annuity routine and subroutines

SIMPLE ANNUITIES

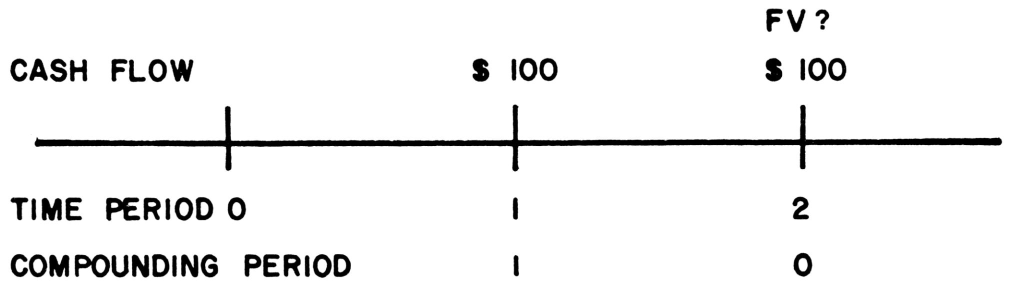
An **annuity** is the term used for a series of equal payments - one each period for a given length of time. The mortgage on your house is an example of an annuity. A **simple annuity** occurs when the interest compounding period is the same as the payment period. In simple annuities, it is assumed that compounding occurs at the end of each period. **Simple ordinary annuities** are simple annuities in which payments are made at the end of each period.

Five variables are involved in the calculation of simple ordinary annuities.

$\mathbf{i_c}$	the interest rate per compounding period
n	the number of payment periods
PMT	the payment per payment period
FV	the future value of the simple ordinary annuity
PV	the present value of the simple ordinary annuity

FV

The **future value** of a simple ordinary annuity is its value at some point in the future. For example, suppose that $100.00 is invested in a money market fund at the end of each year for two years. Assume also that the interest paid on the investment is 10% annually. What is the future value of the account at the end of the second year? This problem may be diagrammed as follows on a time diagram.



Time diagram

The calculation of the future value would procede as follows using the formula given below.

$$FV = PMT * S(n,i_c)$$

where **FV** is the future value

PMT is the constant payment for each period

$S(n,i_c)$ is the sinking fund factor

The sinking fund factor is calculated using the following formula. This factor is so useful that it will be used throughout the remainder of this chapter.

$$S(n,i_c) = \frac{(1 + i_c)^n - 1}{i_c}$$

where i_c is the compound interest rate per compounding period

n is the number of payment periods calculated by multiplying the elapsed time for the annuity (in years) by the number of payments per year

To find the FV for the example being discussed, simply plug the known values into the equation.

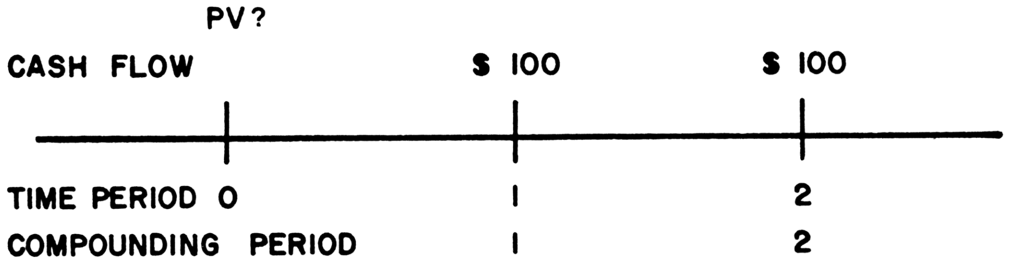
$$i_c = 0.1$$
$$n = 2$$
$$PMT = 100.00$$

$$S(2,0.1) = \frac{(1 + 0.1)^2 - 1}{0.1}$$

$$S(2,0.1) = 2.10$$
$$FV = 100.00 * 2.10$$
$$\mathbf{FV = 210.00}$$

PV

The **present value** of a simple ordinary annuity is the value of the annuity at the beginning of time period 1. Using the example from the previous section, the PV of the money market fund may be diagrammed as follows.



Time diagram

To find the PV of the account, the following formula is used.

$$\mathbf{PV = PMT * A(n,i_c)}$$

where **PV** is the present value of the annuity

PMT is the payment made at the beginning of each period

$A(n,i_c)$ is the present value factor

The present value factor, like the sinking fund factor is an important factor that will be used throughout the remainder of this chapter. The formula to calculate the present value factor is:

$$A(n,i_c) = \frac{1 - (1 + i_c)^{-n}}{i_c}$$

where **i_c** is the compound interest rate per compounding period

n is the number of payment periods calculated by multiplying the elapsed time for the annuity (in years) by the number of payments per year

The PV may be found for this example by plugging the known amounts into the formula.

$$i_c = 0.1$$
$$n = 2$$
$$PMT = 100.00$$

$$A(2,0.1) = \frac{1 - (1 + 0.1)^{-2}}{0.1}$$

$$A(2,0.1) = 1.74$$

$$PV = 100.00 * 1.74$$
$$PV = 173.55$$

PV/FV EQUIVALENCY

If either PV or FV is known, then the other may be calculated using one of the following relationships:

$$FV = PV * (1 + i_c)^n$$
$$PV = FV * (1 + i_c)^{-n}$$

PMT

The equal **periodic payment** for a simple ordinary annuity is easily calculated if either PV or FV is known. The equations below are used to determine PMT.

$$PMT = FV * S(n,i_c)^{-1}$$
$$PMT = PV * A(n,i_c)^{-1}$$

n AND FINAL PAYMENT

If either PV or FV is known, then **n** may be calculated using the appropriate formula:

$$n = \frac{\ln(FV * (i_c/PMT) + 1)}{\ln(1 + i_c)}$$

$$n = \frac{-\ln(1 - PV * (i_c/PMT))}{\ln(1 + i_c)}$$

In most cases, n will be a real number of the form M.X. The integer part, M, will be the number of whole payments. The **final payment** of a simple ordinary annuity is usually less than the other payments and is calculated differently depending on whether PV is known or FV is known.

Case 1 **PV known**

1. determine M.X
2. compute the PV of M payments
3. subtract the PV of M payments from the original PV to determine the final payment
4. compound the present value of the final payment forward M+1 periods to determine the last payment amount

This procedure may be condensed into the following formula:

$$\textbf{Final payment} = (PV - PMT * A(M,i_c)) * (1 + i_c)^{M+1}$$

Case 2 **FV known**

1. calculate M.X
2. calculate the FV at the end of M payments
3. compound the FV at the end of M payments forward one period
4. subtract the amount calculated in step 3 above from the original FV to obtain the final payment

This procedure may be condensed into the following formula:

$$\textbf{Final payment} = FV - ((PMT * S(M,i_c)) * (1 + i_c))$$

i_c

The procedure for calculating the **compound interest** rate is an iterative one. It is not possible to explicitly solve for i_c. Therefore, i_c can only be determined by trial and error. The method used to solve for i_c differs depending on whether PV or FV is known.

Case 1 **FV known**

1. obtain an initial guess for i_c and call it i_0

$$i_0 = (FV / (PMT * n^2)) - (PMT / FV)$$

2. calculate the increment to be subtracted from the initial and later estimates and call it di_k

$$di_k = \frac{(1 + i_k)^{n-1} - ((FV * i_k) / PMT)}{n (1 + i_k)^{n-1} + ((1 - (1 + i_k)^n / i_k)}$$

3. obtain an improved estimate i_{k+1}

$$i_{k+1} = i_k - di_k$$

This process continues until an answer accurate to within some predefined limit is reached. It has been found that 5 iterations of steps 2 and 3 above will result in an answer accurate to 5 decimal places.

Case 2 **PV known**

1. obtain an initial guess for i_c and call it i_0

$$i_0 = (PMT / PV) - (PV / (PMT * n^2))$$

2. calculate the increment to be subtracted from the initial and later estimates and call it di_k

$$di_k = \frac{1 - (1 + i_k)^{-n} - (PV / PMT) * i_k}{n(1 + i_k)^{-n-1} + ((1 + i_k)^{-n} - 1 \ / i_k)}$$

3. obtain an improved estimate i_{k+1}

$$i_{k+1} = i_k - di_k$$

This process continues until an answer accurate to within some predefined limit is reached. It has been found that 5 iterations of steps 2 and 3 above will result in an answer accurate to 5 decimal places.

ANNUITIES PROGRAM DESIGN

Armed with the preceeding background material, we want to design a program which will solve for any of the 5 variables associated with simple ordinary annuities. The program should allow the user to input only what he knows and to solve for the other.

There are 8 values which must be solved - n, i_c, PMT, PV, FV, $S(n,i_c)$, $A(n,i_c)$ and final payment. Since calculating these values requires no interaction with the other values, the use of subroutines would be called for. Thus, in addition to the mainline routine, 8 external closed subroutine program files will be required. These are listed below along with a brief description of each.

ANNUITY	mainline routine which initializes main memory registers, sets up key assignments, and performs the interchangeable solutions logic
CALC-I	subroutine to solve for i_c
CALC-N	subroutine to solve for n
CALC-PV	subroutine to solve for PV
CALC-PY	subroutine to solve for PMT
CALC-FV	subroutine to solve for FV
SINK	subroutine to calculate the sinking fund factor $S(n,i_c)$
PRS-VAL	subroutine to calculate the present value factor $A(n,i_c)$
FIN-PMT	subroutine to calculate the final payment

In addition to the subroutines listed above, the mainline routine contains 5 internal closed subroutines. These subroutines - N, INT, PV, PMT and FV - are associated with the mainline routine and are made closed subroutines only because each subroutine must be assigned to a key.

A good way to visualize the relationship between routines and subroutines within a complex program is to draw a hierarchy diagram of the overall system.

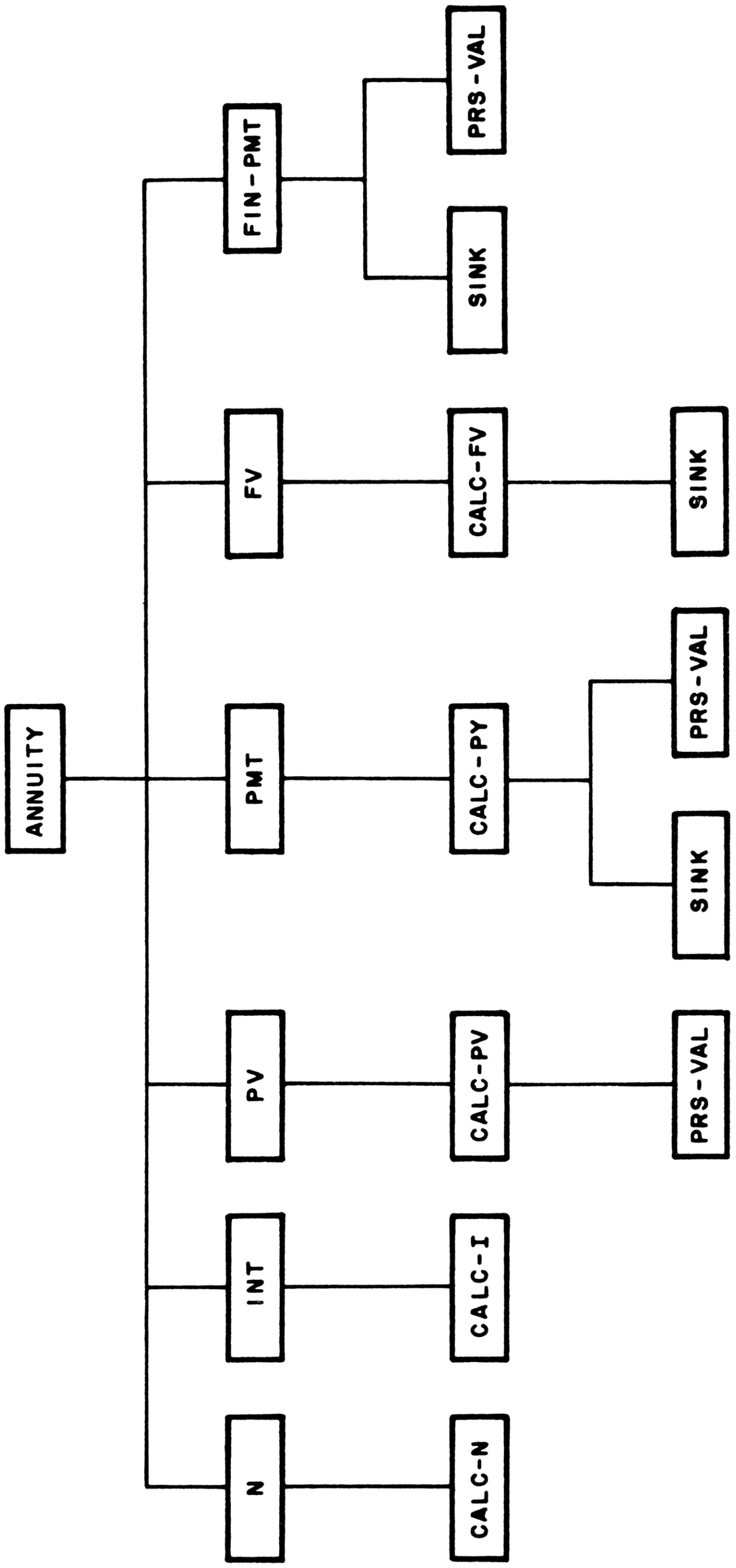
The vertical placement of items within such a diagram represents the relationship between the routines and subroutines.

