 GSBORNE/McGraw-Hill

PET™/CBM™ PERSONAL COMPUTER GUIDE



Carroll S. Donahue
Janice K. Enger

 **commodore**

Authorized

PET-CBM Personal Computer Guide

**Carroll S. Donahue
Janice K. Enger**

**OSBORNE/McGraw-Hill
Berkeley, California**

Published by
OSBORNE/McGraw-Hill
630 Bancroft Way
Berkeley, California 94710
U. S. A.

For information on translations and book distributors outside of the U. S. A. ,
please write OSBORNE/McGraw-Hill at the above address.

PET-CBM PERSONAL COMPUTER GUIDE

1234567890 DODO 89876543210

ISBN 0-931988-30-6

Copyright © 1980 McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of the publishers.

Cartoons by Erfert Nielson.

The definitions on the back cover are adapted, with permission, from Webster's New World Dictionary, Second College Edition. Copyright © 1980 by William Collins Publishers, Inc.

The programs in this book were printed on a Commodore CBM Model 2022 Tractor Printer.

Table of Contents

Preface	xi
1. Introducing the PET	1
Communicating with the PET	1
ROM and RAM	2
BASIC Programming	3
PET Models	3
Features	5
2. Operating the PET	7
Unpacking	7
Cassette Hookup	8
Keytop Film Covering	9
Startup	10
Keyboard	11
Keyboard Entry	14
PET Key Groups	15
Screen Display	32
Cassette Tape Drive	33
Cassette Operation	34
Cassette Tape Controls	36
Cleaning and Maintenance	38
Cleaning and Demagnetizing the Tape Head	39
Care of Cassette Tapes	40
Elementary Troubleshooting	42

3. Programming the PET	47
Calculator or Immediate Mode	47
A One-Line Program	49
Reexecuting in Immediate Mode	51
Modifying a Program	52
Elements	53
Numbers	53
Strings	57
Variables	58
Operators	62
Arrays	69
Functions	71
File Names	74
Writing BASIC Programs	76
Line Numbers	76
BASIC Statement Groups	78
Developing a Program	84
Modification Example Tasks	107
4. PET BASIC	111
Format Conventions	112
BASIC Commands	112
CLR	112
CONT	112
LIST	113
LOAD	114
NEW	117
RUN	117
SAVE	119
VERIFY	120
BASIC Statements	123
CLOSE	123
CMD	124
DATA	125
DIM	126
END	127
FOR. . . NEXT [STEP]	127
GET	129
GET #	130
GOSUB	131
GOTO	131
IF. . . THEN	131
INPUT	133
INPUT #	134
LET. . . =	135
ON. . . GOSUB	136
ON. . . GOTO	136
OPEN	137
POKE	143
PRINT	143
PRINT #	145
READ	146
REM	147
RESTORE	148

4.	(Continued)	
	RETURN	148
	STOP	148
	WAIT	149
	BASIC Arithmetic Functions	150
	ABS	150
	ATN	150
	COS	150
	EXP	151
	INT	151
	LOG	152
	RND	153
	SGN	154
	SIN	154
	SQR	155
	TAN	155
	BASIC String Functions	156
	ASC	156
	CHR\$	156
	LEFT\$	156
	LEN	157
	MID\$	157
	RIGHT\$	158
	STR\$	158
	VAL	158
	BASIC Format Functions	159
	POS	159
	SPC	159
	TAB	160
	BASIC System Functions	161
	FRE	161
	PEEK	162
	ST	162
	SYS	163
	TI, TI\$	163
	USR	163
	User-Defined Function	164
	DEF FN	164
5.	Making the Most of PET Features	167
	Keyboard Rollover	167
	Buffer	174
	Two-Key Repetition	177
	Strings	180
	Concatenating Strings	180
	Numeric Strings	182
	Programmed Cursor Movement	185
	The PRINT Statement	185
	The CHR\$ Function	188
	Arithmetic	189
	Addition	190
	Subtraction	202
	Multiplication	215

5. (Continued)	
Graphics	225
Graphics in Calculator Mode	225
Programming Graphics	229
Summary	231
Animation	232
Time Delay	232
Programming Character Placement	233
Summary	237
Files	237
Program Files	239
Data Files	239
File Handling	240
Writing a Data File	242
Reading a Data File	255
Alternate Character Set	273
Features of the Alternate Character Set	277
BASIC Word Abbreviations	278
Setting the Clock	281
Digital Display Clock	285
POKE to the Screen	288
Calculating the Screen Address	288
Generating Random Numbers	291
6. System Information	303
Organization of the PET System	303
Memory Map	308
PET BASIC Interpreter	310
BASIC Statement Storage	312
User Program Area Initialization	315
Data Formats	317
Array Header	322
Assembly Language Programming	329
Appendices	
A. PET ASCII Codes	347
B. PET Error Messages	361
Operating System Error Messages	366
C. Program Examples Solved	369
D. BASIC Bibliography	387
E. PET Newsletters and References	389
F. Conversion Tables	393
Hexadecimal-Decimal Integer Conversion	394
Powers of Two	400
Mathematical Constants	400
Powers of Sixteen	401
Powers of Ten	401
G. Variations for Original ROMs (Revision Level 2)	403
H. PET Features	425
Index	427

FIGURES

1-1	Three Models of the Commodore PET	4
2-1a	Front View of Compact Keyboard PET	8
2-1b	Front View of Full Size Keyboard PET	9
2-2	Back View of PET	10
2-3	Full Size PET Keyboard	12
2-4	Compact PET Keyboard	13
2-5	The PET Video Screen	32
2-6	The 8x8 Dot Matrix	33
2-7	PET Cassette Tape Drive	34
2-8	Inserting a Tape Cassette	35
2-9	PET Cassette Tape Head	39
2-10	A Typical Tape Head Demagnetizer	40
2-11	Write Protect Notches	41
2-12	PET Internal View	43
2-13	Memory Device Unconnected	44
2-14	Extractor Tool	45
2-15	RAM Numbering	46
4-1	Logical Files	139
5-1	Odd/Even Division of PET Keyboard	170
5-2	Odd/Even Three-key Rollover	171
5-3	Draw the Square	228
5-4	Make Program Statement from Graphics	230
5-5	Using PRINT #	244
5-6	MAIL. PRINT #	252
5-7	Read a Data File	256
5-8	MAIL. INPUT #	264
5-9	Format Printing Using GET #	272
5-10	Standard/Alternate Character Set Decision	274
5-11	Standard to Alternate Character Set	276
6-1	PET Block Diagram	304
6-2	Principal Pointers in User Program Area	311
6-3	BASIC Statement Storage	312
6-4	User Program Area on Power-up	316

TABLES

2-1	Special Symbols as Program Operators	19
2-2	Graphic Character Keys	21
3-1	Special String Symbols	58
3-2	Reserved Words	62
3-3	Boolean Truth Table	67
3-4	Operators	69
4-1	Physical Device Numbers	140
4-2	Secondary Address Codes	141
4-3	ST Values for Tape I/O	162
5-1	Addition Operations	181
5-2	Keyword Abbreviations	280
6-1	PET Memory Allocation by 4K Blocks	304
6-2	PET Memory Map (Rev. 3 ROMs)	334
6-3	PET BASIC Keyword Codes	313
6-4	ASCII Standard 7-Bit Codes	327
6-5	PET Screen Memory 7-Bit Codes	328
G-1	PET Memory Map (Rev. 2 ROMs)	414

Preface

The personal computer field was already brimming with excitement when the Commodore PET® entered the scene in late 1977. A minimum PET system, housed in a self-contained unit with custom keyboard, video display, and tape cassette, cost just \$595.

The PET was an immediate sales success. Many thousands of units have been sold, and PET user groups are springing up all over.

The PET has surely become famous as one of the first popular personal computers. It has also become widely known — indeed infamous — as having the most meager literature a user has ever had to face. That is why this book has been written.

After reading this book you will have a good understanding of what a personal computer — especially the PET — can be expected to do for you. Certainly as important as knowing what you can do with a personal computer is knowing what you can't do. By examining the capabilities and limitations of the PET, you will have a grasp of what you can realistically expect a personal computer to do for you — which is important if you have a personal computer, or you are thinking of getting one.

There are numerous programming examples throughout this book; by reproducing these examples you will quickly get your PET working. These examples are thoroughly described and documented so you can learn how and why the

programs work. It is the “how” and the “why” that are important if you want to learn how to make your PET work efficiently for you.

This book is a complete guide to the use of the PET computer.

Chapter 1 is a general introduction to the personal computer and BASIC language.

Chapter 2 describes how to operate the PET keyboard, display screen, and magnetic tape cassette unit; it also describes rudimentary troubleshooting techniques.

Chapter 3 introduces you to BASIC, the programming language of the PET.

Chapter 4 is an alphabetical reference of PET BASIC commands, statements, and functions.

Chapter 5 describes “features” and programming quirks of the PET — and how to get around limitations of PET BASIC. You will find this chapter highly useful in understanding your PET and how to really go about programming it.

Chapter 6 contains useful system information and describes such techniques as how to execute assembly language programs under PET BASIC.

A summary of reference material is contained in appendices to the book.

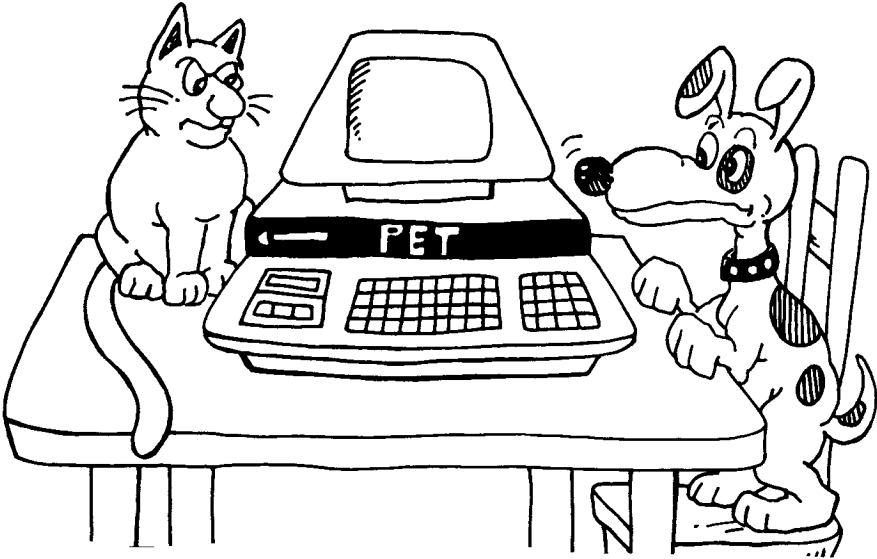
The chapters are divided, roughly, into three technical levels, as follows:

1. If you are a novice who wishes to use pre-written programs, you can get by with the loading and editing techniques described in Chapter 2.

If you find using the PET to be a lot of fun at this stage, and you want to learn more about computers, a recommended book is *An Introduction to Microcomputers: Volume 0 — The Beginner's Book*, Osborne/McGraw-Hill, 2nd Edition, 1979.

2. If you are a beginning programmer, and you want to write your own programs in PET BASIC, then Chapter 3 will get you started. In addition, detailed descriptions of individual BASIC statements are given in Chapter 4.
3. You will probably find that your programs occasionally do odd things. These peculiarities are accounted for in Chapter 5 — necessary reading for the advanced programmer. In Chapter 5 you will find coverage of topics such as Extended Precision Arithmetic and File I/O. The system information in Chapter 6 is written for experienced programmers who need to know about PET's memory layout, for example, so you can write assembly language subroutines to be run under PET BASIC.

Interfacing the PET to external devices is such a large topic in itself that a separate book is devoted to it. This Personal Computer Guide concerns itself only with the PET keyboard, display screen, and magnetic tape cassette unit, the latter either built-in or provided externally. See the references listed in Appendix E for sources of interfacing information on other peripheral input/output devices available for the PET.



"There's a new PET in the House"

ACKNOWLEDGMENTS

The name PET is a registered trademark with regard to any computer product, and is owned by Commodore Business Machines, a division of Commodore International. Permission to use the trademark name PET has been granted by Commodore Business Machines, and is gratefully acknowledged. Special thanks are due Mr. Dennis Barnhart, Mr. Lawrence Perry, and Mr. Robert Leman of Commodore Business Machines, Santa Clara, California for the valuable assistance and support they provided. Unless otherwise indicated, photos are reprinted with the permission of Commodore Business Machines. Use of the Commodore Model 2022 printer was also provided by Commodore.

The Digital Display Clock program listed in Chapter 5 is included courtesy of Mr. Patrick L. McGuire, Leland Enterprises, Oakland, California. The authors wish to extend their appreciation to Mr. McGuire for his contribution.

Thanks are also extended to Mr. Jim Butterfield of Toronto, Canada, for so readily sharing his memory map information. Any errors in the memory map tables in this book remain, of course, the responsibility of the authors.

HOW THIS BOOK HAS BEEN PRINTED

Notice that the text in this book has been printed in **boldface type** and lightface type. The boldface type is used to highlight the key points discussed in this book. The lightface type expands on information presented in the previous boldface type. Therefore, read the lightface type if you wish to know more about a subject presented in boldface type.

CHAPTER 1

Introducing the PET

The Commodore PET is a low-cost personal computer housed in a self-contained unit, with keyboard, video display screen, and built-in or external magnetic tape cassette unit. The PET is portable and can sit on a desk or tabletop. Moreover, the PET operates on normal household current and uses high quality, but still inexpensive, magnetic tape cassettes to store information in a computer-readable form. The PET is powered up simply by plugging it in and turning it on.

COMMUNICATING WITH THE PET

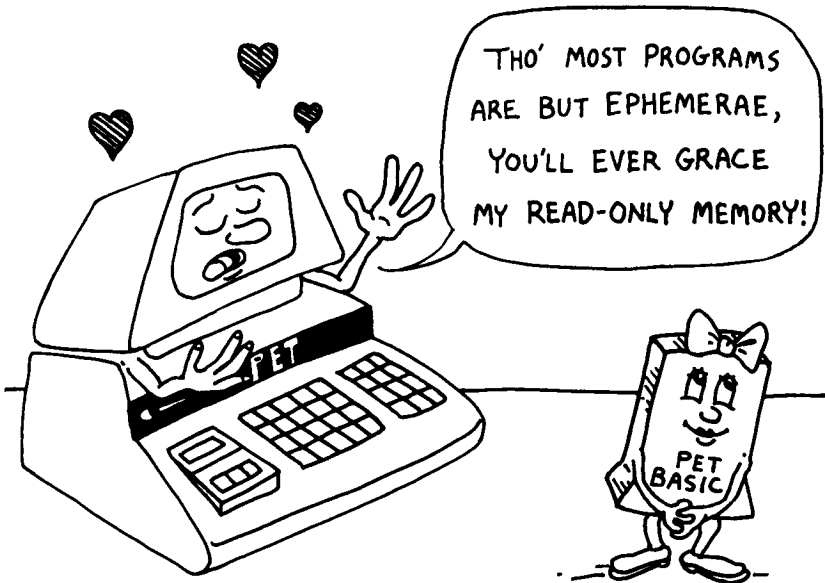
Whenever you power up the PET, you activate its BASIC* “interpreter.” The BASIC interpreter is a computer program that is permanently built into every PET. The PET BASIC interpreter carries out orders you give it by typing in commands at the PET keyboard. It allows you to start programming your PET as soon as you turn the power on.

You can command the PET to load a program from a tape cassette, list the program, execute the program, and perform a number of related chores. You can also type in your own programs at the PET keyboard, edit them, run them, and store them on tape cassettes.

*BASIC is a registered trademark of the Trustees of Dartmouth College.

ROM AND RAM

The BASIC interpreter is held in a type of memory called "Read-Only Memory" (otherwise known as ROM). ROM will keep its information forever, whether electric power is turned on or off. Read-Only Memory, and the information it holds, thus becomes a permanent part of your PET's electronics.



In contrast, a program loaded from a tape cassette, or input from the keyboard, is placed into Read/Write Memory (otherwise known as RAM). Any information stored in RAM is lost when the PET's power is turned off. That is why you need a cassette unit (or similar device) to store programs that you want to keep and use again and again.

BASIC PROGRAMMING

A host of pre-written programs is available on tape cassettes that you can load into memory and execute simply by pressing a couple of keys on the PET. Some of the programs that you can get are listed below. Some sources of pre-written PET programs are listed in Appendix E.

Address Book	ESP Test	Poker
Air-Sea War	Football	Pong
Alphabetizing	Foreign Language	Racetrack
Baseball	Golf	Recipe File
Battleship	Hangman	Rhyming
Biorhythm	Home Accounting System	Road Rally
Blockade	Hostage	Schedule Planner
Calendar	Investment Analysis	Spelling
Checkbook	Inventory	Star Trek
Check Writer	Kite Fight	State Capitols
Chemistry Drills	Life	Stock Portfolio Analysis
Computer Art	Lunar Lander	Tic-tac-toe
Craps	Maze	Unlist
Cryptograms	Memory Test	Vocabulary
Depth Charge	Othello	Wallpaper
Dogfight	Personal Ledger	Wumpus
Editor	Poetry	Yahtzee

In addition to buying programs, you can also write your own programs in the BASIC language, to be developed and executed on the PET.

“BASIC”, Beginners All-purpose Symbolic Instruction Code, was developed at Dartmouth College in 1963. It is one of the most popular computer languages, particularly favored by beginning programmers because it is easy to learn and easy to use. PET BASIC is a variation of Dartmouth BASIC.

A word of caution here: if you have never programmed before, do not be misled into thinking that with your personal computer you will have your menus, personal calendar, Esperanto concordance, and FBI agent file all safely stored and working in a few hours! There are real limitations to the PET — or any personal computer — as this book points out. Even with the best of languages and the most expensive of systems, writing useful, error-free programs is a real challenge. But if you have a bent for programming, and you will have a good idea whether you do or not after reading this book, you will find writing your own programs for the PET to be an exciting, rewarding pastime.

PET MODELS

The first PETs were available with a compact, custom graphics keyboard and built-in tape cassette unit, as shown in Figure 1-1a. Additional PET models are now available that have full-size keyboards. The tradeoff for these full-size keyboards is that the tape cassette unit must be connected externally. The full-size “graphics” keyboard is shown in Figure 1-1b. The CBM (Commodore Business Machine), a PET that has a full complement of small-business programs, and that also functions as a word processor, is shown in Figure 1-1c.

a. PET with compact graphics keyboard and built-in tape cassette unit



b. PET with full-size graphics keyboard



c. CBM, or Business PET



Figure 1-1. Three Models of the Commodore PET

FEATURES

All PET models are housed in very similar cabinets and have a similar set of specifications, as listed in Appendix H. **The only differences that you need to be concerned with in using this book are:**

1. **Whether your PET has the original ROMs, also called "old" ROMs, which are Revision Level 2; or the newer ROMs, which are Revision Level 3.** The newer ROMs eliminate problems that were encountered with the original ROMs.

You have Revision 2 ROMs if your PET powers up with the first line printed as:

```
***COMMODORE BASIC***
```

You have Revision 3 ROMs if your PET powers up with the first line printed as:

```
###COMMODORE BASIC###
```

If you have the old ROMs, read carefully Appendix G, "Variations for Original ROMs (Revision Level 2)." This appendix delineates every change you will have to make, chapter by chapter, to use this book with the old ROMs. The changes are not extensive. The rest of this book assumes that you have the newer ROMs.

2. **Whether your PET has the compact keyboard or one of the full-size keyboards.** This book is written for all PET keyboards, and differences, where they exist, are clearly spelled out.
3. **How much memory your PET has.** Your PET may have 4K, 8K, 16K, or 32K bytes of memory. This is Read/Write Memory (RAM); all PETs have the same amount of Read-Only Memory (ROM). A "K" is, roughly, one thousand. A "byte" is the standard unit of memory storage in the PET; each byte can hold one character of data.

The more memory your PET has, the longer the programs and the larger the amounts of data it can handle at one time. However, this is not an operational difference and is not important in learning how to use the PET. You only need to be aware of the memory size for the few references to the "top of memory," whose value will depend on how much RAM is implemented in your PET.

CHAPTER 2

Operating the PET

This chapter tells you how to operate the PET keyboard, use the screen display, and work with the magnetic tape cassette drive. After reading this chapter, you will be able to load and use pre-written PET programs stored on tape cassettes.

The primary external features of the PET are shown in Figures 2-1 and 2-2. The Model 2001-8 in Figure 2-1a has a compact keyboard and built-in tape cassette unit. The 2001-16, 2001-32, and CBM models have front panels as shown in Figure 2-1b with a full size keyboard and separate tape cassette unit. The back view is the same for all units and is shown in Figure 2-2.

UNPACKING

Assuming you have just brought your PET home in a box, you must unpack it. The PET is packed in protective material, inside a heavy-duty cardboard box. Inspect the box for any evidence of shipping damage. Carefully open the box and remove the PET. Place the PET on a sturdy, level surface such as a table or desk top. Inspect the unit for any evidence of shipping damage. Report any apparent damage to the shipping agent, or to the dealer from whom you purchased the PET. Read any special material enclosed in the box; be sure to complete the war-

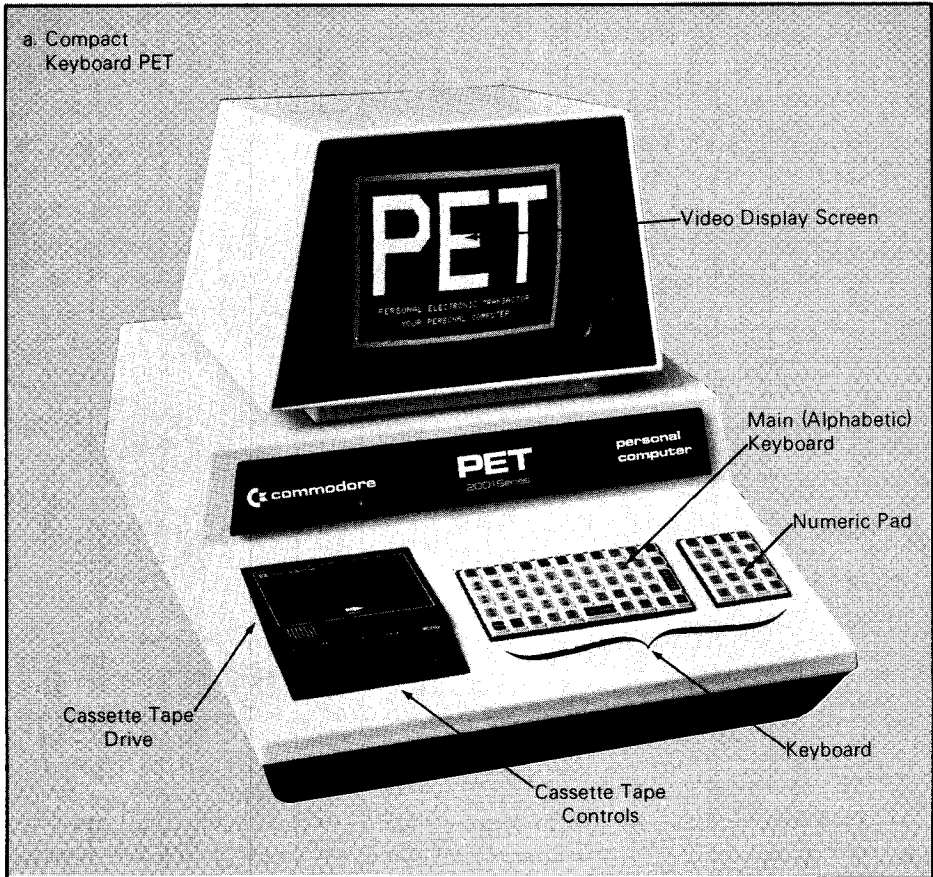


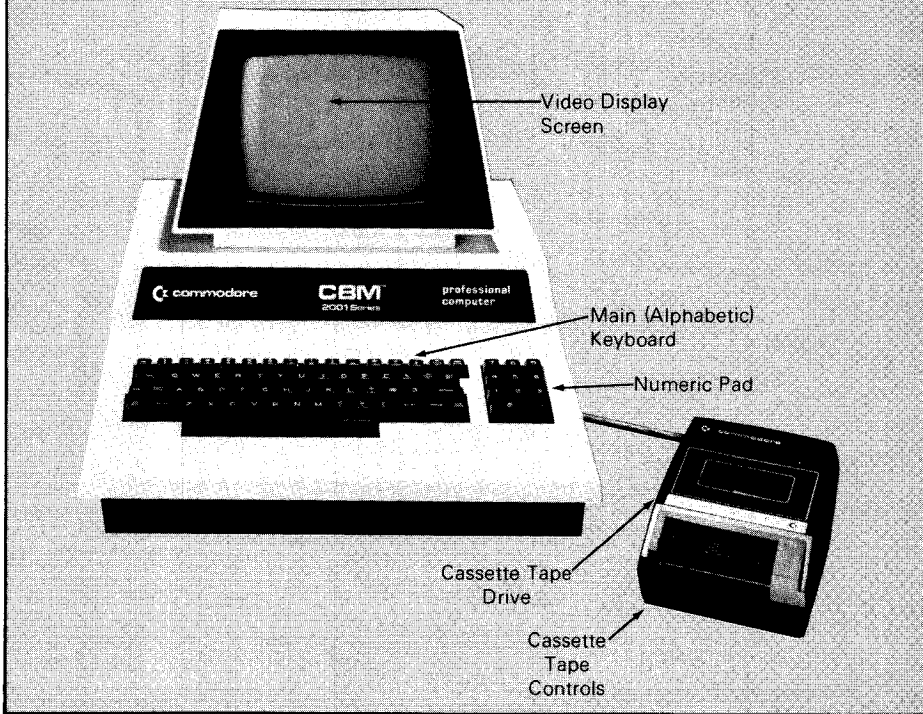
Figure 2-1a. Front View of Compact Keyboard PET

ranty information. Save the box and packing material — you may need to ship your PET back to the factory for repairs at some time, or transport it to other locations.

CASSETTE HOOKUP

On full size keyboard PETs, the cassette tape unit is separate and must be connected to the PET unit. Unpack the cassette unit and place it on the desk or table top within easy reach of the PET. The cassette unit terminal plug is tailor-made to fit the PET cassette port (2nd Cassette Interface J3 as shown in Figure 2-2). Connect the plug to J3 on the PET, securely seating it.

b. Full Size Keyboard PET



KEYTOP FILM COVERING

On a new PET with a compact keyboard, each key is covered with a protective plastic film which you must remove before using the keyboard. It is recommended that this film be removed by pressing a piece of masking tape over each key, sticky side down. Then lift up the masking tape, bringing the plastic film off the key with the tape. The keytop lettering can be scratched off, so be careful.

You can protect the keytops so that letters are not erased by applying a coat of clear nail polish or similar protective coating to the keytops.

This procedure is not required on the full size keyboard.

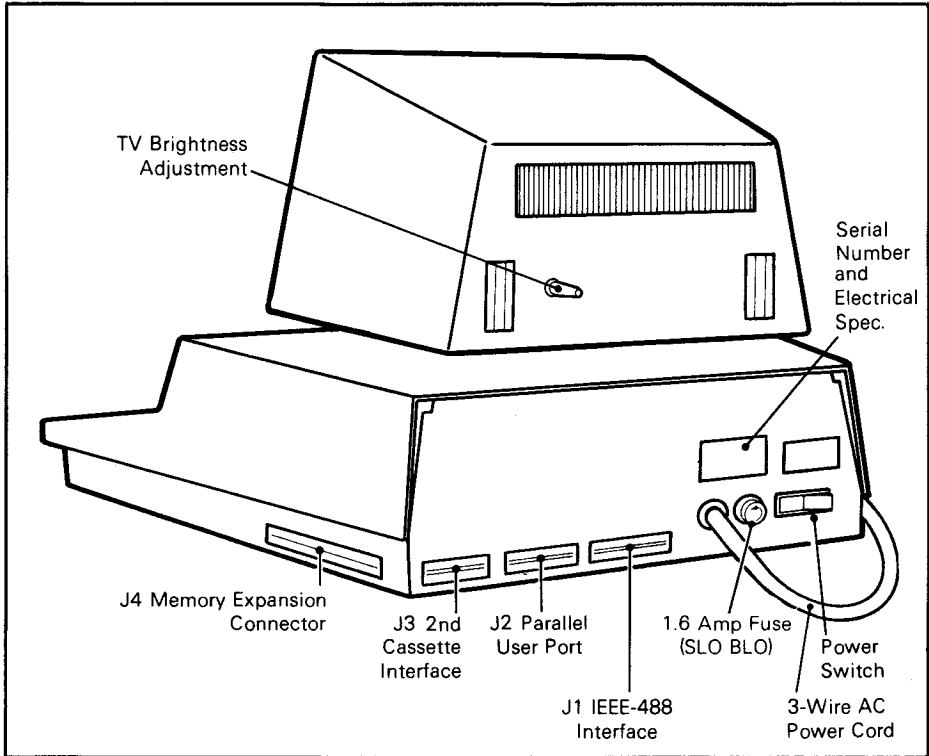


Figure 2-2. Back View of PET

STARTUP

1. **Plug the PET in.** Plug the three-prong power plug into a standard grounded electrical outlet. **DO NOT ATTEMPT TO PLUG THE CORD INTO A TWO-HOLE (UNGROUND) OUTLET. DO NOT ATTEMPT TO REMOVE THE GROUND PRONG.** If the unit is not properly grounded, you may receive an electrical shock. Grounding adapters that convert a two-prong outlet into a three-prong (grounded) outlet are available from your hardware store or electrical supply house.

CAUTION: Do not use a three-prong adapter unless you ground it properly when installing it.

2. **Switch power on.** The power switch is located on the back of the PET, at the left side (see Figure 2-2). It is a two-position "rocker" switch. Pressing on the outer side of the switch turns power on; pressing the inner side of the switch turns power off. A white dot shows on the side of the switch when power is on.

3. **Wait for READY display.** About three seconds after switching power on, the following message is displayed on the screen:

```
### COMMODORE BASIC ###  
XXXX BYTES FREE  
READY.  
⌘
```

The four lines of display have the following meanings:

###COMMODORE BASIC###	This line indicates that the PET BASIC interpreter has been activated.
XXXX BYTES FREE	This line shows how much memory is available to you. 3071 (or a similar number) will be displayed for a 4K PET system. 7167 (or a similar number) will be displayed for an 8K PET system. 15359 (or a similar number) for a 16K PET system. 31743 (or a similar number) for a 32K PET system.
READY.	The PET is ready to receive input from the keyboard.
⌘	The flashing cursor is displayed at the position on the screen where the next character typed in from the keyboard will appear. The cursor must be present in order to enter commands from the keyboard.

If you do not get the display illustrated above after turning power on, then turn power off, wait a few seconds, and turn power back on. The display will first be filled with random characters for a second or so. This is normal; just ignore it. The random character display will appear whenever the PET is turned off and then on again within about ten seconds.

KEYBOARD

The PET keyboard is used to enter programs and commands. The type of keyboard depends on which model PET you have. **The same keys are present on both compact and full size keyboards,** with the exception that the full size keyboard has a shift lock key. However, the locations of some of the keys are different.

The keyboard layouts are introduced separately below, followed by a common description of the individual keys. If you have the full size keyboard, read the description below and then skip to the section titled "PET Key Groups." If you have the compact keyboard, skip the next section and begin reading at the section titled "Compact Keyboard."

Full Size Keyboard

The full size "graphics" keyboard is illustrated in Figure 2-3. The CBM keyboard is very similar in layout, but the graphic symbols are not printed on the keys. If you have a CBM, use Figure 2-3 and Table 2-2 to locate graphic symbols on the CBM keyboard.

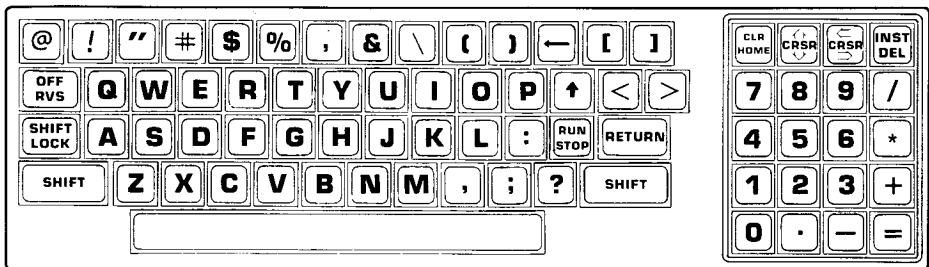


Figure 2-3. Full Size PET Keyboard

The keyboard consists of 74 keys, most of which have an upper shift and a lower shift. Two symbols or operations are shown on the same key. Most of the upper shift symbols are graphic keys. For these keys, the larger top imprint is for unshifted mode, and the graphic symbol is shown on the front side of the key. For non-graphic keys both operations are shown on the top imprint, the upper one being for shifted mode and the lower one for unshifted mode. SHIFT, SHIFT LOCK, the SPACE bar, and RETURN are exceptions.

The letters are arranged like those of a typewriter and, to the right, the numbers are organized into a numeric pad similar to that of an adding machine or calculator. (The number "5" on the CBM has a raised dot to locate it by finger touch.) Those familiar with a typewriter keyboard should feel comfortable with this full size PET keyboard. Where possible, the special symbols are placed where they normally reside on a typewriter keyboard.

One feature of the PET keyboard is that the letters (A-Z), numbers (0-9), and special symbols (*, %, #, etc.) are all typed in unshifted mode. These frequently used characters are thus readily accessible.

The PET has two character sets. The "standard" character set has capital (upper case) letters typed in unshifted mode, and the shifted mode prints the graphic characters. The "alternate" character set gives upper and lower case letters but eliminates some graphic characters. If you have the graphics keyboard, the standard character set is active whenever the PET is powered up. If you have the CBM, however, the alternate character set is active on power-up. The selection of standard or alternate character set is determined by the value contained in a particular system location that you can change. This feature is described in detail in Chapter 5. For now, if you have the CBM, type in the following statement:

```
POKE 59468,12
```

This activates the character set with graphics on CBM units. **Unless stated otherwise, this book assumes that the graphics character set is being used.**

In the following key descriptions, and throughout this book, individual keys from the keyboard are illustrated. To avoid confusion, we have used keys from just one type of keyboard — the compact keyboard. You will find these keys virtually identical to the legends on the full size keys. Thus you should have no trouble recognizing which key is intended on your full size keyboard when you see the corresponding key from the compact keyboard.

Compact Keyboard

The compact keyboard is illustrated in Figure 2-4.

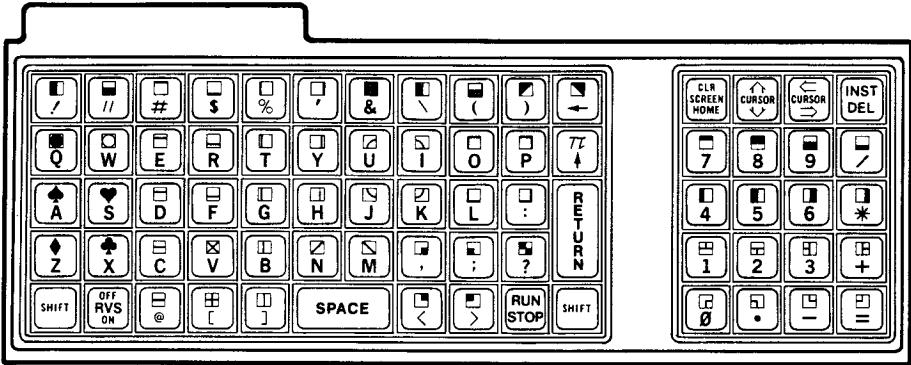


Figure 2-4. Compact PET Keyboard

The compact keyboard consists of 73 keys, most of which have an upper shift and a lower shift. Two symbols or operations are shown on the same key; the top one is for the shifted mode and the lower one is for the unshifted mode. SHIFT, SPACE, and RETURN are the exceptions.

The letters are arranged like those of a typewriter. To the right, numbers are organized into a numeric pad similar to that of an adding machine or a calculator. For those familiar with a typewriter keyboard, there are some real differences between the typewriter keyboard and the compact PET keyboard.:

1. This PET keyboard is more compact than a standard keyboard. Because of this, as well as the need to frequently access the numeric pad keys, you will find it difficult to "touch type" on the compact PET keyboard.
2. The standard character set has upper case letters only; they are typed in unshifted mode. The PET keyboard is arranged so that all letters, numbers, and special characters (+, -, %, etc.) are typed in unshifted mode so that these most frequently used characters are readily accessible.
3. There is no shift lock on the PET shift keys. To type upper case characters or controls, you need to hold down the shift key, while at the same time pressing the desired character or control key.

KEYBOARD ENTRY

Before proceeding with the individual key descriptions, we digress here for a moment to introduce several important concepts related to keyboard entry.

Perhaps the major difference between a PET keyboard and a typewriter keyboard results from the presence of the BASIC interpreter within the PET. The BASIC interpreter can process the same keystroke in completely different ways depending on the context of the keystroke.

When you press a typewriter key, you type a character — nothing more and nothing less. Under the correct circumstances, the PET, like a typewriter, will interpret a keystroke as one character of text. We refer to such text as a "string," because to the PET it **is a string of text characters.**

Variables

Under some circumstances the BASIC interpreter will treat a keystroke as part of a BASIC statement. Or the keystroke may become part of a "symbol" or variable name. We use the word "syntax" to describe the rules governing the form that BASIC statements must have and the way characters within the BASIC statement will be interpreted.

π is a "symbol": it represents the value 3.14159265, the circumference of a circle divided by its radius.

The concept of a "symbol" or "variable" is fundamental to all computer programs, whether they are written in BASIC or in any other programming language. You create variable names by grouping letters and/or other characters, according to specified rules. Each variable becomes an entity, capable of representing information. The information may be a number or a word of text. You may liken variables to letter boxes, each of which has a name, but can contain whatever you choose to put into it.

The information represented by a variable name may be a string of text, an integer, or a floating point (real) number. If the last character of the variable name is %, the variable can represent only integers. If the last character of the label is \$, the label can represent only strings. If the last character of the label is neither % nor \$, then the label can represent only floating point numbers.

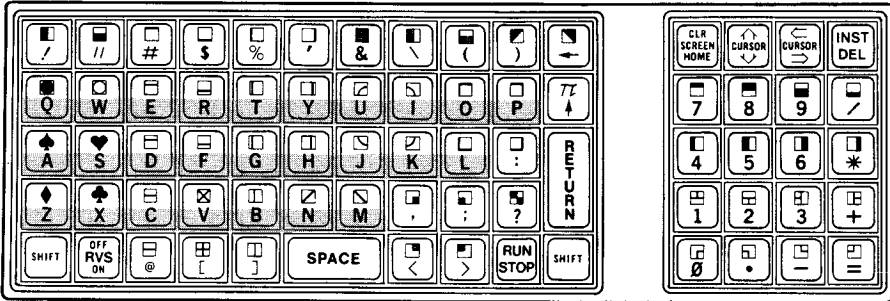
Until a program assigns a value to a variable name, the variable is assumed to have a value of 0 for a number, or no characters at all for a string.

PET KEY GROUPS

The PET keys can be divided into the following groups:

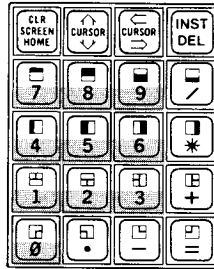
- Alphabetic keys
- Numeric keys
- Special symbols
- Graphic keys
- Function keys
- Cursor control keys

Alphabetic Keys



The alphabetic keys provide the 26 letters of the alphabet, A to Z. These are upper case (capital) letters when used with full graphics. A lower case alphabet is available with the alternate character set (see Chapter 5).

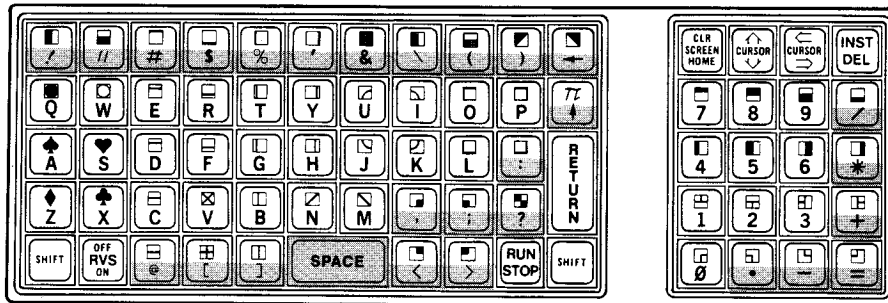
Numeric Keys



The numeric keys provide the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. All numbers input to PET BASIC are decimal numbers (the kind you use every day) such as 3, -14, 10.5.

The π key (upper shift of π) could also be considered a numeric key (or a special symbol), but since it represents a multidigit number and has some special characteristics, it is described below under function keys.

Special Symbols



Special symbols, or special characters, have two general uses:

1. They may **represent standard punctuation marks or commonly used symbols**. For example, there is a period, a comma, arithmetic operations (+ for addition, – for subtraction, etc.), and other characters that have widely recognized interpretations, such as "\$" for dollar sign.
2. These same characters may at other times **represent a specific operation or be part of a BASIC statement**. Remember, we use the word "syntax" to describe the special rules governing the characters that must appear in a BASIC statement and the order in which they must appear.

Thus, the same symbol means different things when used in different contexts. Some English words, called homonyms, also have this property. For example, pool can be a pool of water or a game of pool.

Consider the following PET BASIC statement:

```
IF A=1 THEN B=2
```

This BASIC statement uses the equal sign (=) twice, apparently with the same meaning in each case. But these two equal signs actually mean different things to PET BASIC. The first equal sign, in the phrase IF A=1, has a standard meaning of equality; PET BASIC must determine whether A has the current value of 1. In the second phrase, THEN B=2, however, the equal sign is interpreted as "assign B the value of 2, regardless of what its current value is." These equal signs are called "program operators." A program operator is a special symbol that tells PET BASIC to perform a certain operation.

Note that "A" and "B" have been used as variables; each is a "name" assigned to a "letter box." The BASIC statement above checks to see if letter box "A" contains 1; if it does, 2 gets put into letter box "B".

Special symbols can also be used as data characters, just like letters of the alphabet. Consider the following examples:

Statement 1: PRINT "HI FRED, MARY, AND AL"

Statement 2: PRINT FR, MA, AL

In the first statement, the commas are ordinary punctuation marks; they are data characters just like all the letters (and spaces) inside the quotation marks. In fact, every character occurring between the quotation marks is a simple text character, which the BASIC interpreter treats as a text string, just as a typewriter would treat a keystroke. In statement 2, however, the commas are program operators; they are telling PET BASIC to use a special form of tabular alignment when printing out values assigned to variables FR, MA, and AL.

Table 2-1 summarizes the meanings of the special symbols when they are used as program operators. These usages are described in detail in Chapter 3.

Some of the special symbols do not have a program operator function. Such symbols are merely graphic characters: they are used to create pictures on the display screen. Some program operators are formed by combining special symbols and/or letters of the alphabet just as you combine letters into words. We will describe these program operators later.

Special symbols and letters of the alphabet can have different meanings, depending on where they appear in a BASIC statement. The word PRINT, for example, is the part of a BASIC statement that tells the PET to print something. If the same word were enclosed in quotes, it would be interpreted as a word of text. Consider the at-first confusing statement:

```
PRINT "PRINT"
```

The first PRINT is a command to print, but the second PRINT is a word of text, or a string, because it is enclosed in quotes. What would this statement do? You have probably already guessed — it would print the word PRINT. **(The shaded portions in the screen examples show what you type in; the unshaded portion is the PET's response):**

```
PRINT"PRINT"  
PRINT
```

PET BASIC doesn't just have homonyms — the same word having different meanings; it also has synonyms — different words that have the same meaning. For example, the command PRINT can be abbreviated to a question mark.

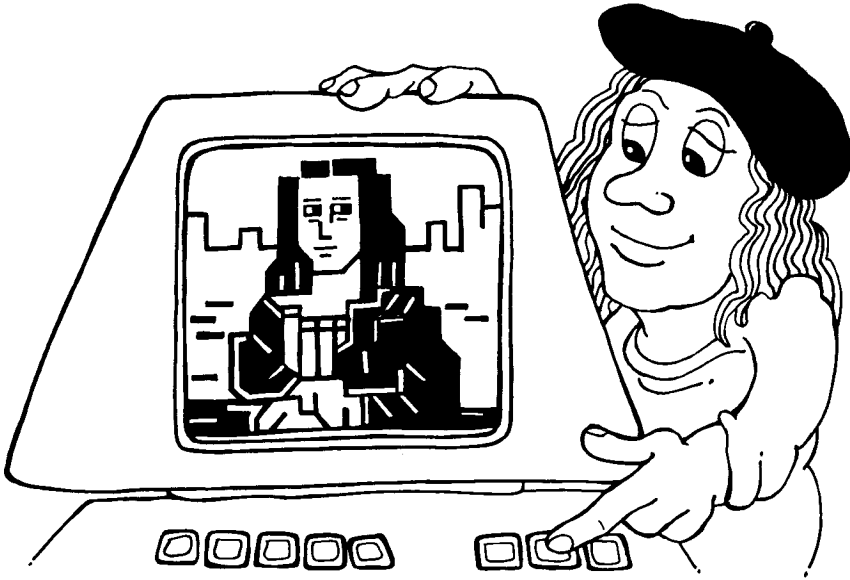
```
? "PRINT"  
PRINT
```

Table 2-1. Special Symbols as Program Operators

Symbol	Description	Program Operator Use	Example
	Space	To improve program legibility; disregarded by PET BASIC except in strings	100 READ Y ↑ ↑
!	Exclamation mark	(none)	
"	Double quotation mark	Begins and ends a string	"STRING"
#	Pound sign	(none)	
\$	Dollar sign	Denotes string variable	C\$
%	Percent sign	Denotes integer variable	C%
&	Ampersand	(none)	
'	Single quotation mark	(none)	
()	Parentheses	1. Enclose array subscripts 2. Enclose function arguments 3. Denote expression evaluation precedence	AR(1,5) SQR(114) ((A+2)/5)12
*	Asterisk	Multiplication	4*16
+	Plus sign	Addition, positive number	2+2 +57
,	Comma	1. Separates items in a list 2. Denotes tabbed format	READ A,B,C PRINT X,Y,Z
-	Minus sign	Subtraction, negative number	14-6 -32
.	Period	Decimal point	65.02
/	Slash	Division	14/2
:	Colon	Separates multiple statements on a line	PRINT A:GOTO 10
;	Semicolon	Denotes continuous-line format	PRINT A;B;C
<	Left carat	Less than	A < 100
=	Equal sign	1. Equal to 2. Replaces	IF A=B C=600
>	Right carat	Greater than	C > D
?	Question mark	Abbreviation for PRINT	? "HII"
@	At sign	(none)	
[]	Brackets	(none)	
\	Backslash	(none)	
↑	Up arrow	Exponentiation	A12
←	Left arrow	(none)	

Graphic Keys

The PET keyboard contains 62 graphic keys. All are the shifted mode of data keys. With so many graphic characters available on the PET, you can create some rather sophisticated display drawings.



The graphic characters are listed in Table 2-2; each is given a name. They are grouped by similarity of line, so that you can get some idea of the sets of design components available when creating graphic displays. The entire key is shown in the table to help you locate keys. **Note that the square enclosing the graphic symbol is not part of the symbol; it is used merely to show the symbol's location within a grid space.**

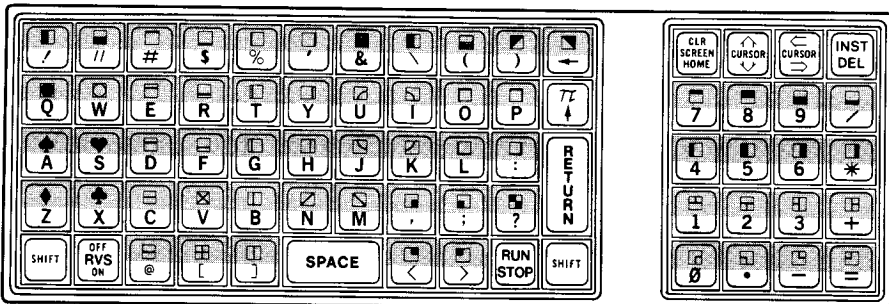
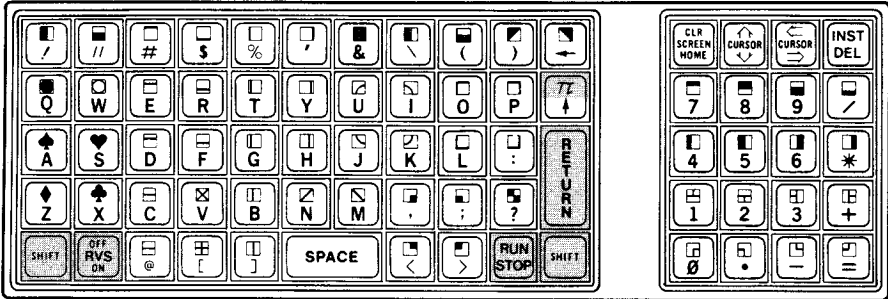


Table 2-2. Graphic Character Keys

Line Horizontal	Thin Bar	Quarter Block Solid	T
Top	Top	Top Left, Top Right	Top
3/4 Top	Bottom	Bottom Left, Bottom Right	Bottom
2/3 Top	Left	Diagonal	Left
Middle	Right	Quarter Block Open (Angle)	Right
Near Middle	Thick Bar	Top Left, Top Right	Symbol
2/3 Bottom	Top	Bottom Left, Bottom Right	X
3/4 Bottom	Bottom	Corner	Cross
Bottom	Left	Top Left, Top Right	Diagonal Acute
Line Vertical	Right	Bottom Left, Bottom Right	Diagonal Grave
Left		Rounded Corner	Grid
3/4 Left	Half Block	Top Left, Top Right	Full
2/3 Left	Left	Bottom Left, Bottom Right	Half Left
Near Middle	Bottom		Half Bottom
Middle	Triangle Solid	Suit	Circle
2/3 Right	Top Left	Spade, Heart	Solid
3/4 Right	Top Right	Diamond, Club	Outline
Right			

Function Keys



SHIFT. The SHIFT key is pressed simultaneously with any of the other keys to give the shifted character of that key. All PET keys have different characters for shifted and unshifted modes, except the SPACE and RETURN keys. The SPACE key generates a blank or space in either mode. The RETURN key provides a carriage return in either mode. On all two-character keys, the lower symbol is the unshifted one and the upper symbol is the shifted one.

There are two identical SHIFT keys located on the main keyboard, one at the lower left and one at the lower right.

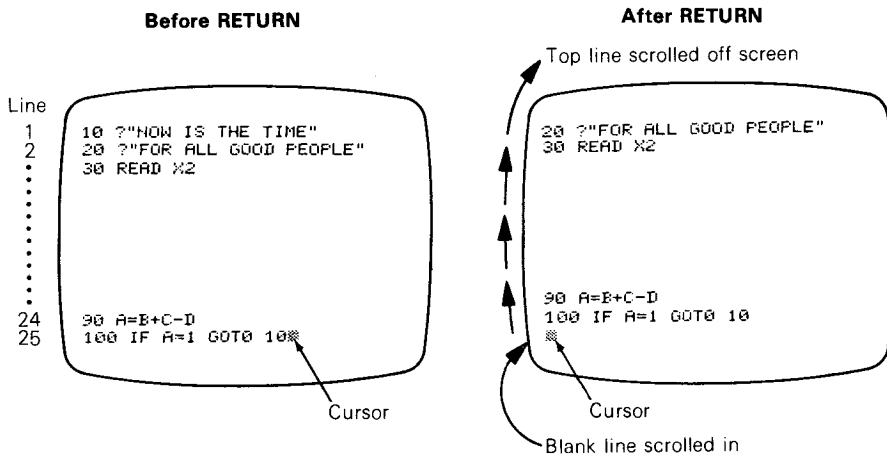
SHIFT LOCK (full size keyboard only). Full size PET keyboards have a SHIFT LOCK key located directly above the left-hand SHIFT key. Pressing the SHIFT LOCK key until it "clicks" into place holds SHIFT down so that both hands are free to type in shifted mode. To release the SHIFT LOCK key, press and release the SHIFT LOCK key; both SHIFT keys return up to their unshifted position.

RETURN. The RETURN key is the equivalent of a carriage return on a typewriter; when depressed, it **causes the cursor to move to the beginning of the next line on the screen.**

When you enter a BASIC statement, the statement line can have up to 40 characters (one display line) or up to 80 characters (two display lines). When you are first entering a statement line, you type in the characters and terminate the line by pressing the RETURN key. If you type in 40 or more characters without pressing RETURN, the cursor automatically drops down to the next display line, and you can continue entering up to 80 characters. If you continue typing onto a third line, however, the PET will give you a SYNTAX ERROR as soon as you press RETURN.

You must depress the RETURN key in order to terminate every line of BASIC input. The PET uses the RETURN character as a signal that the current line is complete and ready to analyze. The "current" line is defined as the line on which the cursor is located, regardless of how many lines may be displayed on the screen.

A RETURN given anywhere on the last line of the screen causes all of the screen text to move, or scroll, up one line. The top line rolls up off the screen and a blank line rolls into the bottom line of the screen, with the cursor left at the beginning of the blank line.



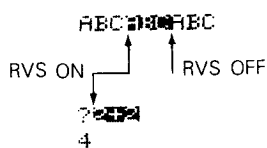
PI (π). Pi (π) is a circle's circumference divided by its diameter. π has its own key on the PET computer. When this symbol is used, alone or in an expression, PET replaces it with the value 3.14159265.

```
?π
3.14159265
```

The only time π is not evaluated as its preassigned number is when it appears within quotation marks. Then it is treated as a graphic character — a part of a string.

```
? "π"
π
```

REVERSE ON/OFF. The Reverse key allows you to reverse the black and white parts of characters; REVERSE is like a photograph and its negative. The normal mode of this key is "off." To activate the REVERSE key, press it in unshifted mode. The next character keys you press will be displayed in reverse field. REVERSE ON stays in effect until you either press REVERSE OFF (the REVERSE key shifted) or press the RETURN key.



Note: Reverse field terminated by carriage return

You can also program the REVERSE key inside quotation marks. In this case, the reverse field is not shown until the display is printed.

```

      RVS ON   RVS OFF
      ↓       ↓
? "ABC" ABC ABC
ABC ABC ABC
  
```

The REVERSE key has another function unrelated to reverse fields: depressing the REVERSE key during rapid scrolling slows down the scroll speed.

RUN/STOP. STOP is the unshifted half of the RUN/STOP key. **STOP stops execution of any current operation** and puts you back in touch with PET BASIC, reconnecting it with the keyboard.

```

FOR I=1 TO 100 :?I NEXT I
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 ← Press STOP key

BREAK
READY.
*
  
```

If you stop a program in mid-execution, you will get a message telling you where in the program you stopped. "Where" becomes a number which identifies the last BASIC statement that was executed. This may be illustrated as follows:

```

100 FOR I=1 TO 100
110 ?I
120 NEXT I
RUN
1
2
3
4
5
6
7
8
9
10
11
12 ← Press STOP key

BREAK IN 110
READY.

```

The STOP key does nothing if there is nothing to stop, i.e., the PET is READY.

RUN is the shifted half of the RUN/STOP key. **RUN causes the next program on the cassette unit to be located, loaded into memory, and then immediately executed.** The display below shows what you will see if you "RUN" the small program illustrated above.

```

READY.
← Press RUN key. Cursor is replaced by:
LOAD

PRESS PLAY ON TAPE #1 ← Press PLAY
OK

SEARCHING
FOUND

PROGRAM 1 TO 10
1
2
3
4
5
6
7
8 ← STOP or RUN here causes a break
} Running program printout

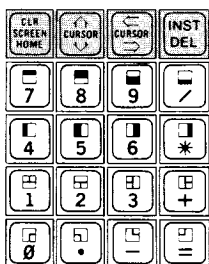
BREAK IN 110
READY.

```

To perform its "load & go" function, RUN must be pressed when the PET is READY; if it is pressed while another program is executing, it acts like a STOP. Using the RUN key is a quick way of running the next program stored on the cassette you have in the PET cassette drive.

Cursor Control Keys

The remaining four function keys are cursor control and edit keys. They are grouped together as the top row on the numeric pad.



CLEAR SCREEN/HOME. HOME is a cursor control key that moves the cursor to its "home" position, the upper left-hand corner of the screen. Any subsequent keyboard entry will be displayed at this position. You "home" the cursor by pressing the CLEAR SCREEN/HOME key in unshifted mode.

Before HOME

```

60 QT=QT+A
70 NEXT X
80 CLOSE1
90 END
1000 POKE 59411,53
1010 T=TI
1020 IF <TI-T><10 GOTO 1020
1030 POKE 59411,61
1040 QT=0 *
    
```

Cursor here

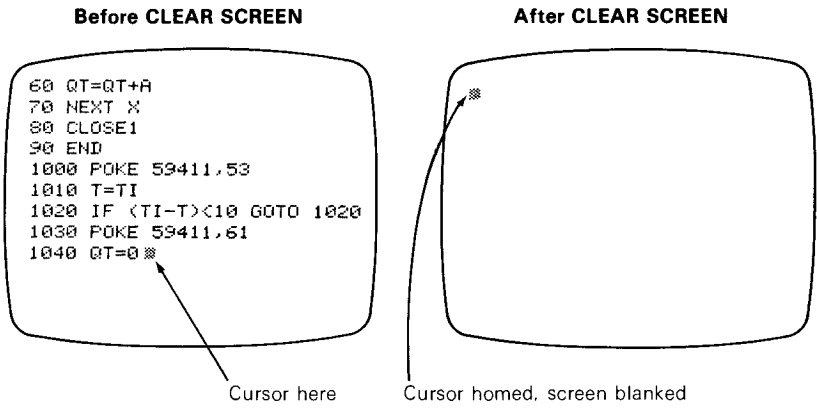
After HOME

```

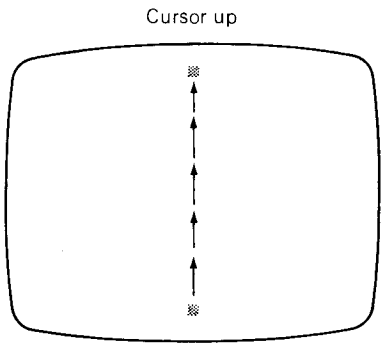
60 QT=QT+A
70 NEXT X
80 CLOSE1
90 END
1000 POKE 59411,53
1010 T=TI
1020 IF <TI-T><10 GOTO 1020
1030 POKE 59411,61
1040 QT=0
    
```

Cursor homed

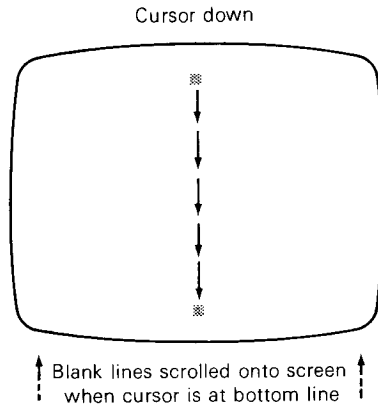
CLEAR SCREEN, obtained by pressing the CLEAR SCREEN/HOME key in shifted mode, **not only homes the cursor but also clears, or blanks, the entire display screen.**



CURSOR UP/DOWN. CURSOR UP, obtained by pressing this key in shifted mode, **moves the cursor up one line within the same physical column of the screen.** When the cursor is at the top line of the screen, CURSOR UP has no effect. The cursor moves over characters without changing them.

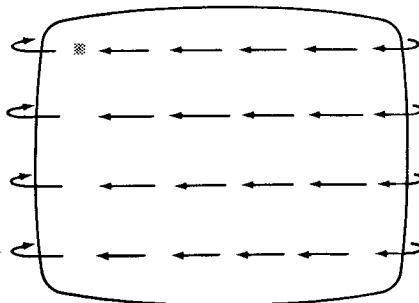


CURSOR DOWN, obtained by pressing this key in unshifted mode, **moves the cursor down one column.** Unlike CURSOR UP, CURSOR DOWN can move the cursor down “infinitely”; if the cursor is at the bottom line of the screen and you press the CURSOR DOWN key, the cursor itself is not moved, but the rest of the screen is moved, or scrolled, up one line. This has the net effect of moving the cursor down one line to a blank line.

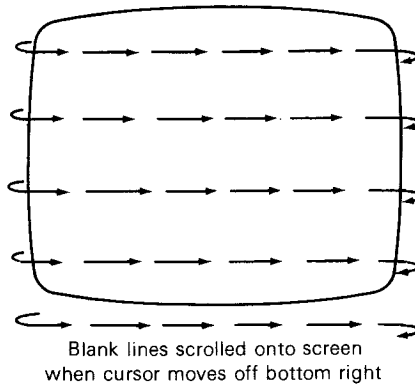


If you rapidly press and re-press the cursor control key as fast as you can, the cursor moves several positions between cursor blinks; this is because cursor movements can be processed faster than cursor blinking. But each key depression in fact moves the cursor one position, even when this is not reflected by the cursor blinking.

CURSOR LEFT/RIGHT. CURSOR LEFT, obtained by pressing this key in shifted mode, **moves the cursor left one position within the same horizontal row.** Multiple pressing of the CURSOR LEFT key rapidly moves the cursor left. When the cursor is at the beginning of a row, pressing the CURSOR LEFT key moves the cursor up one row and to the extreme right-hand end of this row. But the cursor stops at the home position; pressing the CURSOR LEFT key has no effect if the cursor is in its home position. The effect of the the CURSOR LEFT key may be illustrated as follows:



CURSOR RIGHT, obtained by pressing this key in unshifted mode, **moves the cursor right one character position within the same row.** As with CURSOR LEFT, CURSOR RIGHT wraps the cursor around the screen. But CURSOR RIGHT moves the cursor to the beginning of the next line down when the cursor is at the extreme right-hand end of a line. When the cursor reaches the end of the bottom-most row on the screen, the screen scrolls up one line so that the cursor appears at the beginning of the next, blank line. Scrolling continues indefinitely until all text has scrolled off the top of the screen; then the cursor appears to move to the right circularly along the bottom row of the screen.



You use CURSOR LEFT/RIGHT to type over text while editing it. This is equivalent to backspacing and striking over on a typewriter; but PET strikeover replaces the previous character, rather than trying to blot it out. Suppose you have typed:

```
NOW IS TEE TIM
```

and you want to change the first E in TEE to an H. Press CURSOR LEFT until the cursor is on the letter to be corrected:

```
NOW IS TEE TI
NOW IS TEE T
NOW IS TEE
NOW IS TEE
NOW IS TEE
NOW IS TEE
NOW IS TEE
```

Now press H:

```
NOW IS TH TIM
```

The letter H appears where the first E stood, and the cursor moves to the next character position. Now press CURSOR RIGHT (NOT the space bar!) to get back to where you were.

INSERT/DELETE. The INSERT/DELETE key in unshifted mode is **DELETE**. It **deletes the character to the immediate left of the cursor and moves the rest of the line, headed by the cursor, one character position to the left.** To use DELETE, press CURSOR RIGHT or other cursor control keys to place the cursor on the character to the right of the character or characters to be deleted. Then press the DELETE key as many times as needed; each time you will remove one character.

We saw how typeover can be used to correct the following line:

```
NOW IS TEE TIM
```

Alternatively, the DELETE key can be used to make the correction. To do so, press the DELETE key six times; this blanks the line past the T:

```
NOW IS TEE TI
NOW IS TEE T
NOW IS TEE
NOW IS TEE
NOW IS TE
NOW IS T
```

Now type in the correction, and the rest of the line, up to the original point:

```
NOW IS THE TIME
```

The DELETE key can be especially handy when you are initially entering text and you spot an error a few characters in back of the cursor position. If the error is many characters back, you may be better off using CURSOR LEFT, so that you will not have to retype the line, with the possibility of introducing new errors.

INSERT, the INSERT/DELETE key in shifted mode, **opens a single character space in the line at the current cursor position.** You can then insert an additional character into the space. To use INSERT, move the cursor to the character position where you want an insertion to begin. The cursor will flash on the character that will ultimately become the first character beyond the insertion. Press the INSERT key once for each character to be inserted; with each depression of the INSERT key the rest of the line is moved one character position to the right, and one character space appears. Type in the characters being inserted into the space created. You should press as many character keys, including spaces, as you did INSERT keys.

Suppose you want to do the following:

```
NOW IS THE TIM
      NOT
```

Press CURSOR LEFT repeatedly until the cursor is on the T of THE:

```
NOW IS THE TIME
```

Press the INSERT key four times to create spaces for the word NOT plus a space. The cursor remains at the beginning of the insert area:

```
NOW IS#THE TIME
NOW IS# THE TIME
NOW IS#  THE TIME
NOW IS#   THE TIME
```

Enter the word NOT, and a space:

```
NOW IS NOT #HE TIME
```

Use CURSOR RIGHT to move the cursor to the right of TIME if you want to continue entering data.

INSERT can move text to the end of a statement line (up to two display lines), but it cannot move text past the end of the statement line onto a third line. If there is a non-blank character at the last position on the line, or if there are only blanks between the cursor and the beginning of the line, then the INSERT key does nothing.

Similarly, **you can use DELETE to move text to the beginning of the line, but not past the beginning of the line;** pressing the DELETE key with the cursor at the beginning position of the line wraps the cursor up to the end of the preceding line, but the characters pulled to the beginning of the line do not move up with the cursor.

It is important to note here that when you edit screen text using the CURSOR, INSERT, and DELETE keys, what you see on the screen is not always what the PET records. In fact, none of your editing changes are recorded until you depress the RETURN key. Moreover, you must depress the RETURN key once for each statement line that you change.

SCREEN DISPLAY

The PET screen is a high-resolution video display; it is similar to a black-and-white television set. **The display has 1000 character positions, arranged as 25 rows, with 40 characters per row.** The PET screen is illustrated in Figure 2-5.

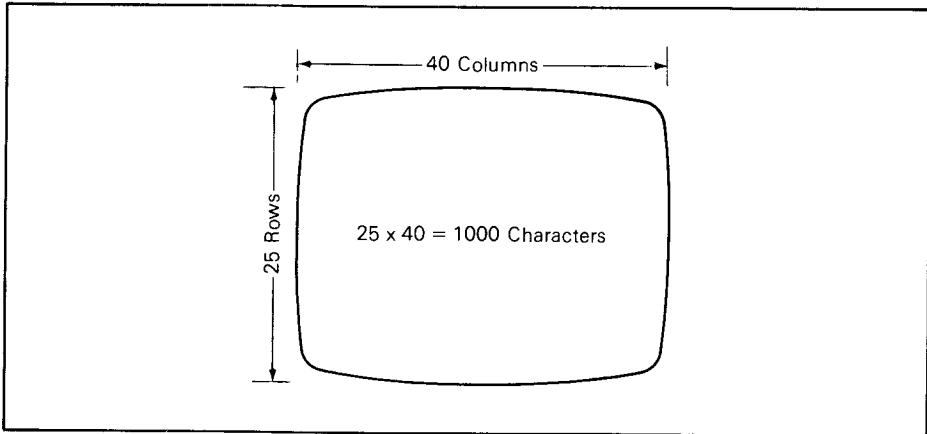


Figure 2-5. The PET Video Screen

One-fortieth of each display line, therefore, becomes one character position. The PET can display 64 dots in a single character position, organized as a block that is 8 dots high and 8 dots wide. **Characters are generated in any character position by displaying appropriate dots within the 8x8 dot block, or matrix.** This is illustrated in Figure 2-6.

Although lines on the PET display screen are only 40 characters wide, logically lines may be up to 80 characters wide. This is another example of what you see differing from what the PET records. To the PET, lines can be up to 80 characters long. The end of a line is recorded by the PET when you press the RETURN key. But the PET will insert its own RETURN after the 80th character, ending the line, if you attempt to type an 81st character as part of a single line.

The PET will display a line of more than 40 characters using two screen display rows. When you type a 41st character, the cursor automatically moves to the first character position of the next row, and displays the character you type; the display looks as though you had typed a RETURN after the 40th character, even though you have not.

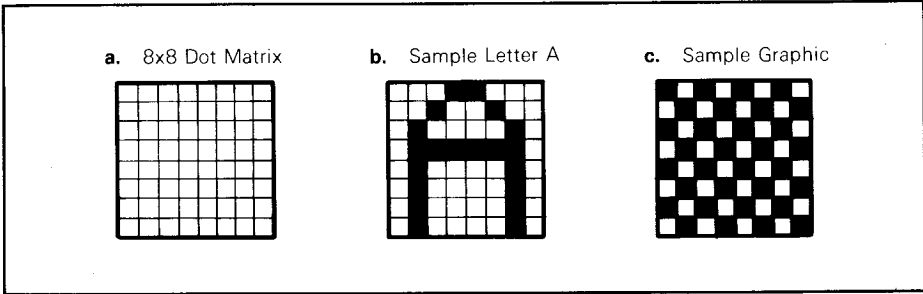


Figure 2-6. The 8x8 Dot Matrix

CASSETTE TAPE DRIVE

The cassette tape drive is built into the PET console on PETs with the compact keyboard but must be connected separately to PETs with the full size keyboard. The tape units themselves are identical. **The tape drive allows you to store programs and data from the PET memory onto cassette tape; you can also load stored programs and data from cassette tape into the PET memory.** The PET can be connected to more than one cassette tape drive. However, only one of the tape drives is the "main" or "console" cassette tape drive. For the compact keyboard PET the console tape drive is the built-in tape unit. For full size keyboards the console tape drive is the tape unit connected at the J1 interface port (refer to Figure 2-2).