The diagram, shown on the next page, illustrates that subroutines N, INT, PV, PMT, FV and FIN-PMT are called from the mainline routine ANNUITY. Subroutine CALC-N is called from subroutine N, subroutine CALC-I is called from subroutine INT, subroutine CALC-PV is called from subroutine PV, subroutine CALC-PY is called from subroutine PMT and subroutine CALC-FV is called from subroutine FV. Subroutine SINK is called from subroutines CALC-PY, CALC-FV and FIN-PMT. Subroutine PRS-VAL is called from subroutines CALC-PV, CALC-PY and FIN-PMT. Do you see how such a diagram helps you to visualize the processing which must occur within a program?

TECHNIQUES USED

The ANNUITY program uses a technique called **interchangeable solutions,** which allows a program to be written which will solve for any term of a multivalued function. The trick is to assign each term of the function to a separate key and then to assign to the appropriate key the logic which will determine whether the user wants to input a value for that term or solve for the term. The logic assigned to each key is generalized as follows:

1. check digit entry flag, F22, to determine if it is SET (ie if a digit key has been pressed) or CLEAR (ie no digit key has been pressed)
2. if F22 is CLEAR, then the user has indicated by pressing the key without first pressing a digit key that he wants to solve for the term associated with the key
3. if F22 is SET, then it is assumed that the user has entered a value for the term thereby indicating that the term is not to be solved for
4. the entered or calculated value for the term is stored into the appropriate main memory register
5. the ALPHA register is formated with the name and value of the term which is then printed



Hierarchy diagram for ANNUITY system

Assume, for example, that the monthly payment required for a \$50,000.00 15% mortgage is to be calculated. The operation of the ANNUITY program would proceed in the following manner. The program is started by pressing R/S after positioning the program pointer to the first line of the program or by assigning the program ANNUITY to an unused key (say key -11) and then pressing that key while the HP-41C/CV is in USER mode. When the program stops, the following key assignments will have been made:

N will be assigned to key 11
INT will be assigned to key 12
PV will be assigned to key 13
PMT will be assigned to key 14
FV will be assigned to key 15
FIN-PMT will be assigned to key -15

Now, the known amounts may be input in any order. The value for n of 360 is keyed in and the "N" key pressed. The keying in of the number 360 (30 years X 12 months per year) causes F22 to be SET. The testing of this flag results in a FALSE condition for the test in line 31 of the ANNUITY program. The value 360 is stored into R00. The ALPHA register is formatted and printed so that the user will know the value of n used in this case.

The value for i_c of 0.0125 (15%/100/12 months) is keyed in and the "INT" key pressed. Again, F22 is SET. The value for i_c is simply stored into R01. The ALPHA register is formatted and printed to give the user a permanent record of the value of i_c used in this case.

The value for PV of 50,000.00 is keyed in and the "PV" key pressed. Again F22 is set by the digit entry and the value is stored into R02. The ALPHA register is formatted and printed to give a permanent record of the value used.

The "PMT" key is now pressed without pressing any other digit keys first. This causes F22 to remain CLEAR and the test in line 58 to be TRUE. Consequently, the subroutine CALC-PY is executed to solve for PMT and the calculated payment is stored into R04. The ALPHA register is formatted and printed to give a permanent record of the calculated payment for this case.

ANNUITY ROUTINE

ANNUITY is the mainline routine or driver for the ANNUITY system. Within ANNUITY, the following processing takes place. The program name and version is printed, user flag F27 is set which turns on USER mode, all allocated main memory registers are cleared and all key assignments are cleared. The necessary key assignments are then made and the user is prompted to supply the known values.

INTERNAL SUBROUTINE N

N is the first internal closed subroutine in ANNUITY and is assigned to key 11. Within this subroutine, the following processing takes place. User flag F22 (the digit entry flag) is tested to see if it is SET or CLEAR. If SET, then the value entered by the user prior to pressing key 11 is stored into main memory register R00. If F22 is CLEAR, then the subroutine CALC-N is called to calculate N. The calculated value of N is then stored into main memory register R00. The ALPHA register is formatted to display the value of N.

INTERNAL SUBROUTINE INT

INT is the second internal closed subroutine in ANNUITY and is assigned to key 12. Within this subroutine, the following processing takes place. User flag F22 (the digit entry flag) is tested to see if it is SET or CLEAR. If SET, then the value entered by the user prior to pressing key 12 is stored into main memory register R01. If F22 is CLEAR, then the subroutine CALC-I is called to calculate INT. The calculated value of INT is then stored into main memory register R01. The ALPHA register is formatted to display the value of INT.

INTERNAL SUBROUTINE PV

PV is the third internal closed subroutine in ANNUITY and is assigned to key 13. Within this subroutine, the following processing takes place. Main memory register R03 is cleared. User flag F22 (the digit entry flag) is tested to see if it is SET or CLEAR. If SET, then the value entered by the user prior to pressing key 13 is stored into main memory register R02. If F22 is CLEAR, then the subroutine CALC-PV is called to calculate PV. The calculated value of PV is then stored into main memory register R02. The ALPHA register is formatted to display the value of PV.

INTERNAL SUBROUTINE PMT

PMT is the fourth internal closed subroutine in ANNUITY and is assigned to key 14. Within this subroutine, the following processing takes place. User flag F22 (the digit entry flag) is tested to see if it is SET or CLEAR. If SET, then the value entered by the user prior to pressing key 14 is stored into main memory register R04. If F22 is CLEAR, then the subroutine CALC-PY is called to calculate PMT. The calculated value of PMT is then stored into main memory register R04. The ALPHA register is formatted to display the value of PMT.

INTERNAL SUBROUTINE FV

FV is the fifth and final internal closed subroutine in ANNUITY and is assigned to key 15. Within this subroutine, the following processing takes place. Main memory register R02 is cleared. User flag F22 (the digit entry flag) is tested to see if it is SET or CLEAR. If SET, then the value entered by the user prior to pressing key 15 is stored into main memory register R03. If F22 is CLEAR, then the subroutine CALC-FV is called to calculate FV. The calculated value of FV is then stored into main memory register R03. The ALPHA register is formatted to display the value of FV.

CALC-N SUBROUTINE

CALC-N is a subroutine called by the internal closed subroutine N. Within CALC-N, the following processing takes place. The subroutines function is printed so that the user is kept informed of what is happening in the program. Main memory register R03 (the register which contains a given or calculated value for FV) is tested to determine if FV was supplied by the user. If R03 is zero meaning that the user did not supply a value for FV, then the section of code which will

```
6:46AM 11/29
01♦LBL "ANNUITY"
02 "ORD. ANNUITIES"
03 PRA
04 "VERSION 1.0"
05 PRA
06 SF 27
07 CLRG
08 CLKEYS
09 "N"
10 11
11 PASN
12 "INT"
13 12
14 PASN
15 "PV"
16 13
17 PASN
18 "PMT"
19 14
20 PASN
21 "FV"
22 15
23 PASN
24 "FIN-PMT"
25 -15
26 PASN
27 "SUPPLY KNOWNS"
28 PRA
29 RTN
30♦LBL "N"
31 FC?C 22
32 XEQ "CALC-N"
33 STO 00
34 "N = "
35 ARCL 00
36 PRA
37 RTN
38♦LBL "INT"
39 FC?C 22
40 XEQ "CALC-I"
41 STO 01
42 "I = "
43 ARCL 01
44 PRA
45 RTN
46♦LBL "PV"
47 0
48 STO 03
49 X<>Y
50 FC?C 22
51 XEQ "CALC-PV"
52 STO 02
53 "PV = "
54 ARCL 02
55 PRA
56 RTN
57♦LBL "PMT"
58 FC?C 22
59 XEQ "CALC-PY"
60 STO 04
61 "PMT = "
62 ARCL 04
63 PRA
64 RTN
65♦LBL "FV"
66 0
67 STO 02
68 X<>Y
69 FC?C 22
70 XEQ "CALC-FV"
71 STO 03
72 "FV = "
73 ARCL 03
74 PRA
75 END
```

Listing of ANNUITY mainline

compute N given PV is entered. If R03 is not zero, then the user supplied a value for FV and the section of code which will compute N given FV is entered.

```
01♦LBL "CALC-N"
02 "CALC N"
03 PRA
04 RCL 03
05 X=0?
06 GTO 10
07 RCL 01
08 RCL 04
09 /
10 RCL 03
11 *
12 1
13 +
14 LN
15 RCL 01
16 1
17 +
18 LN
19 /
20 GTO 19
21♦LBL 10
22 STO 01
23 RCL 04
24 /
25 RCL 02
26 *
27 1
28 X<>Y
29 -
30 CHS
31 RCL 01
32 1
33 +
34 /
35♦LBL 19
36 END
```

Listing of CALC-N subroutine

CALC-I SUBROUTINE

CALC-I is a subroutine called by the internal closed subroutine INT. Within CALC-I the following processing takes place. The subroutines function is printed so the user will be kept informed of what processing is taking place. The control word for the loop is put into main memory register R07. Main memory register R03 is tested to determine if the user inputted a value for FV. If R03 is equal to zero, then it is assumed that i_c is to be calculated using PV and the appropriate section of code is entered. If R03 is not zero, then the section of code to calculate i_c using FV is entered. In either case, an initial guess is made and then a loop is executed 5 times to refine this estimate.

```
01♦LBL "CALC-I"
02 "CALC I"
03 PRA
04 5
05 STO 07
06 RCL 03
07 X=0?
08 GTO 25
09 RCL 04
10 RCL 00
11 X↑2
12 *
13 /
14 RCL 04
15 RCL 03
16 /
17 -
18♦LBL 20
19 STO 01
20 1
21 RCL 01
22 +
23 RCL 00
24 Y↑X
25 STO 09
26 1
27 -
28 RCL 03
29 RCL 01
30 *
31 RCL 04
32 /
33 -
34 STO 08
35 RCL 01
36 1
37 +
38 RCL 00
39 1
40 -
41 Y↑X
42 RCL 00
43 *
44 1
45 RCL 09
46 -
47 RCL 01
48 /
49 +
50 RCL 08
51 X<>Y
52 /
53 RCL 01
54 X<>Y
55 -
56 DSE 07
57 GTO 20
58 GTO 29
59♦LBL 25
60 RCL 04
61 RCL 02
62 /
63 RCL 02
64 RCL 04
65 RCL 00
66 X↑2
67 *
68 /
69 -
70♦LBL 26
71 STO 01
72 1
73 +
74 RCL 00
75 CHS
76 Y↑X
77 STO 09
78 1
79 -
80 RCL 01
81 /
82 RCL 01
83 1
84 +
85 RCL 00
86 CHS
87 1
88 -
89 Y↑X
90 RCL 00
91 *
92 +
93 STO 08
94 1
95 RCL 09
96 -
97 RCL 02
98 RCL 04
99 /
100 RCL 01
101 *
102 -
103 RCL 08
104 /
105 RCL 01
106 X<>Y
107 -
108 DSE 07
109 GTO 26
110♦LBL 29
111 END
```