You can load and store programs on any tape drive (or other peripheral device, such as a disk drive) connected to the PET. However, the console tape drive is the default device implied when performing a load or store without specifying a particular peripheral device.

The cassette tape drive is shown in Figure 2-7.

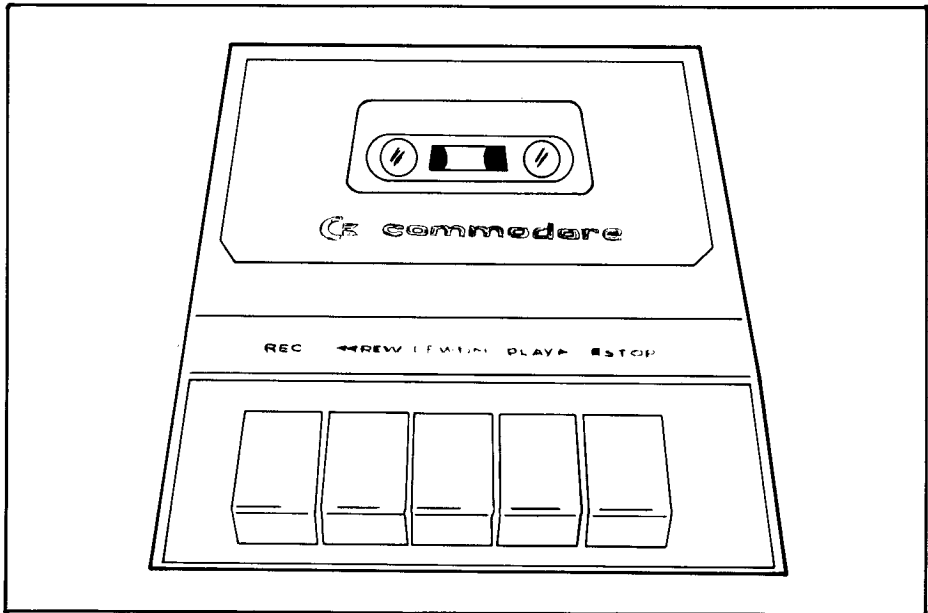


Figure 2-7. PET Cassette Tape Drive

CASSETTE OPERATION

Use the following procedure to insert a tape cassette into the PET tape drive (refer to Figure 2-8):

- a. Lift up the tape drive window.
- b. Holding the magnetic tape cassette as shown, push the cassette along the glide paths on the underside of the tape drive window.
- c. Push down until the cassette clicks into place.
- d. Push down the tape drive window. This aligns the path of the exposed magnetic tape with the tape drive head area.

To remove a cassette tape, lift up the tape drive window, pull the cassette tape out of the tape drive, and then close the tape drive window.



Photo Joe Mauro

a. Tape drive with window cover open—drive empty



Photo Joe Mauro

b. Correct manner to hold a tape cassette prior to inserting into tape drive

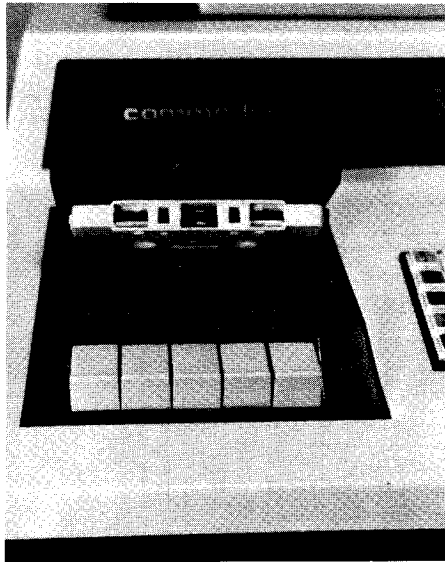


Photo Joe Mauro

c. Open cassette drive with tape cassette inserted

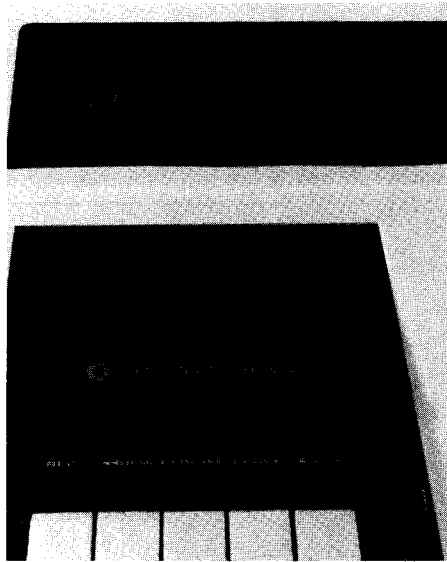


Photo Joe Mauro

d. Closed cassette drive with cassette inside

Figure 2-8. Inserting a Tape Cassette

CASSETTE TAPE CONTROLS

The cassette tape control keys are located at the forefront of the cassette drive.

Record (REC). The RECORD key lets you write onto the magnetic tape cassette from PET memory. To record, you must execute a SAVE command (which is described in Chapter 4), then hold down the REC key while depressing the PLAY key.

Rewind (REW). The REWIND key rewinds the magnetic tape at high speed, to its beginning. To rewind the tape, depress the REWIND key. If the tape does not start moving, press the tape STOP key, and then press REWIND.

You will use REWIND often to rewind tape cassettes back to their beginning point before removing them from the tape drive. You will also use REWIND any time you need the PET to start searching for information beginning at an earlier point on the magnetic tape.

Fast Forward (FFWD). FAST FORWARD winds the magnetic tape forward at high speed. You will not normally use FAST FORWARD.

The PET writes onto magnetic tape, reads from the tape, and searches for information on the tape all at PLAY speed, which is slow.

You may devise your own scheme using Fast Forward to locate information, rather than have the PET search at its Play speed. One method is just to guess. Fast Forward to a point that you think is close to the beginning of the information you want to load and then try to load. Note that you can be anywhere on the tape when you give the PET a LOAD request; you do not have to be at the beginning of a program or at the beginning of the tape. A more scientific method is to use the measuring scale on the tape cassette; the one below goes from 0 to 100 in units of 10, with the beginning of the tape at 30:



As the tape moves forward, the tape radius, measured on the scale, goes from 30 up to 100. 100 is the end of the tape. By keeping a record of information stored on a tape, using the radius numbers, you can Fast Forward to a point just before the desired location.

PLAY. The PLAY key enables the PET to begin searching the tape for a program to load from the tape into PET memory, or to write from PET memory to the tape if the RECORD key is also pressed. When you initiate the first LOAD or SAVE on the cassette, the PET asks you to "PRESS PLAY ON TAPE #1"; the PLAY key remains depressed until you disengage it by pressing the tape STOP key. You can leave the PLAY key pressed until you are through with the tape; the PET reads or writes on the tape only when directed to by commands from the keyboard.

When you first power up the PET, have the tape control keys all up (off). If the PLAY key is depressed, the tape will begin moving on power-up.

STOP. The tape STOP key disengages any of the other tape control keys. If one of the other keys does not respond, press the tape STOP key and then the key that did not respond.

Note that there are two STOP keys on the PET: the tape STOP key and the main keyboard STOP key. You should have no trouble differentiating between the two.

CLEANING AND MAINTENANCE

There are no parts of the PET, except the tape head, that require periodic cleaning for proper operation. You can dust the PET if it becomes dusty. Any household window cleaner or kitchen spray cleaner can be used to clean the PET keyboard, exterior, and display screen if they become heavily soiled. For light cleaning, use a sponge dampened in water or a mild soap solution. **DO NOT CLEAN THE PET WITH LARGE AMOUNTS OF WATER. DO NOT IMMERSE THE PET IN WATER.**

Perform all cleaning with power OFF.

For long-term storage, it is best to pack the PET back into its shipping carton.



RIGHT



WRONG

CLEANING AND DEMAGNETIZING THE TAPE HEAD

The tape deck head area of the PET cassette can be seen by opening the cassette window with power OFF and depressing the PLAY key. This juts the tape head mount out past the bottom edge of the cassette window area, where it is marginally reachable for maintenance. The tape head is shown in Figure 2-9.

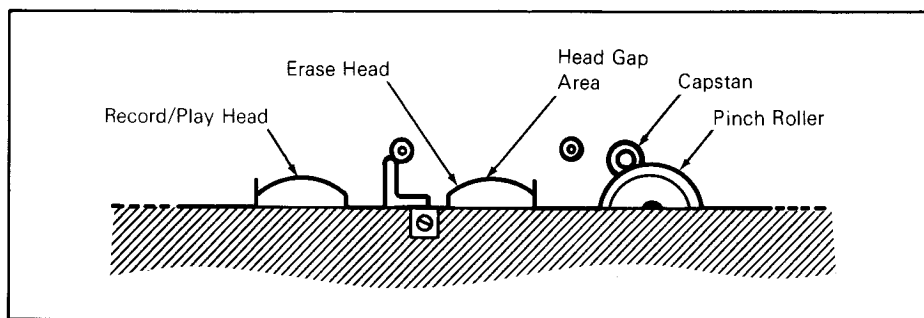


Figure 2-9. PET Cassette Tape Head

The tape head is the portion of the cassette unit that the magnetic tape contacts when you record or play back. The oxide coating on the magnetic tape gradually deposits a film on the tape head and surrounding area; this deposit must be removed periodically by cleaning the tape head to assure reliable operation of the cassette unit. To clean the tape head, use a cotton swab soaked with denatured ethanol. Clean both heads and also the capstan and pinch roller (see Figure 2-9). Allow the area to dry completely before closing the cover.

The tape head also needs to be demagnetized periodically. The tape heads gradually become magnetized through use. This affects recording fidelity and eventually causes recording errors. To demagnetize the tape heads, you will need a tape head demagnetizer (see Figure 2-10); this is an inexpensive unit that can be purchased at most audio equipment stores.

To demagnetize the tape heads, have the cassette window open and the PLAY key depressed. To use the demagnetizer, have it at least two feet away from the PET before plugging in the demagnetizer (the PET should be OFF). SLOWLY bring the demagnetizer towards the PET until it contacts the head surface; carefully move it around on one head surface, then the other head surface, then all metal surfaces that are directly adjacent to the heads. SLOWLY move the demagnetizer back at least two feet before unplugging it.

CAUTION: Keep your pre-recorded cassette tapes away from the demagnetizer. The demagnetizer is an effective tape eraser. Keep it at least five feet away from any pre-recorded cassettes.

The more you use your PET, the more often it will need to have the tape heads cleaned and demagnetized. Do it at least once a month — more often if you are experiencing load problems or tape degradation.



Figure 2-10. A Typical Tape Head Demagnetizer

Photo Joe Mauro

CARE OF CASSETTE TAPES

You will probably buy cassette tapes that have pre-recorded programs on them. You will probably also buy blank tape cassettes on which to record your own programs or data. Below are a few tips on taking care of your tape cassettes.

First, **when you get a new tape cassette (blank or pre-recorded), balance its tension by fast forwarding to the end of the tape and rewinding back to the beginning** before loading the first time. This is just a precautionary measure that may prevent reading errors that are called LOAD errors.

When buying cassette tapes for the PET, do not get very long ones — 15 or 30 minutes is sufficient. This not only cuts down the search time, but gives you tape that is thicker and stronger than long-playing tapes. **Select high quality, low noise, high output, ferric oxide tapes;** bargain brands are generally less satisfactory. **Store the cassettes in a cool place away from things like magnets and electronic equipment.**

Be careful not to touch the oxide coated surface when handling tapes.

Write Protect

You can prevent any cassette from being recorded on by a system called "write protect." A tape cassette has two write protect notches, one for each side of the tape, located on the side opposite the tape access opening (see Figure 2-11). When you buy pre-recorded tapes, or when your own tapes have information stored on them, you can protect these tapes from accidentally being written on by punching out the write protect tabs. To write protect just one side of the tape, punch out the tab that is on the left when you have the side you want to write protect facing upward. When the write protect tab is in place it provides resistance to move a small peg on the PET tape head area that allows the RECORD key to be operational. When the write protect tab is punched out the peg is not moved, so that side cannot be written on.

If you write protect a tape and then decide you want to reuse it, just place a piece of tape over the write protect opening.

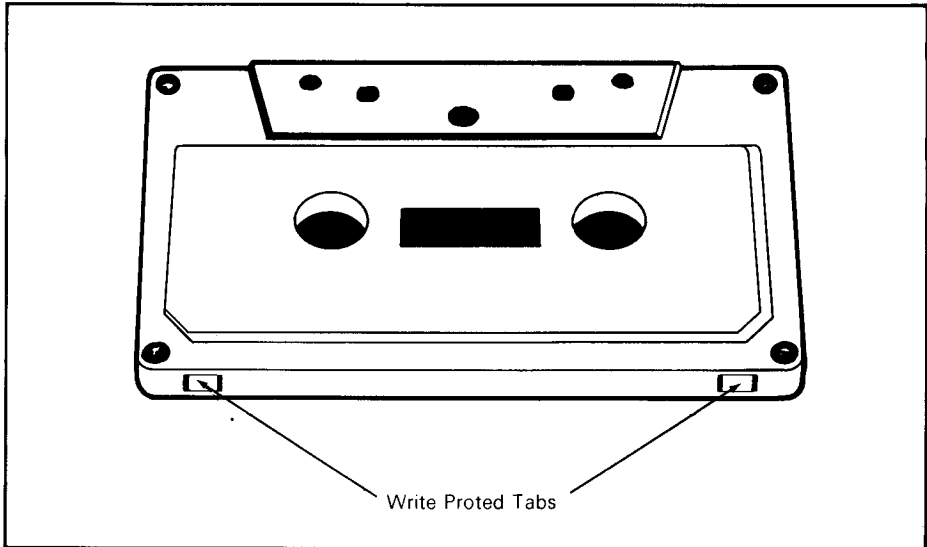


Figure 2-11. Write Protect Notches

ELEMENTARY TROUBLESHOOTING

There are a variety of symptoms that your PET may exhibit when it is not working properly. Here are a few examples: the RETURN key may not work; every alternate key may not work; the screen display may be jittery; the number of "bytes free" may change; the PET may not respond at all.

This section describes some rudimentary troubleshooting procedures that you can follow to check internal cable connections in the PET and to check for a faulty memory chip.

First, here are **a few simple procedures you should follow before opening up the case of the PET:**

- If you power up and get no response, be sure the PET is plugged into an electrical outlet. If it is plugged in, turn power off, wait a few seconds, then turn power back on. Try this a number of times until you get a response.
- If the screen becomes jittery or PET BASIC misreacts intermittently, the PET unit may be overheated. If you cannot cool the room temperature, you will have to turn the PET off for awhile (you will lose whatever information is in memory when you turn it off).
- The problem may not be in the PET hardware at all. There may be program errors, or "bugs," in the PET BASIC interpreter or in the program you are running.

Internal Cable Connections

This is the procedure to open the PET and check for secure positioning of cable connections. The specific items described are in the compact keyboard PET. Follow the same general procedure for full size keyboard PETs.

1. Turn the power off and unplug the PET.
2. Move the front of the PET out from the supporting surface so that you can see the four retaining screws on the bottom. With a screwdriver, unscrew the four retaining screws, placing them in a secure location.
3. Lift the cover all the way up, being careful not to move it so far back that it pulls any of the cords.
4. Locate the supporting rod on the left side in back of the tape cassette. Push the rod up to disengage it from its holder, then move it forward and secure it in the back screwhole on the left side. This holds the PET cover up so you have both hands free. (The supporting rod is shown in place in Figure 2-12.)

5. Locate the two rows of RAM devices (Figure 2-12). There are two rows of eight RAMs each (for an 8K system) making a total of 16 RAMs. Each RAM has nine prongs coming out and down, off each side of the black rectangular top (Figure 2-13). Press gently but firmly on each RAM to be sure it is securely seated in its casing connection.



Figure 2-12. PET Internal View

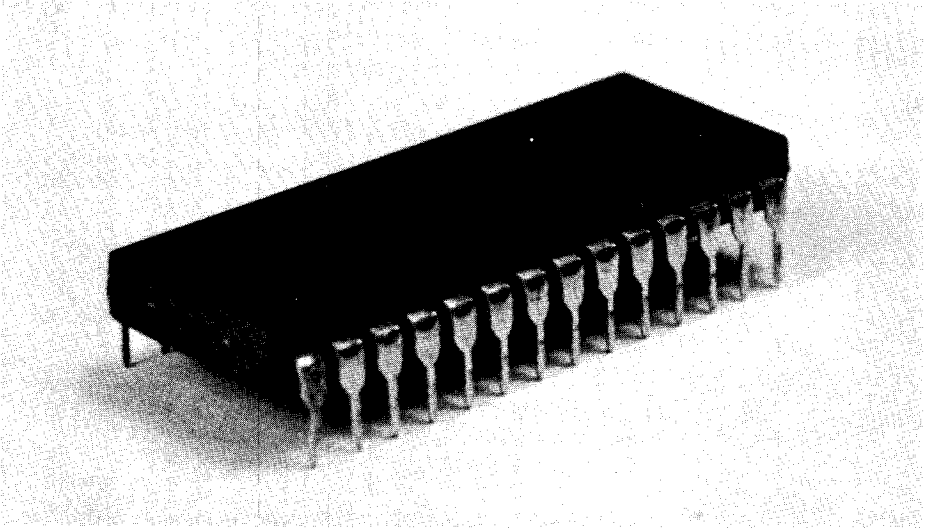


Figure 2-13. Memory Device Unconnected

Photo Joe Mauro

6. The third row of larger rectangles contains ROM devices. Press them down also to be sure that they are firmly seated.
7. The keyboard cable is connected just to the left of the ROMs. Press down along this connector to be sure that it is firmly in place. Optionally unplug the connector, lifting straight up, and plug it back in securely, pushing straight down.
8. Behind the keyboard cable is the video cable. Press firmly down on it to make sure it is firmly seated.
9. In front and to the left of the keyboard connector is the power supply connector. Press down on it to be sure it is firmly connected.
10. Behind and to the left of the power cable is the cassette recorder cable, connected horizontally on the side of the large green board. Push in on the cable to connect it firmly in place.
11. If the inside is dusty, you can vacuum carefully to remove foreign particles. Be very careful not to bend any of the components on the board. Preferably clean with a squeezable bulb brush cleaner or a can of pressurized air for dusting. (These two items are available from photographic supply houses.)
12. Unlatch the retaining rod and put it back into its holder, close the cover, plug the PET in, and turn the power on. If the trouble was due to loose connections, the problem should now be cleared up.

Locating a Defective RAM

On earlier model PETs, the RAM failure rate was high. This prompted the manufacturer to recommend a procedure for owners of the first PETs to check for a defective RAM themselves. We reproduce this procedure below; you may want to learn this procedure if you have an older PET and are experiencing frequent RAM failures. It applies only to PETs with compact keyboards.

If you do not get the standard number of bytes free at power-up (less than 7167 for an 8K PET or 3071 for a 4K PET) or if the RETURN key is not working properly, the problem is probably a defective memory device, or RAM. The following procedure describes how to locate a faulty RAM yourself, so that you do not have to ship the PET to the factory to have this done.

To perform this procedure, you will need an "extractor tool" (see Figure 2-14) available from electronic supply houses.

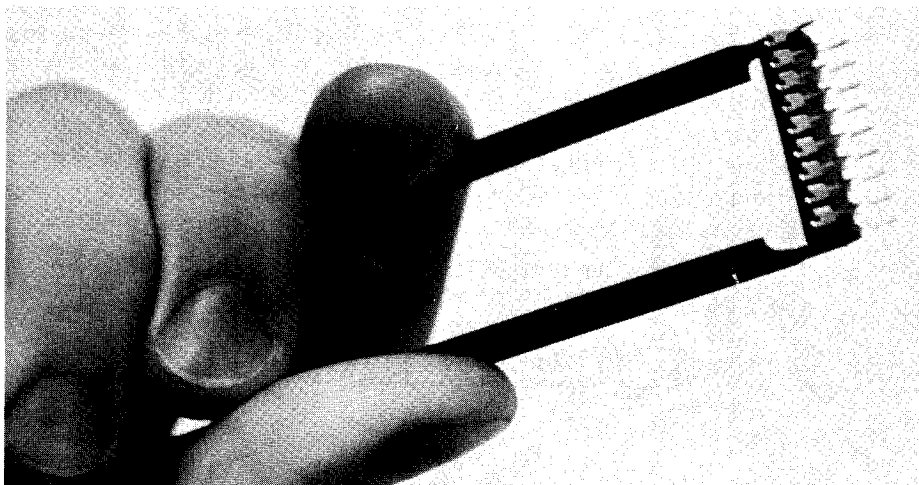


Figure 2-14. Extractor Tool

Photo Joe Mauro

1. Unplug and open the PET as described above.
2. Refer to Figure 2-15 for RAM numbering conventions. If bytes free are less than the number shown in the first column below, then the number of the RAM to be tested is in the second column below.

<u>Bytes Free</u>	<u>RAM Number</u>
1023	2
2047	3
3071	4
4095	5
5119	6
6143	7
7167	8

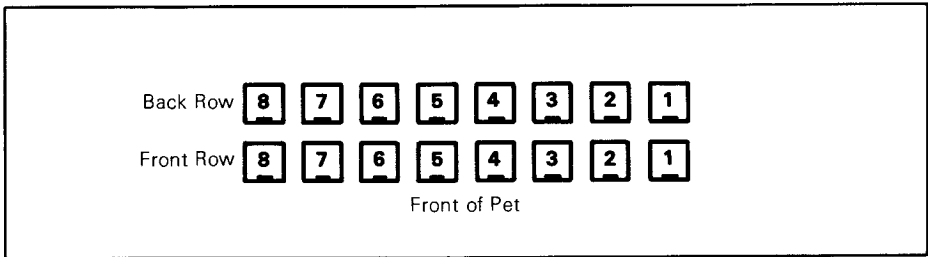
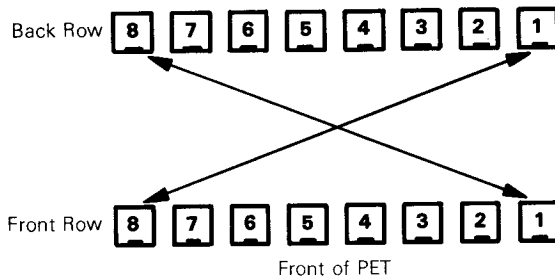


Figure 2-15. RAM Numbering

For example, if bytes free are less than 4095, perform this procedure on the RAM numbered 5 in Figure 2-15.

3. Using the extractor tool, carefully lift the two RAMs straight up from their sockets. **BE SURE THAT YOU DO NOT BEND THE PINS. BE SURE THAT YOU DO NOT TURN THE DEVICES AROUND.** Exchange the two RAMs, pushing each one straight down into the socket with your fingers. Without lowering the cover, plug the PET in and turn it on. Has the number of bytes free changed? If not, the two RAMs are good and you must select another pair. If the number of free bytes decreases, the defective RAM is the one you just put into the front row. If the number increases, the defective RAM is the one in the back row.
4. If the problem is with the RETURN key, exchange RAMs in columns 8 and 1, as shown below.



This (hopefully) puts good RAMs into column 1, and the defective RAM in column 8 can be determined by checking the number of free bytes, as described above.

5. When you have located the defective RAM, lift it out and use a RAM from column 8 in its place. This decreases your total amount of free bytes the least. Return the defective RAM to the manufacturer for a replacement.

CHAPTER 3

Programming the PET

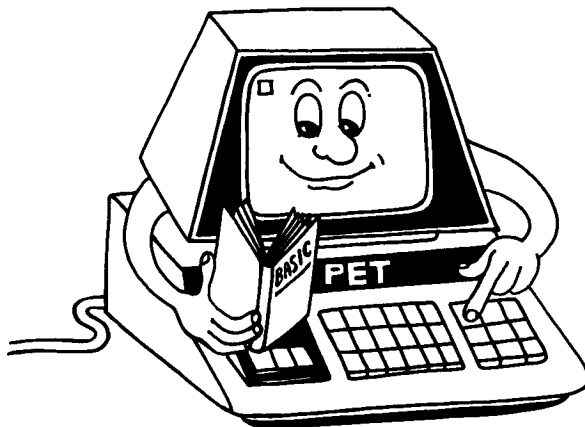
CALCULATOR OR IMMEDIATE MODE

When the PET is powered up it is in calculator mode, also called direct or immediate mode. In calculator mode, as the name implies, you can use the PET as you would a calculator; it responds directly with the answers to arithmetic calculations. This may be illustrated as follows:

74.5+6.42 10.92	Addition
READY. 7500-410 90	Subtraction
READY. ?π*2 6.28318531	Multiplication
READY. ?100/3 33.3333333	Division
READY. ?6/2*4-1 11	Combination

In calculator mode, as illustrated above, answers are displayed immediately, on the next line of the display.

To use calculator mode for arithmetic operations, you must begin the line by typing ? or PRINT; they are both print requests. Next type the calculation for which you seek an answer. Terminate with a RETURN.



We can illustrate the "format" or "syntax" for a general case calculator mode entry as follows:



Statements entered in calculator mode do not, and cannot, begin with line numbers. Later in this chapter we will examine exactly what you can include in the "formula" portion of a calculator mode statement.

You can use immediate mode to display the result of a calculation defined in an earlier statement. Consider the following example:

```
A=π*2
READY.
?A
  6.28318531
```

There were two statements, each given in immediate mode. When you type in the first statement, $A=\pi*2$, the result is not displayed since the statement does not begin with ? or PRINT. But PET BASIC performed the calculation and gave the resulting value to the variable named A. This value is fetched and typed in the second command, ?A.

You can display the value of any variable using the technique illustrated above. The variable does not need to have its value assigned in a preceding immediate mode statement, as was the case for A above. When you have learned how to write and run PET BASIC programs, remember that you can stop the program's execution at any time, then examine the current value assigned to any variable defined by your program. Type ? followed by the variable name, and the value currently assigned to the variable name will be displayed on the next line, just as 6.28318531, the value assigned to A, is shown in the illustration above.

A ONE-LINE PROGRAM

Statements can be grouped together on one line by separating each statement with a colon (:). Thus, the two statements you entered separately earlier:

```
A=π*2
?A
```

can be condensed into one line as follows:

```
A=π*2: ?A
```

Since a line can have up to 80 characters (two display lines), you can put quite a lot in one line, and execute it all in immediate mode. For example, consider the following statement:

```
FOR I=1 TO 800:?"A":NEXT:?"PHEW!"
```

Type this "mini-program" in, exactly as shown, ending with a RETURN. If you type it in successfully, you will see the letter A print across the next 20 lines of the screen, followed by the message PHEW! on the 21st line.

```
FOR I=1 TO 800:"A";:NEXT:"PHEW!"  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
PHEW!
```

READY.
✻

The program line is conveniently left at the top of the screen. This was done by having the program print just enough lines to scroll the program line to the top of the screen but not off it.

MODIFYING A PROGRAM

Before trying any more characters, make one editing modification to the line so that changing the character is just a little easier. By adding an assignment statement at the beginning of the line, and having the variable printed in the PRINT statement, you won't have to move the cursor as far to the right to change the character. The new line, with the display character changed to a W, will look like this:

```
C$="W":FOR I=1 TO 800:PC$:NEXT?"PHEW!"
```

To modify the current line, perform the following steps:

1. Home the cursor so it is blinking at the F in FOR (█ indicates position of cursor).

```
█FOR I=1 TO 800:?"█":NEXT?"PHEW!"
```

2. Press the INSERT key (shift of Insert/Delete) seven times.

```
█ FOR I=1 TO 800:?"█":NEXT?"PHEW!"
```

3. Type in the seven characters:

```
C$="W":
```

The line now looks like this, with the cursor again resting on the F in FOR:

```
C$="W":█FOR I=1 TO 800:?"█":NEXT?"PHEW!"
```

4. Cursor right 14 times to the first quotation mark.

```
C$="W":FOR I=1 TO 800:?"█":NEXT?"PHEW!"
```

5. Type in the two characters:

```
C$
```

The line now looks like this:

```
C$="W":FOR I=1 TO 800:PC$:NEXT?"PHEW!"
```

6. Remove the other quotation mark by pressing one Cursor Right:

```
C$="W":FOR I=1 TO 800:PC$)NEXT?"PHEW!"
```

Followed by one Delete:

```
C$="W":FOR I=1 TO 800:PC$)NEXT?"PHEW!"
```

The changes have all been made; press RETURN to print the new character. Now you can HOME the cursor, then move it right just four positions to change the display character. Display any other characters you want. The graphics are especially interesting.

ELEMENTS

Before modifying this display program further, we will set it aside and describe the elements of PET BASIC. This will introduce you to the full range of statements, functions, and commands available. At the end of the chapter we will come back to this display example and show how you can develop it into a stored program.

Like natural languages, **PET BASIC has an alphabet consisting of letters, numbers, and special characters. These correspond to the alphabetic keys, numeric keys, and special symbol keys on the PET keyboard.** The remaining two groups of keys, graphic keys and function keys, may be used only inside quotation marks, or for keyboard/cursor control purposes not related to the BASIC language.

The PET alphabet is used to form the words, numbers, and other elements that make up a BASIC sentence, or statement.

NUMBERS

You use numbers all the time when working with the PET. There are two kinds of numbers that can be stored in the PET: floating point numbers (also called real numbers) and integers.

Floating Point Numbers

Floating point is the standard number representation used by the PET. The PET does its arithmetic using floating point numbers. **A floating point number can be a whole number, a fractional number preceded by a decimal point, or a combination.** The number can be signed negative (-) or positive (+). If the number has no sign it is assumed to be positive. Here are some examples of floating point numbers:

Whole number, equivalent to an integer:

5
-15
65000
161
0

Numbers with decimal point:

0.5
0.0165432
-0.0000009
1.6
24.0055
-64.2
3.1416

Note that if you put commas in a number and ask the PET to print it, you will get a Syntax Error message. For example, you must use 65000, not 65,000.

Roundoff

Numbers always have at least eight digits of precision; they can have up to nine, depending on the number. The PET rounds off any additional significant digits. It rounds up when the next digit is five or more; it rounds down when the next digit is four or less.

Roundoff will sometimes cause fractional numbers to look inaccurate. Here are some examples:

```
? .555555556  
 .55555555  
↑  
? .555555557  
 .555555556  
↑  
? .111111115  
 .111111111  
↑  
? .111111116  
 .111111112  
↑
```

Appears to round down on 6 or less, up on 7 or more.

Appears to round down on 5 or less, up on 6 or more.

These quirks result from the manner in which computers store floating point numbers.

Scientific Notation

Floating point numbers can also be represented in scientific notation. **When numbers with ten or more digits are entered, the PET automatically converts them to scientific notation.** Scientific notation allows the PET to accurately display these large numbers in a smaller number of spaces. For example:

```
READY.  
?1111111114  
1.11111111E+09
```

```

READY.
?1111111115
1.11111112E+09

```

A number in scientific notation has the form:

$$\text{number}E\pm ee$$

where:

- number is an integer, fraction, or combination, as illustrated above. The "number" portion contains the number's significant digits; it is called the "coefficient." If no decimal point appears, it is assumed to be to the right of the coefficient.
- E the upper case letter E.
- \pm an optional plus sign or minus sign.
- ee a one- or two-digit exponent. The exponent specifies the magnitude of the number; that is, the number of places to the right (positive exponent) or to the left (negative exponent) that the decimal point must be moved to give the true decimal point location.

Here are some examples:

<u>Scientific Notation</u>	<u>Standard Notation</u>
2E1	20
10.5E+4	105000
66E+2	6600
66E-2	0.66
-66E-2	-0.66
1E-10	0.0000000001
94E20	94000000000000000000

As the last two examples show, scientific notation is a much more convenient way of expressing very large or very small numbers. PET BASIC prints numbers ranging between 0.01 and 999,999,999 using standard notation; but numbers outside of this range are printed using scientific notation.

```

?.009
9E-03

```

```

READY.
?.01
.01

```

```

READY.
?999999999.9
999999999

```

```

READY.
?999999999.6
1E+09

```

There is a limit to the magnitude of a number that the PET can handle, even in scientific notation. The range limits are:

largest floating point number: $\pm 1.70141183E+38$

smallest floating point number: $\pm 2.93873588E-39$

Any number of larger magnitude will give an overflow error. Here are some examples of overflow error:

```
71.70141183E+38
 1.70141183E+38

READY.
?-1.70141183E+38
-1.70141183E+38

READY.
?1.70141184E+38

?OVERFLOW ERROR ←
READY.
?-1.70141184E+38

?OVERFLOW ERROR ←
```

Any number of smaller magnitude will yield a zero result. This may be illustrated as follows:

```
?2.93873588E-39
 2.93873588E-39

READY.
?-2.93873588E-39
-2.93873588E-39

READY.
?2.93873587E-39
 0 ←

READY.
?-2.93873587E-39
 0 ←
```

Integers

An integer is a number that has no fractional portion or decimal point.

The number can be signed negative (−) or positive (+). An unsigned number is assumed to be positive. Integer numbers have a limited range of values, from −32767 to +32767. The following are examples of integers:

0
1
44
32699
−15

Any number that is an integer can also be represented in floating point format, since integers are a subset of floating point numbers. **PET BASIC converts any integers to floating point representation before doing arithmetic with them. The most important difference between floating point numbers and integers is that an integer array uses less storage space in memory** (two bytes for an integer, versus five bytes for a floating point number).

STRINGS

We have already used strings as messages to be printed on the display screen. **A string consists of one or more characters enclosed in double quotation marks.** Here are some examples of strings:

















"HI!"
"SYNERGY"
"12345"
"\$10.44 IS THE AMOUNT"
"22 UNION SQUARE, SAN FRANCISCO CA"

All of the data keys (alphabetic, numeric, special symbols, and graphics), as well as the three cursor control keys (Clear Screen/Home, Cursor Up/Down, Cursor Left/Right) and the Reverse On/Off key can be included in a string. The only keys that cannot be used within a string (besides the SHIFT key) are Run/Stop, RETURN, and Insert/Delete.

All characters within the string are displayed as they appear. The cursor control and Reverse On/Off keys, however, normally do not print anything themselves; to show that they are present in a string, certain reverse field symbols are used. They are shown in Table 3-1.

When you enter a string from the keyboard, it can have any length up to the space available within an 80-character line (that is, any character spaces not taken up by the line number and other required parts of the statement). However, **strings of up to 255 characters can be stored in the PET's memory. You get long strings by pushing together, or concatenating, two separate strings to form one longer string.** We will describe this further when we discuss string variables in general.