Listing of CALC-I subroutine

CALC-PV SUBROUTINE

CALC-PV is a subroutine called by the internal closed subroutine PV. Within CALC-PV, the following processing takes place. The subroutines function is printed so that the user is kept informed of the processing taking place. The subroutine to calculate the present value factor is then called. Finally, the present value factor is multiplied by the payment amount.

```
01♦LBL "CALC-PV"
02 "CALC PV"
03 PRA
04 XEQ "PRS-VAL"
05 RCL 04
06 *
07 END
```

Listing of CALC-PV subroutine

CALC-FV SUBROUTINE

CALC-FV is a subroutine called by the internal closed subroutine FV. Within CALC-FV, the following processing takes place. The subroutines function is printed so that the user is kept informed of the processing taking place. The subroutine to calculate the sinking fund factor is then called. Finally, the sinking fund factor is multiplied by the payment amount.

```
01♦LBL "CALC-FV"
02 "CALC FV"
03 PRA
04 XEQ "SINK"
05 RCL 04
06 *
07 END
```

Listing of CALC-FV subroutine

CALC-PY SUBROUTINE

CALC-PY is a subroutine called by the internal closed subroutine PMT. Within CALC-PY, the following processing takes place. The subroutines function is printed so that the user is kept informed of the processing taking place. Next, main memory register R03 is tested to determine if the user has supplied a value for FV. If R03 is zero, then it is assumed that the payment is to be calculated using PV. The reciprocal of the present value factor is calculated and then multiplied by the value of PV. If R03 is not zero, then the reciprocal of the sinking fund factor is calculated and then multiplied by the value for FV.

```
01♦LBL "CALC-PY"          10 *
02 "CALC PMT"             11 GTO 39
03 PRA                    12♦LBL 30
04 RCL 03                 13 XEQ "PRS-VAL"
05 X=0?                   14 1/X
06 GTO 30                 15 RCL 02
07 XEQ "SINK"             16 *
08 1/X                    17♦LBL 39
09 RCL 03                 18 END
```

Listing of CALC-PY subroutine

SINK SUBROUTINE

SINK is a subroutine called by CALC-FV, CALC-PY and FIN-PMT. Within SINK, the sinking fund factor is calculated and stored into main memory register R10.

```
01♦LBL "SINK"
02 1
03 RCL 01
04 +
05 RCL 00
06 Y↑X
07 1
08 -
09 RCL 01
10 /
11 STO 10
12 END
```

Listing of SINK subroutine

PRS-VAL SUBROUTINE

PRS-VAL is a subroutine called by CALC-PV, CALC-PY and FIN-PMT. Within PRS-VAL, the present value factor is calculated and stored into main memory register R11.

```
01♦LBL "PRS-VAL"
02 1
03 ENTER↑
04 ENTER↑
05 RCL 01
06 +
07 RCL 00
08 CHS
09 Y↑X
10 -
11 RCL 01
12 /
13 STO 11
14 END
```

Listing of PRS-VAL subroutine

FIN-PMT SUBROUTINE

FIN-PMT is a subroutine which is invoked by pressing key -15. Within FIN-PMT, the following processing takes place. The subroutine function is printed so that the user is kept informed of the processing taking place. Next, main memory register R03 is tested to determine if the user has supplied a value for FV. If R03 is zero, then it is assumed that the final payment amount is to be calculated using PV in which case the calculation proceeds. If R03 is not zero, the calculation of final payment proceeds using a value for FV. In either case, the final payment amount is stored into main memory register R05 and is printed out.

CONCLUSION

The preceding program ANNUITY illustrates the use of several programming ideas. The program should be studied and understood before proceeding on to the next chapter. It is not as important to understand the mathematics involved as it is to understand the techniques used in the program.

```
01♦LBL "FIN-PMT"
02 "CALC FINAL PMT"
03 PRA
04 RCL 03
05 X=0?
06 GTO 40
07 XEQ "SINK"
08 RCL 04
09 RCL 10
10 *
11 RCL 01
12 1
13 +
14 *
15 RCL 03
16 X<>Y
17 -
18 GTO 49
19♦LBL 40
20 XEQ "PRS-VAL"
21 RCL 11
22 RCL 04
23 *
24 RCL 02
25 X<>Y
26 -
27 1
28 ENTER↑
29 RCL 01
30 +
31 RCL 00
32 1
33 +
34 Y↑X
35 *
36♦LBL 49
37 STO 05
38 "FIN PMT = "
39 ARCL 05
40 PRA
41 END
```

Listing of FIN-PMT subroutine

Chapter 8

Internal Files

INTRODUCTION

There are several file types available for use on the HP-41C/CV. Of these, only the data and ASCII files may be used to store the data used in programs. This chapter will discuss in detail the creation and use of internal data and ASCII files. An understanding of these two file types will be obtained through a discussion of the following topics:

Extended memory
File headers
File pointers
Working files
File creation and management
Recording data into internal files
Retrieving data from internal files

NOTATION

In this chapter, many of the extended functions will be discussed. The general form of an extended function command is

command X(n) ALPHA(string)

X(n) is used to indicate that some value n is to be placed into the X register prior to executing the command. ALPHA(string) is used to indicate that some string of alphanumeric characters is to be placed into the ALPHA register prior to executing the command.

EXTENDED MEMORY

Extended memory is treated by the HP-41C/CV differently from main memory. The data stored in extended memory cannot be directly accessed with basic HP-41C/CV commands such as STO and RCL. Instead, the desired data must be transfered to main memory before the machine can operate on it. The extra steps necessary to transfer this data are offset by the extra capacity offered by the extended memory.

Extended memory registers are organized into structures called internal files. There are three types of files which may be stored into and recalled from external memory - **data (D), ASCII (A)** and **program (P)** files. Program files will be discussed in Chapter 10.

Internal files consist of two registers which contain information about the file and one or more data records. Internal files are stored in extended memory in the order in which they were created.

FILE HEADERS

The first two registers of any internal file are called the file header. The first register of the file header contains the filename. Filenames can be any combination of alphanumeric characters up to seven characters in length. Longer filenames will be truncated to seven characters. Shorter filenames will be right-filled with spaces to bring the name up to seven characters.

The second register of the file header contains information about the length and type of file and one or two pointers used to gain access to data within the file. Internal data files require only a register pointer whereas internal ASCII files require both a record pointer and a character pointer.

FILE POINTERS

The HP-41C/CV uses a portion of the second file header register to store the file pointers. These pointers enable the system to know exactly which register or character within a record is currently being accessed or about to be accessed. In order to understand how the pointers work, we will discuss an internal data file example and an internal ASCII file example.

DATA FILE POINTERS

Internal data files enable the contents of main memory registers to be stored in extended memory. This frees up the main memory registers for other uses. The simplist internal data file consists of one record of n registers containing numeric data. Fig. 8.1 illustrates how an internal data file may be logically viewed.

Suppose it is necessary to access the data stored in register 2 of the record. The register pointer would have to be moved so that it points to register 2. The **SEEKPTA** command will position the record pointer to the register specified in the command and will make the named file the working file. It is never necessary to account for the two registers in the file header when issuing any of the internal file

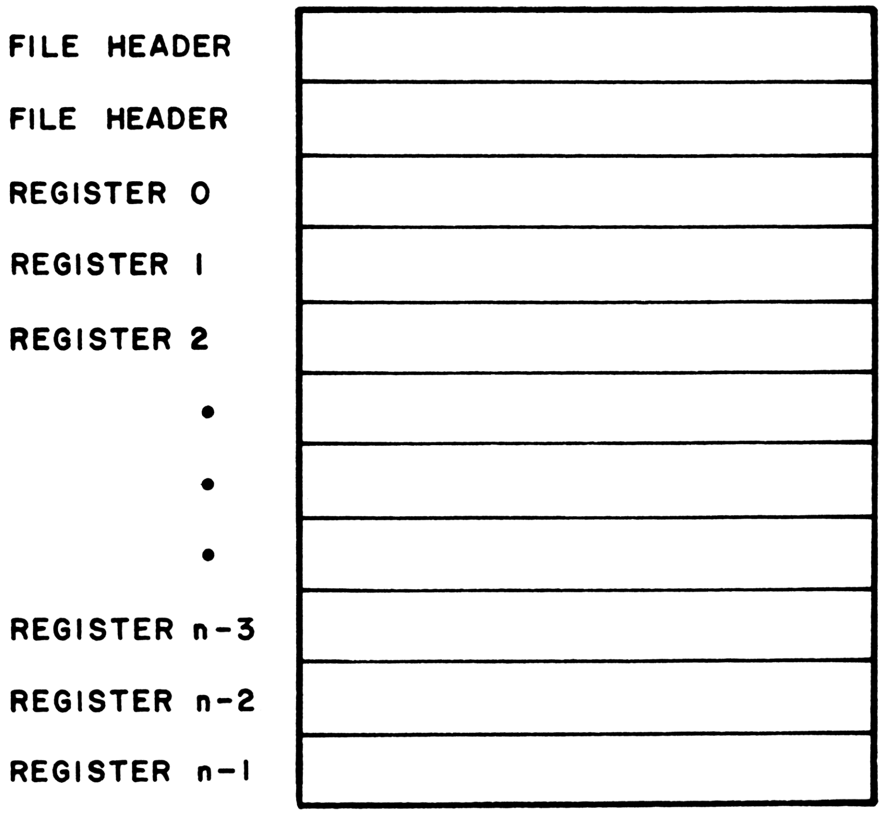


Fig. 8.1 A single record internal data file

commands. The following line of code will access register 2 of the named file and make the file the working file:

```
SEEKPTA  X(2.00) ALPHA(filename)
```

Now, suppose it is necessary to access register 0 of the record. The **SEEKPT** command will position the register pointer in the working file to the register specified in the command. The following line of code will accomplish this:

```
SEEKPT     X(0.00)
```

The register pointer may be moved up or down as desired. If the program calculates the value of the register pointer, only the interger portion of the value in the X register will be used.

A more complicated type of internal data file consists of multiple records each of which contain several registers. Logically, this type of file looks like Fig. 8.2. Since the file is stored as contiguous register locations and the machine does not recognize record boundaries as it does for ASCII files, it is the programmers responsibility to calculate the register pointer.

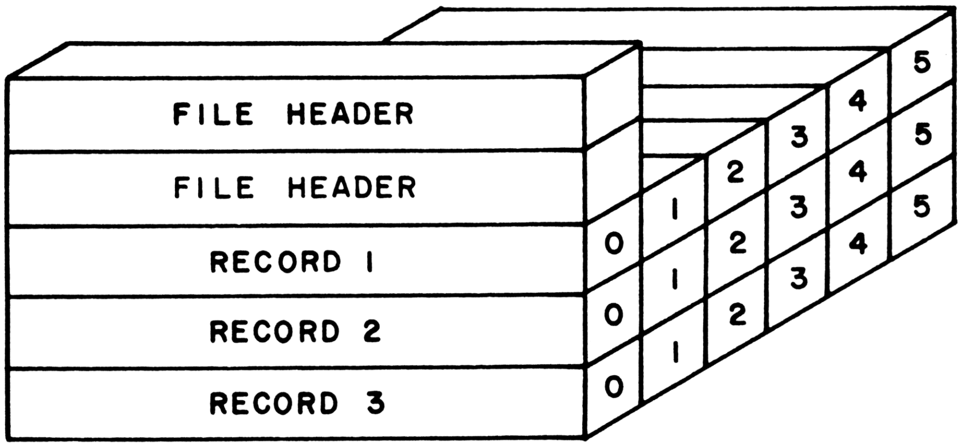


Fig. 8.2 A complex internal data file

Each rectangular block represents a register within the file. The first two registers within the internal file are the file header. Each record consists of of 6 registers numbered 0 through 5. When using this type of internal data file , it is advantageous to assign the same data to the same relative register within each record. For example, if the file contains items for an inventory system, then each record would contain the data for one item in inventory. The registers within each record might contain the following data:

register 0	item number
register 1	cost of the item
register 2	quantity on hand
register 3	quantity on order
register 4	minimum quantity allowed
register 5	lead time

If it is necessary to determine the cost of inventory on hand, the quantity on hand (the contents of register 2) would be multiplied by the cost of the item (the contents of register 1). If this is done for all records in the file, then a total cost of inventory would be obtained. This type of operation would not be possible is the same data had not been assigned to the same relative register within the records.