Table 3-1. Special String Symbols

Function	Key	String Symbol
Reverse On		 (Reverse R)
Reverse Off	Shifted 	 (Reverse Shifted R)
Home Cursor		 (Reverse S)
Clear Screen	Shifted 	 (Reverse Shifted S)
Cursor Down		 (Reverse Q)
Cursor Up	Shifted 	 (Reverse Shifted Q)
Cursor Right		 (Reverse J)
Cursor Left	Shifted 	 (Reverse Shifted J)

VARIABLES

In Chapter 2 we introduced the concept of a variable. We will now describe variables more thoroughly.

A variable is a data item whose value may change. You type the immediate mode statement:

```
PRINT 10,20,30
10      20      30
```

The PET will print the three numbers 10, 20, and 30, as illustrated above. The PET will print the same three numbers whenever this PRINT statement is executed; that is because this PRINT statement uses constant data. In contrast, you can write the immediate mode statement:

```
A=10:B=20:C=30:PRINT A,B,C
10      20      30
```

The same three numbers, 10, 20, and 30, are displayed, but A, B, and C are variables, not constants. By changing the values assigned to A, B, and C, you can change the values printed out by the PRINT statement. Here is an example:

```
A=-4 : B=10 : C=4E2 : PRINT A, B, C
-4          10          400
```

Variables appear in just about every statement of a computer program.

Variable Name

A variable is identified by a name. We used A, B, and C as variable names in the illustrations above. **A variable has two parts: its name and a value. The variable name represents a location at which the current value is stored.** In the illustration below, the current value of A is 14; for B it is 15; and for C it is 0.

<u>Variable Name</u>	<u>Location Contents</u>
A	14
B	15
C	0

If we change A to -1 using the immediate mode statement:

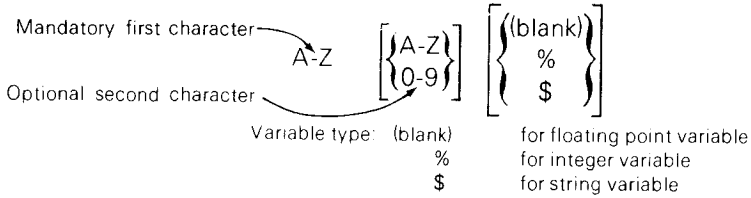
```
A = -1
```

then our illustration must change as follows:

<u>Variable Name</u>	<u>Location Contents</u>
A	-1
B	15
C	0

This is a good way of looking at variables because it is, in fact, the way they are handled by the PET. **A variable name represents an address in PET memory; and at that memory location is the current value of the variable.** The important point to note is that **variable names — which are names that programmers make up — are arbitrary; they have no innate relationship to the value that the variables represent.**

A variable name can have one, two, or three characters, as follows:



A-Z means that you can select any letter of the alphabet A, B, C, . . . X, Y, Z, and 0-9 means that you can select any digit 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

Floating Point Variable

A floating point variable is a variable that represents a floating point number. This is probably the most common type of variable that you will use in PET BASIC.

Names for floating point variables have the form:

$$\text{A-Z} \left[\left\{ \begin{array}{l} \text{A-Z} \\ \text{0-9} \end{array} \right\} \right]$$

Examples:

A
B
C
A1
AA
Z5

Integer Variable

An integer variable is a variable that represents an integer.

Names for integer variables have the form:

$$\text{A-Z} \left[\left\{ \begin{array}{l} \text{A-Z} \\ \text{0-9} \end{array} \right\} \right] \%$$

Examples:

A%
B%
C%
A1%
MN%
X4%

Remember, **floating point variables can also represent integers; but you should use integer variables in arrays whenever possible, since they use less memory** — two bytes, versus five for a floating point array element.

String Variable

A string variable is a variable that represents a string of text. Names for string variables have the form:

$$A-Z \left[\begin{array}{l} \{A-Z\} \\ \{0-9\} \end{array} \right] \$$$

Examples:

A\$
M\$
MN\$
M1\$
ZX\$
F6\$

Longer Variable Names

You can use variable names having more than two alphanumeric characters; but if you do, only the first two characters count. To PET BASIC, therefore, BANANA and BANDAGE are the same name since both begin with BA.

The advantage of using longer variable names is that they make programs easier to read. PARTNO, for example, is more meaningful than PA as a variable name describing a part number in an inventory program.

PET BASIC allows variable names to have up to 255 characters.

Here are some examples of variable names with more than the minimum number of characters:

MAGIC\$
N123456789
MMM\$
ABCDEF%
CALENDAR

If you use extended variable names, keep in mind the following:

1. Only the first two characters, plus the identifier symbol, are significant in identifying a variable. Do not use extended names like LOOP1 and LOOP2; these refer to the same variable: LO.
2. PET BASIC has what are called "reserved words." These are words that have special meaning for the PET BASIC interpreter. No variable can contain a reserved word. In longer names you have to be very careful that a reserved word does not occur embedded anywhere in the name.
3. The additional characters need extra memory space, which you might need for longer programs.

Reserved Words

PET BASIC is programmed to recognize certain words as requests for specific operations. Names that are pegged to certain operations are called "reserved words." **You cannot use these words as variable names because PET BASIC will always recognize the word as a request for the corresponding operation. Moreover, you cannot use a reserved word as any part of your variable names; PET BASIC will still find it and treat it as a request for an operation.**

A list of reserved words is given in Table 3-2. Pay particular attention to the two-character reserved words, since these are the ones you would be most likely to duplicate in a variable name.

Table 3-2. Reserved Words

ABS	GET	OPEN	SPC
AND	GET#	OR	SQR
ASC	GOSUB	PEEK	ST
ATN	GOTO	POKE	STEP
CHR\$	IF	POS	STOP
CLOSE	INPUT	PRINT	STR\$
CLR	INT	PRINT#	SYS
CMD	LEFT\$	READ	TAB
CONT	LEN	READ#	TAN
COS	LET	REM	THEN
DATA	LIST	RESTORE	TI
DEF	LOAD	RETURN	TI\$
DIM	LOG	RIGHT\$	TO
END	MID\$	RND	USR
EXP	NEW	RUN	VAL
FN	NEXT	SAVE	VERIFY
FOR	NOT	SGN	WAIT
FRE	ON	SIN	

OPERATORS

An operator is a special symbol that PET BASIC recognizes as representing an operation to be performed on the variables or constant data. One or more operators, combined with one or more terms, form an "expression."

PET BASIC provides arithmetic operators, relational operators, and Boolean operators.

Arithmetic Operators

An arithmetic operator defines an arithmetic operation to be performed on the adjoining terms. Arithmetic operations are performed using floating point numbers. Integers are converted to floating point numbers before an arithmetic operation is performed; the result is converted back to an integer.

Arithmetic operations and their symbols are:

Addition (+). The plus sign specifies that the term on the left is to be added to the term on the right. For numeric quantities this is straightforward addition.

Examples:

2+2
A+B+C
X%+1
BR+10E-2

The plus sign can be used to "add" strings; but rather than adding their values, they are joined together, or concatenated, forming one longer string. The difference between numeric addition and string concatenation can be visualized as follows:

Addition of Numbers:

num1+num2=num3

Addition of Strings:

string1+string2=string1string2

By concatenation, strings containing up to 255 characters can be developed.

Examples:

"FOR"+"WARD" results in "FORWARD"
"HI"+" "+"THERE" results in "HI THERE"
A\$+B\$
"1"+CH\$+E\$

Subtraction (-). The minus sign specifies that the term to the right of the minus sign is to be subtracted from the term to the left.

Examples:

4-1 results in 3
100-64 results in 36
A-B
55-142 results in -87

The minus can also be used as a unary minus; that is, the minus sign preceding a negative number.

Examples:

-5	
-9E4	
-B	
4--2	same as 4+2

Multiplication (*). An asterisk specifies that the term on the right is multiplied by the term on the left.

Examples:

100*2	results in 200
50*0	results in 0
A*X1	
R%*14	

Division (/). The slash specifies that the term on the left is to be divided by the term on the right.

Examples:

10/2	results in 5
6400/4	results in 1600
A/B	
4E2/XR	

Exponentiation (^). The up arrow specifies that the term on the left is raised to the power specified by the term on the right. If the term on the right is 2, the number on the left is squared; if the term on the right is 3, the number on the left is cubed, etc. The exponent can be any number, variable, or expression, as long as the exponentiation yields a number in the PET's range.

Examples:

2^2	results in 4
12^2	results in 144
1^3	results in 1
A^5	
2^6.4	results in 84.4485064
NM^1-10	
14^F	

Order of Evaluation

When an expression has multiple operations, as in:

$$A+C*10/212$$

there is a built-in hierarchy for evaluating the expression. **First is exponentiation (1), followed by unary minus (-), followed by multiplication and division (* /), followed by addition and subtraction (+ -).** Operators of the same hierarchy are evaluated from left to right.

This natural order of operation can be overridden by the use of parentheses. Any operation within parentheses is performed first.

Examples:

$4+1*2$	results in 6
$(4+1)*2$	results in 10
$100*4/2-1$	results in 199
$100*(4/2-1)$	results in 100
$100*(4/(2-1))$	results in 400

When parentheses are present, PET BASIC evaluates the innermost set first, then the next innermost, etc. Parentheses can be nested to any level and may be used freely to clarify the order of operations being performed in an expression.

Relational Operators

A relational operator specifies a "true" or "false" relationship between adjacent terms. The specified comparison is made, and then the relational expression is replaced by a value of true (-1) or false (0). Relational operators are listed in Table 3-4 at the end of this chapter. Relational operators are evaluated after all arithmetic operations have been performed.

Examples:

$1=5-4$	results in true (-1)
$14>66$	results in false (0)
$15>=15$	results in true (-1)
$A < > B$	

Relational operators can be used to compare strings. For comparison purposes, the letters of the alphabet have the order $A < B$, $B < C$, $C < D$, etc. Strings are compared by comparing their stored character values. Characters are stored using a special binary code called "ASCII." Appendix A lists the ASCII code assigned to every PET character.

Examples:

$"A" < "B"$	results in true (-1)
$"X" = "XX"$	results in false (0)
$C\$ = A\$ + B\$$	

Boolean Operators

The Boolean operators **AND**, **OR**, and **NOT** specify a Boolean logic operation to be performed on two variables, on adjacent sides of the operator. In the case of **NOT**, only the term to the right is considered. Boolean operations are not performed until all arithmetic and relational operations have been completed.

Examples:

```
IF A=100 AND B=100 GOTO 10
```

If both A and B are equal to 100, branch to statement 10.

```
IF X<Y AND B>=44 THEN F=0
```

If X is less than Y, and B is greater than or equal to 44, then set F equal to 0.

```
IF A=100 OR B=100 GOTO 20
```

If either A or B has a value of 100, branch to statement 20.

```
IF X<Y OR B>=44 THEN F=0
```

F is set to 0 if X is less than Y, or B is greater than 43.

```
IF A=1 AND B=2 OR C=3 GOTO 30
```

Take the branch if both A=1 and B=2; also take the branch if C=3.

A single term being tested for "true" or "false" can be specified by the term itself, with an implied "<>0" following it. Any non-zero value is considered true; a zero value is considered false.

Examples:

```
IF A THEN B=2
```

```
IF A<>0 THEN B=2
```

The above two statements are equivalent.

```
IF NOT B GOTO 100
```

Branch if B is false, i.e., equal to zero. This is probably better written as:

```
IF B=0 GOTO 100
```

The three Boolean operators can also be used to perform logic operations on the individual binary digits of two adjacent terms (or just the term to the right in the case of **NOT**). But the terms must be in the integer range. Boolean operations are defined by groups of statements, which taken together constitute a "truth table." Table 3-3 gives the truth tables for the Boolean operators used by PET BASIC.

Table 3-3. Boolean Truth Table

The AND operation results in a 1 only if both bits are 1.	
1 AND 1 = 1	
0 AND 1 = 0	
1 AND 0 = 0	
0 AND 0 = 0	
The OR operation results in a 1 if either bit is 1.	
1 OR 1 = 1	
0 OR 1 = 1	
1 OR 0 = 1	
0 OR 0 = 0	
The NOT operation logically complements each bit.	
NOT 1 = 0	
NOT 0 = 1	

Bit-Oriented Boolean Operations

We include below a discussion of binary digit (bit) oriented Boolean operations. **This discussion is presented for those who are interested in the details of how these operations are performed.** If you do not understand it, skip it. You are not skipping anything you must know.

Bitwise Boolean operations are normally performed on unsigned, positive 16-bit numbers; these numbers range from 0 to FFFF hexadecimal, 0 to 188888 octal, or 0 to 65535 decimal. However, the numbers must be input to the PET in the range ± 32767 decimal. Appendix F contains hexadecimal-PET decimal conversion tables. If you are already familiar with Boolean operations, you have probably used binary, octal, or hexadecimal notation, because the bit-by-bit conversion is straightforward. PET BASIC accepts only decimal numbers, even for these Boolean operations. Here are some examples of bitwise Boolean operations:

<u>Operation</u>		<u>Hexadecimal equivalent</u>
1 AND 1	results in 1	1 AND 1 equals 1
1 AND -1	results in 1	1 AND FFFF equals 1
15 OR 240	results in 255	F OR F0 equals FF
NOT 0	results in -1	NOT 0 equals FFFF
NOT 1	results in -2	NOT 1 equals FFFE

If the terms to be operated on are not already integers, they are converted to integer form; the Boolean operation is performed, and the result is presented as a single integer value. This is also how Boolean operations work on the true/false expressions. That is, there is no operational difference between a mixed Boolean operation such as:

A=1 OR C<2

and a simple Boolean operation such as:

A OR C

The only practical difference is that, since a relational expression is evaluated to -1 or 0 , the Boolean operation will always be performed on quantities of -1 and 0 for a relational expression. For example:

IF A=B AND C<D GOTO 40

First the relational expressions are evaluated. Assume that the first expression is true and the second one is false. In effect, the following Boolean expression is evaluated:

IF -1 AND 0 GOTO 40

Performing the AND yields a 0 result:

IF 0 GOTO 40

Recall that a single term has an implied " $<>0$ " following it. The expression therefore becomes:

IF $0<>0$ GOTO 40

Thus, the branch is not taken.

In contrast, a Boolean operation performed on two variables may yield any integer number:

IF A% AND B% GOTO 40

Assume that $A\%=255$ and $B\%=240$. The Boolean operation 255 AND 240 yields 240 . The statement, therefore, is equivalent to:

IF 240 GOTO 40

or, with the " $<>0$ ":

IF $240 <>0$ GOTO 40

Therefore the branch will be taken.

Now compare the two assignment statements:

A = A AND 10

A = A < 10

In the first example, the current value of A is logically ANDed with 10 and the result becomes the new value of A . A must be in the integer range ± 32767 . In the second example, the relational expression $A < 10$ is evaluated to -1 or 0 , so A must end up with a value of -1 or 0 .

Table 3-4. Operators

	Precedence	Operator	Meaning
	High 9	()	Parentheses denote order of evaluation
Arithmetic Operators	8	↑	Exponentiation
	7	-	Unary Minus
	6	*	Multiplication
	6	/	Division
	5	+	Addition
	5	-	Subtraction
Relational Operators	4	=	Equal
	4	< >	Not equal
	4	<	Less than
	4	>	Greater than
	4	<=or = <	Less than or Equal
	4	>=or = >	Greater than or Equal
Boolean Operators	3	NOT	Logical complement
	2	AND	Logical AND
	1	OR	Logical OR
	Low		

ARRAYS

An array is a sequence of related variables. A table of numbers, for example, may be visualized as an array. The individual numbers within the table become "elements" of the array.

Arrays are a useful shorthand means of describing a large number of related variables. Consider, for example, a table of numbers containing ten rows of numbers, with twenty numbers in each row. There are 200 numbers in the table. How would you like it if you had to assign a unique name to each of the 200 numbers? It would be far simpler to give the entire table one name, and identify individual numbers within the table by their table location. That is precisely what an array does for you.

Arrays can have one or more dimensions. A single-dimensional array is equivalent to a table with just one row of numbers. The dimension identifies a number within the single row. An array with two dimensions yields an ordinary table with rows and columns: one dimension identifies the row, the other dimension identifies the column. An array with three dimensions yields a "cube" of numbers, or perhaps a stack of tables. Four or more dimensions yield an array that is hard to visualize, but mathematically no more complex than a smaller dimensioned array.

Let us examine arrays in detail.

A single-dimensional array element has the form:

name(i)

where:

name is the variable name of the array. The name may be any type of variable name.

i is the array index to that element. On the PET the index i must start at 0.

A single-dimensional array called A, having five elements, would be stored as follows:

A(0)	
A(1)	
A(2)	
A(3)	
A(4)	

The number of elements in the array is equal to the highest element number, plus 1 to count array element 0.

A two-dimensional array element has the form:

name(i,j)

where:

name is the variable name of the array. It may be any type of variable name.

i is the column index.

j is the row index.

A two-dimensional array called A\$, having three column elements and two row elements, would appear as follows:

A\$(0,0)			A\$(0,1)
A\$(1,0)			A\$(1,1)
A\$(2,0)			A\$(2,1)

The size of the array is the product of the highest row element plus 1, multiplied by the highest column element plus 1. For this array, it is $3 \times 2 = 6$ elements.

Additional dimensions can be added to the array:

name (i,j,k,...)

Arrays of up to eleven elements (subscripts 0 to 10 for a one-dimensional array) may be used where needed in PET BASIC, just as variables can be used as needed. Arrays containing more than eleven elements need to be "declared" in a Dimension statement. Dimension statements are described later in this chapter and in Chapter 4. An array (always with subscripts) and a single variable of the same name are treated as separate items by PET BASIC. Once dimensioned, an array cannot be referenced with different dimensions.

FUNCTIONS

Another element of PET BASIC is the built-in function. **A function performs a predefined operation on a specified data item or items;** they are referred to as "arguments." For example, you will use functions to find the square root of a number, to convert a floating point number to integer form, to find the number of characters in a string, or to set a tab to a certain column on the display screen. Functions are a great programming convenience.

We will summarize the functions supported by PET BASIC now so that you can get an overview of what functions are available to you on the PET. The functions are described in detail in Chapter 4.

A function may operate on one or more constants, variables, or expressions; these are termed the arguments of the function. The argument is written in parentheses following the function name:

function(arg)

If a function requires two or more arguments, they are separated by commas:

function(arg1,arg2)

If the arguments of a function (arg1, arg2, etc.) are expressions, which is allowed by PET BASIC, then the expressions are evaluated, reducing each argument to a single number, before the function itself is evaluated.

When a function appears in a BASIC statement, it is evaluated before any other part of the BASIC statement.

The following functions constitute the arithmetic function group:

INT	Converts a floating point argument to its integer equivalent.
SGN	Returns the sign of an argument: +1 for a positive argument, -1 for a negative argument, 0 for a 0 argument.
ABS	Returns the absolute value of an argument. A positive argument does not change; a negative argument is converted to its positive equivalent.
SQR	Computes the square root of the argument.
EXP	Raises the natural logarithm base e to the power of the argument (e^{arg1}).

LOG	Returns the natural logarithm of the argument.
RND	Generates a random number. There are some rules regarding use of RND; they are described in Chapter 4.
SIN	Returns the trigonometric sine of the argument, which is treated as a radian quantity.
COS	Returns the trigonometric cosine of the argument, which is treated as a radian quantity.
TAN	Returns the trigonometric tangent of the argument, which is treated as a radian quantity.
ATN	Returns the trigonometric arctangent of the argument, which is treated as a radian quantity.

Arithmetic functions perform commonly needed arithmetic operations on a single argument. Here are some examples:

INT(A)	Returns the integer value of A.
SQR(144)	Returns the square root of 144 (i.e., 12).
SIN(BX(1,0))	Returns the sine of array element BX(1,0).

The following functions constitute the string function group:

STR\$	Converts a number to its equivalent string of text characters.
VAL	Converts a string of text characters to their equivalent number (if such a conversion is possible).
CHR\$	Converts an 8-bit binary code to its equivalent ASCII character.
ASC	Converts an ASCII character to its 8-bit binary equivalent.
LEN	Returns the number of characters contained in the text string identified by the argument.
LEFT\$	Extracts the left part of a text string. Function arguments identify the string and its left part.
RIGHT\$	Extracts the right part of a text string. Function arguments identify the string and its right part.
MID\$	Extracts the middle section of a text string. Function arguments identify the string and the required mid part.

String functions let you determine the length of a string, extract portions of a string, and convert between numeric, ASCII, and string characters. These functions take one, two, or three arguments. Here are some examples:

STR\$(14)	Converts 14 to "14".
LEN("ABC")	Returns the length of the string. 3 is returned since the string has three characters.

LEN(A\$+B\$) Returns the combined length of the two strings.

LEFT\$(ST\$,1) Returns the leftmost character of the string ST\$.

The following functions constitute the format function group:

SPC	Space over.
TAB	Tab over.
POS	Next print position.

You will use format functions to display or print information using any desired format. SPC and TAB take a single numeric argument. POS uses a dummy argument of 0.

Examples:

SPC(10)	Move cursor over 10 spaces.
TAB(A)	Move cursor to column A+1.

Finally there are some “system” functions you will not use until you are a more experienced programmer. **The system function group includes these functions:**

PEEK	Fetches the contents of a memory byte.
TI\$,TI	Fetches System Time, as maintained by a program clock.
FRE	Returns available Free Space — the number of unused read/write memory bytes.
SYS	Transfers to Subsystem.
USR	Transfers to User Assembly Language Program.

System functions are useful only in certain classes of special programming. They are not used in typical PET programs.

PET BASIC also allows you to define your own functions with the DEF FN function.

These functions are all described in detail in Chapter 4.

FILE NAMES

Another different and important use of PET alphabetic and numeric characters is in the creation of file names.

Since information cannot be held forever in the PET's read/write memory, you must store this information elsewhere — usually on cassettes, although alternatives, such as diskettes, are now available. Cassettes and diskettes both have magnetic surfaces on which information is stored magnetically. Information may consist of programs or data used by programs. In either case the cassette or diskette is the magnetic equivalent of a filing cabinet, and the program or data is therefore referred to as a file. A program or data file is, physically, a piece of magnetic surface on which the program or data has been stored. You do not need to know how or where your program or data is stored, because when you need it, you ask for it by name. **When you first record a file, you assign it a file name. Subsequently you use the file name to identify the program or data,** so that the PET can read it back into read/write memory.

When you buy a pre-programmed tape, it usually has several programs on each side. Each program has a file name. You can load the program you want to run by specifying the name of the program in an appropriate PET command. For example, suppose a tape has programs stored in this order:

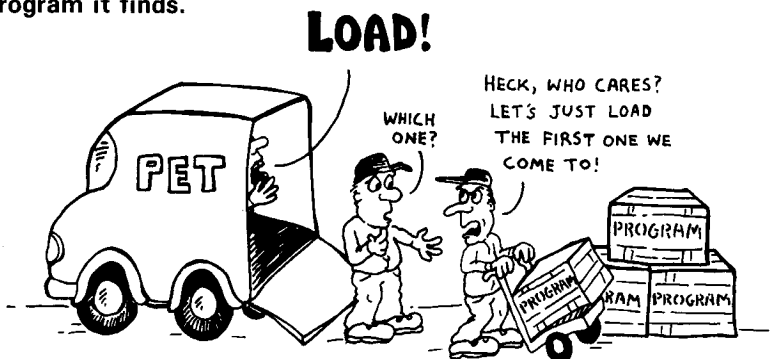
GAME 1
HIHO
BURGLAR

You want to play HIHO. The LOAD command tells the PET to load a file from cassette tape into the PET's read/write memory. To load HIHO into the PET's read/write memory, type in the command:

LOAD "HIHO"

The PET will find GAME 1 first; but continues searching for the program with the name HIHO. It finds it on the second try and loads the program into read/write memory.

If the file name is not specified in a LOAD command, the PET loads the first program it finds.



So if you don't specify the file name, you will have to give multiple LOAD commands, one for each program up to the one you want loaded.

Unlike regular variable names, a file name can have up to 128 significant characters. However, the length is effectively limited to two display lines when typing in file names from the keyboard. Characters in a file name may be any alphanumeric or numeric characters. Here are some examples:

```
TEST PROGRAM 1
TP1
EGGHEAD
XYZ
X1
X2
```

You can abbreviate the file name in the LOAD command. The command:

```
LOAD "TEST"
```

would load TEST PROGRAM 1.

```
LOAD "XY"
```

would load XYZ. However,

```
LOAD "X"
```

would load XYZ, X1, or X2, whichever the PET encountered first on the tape.

The abbreviated file name (like XY and X above) does not have to be the first characters in the file name. The PET searches for a file name match on a character-by-character basis.

This can be a boon if, say, you have forgotten the exact file names of several programs, but you think you saved them as:

```
PROG 1
PROG 2
PROG 3
PROG 4
```

when in fact you had actually saved them as:

```
PGM 1
PGM 2
PGM 3
PGM 4
```

You want to load the last program of the bunch, so you try the command:

```
LOAD "PROG 4"
```

The PET will go right past PGM 4 without loading it, since the file names do not match. Of course, you could give successive LOAD commands without specifying a file name. But you can also try the command:

```
LOAD "4"
```


Since no previous program has the character 4 in it, PGM 4 will be loaded. Because of the character-by-character search for file name matches, you have to be careful when giving abbreviated names, especially if your file names are long. For example, suppose you save three programs successively as:

```
WHILE AWAY THE HOURS
WHILE A
WHILE B
```

You will not be able to load the second program with a LOAD "WHILE A" command. The first program would always be loaded, since it contains that character subgroup.

There is something else you must bear in mind when you make up file names. **Although the file name may have up to 128 characters, only the first 16 characters are displayed by the PET.** If you save two programs successively as:

```
SWEET SIXTEEN TEST 1
SWEET SIXTEEN TEST 2
```

the programs are subsequently listed as:

```
SWEET SIXTEEN TE
SWEET SIXTEEN TE
```

so you can't tell which is which by examining file names as they are displayed. In summary, you will find it helpful to compose file names using 16 characters or less, and you should keep file names about the same length.

WRITING BASIC PROGRAMS

A BASIC program consists of a group of numbered statements. In contrast to immediate mode, statements within a program are not executed until you "run" the program. In other words, when you type in program statements, nothing happens until you specifically initiate program execution. When a statement begins with a line number, PET BASIC stores the statement rather than executing it immediately. A line number is put on a statement to indicate that it is part of a stored program.

LINE NUMBERS

Every program statement begins with a line number that must be unique within the program and therefore uniquely identifies the statement.

A BASIC statement becomes part of a program if you begin the statement with a line number. The number becomes the statement's identification, just as numbers are used to identify individual houses on a street. For example, if you type:

```
10 A=π*2
```

instead of:

```
A=π*2
```

you create a statement that becomes part of a program that will not execute until you specifically run the program.

Every statement in a BASIC program must have a line number. **The numerical value of the line number specifies the order in which the statements in the stored program are to be executed when the program is run. The smallest line number identifies the first statement in the program and the largest line number identifies the last statement in the program.** Statements in between will be organized in order of ascending number sequence. The PET BASIC interpreter takes care of that for you. If you enter a statement out of its number sequence, the BASIC interpreter will move the number to its correct sequential position (whether you want it there or not).

Remember, **the PET BASIC interpreter will always arrange BASIC statements in ascending order of line numbers.** A number at the beginning of a BASIC statement puts the statement into a program which does nothing until you deliberately run it. When running a program, statements are executed one at a time. Statements are executed in sequence of ascending statement numbers (unless a statement explicitly changes the execution sequence by branching to a statement elsewhere in the program).

```
•  
•  
•  
110  
120  
123  
129  
137  
•  
•  
•
```

On the PET, **a line number can be any integer from 0 to 63999.**

$0 \leq \text{line number} \leq 63999.$

Programs and Immediate Mode

Let us look at how **immediate mode can be used to examine values assigned to labeled variables within a program.**

If statement 120 below were in a program and you wished to find out the value of the component variables in that statement, you could use calculator mode to print them.

```
10  
:  
:  
:  
:  
:  
:  
120 A=B*C/D  
999 END
```

} Program stored
in memory

```
READY.  
?B,C,D
```

15 1 8

This illustration assumes that when the program ends, B is assigned the value 15, C is assigned the value 1, and D is assigned the value 8.

The current values of variables remain available until you begin a new program, or rerun the same program.

BASIC STATEMENT GROUPS

We are now going to describe the various BASIC statements that you may use in a PET program. PET BASIC statements are described twice:

1. In the balance of this chapter we will describe PET BASIC statements briefly by functional groups. This description is intended to familiarize you with the types of capabilities available in PET BASIC without becoming buried in detail. This description is also sufficient to provide background for the discussion of programming concepts that follows.
2. In Chapter 4 we describe PET BASIC statements individually, with full detail.

While reviewing these statements, remember that PET BASIC has a number of "characteristics" that give it a unique personality. These characteristics are described in Chapter 5.

Unconditional Branches: GOTO, FOR . . . NEXT, GOSUB/RETURN

Unconditional branches change the flow of program execution from the next sequential statement to a specified statement.

GOTO is the simplest unconditional branch. It designates a line number at which program execution is to continue.

```
55 GOTO 100
```

Transfer program control to the statement with line number 100.

FOR . . . NEXT provides loop control; the statements between the FOR statement and the NEXT statement are executed as many times as specified by the FOR index variable.

```
10 FOR I=1 TO 10
20 ?I
30 NEXT I
```

These statements will print the numbers 1 through 10 in a vertical column on the display screen.

The optional word STEP specifies the size of the increment to be added to the loop index.

```
10 FOR I=10 TO 100 STEP 10
20 ?I
30 NEXT I
```

These statements print the numbers 10, 20, 30, . . . 100 on the display screen.

GOSUB and **RETURN** provide the means of branching program control to a subroutine and then returning to the statement immediately following the subroutine call.

Subroutine:

```
1000 FOR I=A TO Z STEP S
1010 PRINT I
1020 NEXT I
1030 RETURN
```

Calls:

```
50 A=1:Z=10:S=1
60 GOSUB 1000
:
200 A=10:Z=100:S=10
210 GOSUB 1000
:
999 END
```

These statements illustrate the PRINT loop introduced above structured as a subroutine. The first subroutine call will print the numbers 1 through 10 (as in the first FOR . . . NEXT loop example). The second subroutine call will print the values 10, 20, 30, . . . 100 as in the second FOR . . . NEXT loop example.

IF ... THEN is the primary decision making tool in PET BASIC programs. It has two parts: a condition to be tested placed just after the IF; and if the condition is satisfied, one or more actions to be taken placed after the THEN. We have already used this statement frequently. Here are a few more examples:

```
10 IF A=B+5 THEN PRINT MSG1
40 IF CC$<"M" THEN IN=0
41 IF Q<14 AND M<>M1 GOTO 66
```

ON ... GOTO and **ON ... GOSUB** provide a multiple branch capability. The following example provides a three-way branch to line 10, 72, or 50 depending on whether A=1, 2, or 3, respectively. If A is not in this range, no branch is taken.

```
40 ON A GOTO 10,72,50
```

Defining Data: LET ... =, READ/DATA/RESTORE, DIM

Statements for defining data allow you to specify constant data, assign values to variables, and dimension arrays.

The assignment statement **LET ... =**, or just **=**, is probably the most common BASIC statement. The word LET is optional and is usually omitted.

```
10 A=10
20 B=X+Y-Z+3
25 BX(24,2)=AL
21 FOR I=1 TO 10:A(I)=X+1
```

↙ In this statement, only the second = is an assignment statement.

The **READ** statement assigns to variables the values contained in accompanying **DATA** statements.

```
10 DATA 10,20,-4,16E6
20 READ A,B,C,D
```

RESTORE restarts the next READ at the beginning of the first DATA statement.

```
30 RESTORE
40 FOR I=1 TO 4:READ A(I)
```

The dimension statement **DIM** declares the size and dimension of arrays.

```
100 DIM A(3)
```

Console Input/Output: PRINT, INPUT, GET

Console input/output statements provide the ability to interact with the keyboard/display screen during program execution.

PRINT prints data on the display screen. **PRINT** and **?** are equivalent.

```
10 PRINT "NAME"
30 ?A,B,C
```

Both INPUT and GET input data from the keyboard to a program. **INPUT** first prints a question mark, then inputs a line of data, terminated by a carriage return. One or more variables may be input with one INPUT statement.

In response to execution of the statement:

```
1000 INPUT A,B$
```

if the following is input at the keyboard:

```
10,"SIMPLE ONE"
```

then A=10 and B\$="SIMPLE ONE".

INPUT can also print a message before inputting:

```
4001 INPUT "NAME":A$
```

Upon execution, this statement prints on the display:

```
NAME?
```

If you enter the name Susanna:

```
NAME?SUSANNA
```

the result is that A\$=SUSANNA.

GET inputs one character only. This is useful when you want just one-character responses, since the user does not need to enter a carriage return.

Unlike INPUT, where PET BASIC handles the input process, you must program everything yourself when using GET. Your program must wait for a key to be pressed and then check to see if an appropriate key has been pressed. Use a string variable with GET. If you use a numeric variable, the PET returns a zero either if no key has yet been pressed or if a zero is entered. When the GET variable is a string, a null value is returned until a key is struck. The character you input in response to a GET will not be displayed unless you include a statement to cause the display.

The following line waits until any key is pressed:

```
10 GET C$:IF C$="" GOTO 10
```

The following group prints a message and waits until a G is pressed on the keyboard. It does not respond to any other key. When a G is pressed, it displays the character, waits a little bit, then clears the screen and prints the message "HERE WE GO."

```
500 PRINT "PRESS G WHEN READY"  
510 GET C$:IF C$<>G THEN GOTO 510  
520 PRINT C$  
530 FOR I=1 TO 100:NEXT  
540 PRINT "HERE WE GO"
```

File Input/Output: OPEN, CLOSE, PRINT#, CMD, INPUT#, GET#

File input/output statements provide programmed communication with the cassette tape unit or any other external peripheral unit that can input and/or output data.

OPEN assigns an identifier, called a logical file number, to a physical device and readies the physical device for an input/output operation. Physical devices have preassigned numbers. Logical file numbers are made up by the programmer, just as variable names are.

```
10 OPEN 5,1,0,"DATAFILE"    Assigns logical unit 5 to physical device 1
                             (the console cassette unit) and opens the
                             cassette for read (code 0) from file name
                             DATAFILE.
```

CLOSE is executed when all accesses to the logical file have been completed.

```
500 CLOSE 5                Close logical file 5.
```

PRINT# operates essentially like PRINT but outputs to an external device.

```
40 PRINT#3,A              Output the value of variable A to logical file 3.
```

CMD opens communication with an external device so that you can print to an external device what is normally displayed on the screen.

```
40 CMD 3                  Output subsequent log to logical file 3.
```

INPUT# operates somewhat like READ, but it inputs data from an external file rather than from DATA statements in the program, as READ does.

```
50 INPUT#10,A             Input and assign to variable A the next data
                           item from logical file 10.
```

GET# allows you to fetch external data one character at a time.

```
60 GET#7,C$              Input and assign to variable C$ the next data
                           item from logical file 7.
```

None of these statements is used alone in a program; rather, successful input or output to an external device requires a certain sequence of file input/output statements, all having compatible parameters, that together provide an input or output capability. The format of these statements is described in Chapter 4. Using these statements to perform file I/O operations is described further in Chapter 5, File Input/Output.