If the complex internal data file is viewed as it is actually stored in the machine, it would appear as in Fig 8.3. Register 0 of record 1 is physical register 0, register 0 of record 2 is physical register 6 and register 0 of record 3 is physical register 12.

The record/register pointer for a complex data file is easy to calculate if Fig 8.3 is kept in mind. The formula used is:

register = ((record number - 1) * (size of record)) + (relative register number)

where **record number** is the desired "record" within the file

size of record is the number of registers in each "record" of the internal file

relative register number is the desired relative register within the "record"

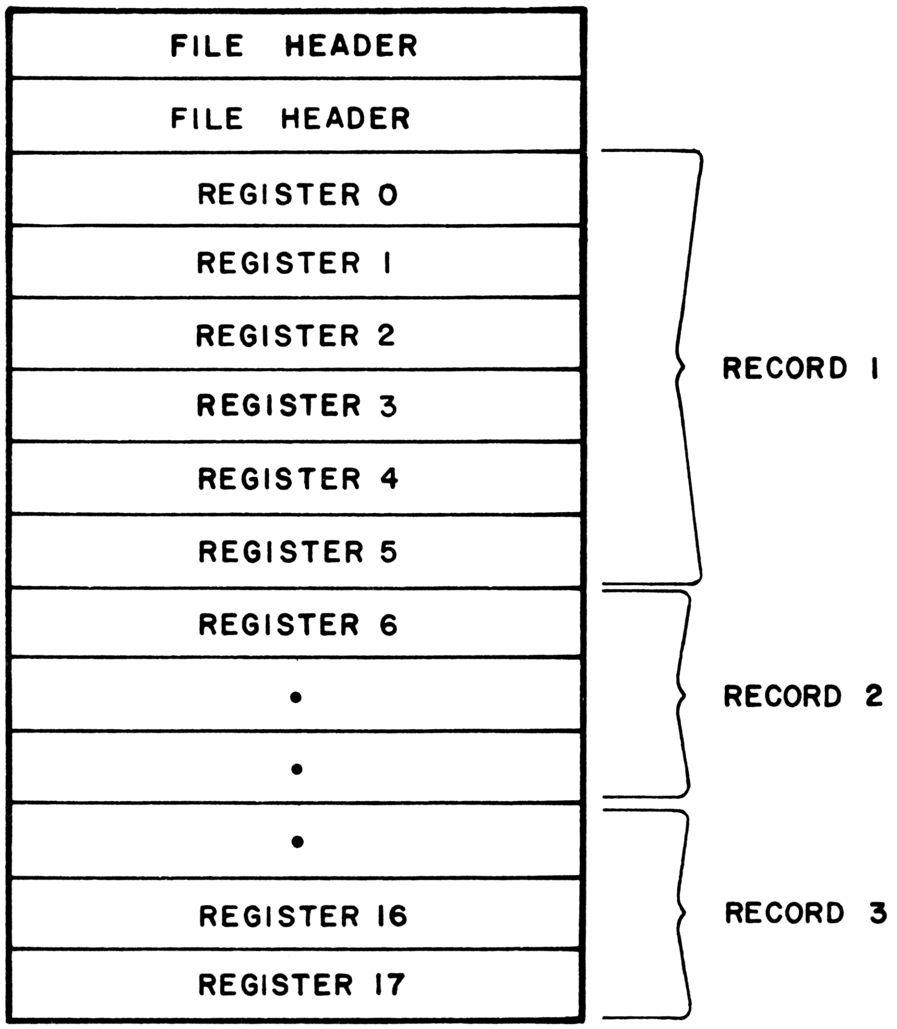


Fig. 8.3 The physical storage of a complex internal data file

The result of the calculation would be left in the X register (remember, only the interger portion of the number will be used). For example, suppose it is necessary to access the quantity on hand (register 2 of each record) of record 3. The calculation would be:

$$\text{register} = ((3 - 1) * (6)) + (2)$$
$$\text{register} = 14$$

Once this calculation is made, the following line of code would set the register pointer of the named file to register 14 and make the named file the working file:

SEEKPTA X(14.00) ALPHA(filename)

The position of the file pointer may be determined at any time using the RCLPTA or the RCLPT commands. The **RCLPTA** command will make the named file the working file and will return the current value of the file pointer to the X register. The general form of this command is:

RCLPTA ALPHA(filename)

The **RCLPT** command will return the current value or the file pointer for the working file to the X register. The general form of this command is:

RCLPT

ASCII FILE POINTER

Internal ASCII files enable the creation and storage of alphanumeric text which can be accessed at the record or character level. The inconvenience of having to calculate the record/register pointer is eliminated because the file header carries two pointers - a record pointer and a character pointer. Fig. 8.4 illustrates how a small ASCII file appears in extended memory.

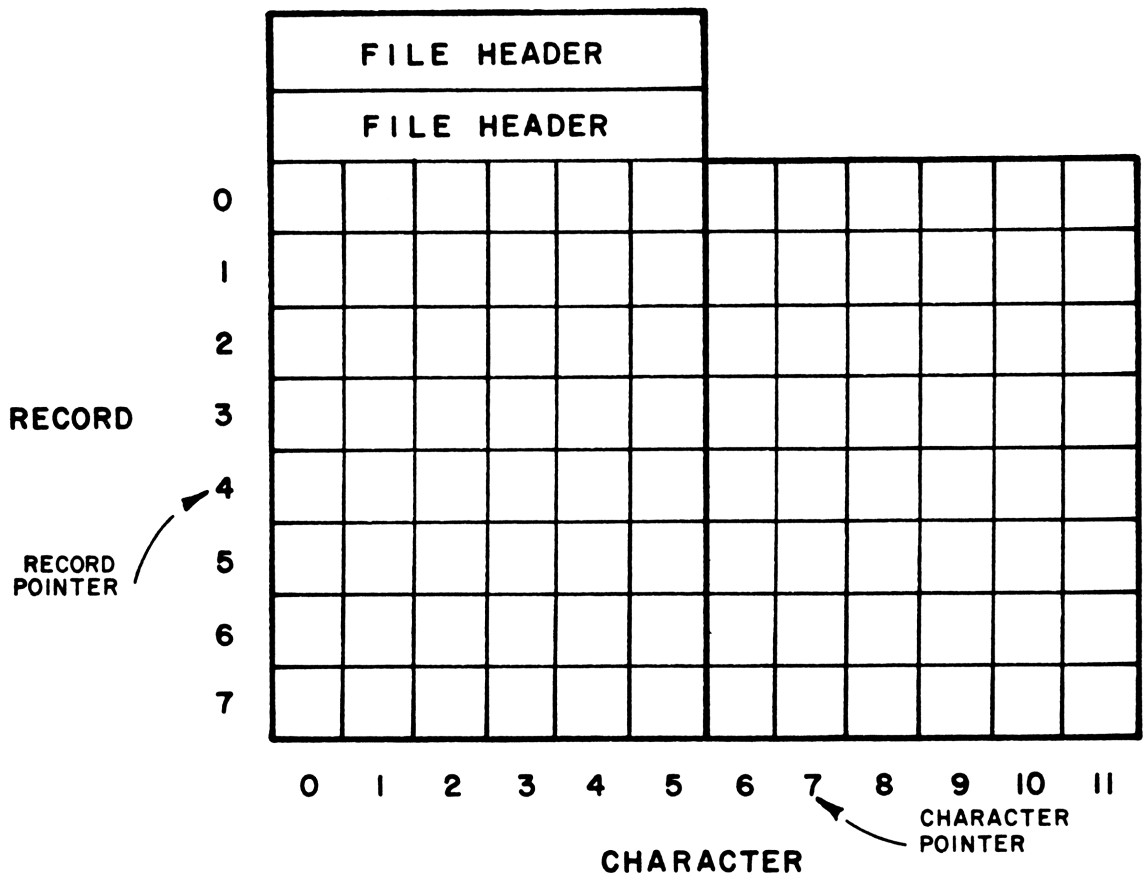


Fig. 8.4 Storage of internal ASCII files

This file is 18 registers in length and consists of 8 records of 12 characters each. Each square represents one character. The record pointer is set to record 4 and the character pointer is set to character 7.

The pointers for internal ASCII files are positioned using the same commands as for data files. The difference is that both the interger and decimal portions of the X register is used. The integer portion is the record pointer and the decimal portion is the character pointer. The following command will position the pointers to character 1 of record 1 and make the named file the working file:

SEEKPTA X(1.001) ALPHA(filename)

Notice that the value in the X register is a real number. The format for the record/character pointer is:

rrr.ccc
where **rrr** is the register number
ccc is the character number

If the character pointer is omitted, character 0 is assumed.

Now reposition the record/character pointer for the working file to character 10 of record 5. The following command will accomplish this:

```
SEEKPT X(5.010)
```

It is possible to recall the pointers for an internal ASCII file so that they may be manipulated or used in some calculation. The **RCLPTA** command will recall the current record/character pointers for the named file to the X register and make the named file the working file.

```
RCLPTA ALPHA(filename)
```

The **RCLPT** command will recall the current record/character pointers for the working file to the X register.

```
RCLPT
```

When the pointers are recalled, the number placed in the X register is a real number. The integer portion of the number is the record pointer and the decimal part of the number is the character pointer. The number may be manipulated just like any other number. For example, the pointers may be recalled to the X register, incremented and the next record retrieved. The following code will do this:

```
   .
   .
RCLPT
ISG X
LBL 00
SEEKPT
RCLREC
   .
   .
```

WORKING FILES

The HP-41C/CV has the ability to remember the name of the file that is currently being manipulated. Several of the internal file manipulation commands require that the filename be placed in the ALPHA register before the first execution of the command. After execution, the named file becomes the working file for the system. File commands issued after a working file is established do not require that a filename be placed into the ALPHA register. The following list of functions will cause the named file to become the working file for the system:

CLFL
CRFLAS
CRFLD
FLSIZE
RCLPTA
SEEKPTA

The following list of commands will operate on the current working file:

APPCHR
APPREC
DELCHR
DELCHR
GETREC
GETRX
GETX
INSCHR
INSREC
POSFL
RCLPT
SAVERX
SAVEX
SEEKPT

FILE CREATION AND MANAGEMENT

The HP-41C/CV operating system must know the file size and other attributes of a file before the file can be processed. The file size, file type and file name are passed to the operating system using the appropriate file creation command. If an internal data file is to be created, the **CRFLD** command is used. The general form of this command is:

CRFLD X(number of registers) ALPHA(data filename)

If an internal ASCII file is to be created, the **CRFLAS** command is used. The general form of this command is:

CRFLAS X(number of registers) ALPHA(ASCII filename)

The number of registers required by the internal file (excluding the two registers required by the header) must be put into the X register and the filename must be put into the ALPHA register. After the file has been created, the new file becomes the working file.

Determining the file size for internal data files is a straight forward process. For simple internal data files, the size is simply the number of registers required to contain the necessary data.

For complex internal data files, the file size is determined by multiplying the number of registers in each record by the number of records. Usually, the exact number of records will not be known ahead of time. In this case, it is a good idea to add 20% to a best guess of the number of records. Although this will increase the file size, it will add a buffer for future file expansion. Registers may be removed easier than they may be added.