Miscellaneous: REM, POKE

REM is a remark, or comment line. Remarks should be used frequently in a program to remind the programmer what is happening.

```
10 REM FETCH DATA IF BUFFER EMPTY'
```

POKE stores a specified byte value into the PET's memory. POKE is similar to a system function in that it is instrumental in performing certain specialized chores; these are described in Chapters 5 and 6. POKE is not used often.

56 POKE 1,AD Store the value of variable AD in memory location 1.

Program Termination: **END, STOP, WAIT**

Program termination statements end, or temporarily halt, a PET BASIC program that is executing. These statements are optional; in particular, an END statement is not required to end a PET BASIC program.

Examples:

500 END Provides two program termination points.

.

.

.

600 END

60 STOP

Performs a programmed BREAK function, like the STOP key from the keyboard. Program execution can be continued by typing CONT from the keyboard.

100 WAIT 525,1

Wait until memory location 525 becomes 1, then continue program execution. WAIT is a special statement not used in normal programming chores.

BASIC Commands: **NEW, CLR, LIST, LOAD, RUN, CONT, SAVE, VERIFY**

BASIC commands are a group of operations that are normally used in immediate mode.

Examples:

NEW Start new program; set variables to zero or null.

CLR Set variable to zero or null.

LIST List current program.

LOAD "ASKME" Search tape for program named ASKME and load it into memory.

RUN Execute program.

CONT Continue program execution at line following last STOP.

SAVE "PETPRO" Write current program to tape cassette unit.

VERIFY "PETPRO" Verify the tape copy of the program named PETPRO by comparing it with the program in memory.

DEVELOPING A PROGRAM

When the PET is powered up it is in immediate, or calculator mode. Entering program mode is simple: just begin entering statements that are preceded by line numbers. Remember that programs are executed in the ascending order of the line numbers, which can range from 0 to 63999.

You will find when developing a program that you will have to change, or edit, your program at least a little, and probably a lot, before you get it running. Because of this, it is helpful to **select line numbers that increment by 5, 10, or even 100**, rather than incrementing by 1. If you have to insert, say, three statements between two existing lines of a program, you can do so without having to renumber any of the existing lines. Do not renumber existing lines in a program unless you absolutely have to; it is easy to cause errors in loops and GOTO statements by not changing the line number everywhere it is referenced.

You will normally key in statements, from your handwritten notes, in the order in which they are to be executed. However, PET BASIC does not require statements to be entered in order. You could key in statement 10, then statement 100, then statement 20. The PET sequences the statements by line number before listing or executing the program.

We return now to the "mini-program" presented at the beginning of this chapter.

```
C$="W":FOR I=1 TO 800:PC$:NEXT?"PHEW!"
```

This line entered in immediate mode prints the letter W (or whatever character you put between the quotes) in each position of the next twenty lines of the display screen. It ends by printing the word PHEW!

We will describe in a little more detail how this immediate program works. Then we will develop this program into a stored program to illustrate the process so that you will feel more comfortable approaching the task of creating, listing, running, debugging, and editing programs on the PET.

Multiple Statements on a Line

There are actually five statements in this one-line immediate mode program. Each statement is separated by a colon (:). The format for multiple statements on a line in immediate mode is:

```
statement:statement . . . :statement
```

The format is the same for one line of a stored program, except that the line has a line number:

```
line number statement:statement . . . :statement
```

Only one line number is needed for multiple statements placed on one line.

In immediate mode, you must put multiple statements on one line. If you attempted to enter the display program one statement at a time, the result would not be the same. Look at the following example:

```
C$="W"

READY.
FOR I=1 TO 800

READY.
?C$
W

READY.
NEXT

READY.
?"PHEW!"
PHEW!

READY.
*
```

The first statement can be entered separately, since variable assignments are stored. But just one W is printed, instead of twenty lines of W's, because the FOR . . . NEXT loop statements are not executed as a unit. In immediate mode, the PET executes one line with no memory of what the previous line was. The only successes were the assignment statement C\$="A" and the ?"PHEW!". In immediate mode you do access to the current values of all variables. Colons can be used to execute "mini-programs" up to 80 characters (two display lines) long in immediate mode.

In program mode, putting multiple statements on one line is handy for some uses, but is indispensable in only one case.

Where two, or sometimes more, statements perform a single operation, they can optionally be placed on one line and separated by colons:

```
10 GET C$:IF C$="" GOTO 10
```

or

```
10 GET C$
20 IF C$="" GOTO 10
```

The only case in a stored program where multiple statements on one line are treated differently than the same statements on separate lines is with an IF statement. **Multiple statements appearing after the IF . . . THEN statement are executed only where the IF condition is satisfied.** Consider the following line:

```
50 IF A=0 THEN B=C:C=C+1
```

Here there is a second statement following the IF . . . THEN statement. If the condition is satisfied (A is equal to 0), both B=C and C=C+1 are executed. If the condition is not satisfied (A is not equal to 0) neither of these actions is executed. Note particularly that C is not incremented, whereas for the statements:

```
50 IF A=0 THEN B=C
55 C=C+1
```

C is incremented whether or not the IF condition is satisfied, since it is on a separate line from the IF . . . THEN statement.

Multiple statements on one line in a program save some memory space, since there is only one line number to keep track of. In the general case, it is suggested that you do not make greater use of multiple statements on a line than necessary, as in the case of performing several actions for a single IF condition. Compacted programs are difficult to read, edit, and debug (errors in programs are commonly referred to as "bugs," therefore the term "debug" means to eliminate errors).

PRINT Formats: Line, Continuous, Tabbed

Normally each PRINT statement prints a new line on the display. That is, a RETURN is forced, so that the next PRINT statement begins printing at the beginning of the next line. **This can be overridden by placing a semicolon (;) at the end of the PRINT statement.** There is an example of this in the display program:

```
C$="W":FOR I=1 TO 800:PRINT C$:NEXT I:"PHEW!"
```

↑
_____Continuous line format (;)

The semicolon in this PRINT statement is what provides continuous line printing of the character over 20 lines of the display. If the semicolon were not there, each character would be printed at the beginning of a new line, or you would see all 800 W's printed in a single column at the left of the screen.

```

C$="W":FOR I=1 TO 800: C$:NEXT:?"PHEW!"
W
W
W
W
W
.
.
.
W
W
W
W
W
.
.
.
W
W
W
W
W
PHEW!

READY.
※

```

} Display appears to blink
as W's are written over W's

The display appears to blink as W's are written over W's. After twenty lines have been printed, the program statements scroll off the top of the screen. To stop the overprinting of W's, change the TO index from 800 to 20 (since we are always printing a total of 20 lines in the program).

```

C$="W":FOR I=1 TO 20: C$:NEXT:?"PHEW!"

```

```

C$="W":FOR I=1 TO 20:? C$:NEXT:?"PHEW!"
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
PHEW!

READY.
※

```

This is not nearly as interesting a display as continuous line printing of the characters. Single line printing is more commonly done for different values. If you print the value of I rather than C\$ you can print the numbers 1 through 20.

```

FOR I=1 TO 20:? I:NEXT:?"PHEW!"
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
PHEW!

READY.
※

```

Return now to the original program line that gives continuous line printing by the presence of a semicolon.

```
C$="W":FOR I=1 TO 800:? C$):NEXT?"PHEW!"
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
PHEW!

READY.
※
```

The FOR . . . NEXT loop index I is used as a counter to indicate the number of W's to be printed, in this case 800. For the first PRINT, a new line is begun and the character W is printed. The semicolon prevents a RETURN to the next line, so the cursor remains at the position following the first W. The second W is then printed, with the cursor left in the position following it, and this pattern continues. At the end of the line, the cursor moves to the next position, which is at the beginning of the next line. This is continuous for 20 lines.

Why does PHEW! print on a new line? It doesn't really; it appears to start a new line because the last character prints in the last position of the previous line. Modify the value of 800 to, say, 780 and PHEW! prints on the same line as the end of the characters. Reenter the line to get the display shown below.

```
C$="-":FOR I=1 TO 780:?C$):NEXT?"PHEW!"
```



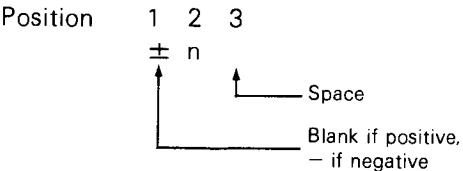
```

C=+5:FOR I=1 TO 267:PC: NEXT:?"PHEW!"
5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
PHEW!

READY.
※

```

When you run this program, note that there is one space between the last number printed and the word PHEW! This shows that the numbers are printed in the following format:



Another example of this format can be obtained by changing the value of C to minus 1:

```

C=-1:FOR I=1 TO 267:PC: NEXT:?"PHEW!"

```



```

C=-1:FOR I=1 TO 267:PC: NEXT:"PHEW!"
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -
PHEW!

```

```

READY.
**

```

Up to now the program line has printed just single characters or single digits. Multiple-digit numbers can be printed, but they will scroll the program line up off the screen unless the TO index is also adjusted. If we change the value of C to 2001, we will have a 6-digit print field, so adjust the TO index from 267 to $800/6=134$:

```

C=2001:FOR I=1 TO 134:PC: NEXT:"PHEW!"

```

```

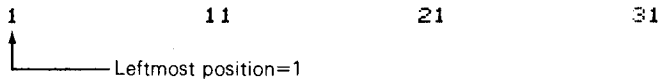
C=2001:FOR I=1 TO 134:?:NEXT:?"PHEW!"
 2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
 2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
 2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
 2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
 2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 PHEW!

```

READY.
 ※

When you print this example, you can see that the continuous line format breaks the numbers between lines. This can be eliminated by using a tabbed line format.

The tabbed line format, indicated by a comma (,), prints at four tab positions every ten positions:



In the display program, **inserting a comma in the PRINT statement**, instead of a semicolon, **causes the numbers to be printed out in four columns**. At four numbers per line, the TO index will be $4 \times 20 = 80$. Type in the line:

```
C=2001:FOR I=1 TO 80:?,NEXT:?"PHEW!"
```

When you run this program, note that the first position in each field is reserved for the sign.

```
C=2001:FOR I=1 TO 80:PC,:NEXT:?"PHEW!"
```

```

2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
2001       2001       2001       2001
PHEW!
```

```

READY.
```

The use of commas between print items is a convenient way to print with tabs in PET BASIC. You don't have to do anything in your program except use commas to separate print items.

Tabs also work for strings. As an example, enter the following immediate mode lines to print out twenty lines of tabbed character data:

```
A$="HUP!":B$="TWO!":C$="THREE!":D$="FOUR!"
FOR I=1 TO 20:PA$,B$,C$,D$:NEXT:?"PHEW!"
```

We use continuous line printing in the display program line because it is the most effective of the three print formats — single line printing, continuous line printing (:), or tabbed printing (,) — for the display program. As we have seen, numeric data does not blanket the screen, even with continuous line printing, because PET BASIC leaves a space between the numbers.

To discover some other features of PET printing, return to the display for the number 2001.

```
C=2001:FOR I=1 TO 134:PC:NEXT:?"PHEW!"
2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 2001 2001 2001 2001 2001 2001
2001 2001 2001 2001 2001 2001 200
1 2001 2001 2001 2001 2001 2001 2
001 PHEW!
```

```
READY.
*
```

Now change the value of C from 2001 to 44 following these instructions: HOME the cursor, CURSOR RIGHT two positions and type in 44, CURSOR RIGHT once and DELETE twice. With a 2-digit number, the print field is 4 (one for the sign, two for the number, and one trailing space), so the TO index will end at $800/4=200$. Change the TO index from 134 to 200: CURSOR RIGHT ten spaces to position the cursor on the 1 of 134 and type in 200. Now press RETURN and (surprise!) you will get the display shown below.

```

C=44:FOR I=1 TO 200:PC):NEXT:"PHEW!"
440 442 44 440 442 44 440 442 44 440
442 44 440 442 44 440 442 44 440 442
44 440 442 44 440 442 44 440 442 44
440 442 44 440 442 44 440 442 44 440
442 44 440 442 44 440 442 44 440 442
44 440 442 44 440 442 44 440 442 44
440 442 44 440 442 44 440 442 44 440
442 44 440 442 44 440 442 44 440 442
44 440 442 44 440 442 44 440 442 44
440 442 44 440 442 44 440 442 44 440
442 44 440 442 44 440 442 44 440 442
44 440 442 44 440 442 44 440 442 44
440 442 44 440 442 44 440 442 44 440
442 44 440 442 44 440 442 44 440 442
PHEW!HEW!

```

READY.

※

Some of the digits from the previous 2001 display were not blanked out. **PET BASIC uses a skip (cursor right) character, not blanks, between fields. When you are printing over existing data, characters between fields — or characters in tabbed format — are not erased.** Also note that there is a remaining "HEW!"; PET BASIC prints the "PHEW!" but leaves the remaining positions of the line just as they were; it does not blank the rest of the line. **This can be a great advantage when you are adding to data already on the screen, and you should bear this capability in mind.** For the display program line, however, it is leaving extraneous characters in the display.

Programmed Cursor Control

To remove extraneous characters from the display, you can have the program clear the screen before printing a new display. To do this, insert a PRINT CLEAR SCREEN literal ahead of the FOR . . . NEXT loop:

```

C=44:"C":FOR I=1 TO 200:PC$):NEXT:"PHEW!"

```

← Clear Screen (shift of CLR/HOME key)

Now when you press RETURN, you will see the screen blank and the numbers begin printing on the second line.

```
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44
PHEW!
```

```
READY.
※
```

To begin printing on the first line, insert a semicolon after the PRINT CLEAR SCREEN. Change the program now to print characters again. With the extra line of forty characters, the program can print 840 characters without scrolling any off the top. Type in the line:

```
C$="A"??"":FOR I=1TO840:PC$;:NEXT?"PHEW!"
```

The only real change from the original line is the CLEAR SCREEN. The CLEAR SCREEN adds a nice touch to the display program. The display blanks immediately, instead of gradually scrolling previous text off the top. End the same line again with the same character. You get a new display with every character reprinted instead of having duplicate characters appear not to be printed again.

Any of the six cursor control functions can be operated under programmed control by printing them as string literals. The cursor control string symbols and keys are listed in Table 3-1. Programming cursor controls is described in Chapter 5.

A Stored Program

The biggest problem with a program entered in immediate mode is that you have to reenter it every time you want to run it, as with the display program. The solution is to turn it into a stored program.

The program has already been developed in immediate mode. Now that the groundwork is done, making a stored program out of the program that we know works in immediate mode is simple. Just type in the statements line by line, beginning each statement with an increasingly larger line number. Remember that you still need the semicolons, which are the PRINT format, but not the colons when putting statements one per line. Terminate each line with a RETURN.

```
READY.  
100 C$="A"  
110 ?"Q";  
120 FOR I=1 TO 840  
130 PRINT C$;  
140 NEXT  
150 PRINT"PHEW!"
```

Each time a numbered line is entered, PET stores the line in memory. You will not have to type the display program line in again now that it is a stored program. You then use the LIST, RUN, and SAVE commands to recall and rerun your program.

Listing a Program

When the display program looks correct, list it and look it over again. To list a program, type the word LIST on any free line and hit RETURN.

```
LIST  
  
100 C$="A"  
110 ?"Q";  
120 FOR I=1 TO 840  
130 PRINT C$;  
140 NEXT  
150 PRINT"PHEW!"  
READY.
```

Note that the ?'s you entered have been expanded to the word PRINT; PET does this automatically. If you see any errors, correct them.

Running a Program

When the program looks completely correct, execute the program by typing RUN, followed by a RETURN.

```
RUN
```

If the program was typed in correctly, you should see the same display we have used throughout this chapter, only with the letter A.

AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
AA
PHEW!

READY.
✳

Enter RUN to run the program again. The program executes the same as before, without having to type in the display program line.

Editing a Program

Running the program to display A's quickly becomes boring. You can change the display character by editing line 100. **To edit a line, the line must appear on the screen.** You don't need to list the entire program, just one line. To list just one line (line 100, for example), use the following command:

```
LIST 100  
  
This lists just the line specified.  
  
LIST 100  
  
100 C#="X"  
  
READY.  
✳
```

Move the cursor to the A, change it to another character, say X, and press RETURN. If you list the entire program, you will see that the change has indeed been made. **Program editing can be done merely by changing a statement that appears on the screen, then pressing RETURN.**

LIST

```
100 C#="X"  
110 ?"Q";  
120 FOR I=1 TO 840  
130 PRINT C#;  
140 NEXT  
150 PRINT"PHEW!"
```

To delete an entire line, type the line number followed immediately by a RETURN. When you list the program, you will see that the entire line and line number are no longer included in the program.

To add a line in a program, type in the line, no matter what the line number, at the first blank line after the end of the current program. When you list the program, you will see that the line you just entered has been placed in its proper numerical sequence within the program.

When you type in a line with the same line number as a line that already exists in the program, the new line replaces the existing line in the program after you press RETURN. You can use this feature to edit lines that are easier to change by retyping than by character correction techniques.

Saving a Program

Now that you have a program in PET's memory that works, **you can store the program on a cassette. Once a program is stored on cassette, you can load it into memory and run it any time.**

To save a program in memory on cassette, put a blank cassette tape in the console cassette unit and type the SAVE command. Although it is not necessary, it is a good idea to name the program; we'll name it BLANKET.

```
SAVE "BLANKET"
```

```
PRESS PLAY & RECORD ON TAPE #1 ← Hold down REC key,  
OK                                then press PLAY key.
```

```
WRITING BLANKET  
READY. ← Tape stops.  
⌘
```

Rewind the tape to the beginning, label it and keep it in a safe place.

When you turn the PET off, the contents of memory are not saved, so saving a program (especially a working program) on tape is a good idea even at intermediate stages of program development.

Loading a Saved Program

If you turn the PET off, the program in memory is lost. To load the saved program, power up the PET, insert the cassette into the console tape unit, and give the LOAD command.

```
LOAD "BLANKET"
```

```
PRESS PLAY ON TAPE #1 ← Press PLAY key; tape begins moving.  
OK
```

```
SEARCHING FOR BLANKET  
FOUND BLANKET  
LOADING
```

```
READY. ← Tape stops.
```

```
※
```

To list the program, use the LIST command. You should see your entire BLANKET program. To execute the program, enter RUN. It will display the character that was last stored in line 100.

Or, to load and execute a program from cassette tape with the RUN key, ready the tape and depress the RUN key. The program will load and execute.

Interactive Programming

This is the current procedure for using program BLANKET once you load it into memory:

1. List line 100.
2. Move the cursor to the character between quotes and type the desired display character, then press RETURN.
3. Enter the command RUN.

It would be better if you could **change the character without having to list a line of the program**. Recall that assignment statements can be entered separately in immediate mode. Can you use an immediate mode assignment statement to change the display character?

To try it, first eliminate the assignment statement in the program. To delete a program statement, type the line number followed immediately by a RETURN.

```
LIST 100
```

```
100 C$="A"
```

```
READY.
```

```
100 ← Type line number, then key RETURN.
```

```
LIST
```

```
110 PRINT "J";  
120 FOR I=1 TO 840  
130 PRINT C$;  
140 NEXT  
150 PRINT "PHEW!"  
READY.
```

```
※
```

Line 100 is no longer in the program.

Type in the statement C\$="X" in immediate mode, then run the program.

You can repeat the command GOTO 110 to get the same display. Now the procedure for running the program is as follows:

1. In immediate mode enter the assignment C\$="y" where y is any display character.
2. Enter the command GOTO 110.

There are only two steps in running the program, but the procedure is still a bit awkward. You must type in a line (the assignment statement), and if you enter RUN instead of GOTO you have to start all over. A much better way would be to **have the program fetch the display character while it is running**. You can do this with the GET statement.

The GET statement assigns a single character entered from the keyboard to the specified string variable.

When PET BASIC begins execution of the GET statement, it sets the string variable (in our case C\$) to null, that is, C\$="". As soon as a key is pressed at the keyboard, it assigns the character to the string variable. Since the program runs much faster than you can press a key, you add an IF statement to check that a key has been pressed before the program continues.

Add the following line to the program by typing in the line. You do not need to list the program before adding the line.

```
100 GET C$:IF C$="" GOTO 100
```

Now list the program and make sure you entered the line correctly.

```
LIST
```

```
100 GET C$:IF C$="" GOTO 100
110 PRINT "C";
120 FOR I=1 TO 840
130 PRINT C$;
140 NEXT
150 PRINT "PHEW!"
READY.
❖
```

Run the program. The screen blanks and the cursor disappears. Press any data key. You should see the display program display the character of the key you pressed. Run the program again. Press any data key. The display appears with the new character.

The procedure for running the program is now:

1. Enter the command RUN.
2. Press any key.

This is a real improvement over the original program. However, it is a little disconcerting to have the screen completely blank out while it waits for you to press a key. You can add a line to the beginning of the program that **prompts the user to press a key**. Type in the line:

```
90 ?"HIT A KEY"
```

List the program and check the new line for any errors.

LIST

```
90 PRINT "HIT A KEY"
100 GET C$:IF C#="" GOTO 100
110 PRINT "J";
120 FOR I=1 TO 840
130 PRINT C$;
140 NEXT
150 PRINT "PHEW!"
READY.
*
```

Now when you run the program it gives instructions for using it, so you are sure of what to do. Run the program several times to display different characters and note how much easier the program is to use.

There is one important modification left to make. You will want to **run the program more than once**— that is, display a character, then display another one, etc. By modifying the program to go back to its beginning point instead of ending, you won't have to type in RUN each time. Add a line at the end of the program to jump back to the beginning of the program:

```
160 GOTO 90
```

Again, display the program and check the new line.

LIST

```
90 PRINT "HIT A KEY"
100 GET C$:IF C#="" GOTO 100
110 PRINT "J";
120 FOR I=1 TO 840
130 PRINT C$;
140 NEXT
150 PRINT "PHEW!"
160 GOTO 90
READY.
*
```

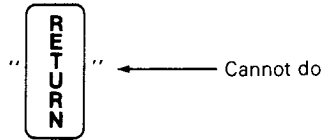
Now it is even easier to use the program. Enter RUN and follow directions.

Of course, **you have to use the STOP key to exit from the program. This can be eliminated by programming for one particular key, when pressed as a data key, to signal termination of the program.** For example, if the RETURN key were pressed instead of a data key, the program could end. To do this, add a check for carriage return after a key has been pressed.

For all data keys, as well as the cursor control keys, you can check for a key as a string literal. For example, to branch to statement 200 if the Y key is pressed, you would enter the following statements:

```
100 GET C$:IF C#="" GOTO 100
105 IF C#="Y" GOTO 200
```

RETURN presents a special problem that other keys do not. You cannot reference RETURN as a string literal:



This is because any time RETURN is pressed, PET BASIC stores the program line in memory and goes to the beginning of the next line. You can, however, use the CHR\$ function to check for a RETURN key. CHR\$ allows you to assign an ASCII code value to a string variable and treat it as a string. The ASCII code value for a RETURN (referring to Appendix A) is 13.

Before programming the check for carriage return, consider what to do if there is one. The last line of the program branches back to the beginning of the program. To terminate program execution, you need to branch beyond that. Add the following line:

```
170 END
```

Now add the check for RETURN as line 105:

```
105 IF C#=CHR$(13) GOTO 170
```

Note that we could have written, in place of line 170 and line 105:

```
105 IF C#=CHR$(13) THEN END ← Option
```

If you choose this option, it is generally good programming practice to have the program termination point at the physical end of the program. It is more difficult to find termination points embedded in the program.

Without the PET's READY message being printed each time, there are two additional lines available on the screen. This allows 80 more characters (at 40 characters per line) to be printed. Change the 840 in line 120 to $840+80=920$. Line 120 will read:

```
120 FOR I=1 TO 920
```

List the program; it should look like this:

```
LIST
90 PRINT "HIT A KEY"
100 GET C#:IF C#="" GOTO 100
105 IF C#=CHR$(13) GOTO 170
110 PRINT "☐";
120 FOR I=1 TO 920
130 PRINT C#;
140 NEXT
150 PRINT "PHEW!"
160 GOTO 90
170 END
READY.
☐
```

When you run the program, you will find that it is scrolling up one line, leaving a blank line at the bottom of the screen. This is because PET BASIC gives a RETURN after printing HIT A KEY to move to a new line in preparation for the next print. We can see this by enabling the cursor to blink.

Normally you cannot see the cursor because its blinking is inhibited by the PET before running a program. However, **you can enable the cursor to blink by adding the following statement to the beginning of the program:**

```
80 POKE 548,0 Enable cursor
```

This is a system location that is discussed further in Chapter 6. Run the program with this line added and you will see the cursor blinking at the bottom line.

```
.
.
PHEW!
HIT A KEY
*
```

This program does not really need the cursor, so delete line 80.

To prevent the blank line, add a semicolon to the PRINT statement in line 90. We should also add a prompt that RETURN is what is used to exit from the program. To incorporate these changes, line 90 should now be edited as follows:

```
90 ?"HIT A KEY OR <CR> TO END";
```

As a final task, you might read over the program and add remarks for any lines that are not clear. Comment on how the number 920 was devised; you can optionally put the remark on the same line, using a colon to separate statements:

```
120 FOR I=1 TO 920 :REM 920/40=23 LINES
```

Add a reminder that `CL` is to clear the screen; optionally align the remarks:

```
110 PRINT"CL"; :REM CLEAR SCREEN
```

Finally, add a few lines at the beginning of the program to describe it. The final program BLANKET appears as follows. Save it on tape.

```
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY OR <CR> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"CL"; :REM CLEAR SCREEN
120 FOR I=1 TO 920 :REM 920/40=23 LINES
130 PRINT C$;
140 NEXT
150 PRINT"PHEW!"
160 GOTO 90
170 END
READY.
```

Figure 3-1. Program BLANKET

Spaces Are Not Needed

Are you struggling with the question of where to put spaces in the line and where not to? Don't worry. PET BASIC interprets a line by the elements in it. Spaces, or blanks, are irrelevant to the PET's decoding of a line. For example, the line:

```
120 FOR I=1 TO 210
```

could read:

```
120 FOR I=1 TO 210
```

or

```
120 FORI=1TO210
```

You can put extra spaces anywhere but within reserved words or other BASIC commands, such as GET# and INPUT#, with the exception of GOTO, which may be written as either GOTO or GO TO. The only place you must put spaces is within quotation marks where you want spaces to be part of the text string. Blanks in a statement are to improve readability of the program; do use them for this.

Character Correction Review

By now you are probably familiar with correcting typing errors on the PET. **If you make any typing errors, use the three character correction methods:**

1. **Typeover.** If you spot an error on the line you are typing, backspace (cursor left) to the character and type the new, correct character over the old, incorrect one. Then cursor right to where you stopped and continue typing.
2. **Delete.** If you spot an error on the line you are typing, press the DELETE key to move the cursor back to the end of what is correct and retype the line from there. **Use DELETE if you make an error inside quotation marks;** other cursor control keys will not function within text strings, as they are programmable in the strings.
3. **Insert.** If you have omitted one or more characters on the line you are typing, cursor left to the right edge of the omission and press the INSERT key once for each character to be inserted. Then type in the missing characters.

MODIFICATION EXAMPLE TASKS

Following are some sample modifications you can make to the BLANKET program. The idea here is that you will be dealing with a familiar program that works, without trivializing the modification. Programs 1 through 4 are loosely classed in the easy category, programs 5 through 8 intermediate, and 9 through 11 advanced. Try any or all of these modifications now or after you have finished reading this book. Working solutions are in Appendix C.

Program 1, CLEAN SCREEN. Have the program clear the screen before printing the HIT A KEY message.

Program 2, MESSAGES. Have the program print "HIT A KEY" when the program is first run but then have it print "HIT A KEY OR <R> TO END" for subsequent display.

Program 3, SINGLE. Combine the last two messages of the display onto one line so that the program prints:

```
PHEW!                      HIT A KEY OR <R> TO END  
          10 spaces
```

Have this line print on the bottom line. Increase character lines printed to 24.

Program 4, CALLING YOU. Structure the program as a subroutine. Add a main program that calls the subroutine and ends the program.

Program 5, SLOW DOWN. Slow down the display. The character display takes about 16 seconds to print. Lengthen this print time to about double.

Program 6, SPEED UP. Speed up the program. The character display takes about 16 seconds to print. Shorten this print time. The time can be shortened, using PRINT, to about two seconds.

Hint: You will need to decrease the number of PRINTs.

Program 7, INDIAN BLANKET. Allow input and display of different characters while the display is being printed. Modify messages as in Program 2, but with the following messages:

```
HIT ANY KEY  
HIT ANY KEY OR <R> TO END
```

At the same time slow down the display, as in Program 5, but make it even slower to allow time to press different keys while viewing the display — slow to about 75 seconds.

Remember when running the program that once you press a key, it automatically repeats until you press the next key. Can you get the display to look like an Indian blanket?

Program 8, PRINTOVER I. Have the program type over the existing display rather than blanking the screen. Do this without scrolling.

Is the display completely satisfactory this way? If not, how would you improve it?

Program 9, PRINTOVER II. The task is the same as Program 4, but fix its faults. Have the character print position show visually, even when printing over the same character. Blank the last two lines before printing the messages, pausing a few seconds after printing PHEW!; then blank the two lines again before printing the next display character.

Program 10, ALL THE WAY. Have the display character print over the entire 1000 character positions of the screen. The HIT A KEY message will appear only once at the beginning of the program. After that the user enters another key "blind." No PHEW! message.

Hint: First do the program for I=1 TO 999. Getting the 1000th character will take a little poking around.

Program 11, BOTTOM UP. Have the program print in reverse order, starting from the lower right-hand corner and proceeding left across rows to the home position. This is not difficult once you have done Programs 9 and 10. Clear the screen before ENDing to avoid typing into a cluttered display.

CHAPTER 4

PET BASIC

This chapter describes the syntax of all PET BASIC commands, statements, and functions that can be included in a PET BASIC program. This chapter is intended to serve as a reference guide. It contains PET BASIC commands, statements, and functions listed alphabetically within each group.

Descriptions given in this chapter have been made as definitive as possible. Frequently this results in the use of terms that you will not understand if you are a novice programmer; these terms are used to make the description as precise as possible. But do not worry if you encounter terms you do not understand; a less precise, general description of each statement, command, or function is also given. A list of BASIC primers is contained in Appendix D.

FORMAT CONVENTIONS

The following conventions are used in the format presentations:

UPPER CASE	Words in upper case are constant and must be typed in exactly as shown.
lower case	Words in lower case are variable; the exact wording or value is supplied by the programmer.
{ }	Braces indicate a choice of items; braces do not appear in an actual statement.
. . .	Ellipses indicate that the preceding item can be repeated; ellipses do not appear in actual statements.
line number	In the format presentations, a beginning line number is implied for all stored program statements.
variable	Unless otherwise specified, a variable may be unsubscripted (a simple variable) or subscripted (an array element).

BASIC COMMANDS

CLR

The Clear command, CLR, reinitializes all of the PET's "system" pointers to user program variables and initializes the Stack Pointer. This is equivalent to turning the PET off, then turning it back on, and loading a fresh copy of the program into memory.

Format:

CLR

After CLR, all numeric variables are assumed to have values of zero and all string variables are assumed to have null values until the variable is given a different value by a program or immediate mode assignment. PET BASIC statements that can assign values to variables are LET=, GET, GET#, INPUT, INPUT#, and READ.

CLR may be given in immediate or program mode.

Example:

CLR

CONT

The Continue command, CONT, resumes program execution after a BREAK. A break is usually caused by pressing the STOP key in immediate mode or executing a STOP statement in program mode.

Format:

CONT

A break is an interruption in program execution that returns control to PET BASIC before the program has executed to completion. A scheduled break is caused by execution of a STOP statement, execution of an END statement that has additional statements following it, or depressing the STOP key while the program is running.

During a program break you can LIST, LOAD, SAVE, VERIFY, and perform any immediate mode operations using PET BASIC statements — including changing the values of program variables — and still continue program execution by issuing a subsequent CONT command typed in at the keyboard. This can be very useful in debugging situations. On resumption of the program, execution continues at the exact point where it left off. A break caused by pressing the RETURN key in response to an INPUT statement can be recovered by typing CONT. In this case the same INPUT request is repeated.

You cannot resume program execution using CONT if during the program break you make any editing changes to the program or issue a CLR or NEW. CONT cannot continue program execution past an error message.

Example:

CONT

LIST

The List Program command, LIST, displays one or more lines of a program. You may edit program lines displayed by the LIST command.

Format:

LIST	{	(blank)	List entire program.
		line	List one line.
		line ₁ -line ₂	List from line ₁ to line ₂ .
		-line	List from start to line.
		line-	List from line to end.

where:

line is a line number. Line₁ is a lower number than line₂. All line numbers are inclusive. The line numbers do not need to be actual line numbers in the program. In the case of listing one non-existent line, a blank line is printed.

LIST is the most common command you will give in PET BASIC. It lists program lines on the display screen for examination and possibly subsequent editing. After a LIST command, you can move the cursor up to any line in the program shown on the screen and make editing changes.

If the program is longer than 23 lines, a simple LIST will scroll the beginning of the program up off the display screen. Use line limiting parameters in this case to retain the desired program lines on the screen. In addition to LIST, the entire program is also listed in response to LIST or LIST0.

If the PET is hooked up to a hard copy output device, such as a line printer or teletype, LIST can be used to obtain printed copies of the current program (see CMD statement).

Examples:

LIST	List entire program.
LIST 50	List line 50.
LIST 60-100	List all lines in the program between lines 60 and 100, beginning with line 60 and ending with line 100.
LIST -140	List all lines in the program from the beginning of the program through line 140.
LIST 20000-	List all lines in the program from line 20000 to the end of the program.

The format of the listed line is virtually the same as that in which the line was entered. PET BASIC does just the following reformatting:

1. '?'s entered as a shorthand for PRINT are expanded to the word PRINT.
Example: ? A becomes PRINT A.
2. Blanks ahead of the line number are eliminated.
For example: 50 A=1 becomes 50 A=1
100 A=A+1 becomes 100 A=A+1
3. A space is inserted between the line number and the rest of the statement if none was entered.
For example: 55A=B-2 becomes 55 A=B-2.
4. The line is printed beginning at column 2 instead of column 1.

LIST is always given in immediate mode. If you put a LIST command in your program, it lists the program but then exits to PET BASIC. Attempting to continue program execution via CONT simply repeats the LIST indefinitely.

LOAD

The LOAD command loads a program from the tape cassette unit, or from another device if specified, into memory.

Format:

}	(blank)	Load the first program found from the console tape.
	filename	Load the named program from the console LOAD tape.
	filename.device	Load the named program from the specified device.
	filename.device.s	Load from specified device with secondary address code.

where:

filename	is a string enclosed in quotes or a string variable representing the file name that was given when the program was written onto tape.
device	is a preassigned device number.
s	is a secondary address code (see OPEN).

LOAD in Immediate Mode

LOAD is most often specified in immediate mode to load a desired program into memory. For a LOAD in immediate mode, PET BASIC automatically performs a CLR function before the program is loaded. Once a program has been loaded into memory, it can be listed for updating or executed by issuing the RUN command.

Begin a tape load with all tape control keys up (off). When you give a LOAD command, PET issues the following dialogue (bracketed items appear only if a filename is specified):

LOAD ["filename"]	Issue LOAD command.
PRESS PLAY ON TAPE #1	Shown if no tape control keys are depressed.
OK	Shown after you press the PLAY key.
SEARCHING [FOR filename]	Tape begins moving.
[FOUND filename a FOUND filename b FOUND filename . . . FOUND filename d FOUND]	Lists the first 16 characters of all programs found, if any, between beginning tape position and filename location.
LOADING	Format for named program.
READY.	Format for unnamed program.
	A program is being loaded.
	Loading is complete.

After the first program has been loaded from a tape, you can for convenience leave the PLAY key depressed if you will be loading again from the tape. However, do not leave the PLAY key depressed if you will subsequently be saving a program. PET can sense that a tape control key is depressed, but it cannot distinguish between them. A common error is to LOAD, then subsequently attempt to SAVE, with only the PLAY key depressed.

Examples:

LOAD	Load the next program found into memory. If you start a LOAD when the cassette is in the middle of a program, the PET will read past the remainder of the current program, then load the next program.
LOAD "EGOR"	Search for the program named EGOR on the tape cassette, and load it into memory.
N\$="WHEELS" LOAD N\$	Search for the program named WHEELS on the tape cassette, and load it into memory.
LOAD "X"	Where a program named X is not on the tape, this command will produce a list of all the programs on the tape in the form: FOUND program 1 FOUND program 2 etc.

Sometimes you may have trouble loading a program from tape into memory. Often the PET will find the program name but does not load the program correctly or does not load it at all. If this happens, the only thing you can do is to try to load the program again. Rewind the tape and reload (several times if necessary). If this does not work, there may have been a problem when saving the program (SAVE command) or in the cassette unit itself.

LOAD in Program Mode

When the LOAD command is executed in program mode, any listing of LOAD messages is suppressed unless the tape PLAY key is up (off). When the PLAY key is off, the following message is shown so that the load can proceed:

```
PRESS PLAY ON TAPE #1  
OK ←————— when PLAY is depressed
```

LOAD in program mode can be used to build program overlays. A LOAD command executed from a program causes that program's execution to stop and another program to be loaded, overlaying the program currently in memory that issued the LOAD request. In this case the PET does not perform a CLR, thereby allowing the first program to pass on all of its variable values to the program being loaded. The overlay program, and all subsequent overlay programs, cannot be larger than the first program in the overlay series.

Example:

```
410 LOAD "OVLY1"
```

NEW

The NEW command clears the PET's memory of the current program.

Format:

```
NEW
```

When a NEW command is executed, the PET performs a CLR for all variables but also reinitializes all pointers that are keeping track of the program statements stored in memory. The program itself is not physically deleted. These NEW operations are automatically performed during a LOAD process.

Issue a NEW command when you want to begin creating a stored program entered from the keyboard and there is already a program in memory. If a NEW were not issued, subsequent program entry lines would simply modify the existing program.

Example:

```
NEW
```

NEW is always issued in immediate mode. If a NEW command is issued from within a program, the program will "self destruct"; it will clear itself out just as if a NEW command had been issued from the keyboard. Of course, the program terminates at that point.

RUN

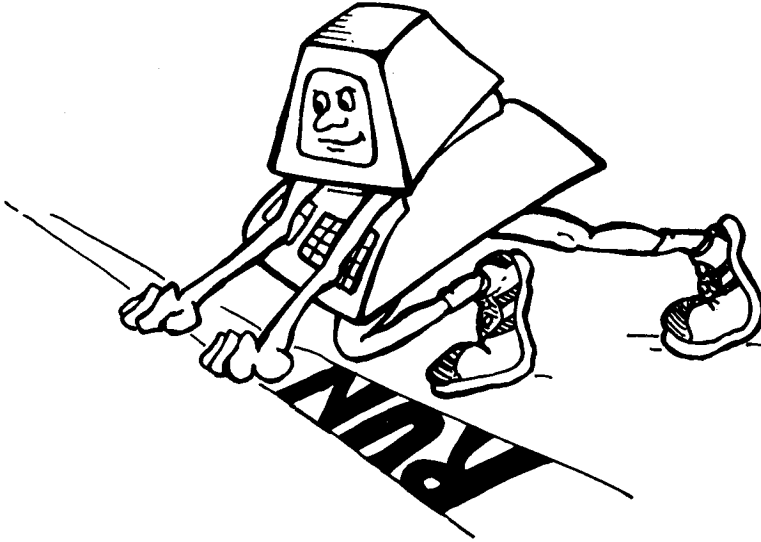
The RUN command begins execution of the program currently stored in memory.

Format:

RUN	{ (blank)	Begin execution at the lowest-numbered line.
	{ line	Begin execution at the specified line.

When the RUN command is executed in immediate mode, the PET performs a CLR of all program variables and performs a RESTORE, then begins executing the program. This is by far the most common use of the RUN command.

If RUN is specified with a line number following it, PET still performs the CLR and RESTORE but begins execution at the specified line number. The RUN with line number command is not for use after a program break — use CONT or GOTO for that purpose. Rather, RUN with a line number specified is used to begin execution of a program within a program. Usually the two programs will be completely separate, each program having its own range of line numbers.



The RUN with no line number command may also be used in program mode. It begins execution of the program from the top with all variables cleared and DATA being called from the beginning again. However, you will probably find that using the CLR, RESTORE, and GOTO commands is preferable.

Examples:

- | | |
|----------|--|
| RUN | Initialize and begin execution of the current program. |
| RUN 1000 | Initialize and begin execution of the program starting at line 1000. |

SAVE

the SAVE command writes a copy of the current program from memory to the console tape unit, or other device if specified.

Format:

SAVE	{	(blank)	Save as an unnamed file.
		filename	Save with filename header.
		filename,device	Save with filename header on specified device.
		filename,device,s	Save on specified device with secondary address code.

where:

filename is a string enclosed in quotes or a string variable representing the file name by which the program is to be called.

device is a predefined device number.

s is a secondary address code (see OPEN).

SAVE is most often used in immediate mode. To save the current program on tape, follow these steps:

1. Insert the tape cassette into the cassette unit. The save begins immediately, so you should be sure to position the tape to the correct point.
2. Depress the tape STOP key to set all keys up.
3. Type in the SAVE command. The following dialogue takes place (bracketed items appear only if a file name is specified):

SAVE [filename]

PET responds with the message:

PRESS PLAY & RECORD ON TAPE #1

First press the RECORD key and then the PLAY key. PET responds with:

OK

WRITING [filename]

READY. ←————— Tape stops

4. Depress the STOP key.
5. Verify the saved program (see VERIFY).

If you want to write the current program onto a tape that is not positioned correctly, use VERIFY commands to position the tape, since VERIFY does not alter the program currently stored in memory. Or, you may write the current program to a spare tape, position the other tape with LOADs, then reload the saved program from the spare tape and proceed with the SAVE procedures on the primary tape.

When the PET performs a SAVE to tape, it automatically writes a tape header and creates a tape file with the specified file name (or blank if no file name is specified). It is recommended that file names be used — you can always load a named program without naming it, but you cannot search for an unnamed one. Error checking and correction techniques are used to help ensure that subsequent loading will be a correct copy of what was originally written. You can help to ensure reliable tapes by verifying after every SAVE command (see VERIFY command).

CAUTION: If any of the four tape control keys REC, REW, FFWD, or PLAY are depressed when the SAVE command is executed, the PET will start “writing” to the tape. Be sure to press the tape STOP key before issuing a SAVE command; this assures that the PET will print the PRESS PLAY & RECORD ON TAPE #1 message. Do not follow the order of these instructions: remember to depress RECORD before you press PLAY.

Examples:

- SAVE Write the current program onto tape, leaving it unnamed.
- SAVE "RED" Write the current program onto tape, assigning the file name of RED.
- A\$="RED" Same as above.
- SAVE A\$

A SAVE command will also work if issued from within a program. You will normally not have any reason to do this. For a programmed SAVE, as with a programmed LOAD, the PET does not issue any messages on the screen unless the device being written to is not ready; for the console tape, the following messages may be displayed:

```
PRESS PLAY & RECORD ON TAPE #1
OK
```

VERIFY

VERIFY is a version of the LOAD command that checks for recording errors after a program is saved. Always verify a program after you save it.

Format:

- VERIFY { (blank) Verify the first program found on the console tape.
- filename Verify the named program on the console tape.
- filename,device Verify the named program from the specified device.
- filename,device,s Verify from specified device with secondary address code.

where:

filename	is a string enclosed in quotes or a string variable representing the file name that was given when the program was written onto tape.
device	is a preassigned device number.
s	is a secondary address code (see OPEN).

Note that LOAD, SAVE, and VERIFY have the same parameters.

VERIFY should always be used following a program SAVE to see if any errors occurred when writing to the tape. By simulating the processes of a program LOAD, it reads and compares the program on the tape to the program in memory without actually loading the program into memory. If an error is detected, the PET displays a screen message to warn the user.

Follow these steps to verify a tape:

1. SAVE the program.
2. Rewind the tape to the beginning of tape (or to a point that you know precedes the load point of the program just saved).
3. Press the tape STOP key.
4. Type in the VERIFY command, giving the same parameters that were used in the SAVE command. The program will be verified.

On being given a VERIFY command, the PET begins the following dialogue (bracketed items appear only if a filename is specified):

VERIFY ["filename"]	Issue VERIFY command.
PRESS PLAY ON TAPE #1	Shown if no tape control keys are depressed.
OK	Shown after you press PLAY.
SEARCHING [FOR filename]	Tape begins moving.
FOUND filename a	Lists the first 16 characters of all programs found, if any, between beginning tape position and filename location.
FOUND filename b	
FOUND filename . . .	
FOUND filename d	
FOUND	Format for named program (SAVE filename").
VERIFYING	Format for unnamed program (SAVE). A program is being verified.
{OK}	Displayed if program verifies.
{READY.}	
?VERIFY ERROR	Displayed if program does not verify.

If the message OK is displayed, it means that you can subsequently load the saved copy into memory and get a duplicate of what is currently in memory.

If the message ?VERIFY ERROR is displayed, it means that the program may not have been correctly recorded onto the tape. Rewind the tape and try the VERIFY again. If the error persists, rewind the tape back to the beginning of the save point, save the program again, and verify the program. If the error still persists, try using another portion of the tape or another tape. To find the problem you may use the Status Word function (ST). ST tells you where the error occurred (see ST function).

The VERIFY command can be used in program mode, but this is not a common practice.

Examples:

VERIFY	Verify the next program found on the ape.
VERIFY "CLIP"	Search for the program named CLIP on the tape cassette, and verify it.
A\$="CLIP"	Same as above.
VERIFY A\$	

BASIC STATEMENTS

CLOSE

The CLOSE statement closes a logical file.

Format:

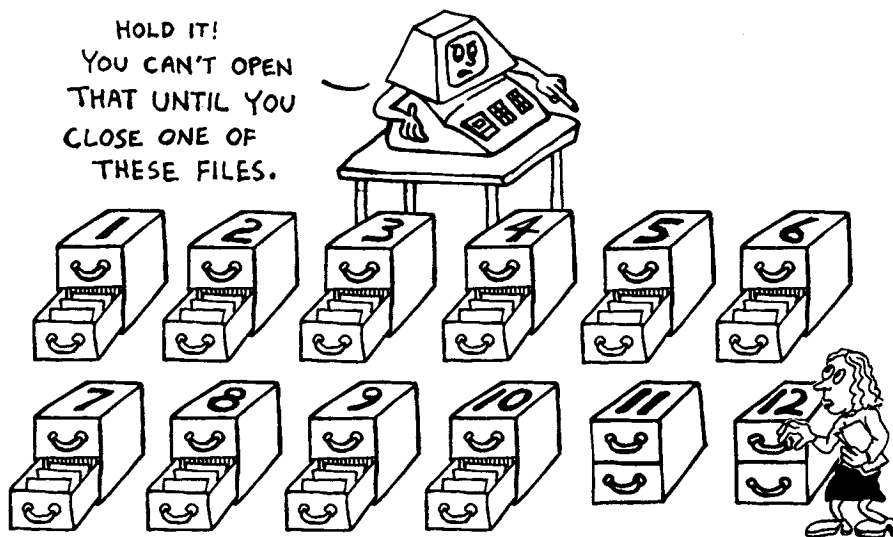
CLOSE file

where:

file

is the logical file number that was given when the file was opened.

Each file should be closed when you are through accessing it. Since the PET allows only ten files to be open at a time, it is good practice to close files as soon as they are not in use to allow other files to be opened. A logical file may be closed only once between file open commands. All open files must be closed before a program END.



The particular operations performed in response to a CLOSE statement depend on the physical device and the type of operation for which the file was opened. For PET cassette tapes, closing a file after reading it will stop the PET from reading the file any further. Closing a file after writing to it is critical: the PET puts an end-of-file mark at that point, an end-of-tape mark for write code 2, and finishes writing the last of the buffer contents onto tape.

If you forget to close a file prior to the program end, you may ruin your file. When a file is opened for write, the data is stored in a buffer area prior to being transferred to the tape. If the file is not closed, the data will probably NOT be transferred to the tape. If the data is transferred under these conditions, other problems can result. If the file is written over a previously recorded tape, and the file is not closed, no end-of-file marker is written. This problem will not present itself until the file is subsequently read, when either a READ ERROR will be generated or the tape will be read past where the end-of-file should be, and the remaining "garbage" on the tape will be read.

CLOSE may be executed in either immediate or program mode. After writing to a file, if no tape control key is depressed when a CLOSE is issued, the PET displays the following message:

```
PRESS PLAY & RECORD ON TAPE #1 ← Press keys
OK ← Tape begins moving to write tape buffer.
```

No tape control keys need to be down for a CLOSE after a READ command.

Examples:

CLOSE 1	Close logical file 1.
CLOSE 14	Close logical file 14.
A=14	Same as above.
CLOSE A	

CMD

The CMD statement directs the PET-user screen dialogue to another output device.

Format:

CMD file
where:

file is the logical file number that was given when the file was opened.

After a CMD statement, output for subsequent PRINT and LIST commands is directed to the "commanded" device rather than to the screen.

CMD may be executed in immediate or program mode.

Example:

The following example sequence uses CMD to obtain program listings and printouts on paper.

OPEN 5,5	Device 5 is a hard-copy printer.
CMD 5	Direct subsequent display log to the logical file 5 device.
LIST	Prints the program listing on paper.
LIST	Prints another copy of the program listing on paper.
PRINT A	Prints the value of variable A on paper.
PRINT #5	Return to display screen log (prints a blank line at the printer and "unlistens" the printer device).

DATA

The DATA statement declares constants that are to be assigned by the READ statement.

Format:

DATA	{	constant	One DATA item
	}	constant.constant...constant	Multiple DATA items

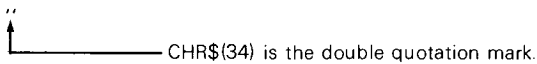
where:

constant is a simple numeric or string constant. Variables, expressions and functions are illegal.

DATA statements may be placed anywhere in the program. See the READ statement for a description of DATA statement processing.

Constants in the DATA statement list may be numeric or string. Numeric constants must agree in format with the variables to which they will be assigned. String constants are usually enclosed in double quotation marks. The quotes are not necessary unless the string contains graphic characters or any of the following special symbols: blanks (spaces), commas, and colons. Blanks and graphic characters are ignored in a string unless they are enclosed in quotes. Any constant may be read into a string variable. Numeric data read into a string variable can be converted to its numeric equivalent by the VAL function. A double quotation mark cannot be represented in a DATA string; an Assignment statement must be used in the following manner:

```
D$=CHR$(34):?D$
```



The DATA statement is valid in program mode only.

Examples:

10 DATA NAME, "C.D."
50 DATA 1E6, -10, XYZ

Defines two string variables.
Defines two numeric variables and one string variable.

DIM

The Dimension statement (DIM) allocates space for arrays.

Format:

DIM { var(i) Dimension one array.
 { var(i), var(i), . . . var(i) Dimension multiple arrays.

where:

- var is a numeric or string variable name.
- i is a constant, variable or expression representing the subscript or subscripts of the array, in the form:
 - var(i) Single-dimension array
 - var(i,j) Two-dimension array
 - var(i,j,k, . . .) Multiple-dimension array

Refer to "Arrays" in Chapter 3 for a complete description of array formats. Arrays with more than eleven elements must be dimensioned in a DIM statement. The DIM statement must be encountered once, and only once, in program execution before any statement calls an element of the array. Arrays of eleven elements or less (subscripts 0 through 10 for a one-dimensional array) may be used without being dimensioned by a DIM statement; for such arrays, eleven array spaces are allocated when the first array element is encountered.

If the array is dimensioned more than once, or if an array having more than eleven elements is not dimensioned, a ?REDIM'ED ARRAY error occurs and the program is aborted. A CLR statement allows a Dimension statement to be re-executed; this is not normally done, however.

DIM may be given in either immediate or program mode.

Examples:

10 DIM A(3)
45 DIM X\$(44,2)
1000 DIM MU(N,3*B), N(12)

DIM statement usage
in program mode.

END

The END statement terminates program execution and returns control to PET BASIC.

Format:

```
END
```

The END statement may be used in a program to provide a program termination point, or multiple program termination points, at locations other than the physical end of the program. END statements can be used to terminate each program when there is more than one program in memory at the same time. An END statement at the physical end of the program is optional. A Continue command (CONT) given after program termination by END will continue program execution at the statement following the END statement.

END is functional only in program mode.

Example:

```
20001 END
```

FOR . . . NEXT [STEP]

The FOR statement sets up a loop where all intervening statements between the FOR statement and the NEXT statement are executed a specified number of times.

Format:

```
FOR index = start TO end [STEP increment]
  [statements in loop]
NEXT [index]
```

where:

index	is a floating point, single variable used to hold the current loop count. The index variable is often used by the statements in the loop.
start	is a numeric constant, variable or expression that specifies the beginning value of the index.
end	is a numeric constant, variable, or expression that specifies the ending value of the index. The loop is completed when the index value is equal to the end value, or when the index value is incremented or decremented past the end value.
increment	if present, is a numeric constant, variable, or expression that specifies the amount by which the index variable is to be incremented with each pass. The step may be incremental (positive) or decremental (negative). If the STEP phrase is omitted, the increment defaults to +1.

FOR . . . NEXT loops may be nested. Each nested loop must have a different index variable name. A nested loop must be wholly contained within the next outer loop; at most, the loops can end at the same point.

The index variable may optionally be included in the NEXT statement. This may be advisable to improve the readability of the program when there are nested FOR . . . NEXT loops. A single NEXT statement is permissible for nested loops that end at the same point. The NEXT statement would then take the following form:

```
NEXT index1,index2,. . .
```

A FOR . . . NEXT loop may contain subroutine calls, but the loop cannot terminate inside a called subroutine. Both FOR . . . NEXT loops and subroutine calls use the Stack (see Chapter 6).

The FOR . . . NEXT loop will always be executed at least once, even when the beginning index value is beyond the end index value range. If the NEXT statement is omitted and no subsequent NEXT statements are found, the loop is executed once.

The start, end, and increment values are read only once, at the first execution of the FOR statement. You cannot change these values inside the loop. You can change the value of the index variable within the loop. This may be used to terminate a FOR . . . NEXT loop before the end value is reached: set the index to the end value, and on the next pass the loop will terminate itself. Do not jump out of the FOR . . . NEXT loop with a GOTO.

Examples:

```
10 FOR IN = 0 TO 100
  .
  .
  .
40 NEXT IN
100 FOR X = A+14 TO C-64+D/2 STEP 4
  .
  .
  .
150 NEXT X
60 FOR A1 = 50 TO 0 STEP -1
  .
  .
  .
90 NEXT
```

```
100 FOR I = 0 TO 10 STEP 0.5
```

```
•  
•  
•
```

```
155 NEXT
```

```
250 FOR I = 1 TO 5
```

```
260 FOR J = A TO B
```

```
•  
•  
•
```

```
300 NEXT I,J same as 300 NEXT I same as 300 NEXT  
310 NEXT J 310 NEXT
```

GET

The GET statement inputs one character from the keyboard into a program.

Format:

```
GET var$
```

where:

var\$ is a string variable into which the input character is to be stored.

When a GET statement is executed, it sets the specified string variable to null. Any previous value of the variable is lost. Then it fetches the next character from the keyboard buffer and assigns it to the specified string variable. If the keyboard buffer is empty, the variable remains null.

GET can be used to handle one-character responses from the keyboard. It has an advantage over INPUT in that the PET does not intercede when the RETURN key is struck; the RETURN key is passed to the program as a value (CHR\$(13)) just like any other key. Because of fast program execution, you will need to program a particular sequence so that the GET command will wait until a key has been pressed at the keyboard. The common program sequence incorporates a check for a null character; if the null character is found, return to the GET statement, and then check the variable value again; if any character is found, proceed with the main program. This sequence is shown in the example below.

The GET statement is valid only in program mode.

Example:

```
10 GET C$:IF C$ = "" GOTO 10
```

Format Variations

GET has other formats, but they are defective and it is not recommended that they be used. These variations are described below.

GET may be used to fetch a numeric variable:

```
GET var
```

where:

var is a numeric variable.

If no key has been pressed a zero is returned. However, a zero is also returned if a zero was pressed at the keyboard. If the character returned is not a digit (0-9) a SYNTAX ERROR message is generated and the program aborts.

The GET statement may have a variable list of the form:

```
GET var,var, . . . ,var
```

where:

var is a string or numeric variable.

GET#

The Get External statement (GET#) inputs one character from an external peripheral device to the PET.

Format:

```
GET#file,var
```

where:

file is the logical file number that was given when the file was opened.

var is a string or numeric variable into which the character is to be input.

GET# functions like the GET command except that the character is fetched from an external device.

GET# is valid only in program mode and only when referencing a logical file that has been opened for input. For input from the tape cassette, PET prints the following message if no tape control keys are pressed when the GET# statement is executed. Otherwise, no messages are printed.

```
PRESS PLAY ON TAPE #1
```

```
OK ← when a tape control key is depressed
```

Example:

```
10 GET#4,C$:IF C$ = "" GOTO 10
```

GOSUB

The GOSUB statement branches program control to specified line and allows a return to the next statement following the GOSUB call.

Format:

GOSUB line

where:

line is a numeric constant specifying the first line of the subroutine.

No special entry to a subroutine is needed. The program branches control to the subroutine after making an entry on the Stack (see Chapter 6). The subroutine must terminate with a RETURN statement to return program control to the statement following the GOSUB statement that activated the subroutine. The RETURN removes the Stack entry. A subroutine may be called from many places in a program. As many as 26 subroutines may be nested (25 GOSUBs before the first RETURN).

Example:

```
10 GOSUB 100
```

GOTO

The GOTO statement branches unconditionally to the specified line.

Format:

GO TO }
GOTO } line

where:

line is a numeric constant specifying the line to be branched to.

Example:

```
10 GOTO 100
```

GOTO executed in immediate mode branches to the specified line in the stored program without clearing the current variable values. GOTO cannot reference immediate mode statements, since they do not have line numbers.

IF . . . THEN

The IF . . . THEN statement provides conditional execution of statements based on a relational expression.

Format:

IF condition THEN $\left\{ \begin{array}{l} \text{statement} \\ \text{statement:statement...} \end{array} \right.$ Conditionally execute statement.
Conditionally execute statements.

IF condition $\left\{ \begin{array}{l} \text{THEN} \\ \text{GOTO} \end{array} \right.$ line Conditionally branch.

where:

condition	is a relational term (with "< >0" implied) or a relational expression.
statement	is any PET BASIC statement valid for the current mode (program or immediate) that may be conditionally executed.
line	is a numeric constant specifying the line to be branched to.

If the specified condition is true, then the statement or statements following the THEN are executed. If the specified condition is false, control passes to the next sequential statement on the next line number without executing the statement or statements following the THEN. A conditional branch may be written in the form of a line number to be branched to placed after the word THEN or after the word GOTO. The compound form THEN GOTO is also acceptable.

IF A = 1 THEN 50	}	equivalent
IF A = 1 GOTO 50		
IF A = 1 THEN GOTO 50		

If an unconditional branch is in a multiple-statement IF . . . THEN, the branch must be the last statement on the line and must be in the traditional "GOTO line" form. If it is not the last statement on the line, the statements following the unconditional branch are never executed. The form "THEN line" is an alternative to "GOTO line" and cannot be used with multiple statements intervening between the "THEN" and the "line."

IF . . . THEN may be given in immediate or program mode.

In an immediate mode IF . . . THEN, the following statements are illegal or non-functional: DATA, GET, GET#, INPUT, INPUT#, REM, RETURN, END, STOP, WAIT. If a line number is specified, or any statement containing a line number, there must be a corresponding statement with that line number in the current stored program if the branch may be taken.

In a program mode IF . . . THEN, the following command and statement are illegal: CONT, DATA. If a FOR . . . NEXT loop will be following the THEN, the loop must be completely contained on the IF . . . THEN line. Additional IF . . . THEN statements may appear following the THEN as long as they are completely contained on the original IF . . . THEN line. However, Boolean connectors are preferred to nested IF . . . THENs. For example, the two statements below are equivalent, but the second is preferred.

```
10 IF A$ = "X" THEN IF B = 2 THEN IF C > D THEN 50
10 IF A$ = "X" AND B = 2 AND C > D THEN 50
```

Examples:

```
400 IF X > Y THEN A = 1
500 IF M + 1 THEN AG = 4.5 : GOSUB 1000
```

INPUT

The INPUT statement inputs data items from the keyboard to a program.

Format:

```
INPUT    { (blank) }      { var                Input data item
          {"msg":}       { var,var, . . . ,var    Input data items
```

where:

- var is a string or numeric variable into which the data item is to be input.
- msg refers to a prompt message which, if present, is a string enclosed in quotes and followed by a semicolon.

When the INPUT statement is executed, PET BASIC prints a question mark on the screen requesting input of data to the variable or variables listed in the INPUT statement. The user enters the data and terminates his or her response with a RETURN. Where there is more than one variable in the INPUT statement, data entry is separated by commas.

```
?data                or     ?data,data, . . . ,data
```

If msg is present, the string is printed before the question mark.

```
msg?data             or     msg?data,data, . . . ,data
```

The input data items must agree in number and type with the variables in the INPUT list. If the variable types are different, a ?REDO FROM START message is printed and the PET waits for another input. Any response is acceptable for string input; for a numeric variable, however, the input must be in proper numeric form. If at least one but still too few data items are input, PET BASIC requests additional input with double question marks (??) until the variable list is satisfied. If too many data items are input, the message ?EXTRA IGNORED is printed but the program continues. Data is input from the first position past the question mark to the position where the RETURN is given. Normally only a single string item is requested to be typed in from the keyboard by each INPUT statement. However, multiple items can be input and items may be input from an existing screen display.

CAUTION: If a null list (RETURN only) is input in response to an INPUT statement, the program aborts and returns control to PET BASIC. The program can be continued, however, by typing CONT in response to the PET's READY message.

INPUT can be given only in program mode.

Examples:

```
100 INPUT A$
200 INPUT A,B$
220 INPUT "ENTER X COORDINATE":X
3060 INPUT "NAME":N$
```

INPUT#

The Input External statement (INPUT#) inputs one or more data items from an external peripheral device to the PET.

Format:

$$\text{INPUT\#file} \begin{cases} \text{var} & \text{Input data item} \\ \text{var,var, . . . ,var} & \text{Input data items} \end{cases}$$

where:

- file is the logical file number that was given when the file was opened.
- var is a numeric or string variable into which the data item is to be input.

Data strings may not be longer than 80 characters (79 characters plus a carriage return) because the input buffer can hold only 80 characters at a time. Item separators such as commas and carriage returns are recognized by the PET when processing the INPUT# statement but are discarded and are not passed on to the program as data.

INPUT# is valid only in program mode and only when referencing a logical file that has been opened for input. For input from the tape cassette, the PET prints the following message if no tape control keys are pressed when INPUT# is executed. Otherwise, no messages are printed.

PRESS PLAY ON TAPE #1

OK ←————— when a tape control key is depressed

Examples:

- 1000 INPUT#10,A Input the next data item, which must be in numeric form, and assign the value to variable A.
- 946 INPUT#12,A\$ Input the next data item as a string and assign it to variable A\$.
- 900 INPUT#5,B,C\$ Input the next two data items and assign the first to numeric variable B and the second to string variable C\$.

LET . . . =

The Assignment statement, LET . . . =, or simply =, assigns a value to a specified variable.

Format:

$\left. \begin{matrix} \text{(blank)} \\ \text{LET} \end{matrix} \right\} \text{var=data}$

where:

- var is a numeric or string variable.
- data is a constant, variable, or expression representing the value to be assigned to var. Data must agree in type with var.

The word LET is optional and is usually omitted in PET BASIC programs. The assignment statement may be used in program or immediate mode.

Examples:

- 10 A=2
- 450 C\$=" " " "
- 300 M(1,3)=SGN(X)
- 310 XX\$(I,J,K,L)="STRINGALONG"

ON . . . GOSUB

The ON . . . GOSUB statement provides conditional subroutine calls to one of several subroutines in a program, depending on the current value of the specified index.

Format:

ON index GOSUB line₁,line₂, . . . ,line_n

where:

index is a constant, variable, or expression that must be in the range $0 < \text{index} \leq 255$.

line_i is one or more line numbers specifying the beginning lines of subroutines.

ON . . . GOSUB has the same format as ON . . . GOTO, and the statement is evaluated in the same way (see ON . . . GOTO).

For index=1, the subroutine beginning at line₁ is called. When that subroutine is completed (a RETURN statement), the program continues with the statement following the ON . . . GOSUB. If index=2, the subroutine beginning with line₂ is called, etc. The ON . . . GOSUB statement should be used only to call subroutines that return control via RETURN statements to the statement following the ON . . . GOSUB.

ON . . . GOSUB is normally executed in program mode. It may be executed in immediate mode as long as there are corresponding line numbers in the current stored program that may be branched to.

Example:

```
10 ON A GOSUB 100,200,300
```

ON . . . GOTO

The ON . . . GOTO statement provides conditional branching to one of several points in a program, depending on the current value of the specified index.

Format:

ON index GOTO line₁,line₂, . . . ,line_n

where:

index is a constant, variable, or expression that must be in the range $0 < \text{index} \leq 255$.

line_i is one or more line numbers.

The specified index is evaluated and, if necessary, truncated to an integer number. If index=1, a branch is taken to the statement with line number line₁. If index=2, a branch is taken to the statement with line number line₂. If index=3, a branch is taken to the statement with line number line₃. If index=0, no branch is taken. If index is in the allowed range but there is no corresponding line number in that position of the line number list, no branch is taken. If a branch is not taken, program control proceeds to the next statement following the ON . . . GOTO. The next statement may be on the same line as the ON . . . GOTO (separated by a colon).

If index has any value other than zero outside the allowed range, the program aborts with an error message. As many line numbers may be specified as will fit on the 80-character line.

ON . . . GOTO is normally executed in program mode. It may be executed in immediate mode as long as there are corresponding line numbers in the current stored program that may be branched to.

Example:

```

40  A=B<10
50  ON A+2 GOTO 100,200      Branch to statement 100 if A is true
                               (-1) or branch to statement 200 if A
                               is false (0).

50  X=X+1                    Branch to statement 500
60  ON X GOTO 500,600,700    if X=1, to statement 600
70                               if X=2, or to statement 700
.                               if X=3. No branch is taken if
.                               X > 3
.

```

OPEN

The OPEN statement opens a logical file and readies the assigned physical device.

Format:

OPEN	}	file	Open code 0 of device 1 and assign file.
		file.device	Open code 0 of device and assign file.
		file.device,s	Open code s of device and assign file.
		file.device,s,filename	Open filename, code s, of device and assign file.

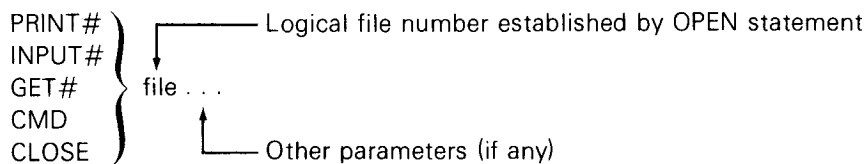
where:

- file is the logical file number and may be any number, variable or expression in the range 1 through 255. The logical file number must be present in an OPEN statement.
- device is the physical device number and may be any number, variable or expression in the range 0 through 30. If omitted, the device number defaults to 1, the console tape unit.
- s is the secondary address code. This parameter may be any number, variable, or expression in the range 0 through 31 that is valid for the specified device. If omitted, the secondary address code defaults to 0.
- filename is a string enclosed in quotes or a string variable representing a file name.

On the PET, a logical file number is the way of referring to a particular operation on a particular portion of a particular I/O unit. The particular operation is specified by the secondary address code; subsequent I/O statements for the file must conform to the secondary address code specification. For example, you cannot open a file for read and then try to write to it. The particular I/O unit is specified by the device number. The particular portion of the device is specified by the filename. The relationships between logical file number, physical device number, secondary address code, and file name are illustrated in Figure 4-1.

Logical File Number

The PET performs programmed input/output (I/O) operations via logical file designations. A logical file must be opened for each type of I/O operation to be performed on each device accessed. The OPEN statement specifies the particular type of operation to be performed via the logical file. Subsequent I/O operations all refer to the logic file number set up by the OPEN statement:



The programmer makes up the logical file number, just as he does program variable names; the logical file number may be any number from 1 to 255. Once opened, the logical file can be used only for the type of operation specified by the OPEN statement until the logical file is closed; then it becomes free to use again in a different operation. A logical file may be opened only once before it is closed; if a file is opened twice before it is closed, a ?FILE OPEN error results. No more than ten logical files may be opened at the same time.