The procedure for determining the size of an internal ASCII file is not quite as easy as for data files. The thing to remember here is that a register will hold up to six characters. If the input and output requirements of a program have been determined accurately, then the number of characters for each record of the internal ASCII file will be known. The number of records will not be known, but the 20% rule could be used to come up with a good estimate. To calculate the file size, the number of characters per record is divided by 6 and the result rounded up to the next whole number. This number is then multiplied by the estimated number of records to arrive at the number of registers required by the file.

Occasionally, it may be necessary to clear a file. For internal data files, this means setting all registers of the file to zeroes and for internal ASCII files it means setting the character and record pointers to 0. The clear file command will clear the named file and retain the file for further use. The general form of this command is:

CLFL ALPHA(filename)

Notice that it is not necessary to indicate the file type as this information is carried in the file header. Files that have just been created will be clear of any data that may have been in the registers prior to the creation of the file.

If a file is no longer needed, it may be purged (removed) from extended memory by using the purge file command. The general form of this command is:

PURFL **ALPHA(filename)**

The **FLSIZE** command will return the file size (the number of registers allocated to the file) of the named file to the X register. If the ALPHA register is clear at the time the command is executed, the size of the working file will be returned to the X register. If the size of some other file is desired, the name of the file is placed into the ALPHA register before the command is executed. The general form of the FLSIZE command is:

FLSIZE ALPHA(filename)

RECORDING DATA INTO INTERNAL DATA FILES

Data is stored into an internal data file from main memeory registers. The **SAVER**, **SAVERX** and **SAVEX** commands are used to store the data (a sequence of main memory registers or the contents of the X register) into the internal data file.

The **SAVER** command copies all the main memory registers currently allocated into the named internal data file. If the ALPHA register is clear, then the main memory registers will be copied into the working file. Main memory register R00 will be copied into internal data file regsiter 0, main memory register R01 will be copied into internal data file register 1, and so on until the last main memory register allocated has been copied into the internal data file. When the process is complete, main memory registers R00 through Rnn will have been copied into data file registers 0 through mm. The process terminates when the last main memory register (Rnn) has been copied or when the last internal data file register (Rmm) has been filled, whichever comes first. At the end of the operation, the file pointer will be positioned to the next available register in the internal data file or to the end-of-file (EOF). The general form of the command is:

SAVER ALPHA(data filename)

The **SAVERX** command copies all main memory registers specified by a control word in the X register to the working file. The control word is of the form:

bbb.eee
where **bbb** is the beginning main memory register
eee is the ending main memory register

Main memory register Rbbb is copied to internal data file register 0, main memory register Rbbb+1 is copied to internal data file register 1, and so on until main memory register eee has been copied to the internal data file. When using this operation, bbb must be less than eee. The general form of this command is:

SAVERX X(bbb.eee)

At the end of the operation, the register pointer will be positioned to the register following the last internal data file register affected or to EOF. The operation will not be attempted if there is not enough room in the internal data file for the data.

The **SAVEX** command will copy the contents of the X register into the internal data file register pointed to by the register pointer. The general form of this command is:

SAVEX

RETRIEVING DATA FROM INTERNAL DATA FILES

Data is retrieved from internal data files and placed into contiguous main memory registers or into the X register. The **GETR, GETRX** and **GETX** commands are used to retrieve the data from internal data files.

The **GETR** command retrieves the contents of the named internal data file (or working file if the ALPHA register is clear) into consecutive main memory registers. The contents of internal data file register 0 is placed into main memory register R00, the contents of internal data file register 1 is placed into main memory register R01, and so on. The process is complete or is terminated when the last main memory register has been filled or when EOF on the internal data file is encountered, whichever comes first. The general form of the command is:

GETR ALPHA (internal data filename)

At the end of the operation, the register pointer will be positioned to the next available register in the internal data file or at EOF.

The **GETRX** command retrieves the contents of the working file into the block of main memory registers specified by the control word in the X register. The control word is of the form:

bbb.eee

where **bbb** is the beginning main memory register used to receive the data
eee is the ending main memory register of the block

The contents of internal data file register 0 is placed into main memory register Rbbb, the contents of internal data file register 1 is placed into main memory register Rbbb+1, and so on. When using this command, bbb must be less than eee. The general form of the command is:

GETRX X(bbb.eee)

At the end of the operation, the register pointer will be positioned to the next register following the last register accessed or to EOF.

The **GETX** command copies the internal data file register pointed to by the register pointer into the X regsiter. The general form of the command is:

GETX

RECORDING DATA INTO INTERNAL ASCII FILES

Whereas internal data files are built by copying blocks of main memory registers, internal ASCII files are built by appending or inserting the contents of the ALPHA register into the current record of the file. The ALPHA register may contain single characters, parts of records or entire records. There are two commands which will put entire records into the working ASCII file and two commands which will append (add) characters to the end of existing records in the working ASCII file.

The **append record** command will append the contents of the ALPHA register to the end of the working file as a new record. Since the ALPHA register may contain at most 24 characters, records added to the file in this manner can be no longer than 24 characters. To overcome this restriction, the **append characters** command is used to add characters onto the end of the current record in the internal ASCII file (the record being pointed to by the record/character pointers). The **insert characters** command will also overcome the 24 character restriction by inserting characters ahead of the character being pointed to by the record/character pointers. The only restriction to adding or inserting characters to the current record is that the total record length may not exceed 254 characters. The general form of the append record command is:

APPREC ALPHA(up to 24 characters of text)

The general form of the append characters command is:

APPCHR ALPHA(up to 24 characters of text)

The general form of the insert characters command is:

INSCHR ALPHA(up to 24 characters of text)

Suppose it is necessary to add another record to the working file. The text to be in the record is 'LAMP, 3 BULB W/ SHADE   1015810055.00'. The following lines of code will create this record and insert the appropriate text:

```
01♦LBL "AA"
02 "."
03 "."
04 "."
05 CLA
06 "LAMP, 3BULB W/S"
07 "⊦HADE"
08 APPREC
09 RCLPT
10 INT
11 0.024
12 +
13 SEEKPT
14 "1015810055.00"
15 APPCHR
16 "."
17 "."
18 "."
19 END
```

Occasionally it may be necessary to add a record in front of another record. The insert record command will insert the contents of the ALPHA register as a new record in front of the record being pointed to by the record/character pointers. The general form of this command is:

INSREC ALPHA(up to 24 characters of text)

Recall that the **SEEKPT** command will position the record/character pointers within an internal ASCII file. This command must be issued prior to inserting records or characters. Otherwise, the record or characters being inserted will will be put in the wrong place.

RETRIEVING DATA FROM INTERNAL ASCII FILES

Internal ASCII files are stored in extended memory as strings of alphanumeric characters. Each record can be from 1 to 254 characters in length. Because of the length limitation on the ALPHA register, only 24 characters at a time may be transfered from an internal ASCII file to the ALPHA register. One of the system flags (F17) is used to signal the end of record (EOR) during a transfer from the internal ASCII file to the ALPHA register. There are two commands available for transferring data from an internal ASCII file to the ALPHA register. The **get record** command will clear the ALPHA register and then transfer up to 24 characters from the working file to the ALPHA register. The fetch begins at the current record/character pointer position. At the end of the fetch, the character pointer is set to the next character to be transfered. Flag 17 is set at the end of each transfer if EOR has not been reached. If the EOR has been encountered, flag 17 will be cleared. The usefulness of flag 17 arises from the fact that a loop may be coded which will allow the contents of an internal ASCII file to be printed one record at a time with blank lines between the records. The general form of the get record command is:

GETREC

The following code will cause the contents of the working file to be printed in its entirity:

```
01♦LBL "BB"         11 FS? 17
02 "."              12 GTO 02
03 "."              13 ADV
04 "."              14 ADV
05♦LBL 00           15 DSE 00
06 ISG 01           16 GTO 00
07 SEEKPT           17 "."
08♦LBL 02           18 "."
09 GETREC           19 "."
10 OUTA             20 END
```

The **alpha recall record** command will append characters from the working file onto the end of the ALPHA register until the ALPHA register is full. Flag 17 will be set if EOR is <u>not</u> reached and cleared if EOR is reached. This function is useful for constructing lines of print. For example, if a mailing list is to be produced from name, address and phone records, each triple of records would be accumulated 24 characters at a time into the ALPHA register. The ALPHA register would then be output to a printer. The general form of the command is:

ARCLREC

CONCLUSION

Internal files and the commands used to create and access them have been discussed. Alone, these files add an order of magnitude of capability to the basic HP-41C/CV. The understanding of these files will allow the user to move out of the realm of simply storing seemingly unorganized data in main memory into the realm of creating and maintaining organized records and files of data.

Chapter 9

I/O Operations

INTRODUCTION

In the last chapter, methods of creating and using internal data and ASCII files were discussed. The reader may realize by now that as large as the combined main memory and external memory is, it will quickly be used up if several internal files are created and kept. To overcome this limitation, mass storage devices may be put to good use. This chapter will discuss the commands available to transfer internal files to mass storage and to transfer external files on mass storage to internal files. The discussion will center around the following topics:

Initialization procedures
File pointers
File creation and management
Recording data onto and retrieving data from external files
Checking files
Protecting files
External ASCII files
Transfering internal ASCII files to external ASCII files
Transfering external ASCII files to internal ASCII files

NOTATION

In this chapter, many of the HP-IL I/O functions will be discussed. The general form of an HP-IL command is

command X(n) ALPHA(string)

X(n) is used to indicate that some value n is to be placed into the X register prior to executing the command. ALPHA(string) is used to indicate that some string of alphanumeric characters is to be placed into the ALPHA register prior to executing the command.

INITIALIZATION PROCEDURES

The medium on which the external file will be recorded contains not only the data itself, but also a directory and two other records used by the system. The directory is a file which contains a catalog of all files on the medium and their location. The directory contains one entry for each file stored on the medium. Each directory record may contain up to 8 entries. The last entry in the directory is used by the system. The directory must be large enough to accomodate as many files as will be written to the medium. Although as many as 447 files may be accounted for in the directory, it is not a good idea to build a directory which will accomodate more than the necessary number of files. The reason for this is that the more entries there are in the directory the longer the system will take to locate a file. Therefore, you should allocate a directory which is only large enough to handle the expected number of files.

The command used to initialize a new medium is the **NEWM** command. This command will build a directory large enough to accomodate the number of files specified by the user and will fill all registers on the medium with zeroes. The NEWM command need only be executed one time for each new medium. The command takes about 3 minutes to execute. The general form of the NEWM command is:

NEWM

The command will prompt for a three digit number which is the number of entries to be built into the directory.

FILE POINTERS

The individual registers within an external data file are located by means of the **SEEKR** command. The basic form of this command is:

SEEKR X(register) ALPHA(filename)

This command must be executed prior to any operation which will retrieve from or store into an external data file. Registers within an external data file are numbered beginning at register 0.

External ASCII files do not have a corresponding SEEKR command. ASCII files may only be stored onto or retrieved from the medium. Any file manipulation must occur within the internal ASCII file.

FILE CREATION AND MANAGEMENT

The NEWM command builds a directory and clears the medium of any previously stored data. It is necessary to define to the system the name and size of each external data file to be stored. The **CREATE** command will allocate a part of the medium to the new external data file and fill the space with zeroes. The number in the X register and the name in the ALPHA register will be recorded into the directory. If the name already exists on the directory, an error message will be displayed by the system. The general form of the CREATE command is:

CREATE X(filesize) ALPHA(filename)

If the duplicate filename message is issued, you have several options available to handle the problem. Assuming that the name used in the CREATE command is the name intended, you may PURGE the existing file from the medium or RENAME the existing file.

To purge a file means to remove its name from the medium so that access to the data on the file is no longer possible. The general form of the **PURGE** command is:

PURGE ALPHA(filename)

The PURGE command works with both external data and external ASCII files. Since the results of this command are permanent, it is a good idea to be sure that the file is no longer needed.

Renaming a file means to change the name of a file stored on the medium. The general form of the **RENAME** command is:

RENAME ALPHA(oldname,newname)

Occasionally, you may not want to remove a file. Renaming a file is a good alternative to purging a file. This command may also be used with external ASCII files.