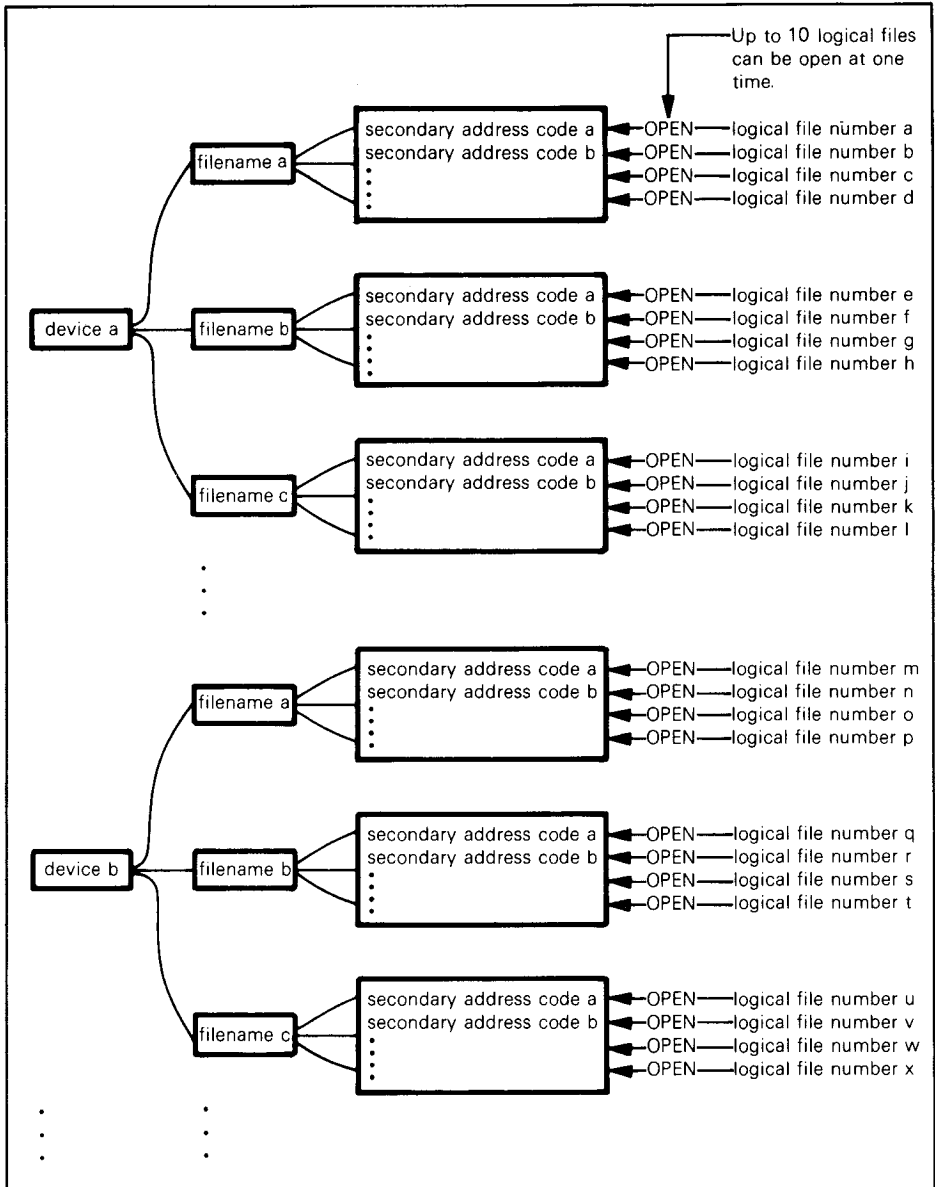


Figure 4-1. Logical Files

Physical Device Number

The physical device number is a predefined number that refers to a particular peripheral unit capable of input and/or output with the PET. The device number must be within the range 0 to 255, although currently only numbers 0 to 30 will assign a device. Device numbers are listed in Table 4-1.

Table 4-1. Physical Device Numbers

Device Number	Device
0	Keyboard
1 (default)	#1 - Console tape cassette unit
2	#2 - Second tape cassette unit
3	Video display screen
4 through 30	IEEE port devices
31 through 255	Currently unassignable

Secondary Address Code

The secondary address code specifies the type of operation to be performed on the selected device. The meanings of the codes for a particular device are predetermined. For the PET tape cassettes, the secondary address code is used by the PET software primarily to read or write tape headers. For IEEE 488-interfaced devices, the secondary address code is passed on directly to the device, allowing the program to select a variety of operational modes. Secondary address codes are not used for keyboard input or output to the display screen. Secondary address codes for PET devices are listed in Table 4-2. For IEEE devices, refer to the device manual for secondary address codes recognized by the device.

Table 4-2. Secondary Address Codes

	Secondary Address Code	Operation
PET Tape Cassette Units	0 (default) 1 2	Open for read Open for write Open for write and end-of-tape mark (EOT) when file is closed
PET Line Printer	0 (default) 1 2 3 4 5	Normal print Print under Format statement control Transfer data from PET to Format statement Set variable lines per page Use expanded diagnostic messages Byte data for programmable character
PET IEEE 488 Devices	0 (default) 1 through 31	No secondary address code sent Refer to device manual

File Names

The filename parameter in the OPEN statement specifies a program or data file by name. If the OPEN is for a write, this is the name that will be assigned to the file being created. If the OPEN is for a read, this is the name that was written when the program was created.

For cassette I/O, the filename parameter is optional. If omitted, the PET defaults the file name to a null string ""; the null string will match the first file name encountered on the tape. For other devices the file name may or may not be optional, depending on the requirements of the device. Note that the logical file number is separate and distinct from the file name. The logical file number (first parameter in the OPEN statement) is for internal PET use; the file name, for tape cassettes, is the name written in the tape header to identify a program or data file stored on tape.

OPEN may be executed in immediate or program mode.

OPEN for Read

When the OPEN statement is executed to open a tape cassette unit for a read, the PET will display the following message if no tape control keys are pressed:

PRESS PLAY ON TAPE #1

OK ← when a tape control key is depressed; tape begins moving.

It then reads the tape header. In immediate mode the messages continue as follows (bracketed items are shown if a filename was specified):

SEARCHING [FOR filename]

FOUND filename a

FOUND filename b

FOUND filename c

FOUND filename d

FOUND

FOUND [filename]

READY.

⌘

Lists the first 16 characters of all files found, if any, between beginning tape position and filename location.

Format for named file.

Format for unnamed file.

File is opened for read.

In program mode the above block of messages is not displayed.

OPEN for Write

When the OPEN statement is executed to open a tape cassette unit for a write, PET will display the following message if no tape control keys are pressed:

PRESS PLAY & RECORD ON TAPE #1

OK ← when a tape control key is depressed; tape begins moving.

It then writes the tape header and tape movement stops.

Examples:

OPEN 1	Open logical file 1 for cassette #1 (default), code read (default) from the first file encountered on the tape (no filename specified).
OPEN 1,1	Same as above.
OPEN 1,1,0	Same as above.
OPEN 1,1,0,"DAT"	Same as above but from the file named DAT.
OPEN 3,1,2	Open logical file 3 for cassette #1, code write and EOT to the next space on the tape. The new file is unnamed.
OPEN 3,1,2,"PENTAGRAM"	Same as above but give the file name PENTAGRAM.

OPEN 54,8,10

Open logical file 54 for IEEE device 8 and send secondary address code 10.

POKE

The POKE statement provides the ability to store a specified byte value into a specified location in PET memory.

Format:

POKE address,data

where:

address	is a numeric constant, variable, or expression in the range $0 < \text{address} \leq 65535$ representing an address in PET's read/write (RAM) memory.
data	is a numeric constant, variable, or expression in the range $0 < \text{data} \leq 255$ representing a byte value to be stored.

Only RAM and display memory locations are affected by a POKE.

Examples:

```
10 POKE 1,A
POKE 32768,ASC("A")-64
```

PRINT

The PRINT statement prints data items on the display screen.

Format:

{	PRINT	}	data	Print data item.
			data,data,..,data	Print data items tabbed.
			data;data;..;data	Print data items continuous.

where:

?	(question mark) is an abbreviation accepted by PET BASIC for the word PRINT.
data	is a constant, variable, or expression representing a data item to be printed. If the data item is a string, it is enclosed in double quotation marks.

Data Conversions

When printing a number, PET BASIC prints it in standard representation if it is in the range $0.01 \leq n \leq 999999999$ and in scientific notation if it is outside that range.

Strings

Strings may be printed by assigning the string to a string variable or enclosing the string within quotation marks. For example,

```
A$="HELLO MARY"  
PRINT A$           Prints HELLO MARY  
PRINT "HELLO MARY" Prints HELLO MARY
```

Further, if you print a string within quotation marks, you may leave off the final set of quotation marks if the PRINT statement is the last statement on a line. For example,

```
PRINT "HELLO MARY Prints HELLO MARY
```

PRINT Formats

First data item. The first data item to be printed in a program is printed at the current cursor position. The PRINT format character (comma or semicolon) following the first data item specifies the format in which the next data item is to be printed. The next data item may be in the same PRINT statement or in a separate PRINT statement.

New Line. When no PRINT format character (comma or semicolon) follows a data item, a RETURN is forced following printing of the data item.

Tabbed. A comma following a data item causes the next data item to be printed at the next tab column. Tabs are at locations 1, 21, 31, and 41 of a line. (A comma may precede the first item to tab before printing.)

Continuous. A semicolon following a data item causes the next item to be printed at the next available location on the line. For numeric data, this is the second position to the right of the previous data item to allow room for a "skip" character between numbers. For string data, items are printed continuously with no spaces or skips in between. The semicolon is optional to specify continuous line printing of character data.

PRINT may be executed in immediate or program mode.

Examples:

```
10 PRINT A  
20 ?A,B,C$  
?"MOOPOO";MM$  
PRINT C1$:C2$;  
PRINT A+B
```

PRINT#

The Print External statement (PRINT#) outputs one or more data items from the PET to an external peripheral device.

Format:

PRINT#	{ file,data	Output data item.
	{ file,data;c:data;c; . . . data	Output data items.

where:

file is the logical file number that was given when the file was opened.

data is a constant, variable, or expression representing a data item to be output. The type of variable — string or numeric — determines the format in which the data is written.

c is an item separator to be output between multiple data items. This may be a stng or string variable. Item separators recognized by INPUT# statements are:

Comma: “,” or CHR\$(44)

RETURN: CHR\$(141)

When outputting a single data item, the data item is output and then a RETURN code is output. On subsequent reading of the file, the INPUT# statement recognizes separate data items by a <RETURN> between them:

data <RETURN> data <RETURN> data <RETURN> . . .data <RETURN>

When outputting multiple data items in one PRINT# statement, the items must be separated by a valid item separator for subsequent INPUT# reads. Standard punctuation marks, like commas between items, are recognized by PET BASIC but are not transmitted to the output file. The writing of the item separators must be forced by including them as strings:

PRINT#file,data;“,”;data;“,”;data. . .

or (equivalent):

PRINT#file,data;CHR\$(44);data;CHR\$(44);data. . .

This results in a file that is stored as:

data,data,data. . .data <RETURN> . . . EOF

For output to cassette tape, the data item or items in a PRINT# statement are accumulated in a 192-character buffer. When the buffer is filled, its contents are output to the external device. Note that for INPUT#, no more than 80 characters can be read between each RETURN.

PRINT# may be executed in immediate or program mode but is valid only when referencing a logical file that has been opened for output.

CAUTION: The form ?# cannot be used as an abbreviation for PRINT#.

Examples:

100 PRINT#1,A

Output numeric variable A and a RETURN code to logical file 1.

200 PRINT#4,A\$

Output string variable A\$ and a RETURN code to logical file 4.

300 PRINT#10,B%;",":C\$

Output numeric variable B%, a comma, string variable C\$, and a RETURN code to logical file 10.

10 OPEN 1,1,2

Open cassette #1 for write.

20 PRINT#1,"HI"

Output HI to tape.

55 OPEN 3,3

Open display screen.

60 PRINT#3,"HI"

Same as ?"HI"

READ

The READ statement assigns values from a corresponding DATA statement to variables named in the READ list.

Format:

READ	{	var	Read one data item.
		var,var,...var	Read multiple data items.

where:

var is a numeric or string variable.

READ is used to assign values to variables. READ can take the place of multiple assignment statements (see LET. . . =).

READ statements with variable lists require corresponding DATA statements with lists of constant values. READ does the actual assigning of data items to the list of variables. On executing the first READ statement in the program, the first DATA constant is assigned to the first READ variable. If the READ statement has more than one variable listed, then the second DATA constant is assigned to the second READ variable. The assignments continue until all of the READ variables in the READ statement have been assigned values from successive DATA constants. A pointer is maintained to the next available DATA constant. The next READ always assigns the next DATA constant. The number of READ and DATA statements can differ, but there has to be an available DATA constant whenever a READ statement is encountered. The data constants and corresponding variables have to agree in type. A string variable can accept any specified constant; a numeric variable can accept only numeric data.

There may be more data than READ statements assigned; however, if the end of the DATA statements is reached before the end of the READ variables, the program aborts with an ?OUT OF DATA error message.

READ is generally executed in program mode. It can be executed in immediate mode as long as there are corresponding DATA constants in the current stored program to read from.

Examples:

10 DATA 1,2,3	On completion, A=1
20 READ A,B,C	B=2
	C=3
150 READ C\$,D,F\$	On completion, C\$="STR"
160 DATA STR	D=14.5
170 DATA 14.5,"TM"	F\$="TM"

REM

The Remark statement (REM) allows comments to be placed in the program for program documentation purposes.

Format:

REM comment
 where:
 comment is any sequence of characters.

PET BASIC ignores Remark statements other than reproducing them in program listings. Remark statements may be placed in the path of program execution, and they may be branched to. A Remark statement is terminated only by a RETURN (end of line). A Remark statement may be placed on a line of its own:

100 REM comment

or it may be placed as the last statement on a multiple-statement line:

100 A=6+B:IF A>10 THEN 20:REM comment

A Remark cannot be placed ahead of any other statements on a multiple-statement line, since anything after a REM to the end of the line is treated as a comment.

10 REM BRANCH IF DONE:GOTO 200
 treated as part of remark ↑ not executed

REM is intended for program mode usage to document a stored program. It may be given in immediate mode.

Examples:

```
10 REM *****
20 REM ***PROGRAM EXCALIBUR***
30 GOTO 55:REM BRANCH IF OUT OF DATA
```


RESTORE

The RESTORE statement resets the DATA statement pointer to the beginning of data.

Format:

```
RESTORE
```

RESTORE may be given in immediate or program mode.

Example:

```
10 DATA 1,2,N44
20 READ A,B,B$           A=1,B=2,B$="N44"
30 RESTORE
40 READ X,Y,Y$           X=1,Y=2,Y$="N44"
```

RETURN

The RETURN statement branches program control to the statement in the program following the most recent GOSUB call.

Format:

```
RETURN
```

Each subroutine must terminate with a RETURN statement.

Example:

```
100 RETURN
```

Note that the RETURN statement, for returning program control from a subroutine, and the RETURN key, for moving the cursor to the beginning of the next display line, are entirely different operations. You should have no trouble distinguishing these two functions.

STOP

The STOP statement causes the program to stop execution and return control to PET BASIC with a break message printed.

Format:

```
STOP
```

On encountering a STOP statement, execution control is returned to PET BASIC. The following message is printed:

```
BREAK IN line
```

The line number in the statement above indicates where the STOP statement was executed. Typing in CONT from the keyboard continues program execution at the statement following the STOP statement.

Example:

655 STOP

will cause the message
BREAK IN 655 to be printed.**WAIT**

The WAIT statement halts program execution until a memory location becomes non-zero.

Format:
$$\text{WAIT} \quad \left\{ \begin{array}{l} \text{address, mask} \\ \text{address, mask, xor} \end{array} \right.$$

where:

address	is a memory address.
mask	is a one-byte mask value.
xor	is a one-byte mask value.

The WAIT statement does the following:

1. Fetches the contents of the addressed memory location.
2. Exclusive-ORs the value obtained in step 1 with xor, if present. If xor is not specified, it defaults to 0. When xor is 0, this step has no effect.
3. ANDs the value obtained in step 2 with the specified mask value.
4. If the result is 0, WAIT returns to step 1, remaining in a loop that halts program execution at the WAIT.
5. If the result is not 0, WAIT processing is completed and program execution continues with the next statement following the WAIT statement.

The keyboard STOP key cannot interrupt execution of a WAIT statement.

BASIC ARITHMETIC FUNCTIONS

ABS

ABS returns the absolute value, or magnitude, of a number. This is the value of the number without regard to sign.

Format:

ABS(arg)

where:

arg is a numeric constant, variable or expression.

Examples:

A=ABS(10) Results in A=10.
A=ABS(-10) Results in A=10.
PRINT ABS(X),ABS(Y),ABS(Z)

ATN

ATN returns the arctangent of the argument.

Format:

ATN(arg)

where:

arg is a numeric constant, variable or expression.

ATN returns the value in radians in the range ± 1.1071487 .

Examples:

A=ATN(AG)
?180/ π *ATN(A) Prints the value in degrees.

COS

COS returns the cosine of the argument.

Format:

COS(arg)

where:

arg is a numeric constant, variable, or expression representing an angle in radians.

Example:

A=COS(AG)

EXP

EXP returns the value e^{arg} . The value of e used is 2.71828183.

Format:

EXP(arg)

where:

arg is a numeric constant, variable, or expression in the range ± 88.0296910 .

A number larger than the allowed range will result in an overflow error message. A number smaller than the allowed range will yield a zero result.

Examples:

?EXP(0)	Prints 1.
?EXP(1)	Prints 2.71828183.
EV=EXP(2)	Results in EV=7.3890561.
EV=EXP(50.24)	Results in EV=6.59105247E+21.
?EXP(88.0296919)	Largest allowable number, yields. 1.70141183E+38.
?EXP(-88.0296919)	Smallest allowable number, yields. 5.87747176E-39.
?EXP(88.029692)	Out of range, overflow error message.
?EXP(-88.029692)	Out of range, returns 0.

INT

INT returns the integer portion of a number, rounding to the next lower signed number.

Format:

INT(arg)

where:

arg is a numeric constant, variable, or expression.

For positive numbers, INT is equivalent to dropping the fractional portion of the number without rounding. For negative numbers, INT is equivalent to dropping the fractional portion of the number and adding 1. Note that INT does not convert a floating point number (5 bytes) to integer type (2 bytes).

Examples:

A=INT(1.5)	Results in =1.
A=INT(-1.5)	Results in A=-2.
A=INT(A+A%)	
X=INT(-0.1)	Results in X=-1.

A caution here: since floating point numbers are only close approximations of real numbers, an argument may not yield the exact INT function value you might expect. For instance, consider a simple loop going from 2 to 4 by successively adding one-tenth (.1).

```
FOR I=2 TO 4 STEP.1:?I:NEXT
2
2.1
2.2
2.3
2.4
2.5
2.6
2.7
2.8
2.9
3
3.1
3.2
3.3
3.4
3.49999999
3.59999999
3.69999999
3.79999999
3.89999999
3.99999999
```

READY.

You will see that the six numbers 3.5 through 4 are represented only approximately. If these values were in an array A(0) through A(20), the function INT(A(20)) would equal 3, not 4 as would be expected.

LOG

LOG returns the natural logarithm, or log to the base e. The value of e used is 2.71828183.

Format:

LOG(arg)

where:

arg is a positive, non-zero number, variable, or expression.

An Illegal Quantity error message is issued if the argument is zero or negative.

Examples:

?LOG(1)	Prints 0.
A=LOG(4)	Results in A=1.38629436.
A=LOG(10)	Results in A=2.30258509.
A=LOG(1E6)	Results in A=13.8155106.
A=LOG(X)/LOG(10)	Calculates log to the base 10.

RND

RND provides the ability to generate random number sequences in PET BASIC. At each reference, RND returns a number between 0 and 1, $0 < n < 1$.

Format:

RND(arg)	Return random number.
RND(-arg)	Store new seed number.

where:

arg	is a positive, non-zero number, variable, or expression. This RND function returns the next random number in sequence. The value of arg in this case is arbitrary; that is, the random number returned for a positive argument does not depend on the value of the argument.
-arg	is a negative number, variable, or expression. The RND function stores a new seed. This causes subsequent random numbers to begin a new sequence based on the -arg value. RND (-arg) returns a small value, constant for the given -arg, that should be disregarded.

Examples:

A=RND(-1)	Store a new seed based on the value -1.
A=RND(1)	Fetch the next random number in sequence.

An argument of zero is treated as a special case in PET BASIC. It does not store a new seed, nor does it return a random number. RND(0) uses the current system time value TI to introduce an additional random element into play.

A pseudo-random seed is stored by the function:

RND(-TI)	Store pseudo-random seed.
----------	---------------------------

RND(0) can be used to store a new seed that is more truly random, by using the following function:

RND(-RND(0))	Store random seed.
--------------	--------------------

You can compare these operations to a deck of cards: $RND(-T)$ is like shuffling the deck, $RND(0)$ is like cutting the cards, and $RND(-RND(0))$ is like giving the cards both a shuffle and a cut.

To generate random numbers in non-repeating sequences as would be needed, say, in games of chance, use $RND(-RND(0))$ to store a random seed and then fetch as many random numbers as needed with $RND(1)$ references, where 1 can be 1 or any other positive, non-zero number.

Examples of using the RND function to generate random numbers on the PET are given in Chapter 5.

SGN

SGN , for sign of a number, can be used to determine if a number is positive, negative, or zero.

Format:

$SGN(arg)$

where:

arg is a numeric constant, variable, or expression.

The SGN function returns: +1 if the number is positive, non-zero.
0 if the number is zero.
-1 if the number is negative.

Examples:

```
?SGN(-6)           Prints -1.  
?SGN(0)            Prints 0.  
?SGN(44)           Prints 1.  
IF A > C THEN SA=SGN(X)  
IF SGN(M) >= 0 THEN PRINT "POSITIVE NUMBER"
```

SIN

SIN returns the sine of the argument.

Format:

$SIN(arg)$

where:

arg is a numeric constant, variable, or expression representing an angle in radians.

Examples:

```
A=SIN(AG)  
?SIN(45*π/180)     Prints sine of 45 degrees.
```

SQR

SQR finds the square root of a positive number. A negative number results in an error message.

Format:

SQR(arg)

where:

arg

is a positive number, variable, or expression.

Examples:

A=SQR(4)

Results in A=2.

A=SQR(4.84)

Results in A=2.2.

?SQR(10)

Prints 3.16227766.

?SQR(144E30)

Prints 1.2E+16.

TAN

TAN returns the tangent of the argument.

Format:

TAN(arg)

where:

arg

is a numeric constant, variable or expression representing an angle in radians.

Example:

XY(1)=TAN(180* π /180)

BASIC STRING FUNCTIONS

ASC

ASC returns the ASCII code number for a specified character.

Format:

ASC(arg)

where:

arg is a string constant, variable, or expression.

If the string is longer than one character, ASC returns the ASCII code for the first character in the string. The returned argument is a number and may be used in arithmetic operations. ASCII codes are listed in Appendix A.

Examples:

?ASC("A") Prints 65.

N=ASC(B\$)

X=ASC("STR") Prints the ASCII value of "S" (83).

CHR\$

CHR\$ returns the string representation of the specified ASCII code.

Format:

CHR\$(arg)

where:

arg is a valid ASCII code (see Appendix A) in the range $0 < \text{arg} \leq 255$.

CHR\$ can be used to specify characters that cannot be represented in strings, such as the carriage return code and the double quotation mark.

Examples:

IF C\$=CHR\$(13) GOTO 10 Branches if C\$ is a carriage return.

?CHR\$(34);"HOHOHO";CHR\$(34) Prints the eight characters
"HOHOHO"

LEFT\$

LEFT\$ returns the leftmost characters of a string.

Format:

LEFT\$(arg,count)

where:

arg is a string constant, variable, or expression.

count is a numeric constant, variable, or expression specifying the number of leftmost characters to be returned, in the range $0 < \text{count} \leq 255$.

Examples:

```
?LEFT$("ARG",2)           Prints AR.
A$=LEFT$(B$,10)
```

LEN

LEN returns the length of the string argument.

Format:

```
LEN(arg)
```

where:

arg is a string constant, variable, or expression.

LEN returns a number that is the count of characters in the specified string.

Examples:

```
?LEN("ABCDEF")           Prints 6.
N=LEN(C$+D$)
```

MID\$

MID\$ returns any specified portion of a string.

Format:

```
MID$(arg,position { (blank) } )
```

Return all characters from position to end.
Return count characters starting from position.

where:

arg is a string constant, variable, or expression.

position is a numeric constant, variable, or expression specifying the character position in the string at which retrieval is to begin. Position 1 is the beginning of the string. If the position is beyond the end of the string, a null string is returned. This parameter must be in the range $0 < \text{position} \leq 255$.

count if present, is a numeric constant, variable, or expression that specifies the number of characters to be returned. If omitted, or if it is larger than the count to the end of the string, all characters from the starting position to the end of the string are returned. This parameter must be in the range $0 < \text{count} \leq 255$.

An Illegal Quantity error message is printed if a parameter is out of range.

Examples:

?MID\$("ABCDE",2,1)	Prints B.
?MID\$("ABCDE",3,2)	Prints CD.
?MID\$("ABCDE",3)	Prints CDE.

RIGHT\$

RIGHT\$ returns the rightmost characters in a string.

Format:

RIGHT\$(arg,count)

where:

arg	is a string constant, variable, or expression.
count	is a numeric constant, variable, or expression specifying the number of rightmost characters to be returned, in the range $0 < \text{count} \leq 255$.

Examples:

?RIGHT\$(ARG,2)	Prints RG.
MM\$=RIGHT\$(X\$+"#").	

STR\$

STR\$ returns the string equivalent of the numeric argument.

Format:

STR\$(arg)

where:

arg	is a numeric constant, variable, or expression.
-----	---

STR\$ converts the number to the same format as it would be printed out in a PRINT statement. The first character of the string is a blank if a number is positive, or it is a minus sign if the number is negative.

Examples:

A\$=STR\$(14.6)	
?STR\$(1E2)	Prints 100.
?STR\$(1E10)	Prints 1E+10.

VAL

VAL returns the numeric equivalent of the string argument.

Format:

VAL(arg)

where:

arg	is a string constant, variable, or expression.
-----	--

The number returned by VAL may be used in arithmetic computations.

VAL converts the string argument by first discarding any leading blanks. If the first non-blank character is not a numeric digit (0-9), the argument is returned as a value of 0. If the first non-blank is a digit, VAL begins converting the string into real number format. If it subsequently encounters a non-digit character, it stops processing so that the argument returned is the numerical equivalent of the string up to the first non-digit character.

Examples:

```
A=VAL("123")
NN=VAL(B$)
```

BASIC FORMAT FUNCTIONS

POS

POS is a limited use function that returns the column position of the cursor.

Format:

```
POS(arg)
```

where:

arg is a dummy argument and may be string or numeric.

For POS purposes, the character positions along each row are numbered 0 through 39. Character positions for a string are numbered 0 through 255. POS returns the current cursor position (0-255) except after a Home, Clear Screen, or POKE, which do not reset the POS indicator. The position is taken as the current cursor position before the POS function itself is typed. POS is reset to 0 at each RETURN.

Examples:

```
?POS(1)           At the beginning of a line,
                  returns 0.
?"xyz ABC";POS(1) With a previous POS value of 0,
                  prints a POS value of 6.
```

SPC

SPC moves the cursor right a specified number of positions.

Format:

SPC(arg)

where:

arg

is a numeric constant, variable, or expression that specifies the number of skips to be printed. This parameter must be in the range $0 \leq \text{arg} \leq 255$. A value of 0 prints 256 skips.

SPC begins counting positions at the current cursor position before the SPC function itself is typed.

Examples:

?SPC(1);"H"

H

?SPC(5);"H"

H

TAB

TAB moves the cursor right to the specified column position.

Format:

TAB(arg)

where:

arg

is a numeric constant, variable, or expression that specifies the column position to which the cursor is to be moved. This parameter must be in the range $0 \leq \text{arg} \leq 255$.

TAB moves the cursor to the n+1 position, where n is the value of the specified argument.

Examples:

?"QUARK";SPC(10);"W"

QUARK W

?"QUARK";TAB(10);"W"

QUARK W

These two examples show the difference between SPC and TAB. SPC skips ten positions from the last cursor location, whereas TAB skips to the 10+1th position on the row.

BASIC SYSTEM FUNCTIONS

FRE

FRE is a system function that collects all unused bytes of memory into one block (called "garbage collection") and returns the number of free bytes.

Format:

FRE(arg)

where:

arg is a dummy argument, and may be string or numeric.

FRE can be used anywhere a function may appear, but it is normally used in an immediate mode PRINT request.

Examples:

?FRE(1)

Institute garbage collection and print the number of free bytes.



PEEK

PEEK returns the contents of the specified memory location. PEEK is the function counterpart of the POKE statement.

Format:

PEEK(address)

where:

address is a numeric constant, variable, or expression specifying the address of the location whose contents are to be fetched.
 $0 \leq \text{address} \leq 65535$.

Any memory locations from 0 to 65535 can be PEEKed except the block of system locations from 49152 to 57599; these locations contain the 8K BASIC interpreter and have been PEEK-protected to discourage examination of proprietary software. The protected area returns a PEEK value of 0. Locations of interest that you might want to PEEK at are discussed in Chapter 6.

Examples:

?PEEK(1) Prints contents of memory location 1.
A=PEEK(20000)

ST

ST represents the current value of the PET's I/O status word. This word is set to certain values depending on the results of the last input/output operation that affects the status word.

Format:

ST

ST values for tape cassette I/O operations are shown in Table 4-3.

Table 4-3. ST Values for Tape I/O

ST Bit Position	ST Numeric Value	Tape READ	Tape VERIFY
0	1		
1	2		
2	4	Short block	Short block
3	8	Long block	Long block
4	16	Unrecoverable read error	Any mismatch
5	32	Checksum error	Checksum error
6	64	End of file	
7	-128	End of tape	End of tape

The status should be checked, if it is going to be checked, immediately after an INPUT# or GET# statement.

Examples:

```
10 IF ST < > 0 GOTO 500           Branch on any error.
50 IF ST=4 THEN ?"SHORT BLOCK"
```

SYS

SYS is a system function that transfers program control to an independent subsystem.

Format:

SYS(address)

where:

address is a numeric constant, variable, or expression representing the starting address at which execution of the subsystem is to begin. The value must be in the range $0 \leq \text{address} \leq 65535$.

Unlike other functions, SYS can be specified alone in an immediate program statement. Use of the SYS command is described further in Chapter 6.

TI, TI\$

TI and TI\$ represent two system time variables.

Format:

TI Number of jiffies since current startup.
TI\$ Time of day string.

Examples:

```
?TI
TI$="081000"
```

Usages of TI and TI\$ are described in Chapter 5, under "Setting Time of Day."

USR

USR is a system function that passes a parameter to a user-written assembly language subroutine whose address is contained in memory locations 1 and 2, and fetches a return parameter from the subroutine.

Format:

USR(arg)

where:

arg is the parameter value passed to the subroutine.

The USR function is described in more detail in Chapter 6.

USER-DEFINED FUNCTION

DEF FN

The DEF function (DEF FN) allows the user to define his own special purpose functions and use the functions in his program in the same way that the intrinsic PET BASIC functions are used.

Format:

DEF FN name(arg)=formula Define user function.

where:

name	is a simple floating point variable name <u>one or two characters in length</u> . The name can be longer, but an alphabetic name of <u>more than five letters yields a syntax error</u> . The function is used in the program by referring to the name FNname.
arg	is a simple floating point variable name. This is a dummy assignment that represents the argument value in the formula to the right. A string (\$) or integer (%) arg is not allowed.
formula	is any arithmetic expression containing any combination of numeric constants, variables, and operators. Normally the arg appears in the formula.

A user-defined function is limited to one 80-character line; however, a previously defined function can be used in the formula in another user-defined function, so user-defined functions of any desired complexity can be developed.

A user-defined function is used by typing in the function name, FNname, followed by the actual argument enclosed in parentheses.

FNname(arg) Reference user function.

Only a single argument can be passed to a user-defined function. The function name can be redefined by another DEF FN with the same name in the same program.

The DEF FN definition statement is illegal in immediate mode. However, a user-defined function that has been defined by a DEF FN statement in the current stored program can be referenced in an immediate mode statement.

Examples:

```
10 DEF FNC(R) =  $\pi$ *R2
```

Defines a function that calculates the circumference of a circle. It takes a single argument R, the radius of the circle, and returns a single numeric value, the circumference of the circle.

```
?FNC(1)
```

Prints 3.14159265 (the value of π).

```
A=FNC(14)
```

Assigns to A the value calculated by the user-defined function FNC.

```
55 IF FNC(X) > 60 GOTO 150
```

Uses the value calculated by the user-defined function FNC as a branch condition.

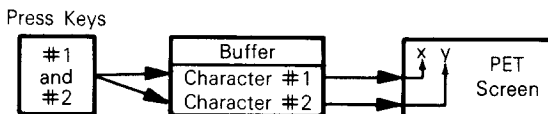
Making the Most of PET Features

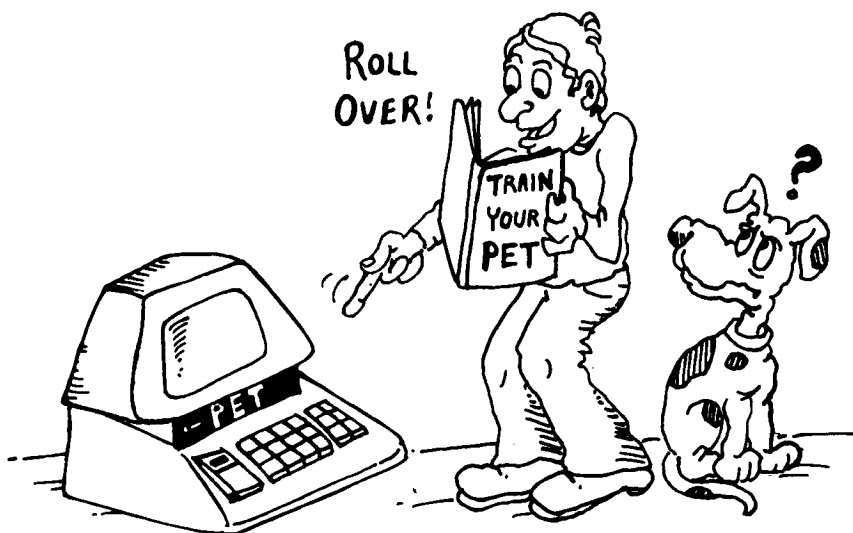
In this chapter we go over features of the PET that you will come across sooner or later in your PET BASIC programs. Where the "feature" is a limitation or other stumbling block, we tell you how to get around it (if possible) or at least warn you about it. Where the feature is a real capability, we tell you how to use it to its fullest advantage.