If you want to reuse a file already on the medium, it may be necessary to clear the file to zeroes so that unexpected data will not be encountered. The **ZERO** command will clear the named file to zeroes. The general form of this command is:

ZERO ALPHA(filename)

If a medium contains more than one file, a listing of the contents of the directory should be obtained. The **DIR** command will display the name of each file on the medium along with the length of the file and the file type. If a printer is attached, then this information will be printed. If a directory listing is to be obtained, it should always be printed and kept with the medium in its container. It is also a good idea to have the date and time appear on the listing. This is a good use of the TIME module. The following program will allow the user to obtain a date and time stamped directory listing. The listing produced will assure that the current contents of the medium will be known.

```
01♦LBL "DTETIME"
02 FIX 4
03 DATE
04 CLA
05 ADATE
06 PRA
07 TIME
08 CLA
09 ATIME
10 PRA
11 FIX 2
12 DIR
13 END
```

Listing of a directory listing routine

RECORDING DATA INTO EXTERNAL DATA FILES

There are two commands available which will cause the data stored in main memory registers to be copied into the appropriate external file. The **WRTR** command will copy data from all main memory registers into the named external file. If an internal data file is to be copied to an external data file, the data must first be transfered from the internal file in extended memory into main memory. The general form of the WRTR command is:

WRTR ALPHA(filename)

The copying begins at main memory register R00 and continues until all main memory registers have been copied. Data is stored in the external file beginning at register 0. At the end of the copy, the external file pointer will be positioned at the next available register in the external file or at the end of file (EOF).

The **WRTRX** command will copy data from the specified main memory register block into the external file beginning at the register pointed to by the external file pointer. The medium must be positioned prior to executing this command if the data is to occupy the proper registers. The SEEKR command or a previously executed WRTR, WRTRX, READR or READRX command will leave the external file pointer positioned at the next available register in the file. The basic form of the WRTRX command is:

WRTRX X(bbb.eee)

Copying begins at main memory register Rbbb and continues until main memory register Reee has been copied or until the last main memory register has been copied. If bbb is greater than eee, then only main memory register Rbbb will be copied. At the end of the operation, the external file pointer will be positioned at the next available register in the external file or to EOF.

CHECKING FILES

The **VERIFY** command will allow the user to check the named external file to ensure that the data recorded in the file can be accessed. It is a good idea to execute this command immediately after recording the data. If the result of the verification is such that the file cannot be read, then the same data may be recorded again on another medium. The basic form of the VERIFY command is:

```
VERIFY  ALPHA(filename)
```

PROTECTING FILES

Once an external file has been created and verified, it may be a good idea to protect the file from accidental erasure or alteration. This may be done with the **SEC** command. The basic form of this command is:

```
SEC   ALPHA(filename)
```

A secured external file has an S following the filename in a directory listing. There are two ways in which a secured file may be altered. First, a NEWM command may be executed to completely initialize the medium. Secured files are not immune to the effects of the NEWM command. Secondly, the file may be unsecured.

The **UNSEC** command will cancel the security given an external file by the SEC command. The basic form of the UNSEC command is:

```
UNSEC  ALPHA(filename)
```

After a file has been unsecured, it may be altered as desired.

RETRIEVING DATA FROM AN EXTERNAL DATA FILE

There are two commands available which will cause the data stored in an external file to be copied into main memory registers. The **READR** command will cause the data in an external file to be copied into main memory beginning at main memory register R00. External file registers are copied beginning at register 0 and continuing until the end of the external file is reached or until all allocated main memory registers have been filled. The external file pointer will be left positioned to the next available external file register or EOF. The general form of the READR command is:

READR ALPHA(filename)

The **READRX** command will copy data beginning at the current position of the external data file pointer into the specified main memory register block. The medium must be positioned to the first external data file register to be copied prior to executing this command. The SEEKR command or a previously executed WRTR, WRTRX, READR or READRX command will leave the external data file pointer positioned at the next available register in the file. The basic form of the READRX command is:

READRX X(bbb.eee)

Copying begins at the current location of the external file pointer and data is placed into main memory beginning at Rbbb. Copying continues until main memory register Reee is filled or until the end of the external file is reached. If bbb is greater than eee, then only the data from the current external file register is copied into main memory register Rbbb. At the end of the operation, the external file pointer will be positioned at the next available register or to EOF.

The data placed into the HP-41C/CV through the use of a READR or READRX command goes directly into main memory. If the data is destined for an internal data file, then the data must be copied into extended memory from main memory.

EXTERNAL ASCII FILES

Internal ASCII files may be stored as external ASCII files in much the same way as internal data files are stored as external data files. The major difference between external ASCII files and external data files is that external ASCII files may only be read or written, they may not be altered in whole or in part while on the medium. Before an internal ASCII file may be stored as an external ASCII file, the external file must have been created using the CREATE command. If an external ASCII file is to be transferred to an internal ASCII file, the internal file must have been allocated using the CRFLAS command.

External ASCII files may be secured with the SEC command, unsecured with the UNSEC command, renamed with the RENAME command and verified with the VERIFY command.

TRANSFERING INTERNAL ASCII FILES TO EXTERNAL ASCII FILES

The **SAVEAS** command will copy the named internal ASCII file to the specified external ASCII file. This command requires that the names of both the internal and external files be supplied at the time of issuing the command. If only the internal file name is given, then the system will assume that the external file is to be called by the internal filename. The general form of the SAVEAS command is:

SAVEAS **ALPHA(internal name,external name)**

If the external filename is not the same as the name specified on the CREATE command, then the system will issue an error message. Copying will continue until the end of the internal file is reached or until the end of the external file is reached.

TRANSFERING EXTERNAL ASCII FILES TO INTERNAL ASCII FILES

The **GETAS** command will copy the named external ASCII file to the specified internal ASCII file. This command requires that both the internal filename and external filename be specified in the ALPHA register at the time the command is issued. If only the external filename is supplied, then the system will assume that the name of the internal file is the same as the name of the external file. Whatever name is used for the internal file must have been specified on a CRFLAS command. The general form of the GETAS command is:

```
GETAS  ALPHA(external name,internal name)
```

Copying will continue until either the end of the external file is reached or the end of the internal file is reached.

CONCLUSION

This chapter has presented the commands necessary to create and access external files. With the current capacity of mass storage devices (131K characters) and the expanded capabilities to the HP-41C/CV and the Extended Functions, you will be hard pressed to find applications which cannot be accomodated on the HP-41C/CV personal computer.

Chapter 10

Sales Program Example

INTRODUCTION

This chapter will present a program which uses both internal and external data files. Careful study will give you a better understanding of how to setup internal and external data files and how to use the powerful I/O commands available in the HP-IL and Extended Functions modules.

SALES PROGRAM BACKGROUND

SALES is a program designed to prompt the user for an item number and the quantity sold. The program will obtain the price of the item sold from an internal price file, calculate the extended cost and print a line on the receipt. At the end of the sale, the program will subtotal the sale, calculate the appropriate sales tax and total the sale. In addition to creating a receipt, the program will create a record of the sale and write this out to tape. At a later time, this tape will be used as input to an inventory program.

There are three auxillary programs associated with SALES which help the user with loading the internal price file and with loading and unloading the programs in the SALES system.

LODPRI is a short program which prompts the user for item number and price and then loads this information into an internal data file called PRIC.DA. The program first positions the internal file pointer to register 0 (it assumes that the file PRIC.DA has already been created and is of such a size as to allow for expansion). The user is then prompted for an item number between 100 and some predefined upper limit found by multiplying the size of the internal file PRIC.DA minus 1 by 100. If a proper number is keyed in and R/S pressed, the program will go on to prompt the user for the price of the item associated with the item number. The program uses the item number as an absolute index (register number) into the file and stores the price there. This process of prompting the user for an item number and price continues until the user presses R/S without first keying in an item number.

This technique of calculating a register number into which to store data is useful for two reasons. First, space is saved in the internal file because the file does not have to contain the item number associated with the price as this information is determined from the location within the file of the price. If a price is stored in register 5, for example, then the program is "intelligent" enough to know that the price must be associated with item number 105. Secondly, the location of items within the file are easy to determine and remember.

```
♦LBL "LODPRI"
CLA
"PRIC.DA"
0
SEEKPTA
FIX 0
♦LBL 00
"ITEM-NUM"
PROMPT
FC?C 22
GTO 09
100
-
"PRICE ?"
PROMPT
CF 22
X<>Y
SEEKPT
RDN
SAVEX
GTO 00
♦LBL 09
FIX 2
.END.
```

Listing of program LODPRI

LOD-SAL is a program which resides in program memory at all times. This program will prompt the user for the physical location (tape or extended memory) of the SALES system. If the SALES system is located on tape, then LOD-SAL will load each routine and subroutine into program memory using the READP command.

If the SALES system is located in extended memory, then LOD-SAL will load each routine and subroutine into program memory using the GETP command. After the SALES system is loaded, the program UNL-SAL is loaded. The necessary key assignments are then made and LOD-SAL is exited.

```
01♦LBL "LOD-SAL"
02 SF 27
03 0
04 PSIZE
05 "TAPE<Y,N> ?"
06 AON
07 PROMPT
08 AOFF
09 ATOX
10 78
11 -
12 X≠0?
13 GTO 00
14 "SALES"
15 GETSUB
16 "GETSAL"
17 GETSUB
18 "GETPRIC"
19 GETSUB
20 "RECSAL"
21 GETSUB
22 "TOTAL"
23 GETSUB
24 "SUBTOTL"
25 GETSUB
26 "TAX"
27 GETSUB
28 "RECEIPT"
29 GETSUB
30 "UNL-SAL"
31 GETSUB
32 GTO 01
33♦LBL 00
34 "SALES"
35 READSUB
36 "GETSAL"
37 READSUB
38 "GETPRIC"
39 READSUB
40 "RECSAL"
41 READSUB
42 "TOTAL"
43 READSUB
44 "SUBTOTL"
45 READSUB
46 "TAX"
47 READSUB
48 "RECEIPT"
49 READSUB
50 "UNL-SAL"
51 READSUB
52♦LBL 01
53 "SALES"
54 11
55 PASN
56 "TAX"
57 13
58 PASN
59 "UNL-SAL"
60 15
61 PASN
62 END
```

Listing of program LOD-SAL

UNL-SAL is a program which will move the SALES system out of program memory. The program will prompt the user as to whether the SALES system is to be recorded onto tape in addition to being copied into extended memory. After the programs have been copied, any key assignments which have been made are erased and the SALES system is removed from program memory.

```
01♦LBL "UNL-SAL"
02 "RECORD ?"
03 AON
04 PROMPT
05 AOFF
06 ATOX
07 78
08 -
09 X≠0?
10 SF 00
11 "SALES"
12 SAVEP
13 FS? 00
14 WRTP
15 "GETSAL"
16 SAVEP
17 FS? 00
18 WRTP
19 "GETPRIC"
20 SAVEP
21 FS? 00
22 WRTP
23 "RECSAL"
24 SAVEP
25 FS? 00
26 WRTP
27 "TOTAL"
28 SAVEP
29 FS? 00
30 WRTP
31 "SUBTOTL"
32 SAVEP
33 FS? 00
34 WRTP
35 "TAX"
36 SAVEP
37 FS? 00
38 WRTP
39 "RECEIPT"
40 SAVEP
41 FS? 06
42 WRTP
43 "UNL-SAL"
44 SAVEP
45 FS?C 00
46 WRTP
47 CLA
48 11
49 PASN
50 13
51 PASN
52 15
53 PASN
54 "SALES"
55 PCLPS
56 CF 11
57 CF 27
58 .END.
```

Listing of program UNL-SAL

PROGRAM FILES

Within main memory, there may be two types of information - data and programs. The only way the system has of knowing whether something in main memory is program or data is by the location of the curtain. Programs may also be stored in extended memory or on an external medium. When stored in one of these other areas, the system must know that they are programs and not some other type of data. This problem is handled by the definition of a new type of file called program files. Program files in extended memory are labeled as P files and program files on an external medium are labeled as PR files.

Programs files in program memory may be stored into extended memory using the **SAVEP** command or they may be copied onto an external medium using the **WRTP** command. When copying program files from main memory to one of these other areas, there is no distinction between program and subroutine. The basic form of the SAVEP and WRTP commands is:

SAVEP ALPHA(program name,file name)

WRTP ALPHA(program name,file name)

These commands will copy a program from program memory to either extended memory or to an external medium as a program file. If the HP-41C/CV is in USER mode and some keys are reassigned, then the key assignments will also be copied. If the file name is omitted at the time of copying, the name of the program will become the name of the file. Program files do not need to be allocated prior to being copied by the SAVEP or WRTP commands.

Program files stored in either extended memory or on an external medium may also be copied into program memory. The location where the program file will be placed depends upon whether the program file is copied into program memory as a program or as a subroutine. If the program file is copied as a program, then the file will be placed in main memory program space replacing the last program. If the program file is copied as a subroutine, then it will be placed into program memory following the last program. If program files are coming from extended memory, then either the **GETP** or the **GETSUB** command will copy the program file into program memory. The basic form of these commands is:

GETP ALPHA(file name)

GETSUB ALPHA(file name)

If program files are coming from an external medium, then either the **READP** or the **READSUB** command will copy the program file into program memory. The basic form of these commands is:

READP ALPHA(file name)

READSUB ALPHA(file name)

Any key assignments recorded with the program will become active when the program file is copied into main memory program space.

SALES PROGRAM DESIGN

The SALES program was designed to meet two objectives. First, it had to be as easy to use as possible. Secondly, it was to be part of a larger system to be used in small businesses to function as a point-of-sale terminal and to create input for an inventory system.

To make the SALES system as easy to use as possible, many of the preliminary tasks are handled by the SALES program and the auxilliary programs. File pointers are prepositioned, main memory registers are allocated and the price file is made the working file. All that is left for the user to do is answer the questions asked by the system.

The SALES system was designed to interface with an inventory system. The purpose of this interface is to allow sales transactions to be passed to an inventory system so that the appropriate inventory records may be updated.

The SALES system consists of the following routines and subroutines. Along with the name is the function performed by the routine or subroutine.

SALES the mainline routine which sets user flag F11 so that the program will start each time the HP-41C/CV is turned on, initializes control registers and continually calls GETSAL until user flag F00 is set at which time TOTAL is called

GETSAL this subroutine prompts the user for the quantity and item sold. If the user pressed R/S without pressing a digit key, the subroutine will set F00 and exit

GETPRIC this subroutine calculates a register number in PRIC.DA from which the price of the item is obtained

RECSAL this subroutine formats the sale information into a data record and writes the record out to tape

TOTAL this subroutine calculates a total for the sale and then produces two receipts. The subroutine SUBTOTL is called to calculate a subtotal and TAX is called to determine the appropriate sales tax

SUBTOTL this subroutine subtotals the sale

TAX this subroutine determines the appropriate sales tax

RECEIPT this subroutine formats and produces two receipts - one for the customer and one for the user

The hierarchy diagram for the SALES system is on the page after next. Referring to this diagram will allow you to visualize the relationship of the subroutines to each other and to the routine SALES.

SALES ROUTINE

SALES is the mainline routine or driver for the SALES system. Within SALES, the following processing takes place. User flag F11 is set so that the program will automatically begin whenever power is applied to the machine. User flag F27 is set to place the machine in USER mode.

The **PWRDN** command is issued so that the printer and tape drive, both of which should be in STANDBY mode, will be powered down.

The internal data file, PRIC.DA, is made the current file and its pointer is set to register 0.

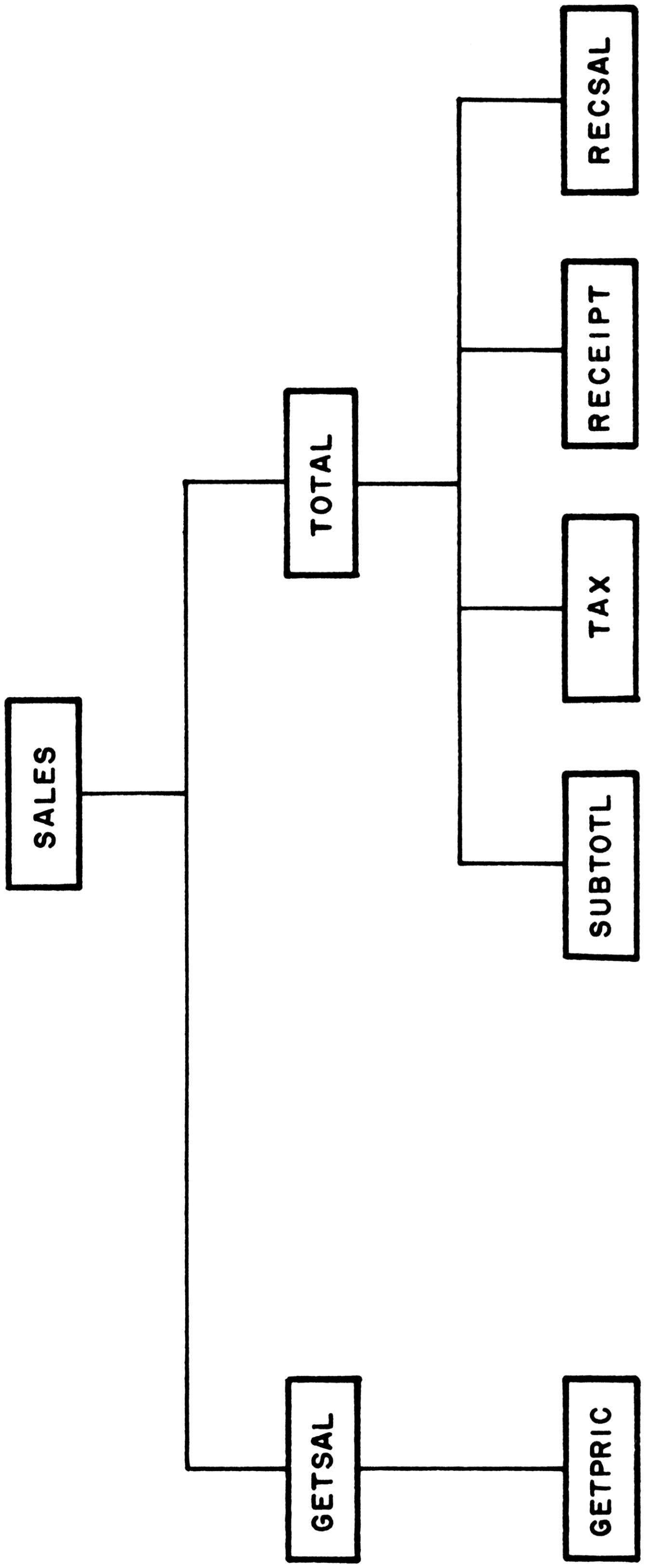
Main memory register R04, which will be used in a later subroutine to count the number of groups of items sold, is initialized to 0. Main memory register R03, which will be used in a later subroutine as a pointer to a block of main memory registers used to store sale information, is initialized to 10.

Main memory is sized to 40.

The subroutine GETSAL is called repeatedly until user flag F00 is set.

When user flag F00 is set, subroutine TOTAL is called.

The **OFF** command is issued to turn the HP-41C/CV off. When the machine is turned on again, the mainline routine SALES will begin automatic execution at the GTO "SALES" statement.



Hierarchy diagram for the SALES system

```
01♦LBL "SALES"
02 SF 11
03 SF 27
04 PWRDN
05 CLA
06 "PRIC.DA"
07 0
08 SEEKPTA
09 STO 04
10 10
11 STO 03
12 40
13 PSIZE
14♦LBL 04
15 XEQ "GETSAL"
16 FC?C 00
17 GTO 04
18 XEQ "TOTAL"
19 OFF
20 GTO "SALES"
21 .END.
```

SALES program listing

GETSAL SUBROUTINE

GETSAL is a subroutine called by SALES to prompt the user for the appropriate input. Within the subroutine, the following processing takes place. The user is prompted for the quantity sold. If a number is entered before pressing R/S, then the program will continue and store the quantity indirectly according to R03.

Main memory register R03 is incremented and the user is prompted for the item number. This number is stored indirectly according to R03.

The subroutine GETPRIC is called to obtain the price of the item from the internal data file PRIC.DA and then to store this price indirectly according to R03. Main memory register R04 is incremented and the subroutine is executed again. At this point, the Sales Storage Block which is pointed to by R03 contains the quantity, item number and price of the item sold. The block may be pictured as follows:

QUANTITY	ITEM NUMBER	PRICE
Rn	Rn + l	Rn+2

R_n	begins at main memory register R10
R_{n+1}	begins at main memory register R11
R_{n+2}	begins at main memory register R12

Sales storage block

The way this storage block is defined, up to 10 items may be recorded in it. If more than 10 items may be sold per transaction, the initial size and the initial value set in R03 must be adjusted accordingly. If R/S was pressed in response to the "QUANTITY" prompt without first pressing a digit key, the user flag F00 will be set and the subroutine will exit.

GETPRIC SUBROUTINE

GETPRIC is a subroutine called by GETSAL to obtain a price in the internal data file PRIC.DA. The subroutine assumes that register X contains a number between 100 and some upper limit as defined earlier. The value in register X has 100 subtracted from it and this new value is used in a **SEEKPT** command to position the internal data file pointer to the appropriate file register. The **GETX** command is then issued to return the value in the file register to the X register. The number returned to the X register is the price associated with the item sold. This is an example of a function subroutine, a subroutine which receives a value and transforms that value into another value. SIN and COS are also function subroutines.

```
01♦LBL "GETSAL"
02 CLA
03 "QUANTITY ?"
04 PROMPT
05 FS?C 22
06 GTO 02
07 SF 00
08 GTO 01
09♦LBL 02
10 STO IND 03
11 ISG 03
12♦LBL 00
13 "ITEM-NUM ?"
14 PROMPT
15 CF 22
16 STO IND 03
17 XEQ "GETPRIC"
18 ISG 03
19♦LBL 00
20 STO IND 03
21 ISG 03
22♦LBL 00
23 ISG 04
24♦LBL 00
25♦LBL 01
26 .END.
```

Listing of GETSAL subroutine

```
01♦LBL "GETPRIC"
02 100
03 -
04 SEEKPT
05 GETX
06 .END.
```

Listing of subroutine GETPRIC

TOTAL SUBROUTINE

TOTAL is a subroutine called by SALES to calculate a total for the sale and to produce two receipts. Within the subroutine the following processing takes place. The printer and tape drive are powered up in preparation for use in printing the receipts and recording the inventory record. Subroutine SUBTOTL is called to calculate the subtotal for the sale. The subtotal is placed into main memory register R00.

The subroutine TAX is called to calculate the appropriate sales tax on the sale. The sales tax is placed into main memory register R01.

The total is calculated and printed. The user is then prompted to enter the amount tendered by the customer which is stored in main memory register R03. A calculation is then done to determine the amount of change due the customer or the additional amount needed from the customer. The amount of change or the additional amount needed from the customer is stored in main memory register R05. If change is due the customer, then a line is printed to tell the user the amount of change to give. If an additional amount is needed from the customer, then a line is printed to tell the user the amount to collect.

The subroutine RECEIPT is then called twice to print two receipts - the first to be kept by the user and the second to be given to the customer.

The subroutine RECSAL is then called to build and record an inventory record for each item sold.

The printer and tape drive are then powered down and the subroutine is exited.

SUBTOTL SUBROUTINE

SUBTOTL is a subroutine called by TOTAL to calculate a subtotal for the sale. This subroutine first calculates a control number and stores it in main memory registers R03 and R07. The control number is calculated according to the following equation:

$$c(R03) = ((c(R04) \times 3) + 9) / 1000$$

```
01♦LBL "TOTAL"
02 PWRUP
03 XEQ "SUBTOTL"
04 RCL 00
05 XEQ "TAX"
06 STO 01
07 +
08 STO 02
09 0
10 STO 03
11 "TOTAL = "
12 ARCL 02
13 PRA
14♦LBL 05
15 "TENDERED ?"
16 PROMPT
17 ST+ 03
18 RCL 03
19 RCL 02
20 -
21 STO 05
22 X=0?
23 GTO 06
24 X>0?
25 GTO 07
26 TONE 0
27 "COLLECT = "
28 ARCL 05
29 PRA
30 GTO 05
31♦LBL 07
32 TONE 9
33 "CHANGE = "
34 ARCL 05
35 PRA
36♦LBL 06
37 XEQ "RECEIPT"
38 XEQ "RECEIPT"
39 PWRDN
40 XEQ "RECSAL"
41 .END.
```

Listing of TOTAL subroutine

After this control number is calculated, the extended costs for all items sold is calculated and summed into main memory register R00. This process uses the information stored in the Sales Register Block.

```
01♦LBL "SUBTOTL"
02 0
03 STO 00
04 RCL 04
05 3
06 *
07 9
08 +
09 1 E-3
10 *
11 10
12 +
13 STO 03
14 STO 07
15♦LBL 02
16 RCL IND 03
17 2
18 ST+ 03
19 RDN
20 RCL IND 03
21 *
22 ST+ 00
23 ISG 03
24 GTO 02
25 .END.
```

Listing of SUBTOTL subroutine

TAX SUBROUTINE

TAX is another function subroutine called by TOTAL and which takes 5% of whatever is passed to it in the X register. At the end of the subroutine, register X will contain the tax and register Y the amount that was taxed.

```
01♦LBL "TAX"
02 5
03 %
04 .END.
```

Listing of subroutine TAX

RECEIPT SUBROUTINE

RECEIPT is a subroutine called by TOTAL to format and print a receipt of the sale. Within the subroutine, the following processing takes place. The heading of the receipt is formatted with the name of the company and its phone number. The **FMT** function of the printer is put to use here to center the information on the line. The current date is then obtained from the TIME module and centered on the receipt.

The secondary heading for the receipt is then formatted with the column headings QUANT, ITEM and COST.

The control number calculated in subroutine SUBTOTL is recalled from main memory register R07 and is placed into R03. A loop to retrieve sale data is then entered.

Sale data is obtained from the Sales Register Block, one item at a time, and is formatted onto a detail print line. After the detail line is formatted, it is printed and then the next detail line is formatted and printed, and so on until all details have been formatted and printed.

A subtotal line is then formatted and printed. The subtotal is obtained from main memory register R00. A sales tax line is then formatted and printed. The sales tax is obtained from main memory register R01. A total line is then formatted and printed. The sales total is obtained from main memory register R02.

Finally, a THANK YOU line is formatted and printed, the printer paper is advanced 4 times and the subroutine is exited.

RECSAL SUBROUTINE

RECSAL is a subroutine called from TOTAL to create and record an inventory record for use later in an inventory system. Within this subroutine, the following processing takes place. The internal data file SALH.DA (assumed to already exist) is made the working file and its pointer set to 0. A loop is then entered which will extract each block of information from the Sales Storage Block and store it into SALH.DA.

```
01♦LBL "RECEIPT"
02 CLA
03 ADV
04 FMT
05 "SPICE OF LIFE"
06 ACA
07 " CRAFTS"
08 ACA
09 PRBUF
10 CLA
11 FMT
12 "(813) 397-3673"
13 ACA
14 PRBUF
15 CLA
16 FIX 4
17 DATE
18 FMT
19 ADATE
20 ACA
21 PRBUF
22 CLA
23 FIX 2
24 ADV
25 FMT
26 "QUANT"
27 ACA
28 "   "
29 ACA
30 "ITEM"
31 ACA
32 "    "
33 ACA
34 "COST"
35 ACA
36 PRBUF
37 CLA
38 ADV
39 RCL 07
40 STO 03
41♦LBL 10
42 FIX 0
43 CF 28
44 CF 29
45 RCL IND 03
46 STO 06
47 FMT
48 ARCL IND 03
49 ACA
50 "   "
51 ACA
52 ISG 03
53 "├"
54 ARCL IND 03
55 ACA
56 FIX 2
57 SF 28
58 SF 29
59 ISG 03
60 "├"
61 RCL IND 03
62 ST* 06
63 "  "
64 ACA
65 ARCL 06
66 ACA
67 PRBUF
68 CLA
69 ISG 03
70 GTO 10
71 ADV
72 "SUB"
73 RCL 00
74 ACA
75 FMT
76 ACX
77 PRBUF
78 CLA
79 "TAX"
80 RCL 01
81 ACA
82 FMT
83 ACX
84 PRBUF
85 CLA
86 "TOTAL"
87 RCL 02
88 ACA
89 FMT
90 ACX
91 PRBUF
92 CLA
93 ADV
94 ADV
95 SF 12
96 FMT
97 "THANK YOU"
98 ACA
99 PRBUF
100 CF 12
101 ADV
102 ADV
103 ADV
104 ADV
105 END
```

Listing of RECEIPT subroutine

```
01♦LBL "RECSAL"
02 "SALH.DA"
03 0
04 SEEKR
05 RCL 07
06 STO 03
07♦LBL 08
08 RCL IND 03
09 ISG 03
10♦LBL 00
11 RCL IND 03
12 ISG 03
13♦LBL 00
14 RCL IND 03
15 X<>Y
16 100
17 -
18 2
19 *
20 STO 00
21 SEEKR
22 RDN
23 8.009
24 READRX
25 RCL 00
26 SEEKR
27 RDN
28 RDN
29 X<>Y
30 ST+ 08
31 *
32 ST+ 09
33 RDN
34 RDN
35 RDN
36 WRTRX
37 ISG 03
38 GTO 08
39 .END.
```

Listing of RECSAL

CONCLUSION

This program has illustrated several important concepts. The use of internal and external data files was illustrated as well as HP-IL and Extended Functions. This program should be studied carefully so that it is understood. The techniques used here may be used equally well in other applications you may have.

Index

adjustment step	67
annuities	78
APPCHR	113
APPREC	112
ARCLREC	115
ASCII files	99
ASCII file pointers	104
auto-push state	29
bbb.eee	110, 111
BST	41
calling sequence	73
CASE	50
CAT 1	36
cccc.tttdd	60
cccc.tttii	57
CLFL	109
closed subroutine	73
compiling	44
complex data file	101
conditional transfers	56
CREATE	118
CRFLD	108
CRFLAS	108
data files	99
data file pointer	100
DEL	42
directory	117
DO-UNTIL	69, 50
DSE	59

DO-WHILE	68, 50
END	40
entry point	48
exit point	48
external ASCII files	122
file headers	99
file pointers	99
FLSIZE	109
FMT	139
functionally cohesive	76
functors	29
GETAS	123
GETP	129
GETR	111
GETREC	114
GETRX	111
GETSUB	129
GETX	112
GTO	54
GTO..	38
GTO.nnn	41
hard addressed	11
hierarchy diagrams	85
IF-THEN-ELSE	49, 62
infinite loops	48
initialization step	67
INSCHR	113
INSREC	113
interchangable solutions	86

internal files	99
ISG	57
loops	67
loop body	67
manipulators	29
NEWM	117
next sequential instruction	74
NSI	74
open subroutine	73
PACK	38
parameter	72
PCLPS	43
port-addressed	11
PPC	73
program body	36
program files	99, 128
program header	36
program trailer	36
program memory	36
PRP	43
PSIZE	38
PRSTK	30
PURFL	109
PURGE	118
PWRDN	131
RCLPT	104
RCLPTA	104
READP	129
READR	122

READRX	122
READSUB	129
RENAME	119
RPN	23
rrr.ccc	105
SAVEAS	123
SAVEP	129
SAVER	110
SAVERX	110
SAVEX	111
SEC	121
sequence control structure	48
SEEKPTA	101
SEEKPT	101
SEEKR	117
SIZE	38
SST	41
status registers	8
structured design	47
structured programming	48
structured techniques	47
subroutine body	76
subroutine header	75
subroutine trailer	76
test for loop exit	67
unconditional transfers	54
unreachable code	48
UNSEC	121
VERIFY	121

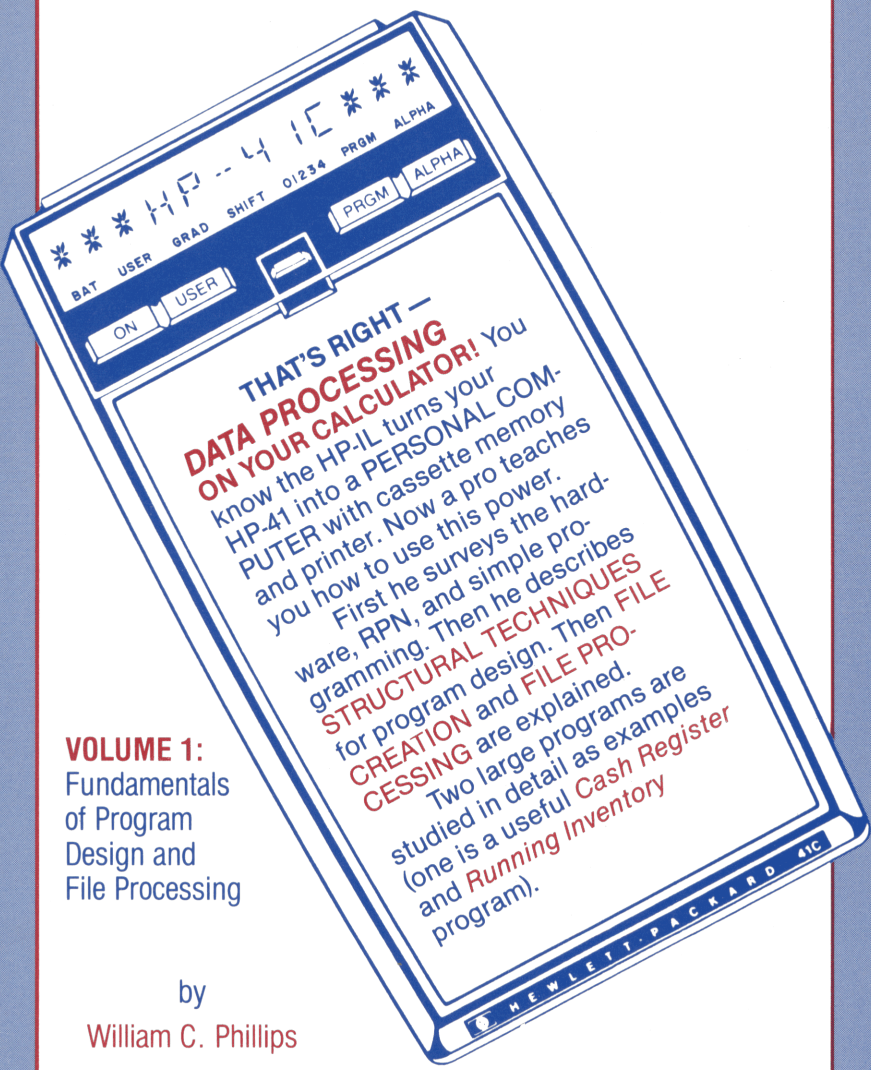
working files 107
WRTP 129
WRTR 120
WRTRX 120

XEQ 73

ZERO 119

20% rule 108

DATA PROCESSING ON THE HP-41 C/CV



THAT'S RIGHT —
DATA PROCESSING ON YOUR CALCULATOR! You know the HP-IL turns your HP-41 into a PERSONAL COMPUTER with cassette memory and printer. Now a pro teaches you how to use this power. First he surveys the hardware, RPN, and simple programming. Then he describes STRUCTURAL TECHNIQUES for program design. Then FILE CREATION and FILE PROCESSING are explained. Two large programs are studied in detail as examples (one is a useful *Cash Register* and *Running Inventory* program).

VOLUME 1:
Fundamentals of Program Design and File Processing

by
William C. Phillips

The EduCALC Technical Series

ISBN 0-936356-02-2
Library of Congress Catalogue Card Number 83-81097