KEYBOARD ROLLOVER

If you type rapidly you may type faster than the characters can be printed. If you press two or more keys simultaneously or press the second key before the first character is printed, a keystroke will be ignored — unless your keyboard has "rollover." Rollover "remembers" a keystroke until it is printed. Fortunately, the PET keyboard has rollover.

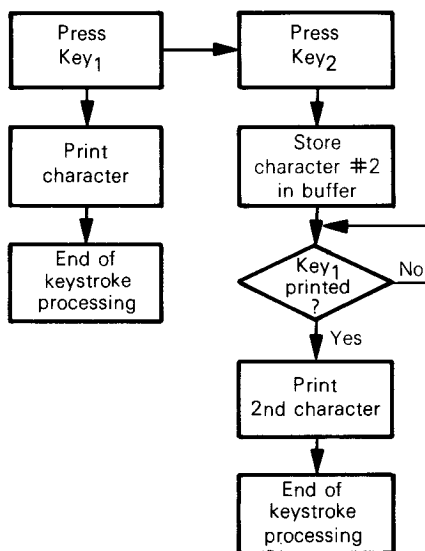
Rollover remembers incoming keystrokes while a preceding keystroke is being processed. The "remembered" keystrokes are then processed. The keystrokes are remembered by storing them in a buffer until they are printed:





Without this buffer, rapidly incoming keystrokes would be lost before printing — and rollover would be impossible.

The following flowchart shows how rollover works in greater detail. The PET must store incoming keystroke #2 in the buffer until keystroke #1 is printed. Once keystroke #1 is printed, keystroke #2 is taken from the buffer and printed.

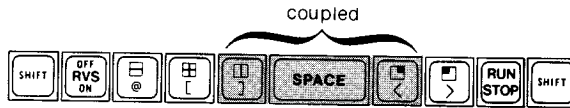



All you need to remember about rollover is that it is a beneficial feature of the PET keyboard that allows you to type in data faster than you could if rollover were not present.

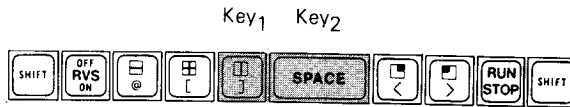
Because of the size and type of keys used for the small PET keyboard, you may encounter minor data entry problems if you try to type fast on the small keyboard. These problem areas are described in detail below. If you have a full size PET keyboard, you might proceed to the next section, "The Keyboard Buffer."

Two-Key Rollover (Small Keyboard)

There are areas on the compact PET keyboard which do not give predictable results during rollover. The space key on the bottom row (a SPACE character is indicated hereafter by the symbol "␣") is coupled with the keys directly to the left and right, giving incorrect results.



When the RIGHT BRACKET  key and the SPACE key to its left are pressed simultaneously:



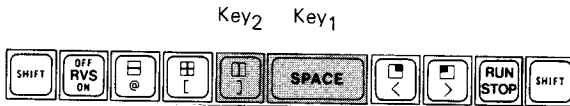
correct rollover should print:

]
␣

but the PET often prints:

] < ␣ or] ␣ < ␣

The key coupling erroneously puts the "<" character (to the right of the "␣" key) and sometimes an extra "␣" into the buffer. The same problem occurs when the keys are reversed:



Instead of printing:

]
␣

the PET often prints:

< ␣] or ␣ ␣ <]

The above examples show that **rollover should not be depended upon when typing on the bottom row of the small keyboard** due to the coupling problem with the SPACE key. **But the rest of the small PET keyboard can be depended upon for two-key rollover as a valuable safeguard against losing keystrokes.**

Three-Key Rollover (Small Keyboard)

Three-key rollover remembers the second and third keystrokes when three keys are pressed before the first key is released. **Three-key rollover is unpredictable on the PET and should not be depended upon**, as correct printing depends upon the keyboard position of all three keys. **This section is only for those readers who are interested in PET trivia and who want to experiment in three-key rollover.**

The compact keyboard may be divided into odd and even rows, as shown in Figure 5-1.

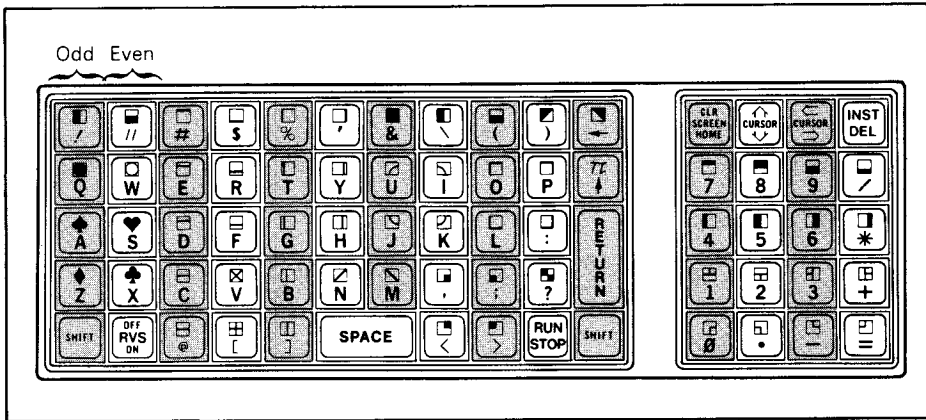


Figure 5-1. Odd/Even Division of PET Keyboard

If the three keystrokes, Key₁, Key₂, and Key₃, are not adjacent to each other, then the result is correct and will be printed as:

(characters of) Key₁ Key₂ Key₃

But if the three keystrokes are adjacent, the result is incorrect, and the type of error depends upon whether Key₁ is in an odd or even column. Figure 5-2 illustrates the two different types of errors that occur depending on the position of Key₁.

To demonstrate this phenomenon, let us use three actual examples from the keyboard.

The first example shows three non-adjacent keystrokes simultaneously entered. The diagram shows F, U, N entering the buffer and then printing to the screen in the correct order.

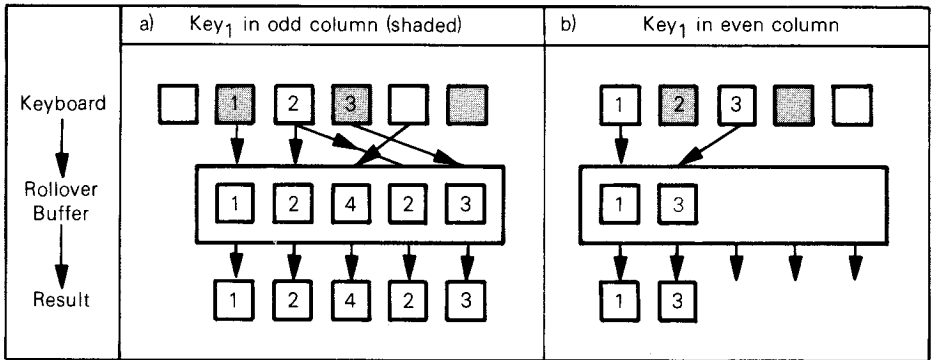
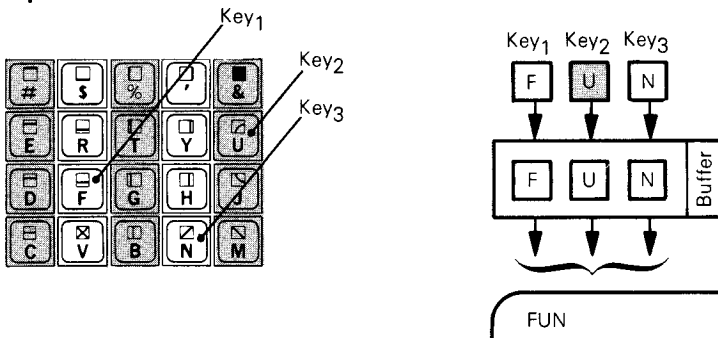


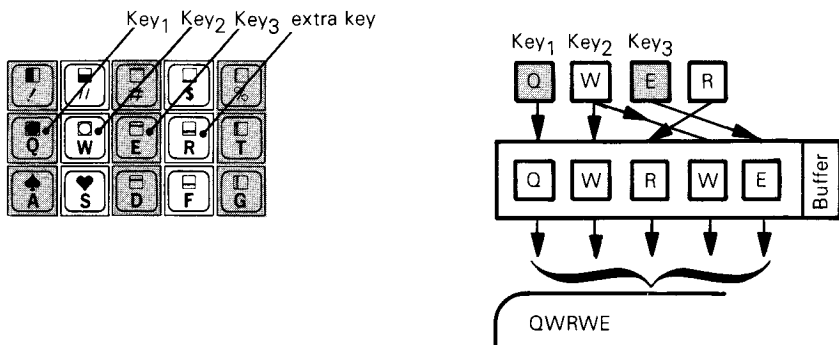
Figure 5-2. Odd/Even Three-Key Rollover

Example 1:



The second example illustrates three adjacent keys with Key₁ in an odd column. The three keys used are Q, W, and E. Observe that an extra character has been put into the buffer without actually being typed in. This extra character is always from the key directly to the right of Key₃; in this example, R is directly to the right of Key₃, E. Not only is Key₃ not entered as the third key, it is not entered as the fourth key either; Key₂ is again put into the buffer before Key₃ is finally inserted.

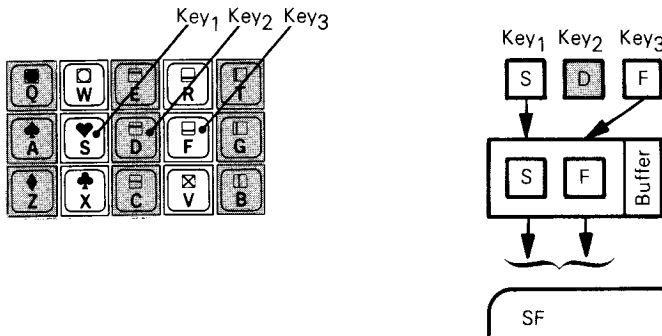
Example 2:



This is another example of a key coupling error. The keys throughout the keyboard are coupled together in various combinations, but only give problems in these certain combinations. Because of key coupling, the extra key is thrown in erroneously.

The third example shows a set of three adjacent keys with Key₁ in an even column. When Key₁ is positioned in an even column, Key₂ is passed by and never put into the buffer for printing.

Example 3:



The bottom row of the small keyboard is a special problem area for three-key rollover. It is inconsistent in relation to the three-key rollover for the remaining keyboard.



The simultaneous hitting of the three following keys:



produces only the reverse graphic representation of a fourth key:



The change from normal to graphic mode might be due to reverse coupling of the SHIFT and RVS keys.

The following sets show more of the rollover inconsistency that occurs with keys in the bottom row of the keyboard:

Key ₁ Column	Keys	Rollover Result	Key Order
Even		[<∅ [∅ <	Keys #1, 4, 3 Keys #1, 3, 4
Even		∅ <<>	Keys #1, 2, 2, 3
Odd]<]<∅<	Keys #1, 3 Keys #1, 3, 2, 3
Odd		<	Key #1

These three-key rollover problems are not unique to the alphabetic and special character keys alone. The right-hand portion of the keyboard, containing the cursor controls and numerical keys, is a continuation of the even/odd column format as shown in Figure 5-1. Cursor control and numeric rollover respond like the alphabetic and graphic rollover.

In summary, to avoid extraneous characters and combinations, hit only one key at a time on the small PET keyboard. Be especially careful when depressing any of the bottom row keys. Three-key rollover on the PET is a desirable option, but, as the examples show, should not be depended upon for completely accurate keyboard entry when typing is very fast or erratic.

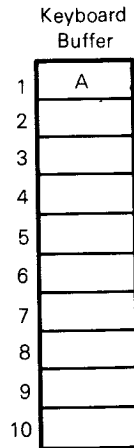
BUFFER

The PET has a 10-character keyboard buffer to hold the ASCII code equivalents of characters whose keys are pressed at the keyboard. This is normally of no concern to you, but you should be aware of how the buffer operates for when you are programming PET inputs.

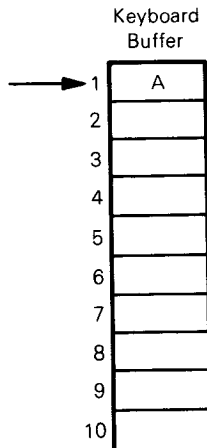
To illustrate, load the final saved version of the BLANKET program (from Chapter 3) and press a key for display. While the display is occurring, press up to ten more keys, then sit back and relax. Each of the keyed-in characters will be fetched, in turn, and displayed by the BLANKET program.

Let's look at this process in more detail.

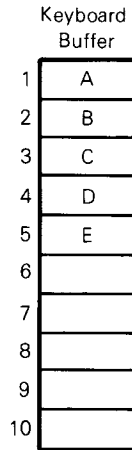
Whenever you press, say, the A key, it goes into the first storage location in the 10-character keyboard buffer:



The PET keeps track of the number of characters in the buffer and the location of the next character to be displayed. When GET fetches this character, a pointer is changed to indicate that this character has been fetched:

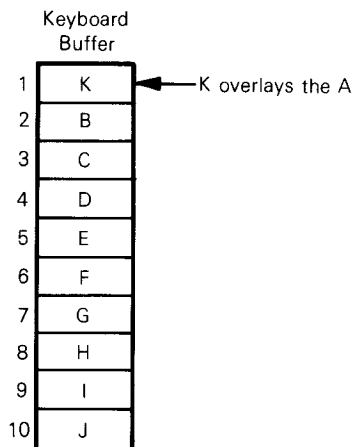


When you press additional keys while the A is being displayed, the additional characters are stored in the keyboard buffer beginning at the next available location. Suppose you type in A and while A is being displayed you type in B, C, D, and E. These characters are all stored in the keyboard buffer:



If you let the program continue, it will successively display all the letters stored in the keyboard buffer. After A is finished, the program fetches B and displays it across the 20 lines, then it fetches C and displays it, etc.

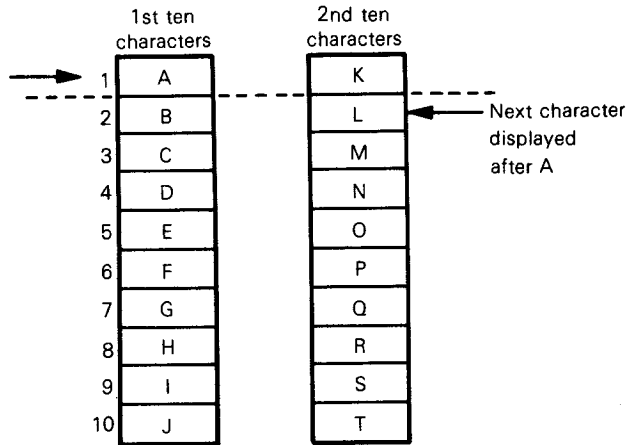
What happens if you type in more than ten characters? At the tenth character the buffer pointer wraps around, and continues storing the characters being keyed in from position 1 of the buffer. If you type in the first eleven letters of the alphabet, A-K, the first ten letters are stored in the ten buffer locations, then the letter K is stored back in the first buffer location, overlaying the A:



When the program finishes displaying the A, it returns to fetch another character. But the PET has already fetched the character in location 1, so it considers the buffer empty! Keying in exactly eleven characters, or multiples of

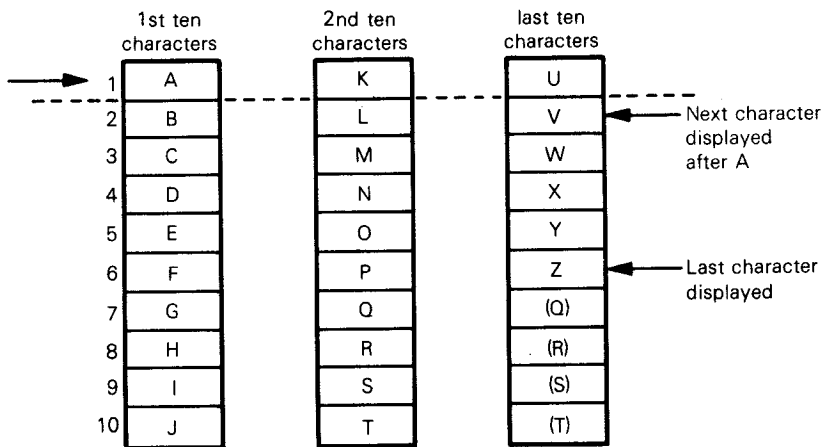
eleven characters, produces no additional automatic displays in program BLANKET.

Typing in 12 to 20 characters causes the most recent string of characters 12 and on to be printed. For example, type in A, and while A is being displayed type in B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, and T.



The order of display is A, L, M, N, O, P, Q, R, S, T.

The same holds true for additional multiple characters. Type in A and while A is being displayed type in the rest of the alphabet in order (you will have to be fairly quick to do this).



The negating effect occurs for every 11 characters. For instance, type in A and B, and let A display completely. Then, while B is displaying, type in C, D, E, F, G, H, I, J, K, and L. The additional ten characters are cancelled out, just as the additional ten characters B through K were when entered while A was being displayed.

Emptying the Buffer Before a GET

This illustration of stuffing the keyboard buffer in advance is mostly a mild surprise, and a rather pleasant one, for program BLANKET. You can save up the characters you want displayed rather than keying them in one at a time in response to the HIT A KEY message. For other programs, however, it could come as a rude shock that idle toying with the keyboard during, say, a game could cause the game program to fetch an unwanted string of characters from the keyboard buffer. To avoid this, you can program a loop to empty the keyboard buffer, discarding the fetched characters, before fetching an intended response character. Implement a GET fetch by the code example below:

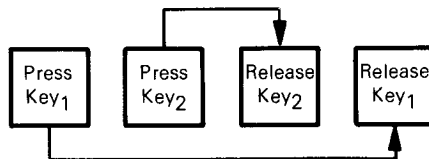
```
95 FOR I=1TO10:GET C$:NEXTI: REM EMPTY KYBD BFR
100 GET C$:IF C$="" GOTO 100
```

Line 95 is the added line to empty the keyboard buffer by getting all ten possible characters that might have inadvertently been "stuffed" there. Line 100 is the standard BLANKET GET line.

Edit program BLANKET by adding line 95 as shown above. Now press any combination of keys while a character is being displayed. Any stored characters are fetched and discarded by the GET loop, so you will not have any automatic display continuations with this loop added.

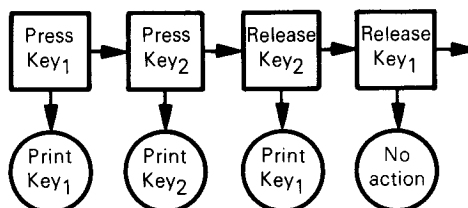
TWO-KEY REPETITION

A commonly used keyboard typing technique is to hold down one key while pressing and releasing another.



Holding down one key while pressing another key is often an unnoticed, automatic typing action used during a conversion from alphabetic to graphic or shifted mode. But it may be expanded to other uses, such as repeating sequences, to produce some interesting effects.

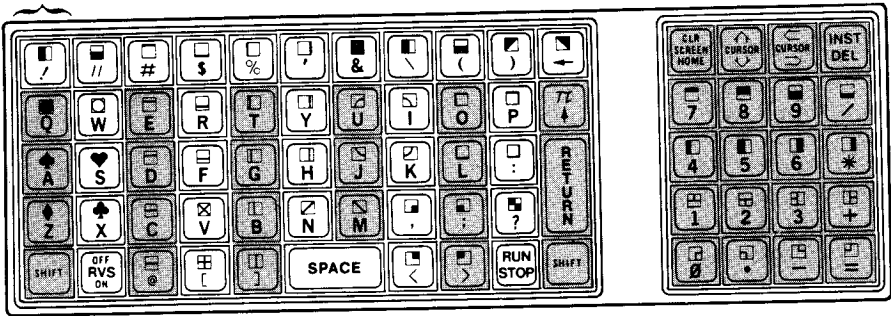
Two-key repetition causes an alternating sequence of two characters. This method is similar to rollover because a sequence is produced, but the results are different. The following diagram shows what happens:



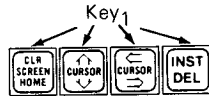
Two-key repetition is a feature of any PET keyboard, small or full size. For simplicity, only the compact keyboard is shown below. Keys available for Key₁ will differ, depending on which keyboard you have.

As with rollover, **the keyboard is divided into odd and even columns, with only the odd column keys available for Key₁.** However, the entire right-hand numeric board and cursor controls are available for Key₁, regardless of odd and even columns.

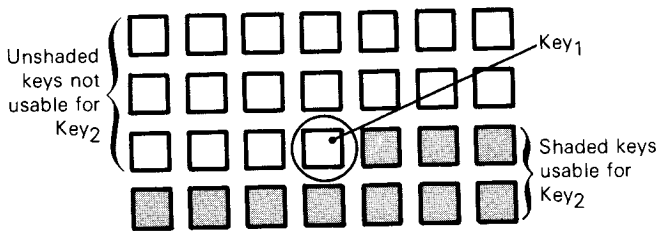
Keys available as Key₁ (shaded)



Any key in the even columns used for Key₁ will not produce a repeating sequence. Another peculiarity is that the cursor keys must be used as Key₁ in either programmed or immediate mode.



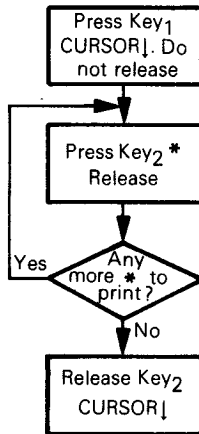
Further limiting this feature is the requirement that Key₂ be to the right of Key₁ in the same row, or below the row of Key₁. If Key₂ is located above Key₁ or to the left, the process stops after the first printing of Key₁.



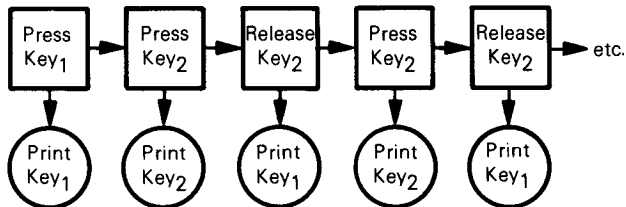
A useful application to demonstrate two-key repetition is creating graphic diagrams with cursor movements. For example, draw the diagonal line shown in the diagram below.



To create this pattern, use the CURSOR↓ key as Key₁ and the asterisk (*) key as Key₂. As you hold down Key₁, repeatedly press and release Key₂; the cursor will automatically move and print the "*" in a diagonal line. Following is a flowchart of the process:



As you can see, if you hold down Key₁ as you press and release Key₂, it appears as if the two keys were being pressed alternately.



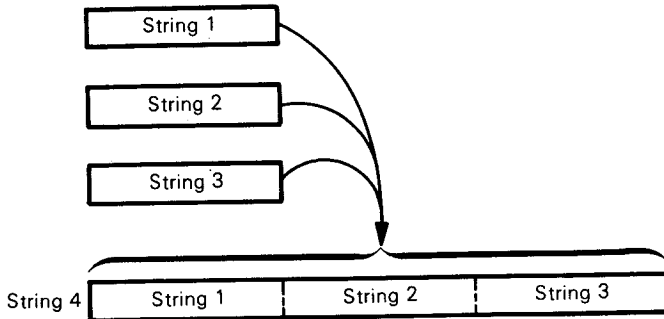
But if Key₁ is released while Key₂ is depressed, Key₁ will not be printed a second time.

There are countless combinations available for repeating sequences on the PET. Since three-or-more-key repeating sequences have limited use and varied and unpredictable results, they will not be covered in this book, but may serve as an interesting exercise for the curious programmer.

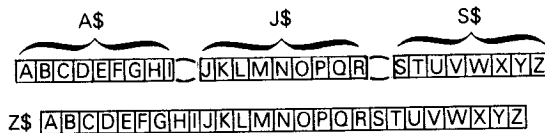
STRINGS

CONCATENATING STRINGS

Recall that when creating strings the PET will accept alphabetic, graphic, and numeric characters, or combinations of these. While handling strings, it may be useful to connect one or more strings together into a single string by linking them end to end in a chain-like fashion:



Suppose, for example, we want to create one large string, Z\$, containing the alphabet A through Z. By linking together the last character of A\$, shown below, with the first character of J\$, and the last character of J\$ to the first character of S\$, Z\$ may be created:



Concatenating alphabetic strings, such as A\$, J\$ and S\$, may be done using one simple statement:

$$Z\$=A\$+J\$+S\$\$$$

The arithmetic operator “+” adds the contents of numeric variables or the numbers themselves. But when used with strings the “+” serves as a linkage sign to concatenate the strings together. Table 5-1 summarizes the addition of strings and numbers.

A word of caution: strings cannot be separated or broken apart in the same fashion as they are concatenated; they cannot be “subtracted” the way they are “added.” For instance, to create string X\$ containing the contents of J\$ and S\$ from our original strings A\$, J\$, S\$, and Z\$, it would be incorrect to type:

$$X\$=Z\$-A\$\$ \quad \text{Incorrect}$$

Table 5-1. Addition (+) Operations

Sign	Type	Example Statement.	Operation	Result																															
+	numbers	P = 2 + 3	2 + 3	P = 5																															
+	numeric variables	Q = T + S T = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> S = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	1	1	1	1	1	$\begin{array}{r} 12345 \\ +11111 \\ \hline 23456 \end{array}$	Q = 23456																					
1	2	3	4	5																															
1	1	1	1	1																															
+	alphabetic strings	R\$ = A\$ + F\$ A\$ = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr></table> F\$ = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td></tr></table>	A	B	C	D	E	F	G	H	I	J	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td></td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td></tr></table>	A	B	C	D	E		F	G	H	I	J	R\$ = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td></tr></table>	A	B	C	D	E	F	G	H	I	J
A	B	C	D	E																															
F	G	H	I	J																															
A	B	C	D	E		F	G	H	I	J																									
A	B	C	D	E	F	G	H	I	J																										
+	numeric strings	Q\$ = T\$ + S\$ T\$ = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> S\$ = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	1	1	1	1	1	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td></td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5		1	1	1	1	1	Q\$ = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	2	3	4	5	1	1	1	1	1
1	2	3	4	5																															
1	1	1	1	1																															
1	2	3	4	5		1	1	1	1	1																									
1	2	3	4	5	1	1	1	1	1																										

Try it yourself. Type the values of A\$, J\$, S\$, Z\$, and X\$=Z\$-A\$ into the PET as shown below. Your PET will respond with a ?TYPE MISMATCH ERROR IN LINE 50.

```

10 A$="ABCDEFGH I"
20 J$="JKLMNOPQR"
30 S$="STUVWXYZ"
40 Z$=A$+J$+S$
50 X$=Z$-A$ ← Incorrect attempt to get J through Z string
60 PRINT X$

```

RUN

?TYPE MISMATCH ERROR IN LINE 50

The only valid arithmetic operator for strings is the addition sign (+). The other arithmetic operators (-, *, /) will not work. The Boolean operators (<, >, =) may be used for string comparison.

The correct method of extracting part of a larger string is to use the string shift functions listed in Chapter 4. With the LEFT\$, MID\$, and RIGHT\$ functions it is possible to extract any desired portion of a string. In our example, the letters J through Z can be extracted as follows:

```

50 X$=RIGHT$(Z$, 17)
X$ = RIGHT$(

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

, 17)
X$ = 

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|


```

or, the string may be built by the concatenation of J\$ and S\$:

```

50 X$=J$+S$
X$ = 

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| J | K | L | M | N | O | P | Q | R |
|---|---|---|---|---|---|---|---|---|

+

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|


X$ = 

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|


```

Printer/Screen Concatenation

Another method is available if you want the strings concatenated for screen or printer output only, and not retained in the PET memory. This is done with **the PRINT statement and semicolon separators (;)** between the strings:

```
PRINT A$;J$;S$  
  
RUN  
  
ABCDEFGHIJKLMNQPQRSTUVWXYZ
```

The above result (A through Z) is equivalent to the contents of Z\$ because the semicolons allow no embedded blanks between the strings. The only difference is that the screen result (A through Z) is not retained anywhere in PET memory. However, this method will not concatenate strings without embedded blanks if the strings were previously converted from a numeric variable with the STR\$ or VAL functions; this will be discussed further in the "Numeric Strings" section.

Concatenating Graphic Strings

Concatenating graphic strings can be very useful in creating pictures and diagrams. **Graphic strings are concatenated in the same way as alphabetic strings.**

NUMERIC STRINGS

A numeric string is a string composed of numbers — or a combination of numbers and decimal point, negative sign, or exponent — that evaluates to a real number. A numeric string must be identified by a "\$" following the string name.

The contents of numeric strings may be assigned in two different ways, each yielding slightly different results. **When numeric variables are assigned to numeric strings using the STR\$ function, the sign value preceding the number (blank if positive, "-" if negative) is transferred along with the number.**

This is shown in the short program below:

```
10 AB=12345  
20 T$=STR$(AB)  
30 PRINT"AB=";AB  
40 PRINT"T$=";T$  
  
RUN  
  
AB= 12345  
T$= 12345
```

However, **if a number is entered enclosed within quotation marks or entered as a string with an INPUT statement** (or other similar statement such as GET or READ), the numeric string is treated like any other alphabetic or graphic string and **no blank for a positive sign value is inserted before the number**. This is demonstrated in the following program:

```

10 AB=12345
20 T$="12345"
30 PRINT"AB=";AB
40 PRINT"T$=";T$

RUN

AB= 12345
T$=12345

```

Notice that there is no preceding blank printed in the T\$ string above.

Concatenating Numeric Strings

Let's concatenate two strings, T\$ and Q\$, to make a new string, W\$. W\$ should contain the ten digits 1,2,3,4,5,6,7,8,9,0.

```

10 T=12345
20 Q=67890
30 T$=STR$(T)
40 Q$=STR$(Q)
50 W$=T$+Q$ ← Create new string W$
60 PRINT"W$=";W$

```

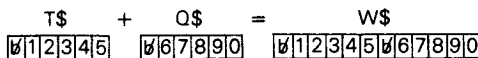
The result would be:

```
W$= 12345 67890
```

Why the embedded blanks before the 1 and 6? T\$ and Q\$ were originally positive numeric variables T and Q; when T and Q were converted from numbers into strings, the sign position was transferred along with the number.



Therefore, when T\$ and Q\$ are concatenated, the new string W\$ contains a first-digit blank and an embedded blank before the first digit of Q\$.



How do we get rid of the embedded blanks? The solution to this problem is much simpler than it may at first appear. The method is not to attack the contents of W\$ and try to get rid of the blanks there, but to go back to where the blanks first pose a problem; when they are in the separate strings T\$ and Q\$. Look again at the contents of T\$ and Q\$:



The only values we want to appear in W\$ are the numbers to the right of the sign value in both T\$ and Q\$. Here again the string functions will aid us. The LEFT\$, MID\$, and RIGHT\$ functions allow the programmer to select any character or group of characters from within a given string. For our problem, we want just the characters to the right of the first character, the first character being the sign value (either blank or "-"). For instance, T\$=RIGHT\$(T\$,LEN(T\$)-1) would result in:

Before: After:
 T\$ 12345 → T\$ 12345

Since the first digit desired is in the second position of the string, by telling the PET to use only the values starting in position #2 and moving to the right, it will drop the sign character. Therefore, to save programming steps and memory space, we can program the PET to concatenate T\$ and Q\$ and drop the leading blank all in one statement:

$$W\$ = \underbrace{\text{RIGHT}\$(T\$, \text{LEN}(T\$) - 1)}_{\substack{\text{Drop leading blank} \\ \text{of } T\$}} + \underbrace{\text{RIGHT}\$(Q\$, \text{LEN}(Q\$) - 1)}_{\substack{\text{Drop leading blank} \\ \text{of } Q\$}}$$

↑
Concatenate
T\$ and Q\$

Our example program, amended to eliminate the sign digits, appears as follows:

```

10 T=12345
   T =  12345
20 Q=67890
   Q =  67890
30 T$=STR$(T)
   T$ =  12345
40 Q$=STR$(Q)
   Q$ =  67890
50 W$=RIGHT$(T$, LEN(T$)-1)+RIGHT$(LEN(Q$)-1)
   W$ = RIGHT$(T$,6-1)           +RIGHT$(Q$,6-1)
   W$ = RIGHT$(T$,5)           +RIGHT$(Q$,5)
   W$ = T$ 12345           +Q$ 67890
   W$ = 1234567890
60 PRINT "W$=";W$
RUN
W$=1234567890

```

In the example above, note that line 50 does not check for negative numbers. If both numbers are negative, then the leading character of T\$ should not be dropped; this allows the negative sign to appear in front of the entire number W\$. If the two strings have different signs, they should not be concatenated. When concatenating two strings with different signs, the following answers result:

$$\begin{array}{l}
 \text{T\$ } \boxed{\cancel{0}12345} + \text{Q\$ } \boxed{-67890} = \text{W\$ } \boxed{\cancel{0}12345\cancel{-}67890} \quad \text{Incorrect} \\
 \text{T\$ } \boxed{-12345} + \text{Q\$ } \boxed{\cancel{0}67890} = \text{W\$ } \boxed{-1234\cancel{5}67890} \quad \text{Incorrect}
 \end{array}$$

In summary, any type of string can be concatenated. The only string type that must be treated differently is a numeric string converted from a non-string source. When the extra space for the sign of numeric strings is taken into consideration, then string concatenation becomes an easy, useful tool.

PROGRAMMED CURSOR MOVEMENT

In Chapter 2 we discussed the screen editing capabilities of six PET cursor control keys: CLEAR SCREEN/HOME, CURSOR UP/DOWN, CURSOR LEFT/RIGHT, INSERT/DELETE, RETURN, and REVERSE.



Although these six cursor keys function in immediate mode, it is also possible to program cursor movement. This allows you to move output on and around the screen during program execution, when manual cursor movement via the keyboard is inhibited. By programming cursor movement you can format output without using programmed spaces and end-of-line checks.

You can program cursor movement using 1) the PRINT statement, and 2) the CHR\$ function. This section will concentrate on programming cursor movement with the PRINT statement. Cursor movement using the CHR\$ function will be discussed in the following section, "Programming Characters in ASCII."

THE PRINT STATEMENT

Only four of the six cursor keys are programmable using the PRINT statement: CLEAR SCREEN/HOME, CURSOR UP/DOWN, CURSOR LEFT/RIGHT, and REVERSE. **The INSERT/DELETE key and the RETURN key are not programmable with the PRINT statement.**

The programming rules to print any character apply also to cursor movement keys. The first step is to type PRINT followed by a set of quotation marks:

PRINT"

Next, enter cursor movements, along with any other characters, ending with another set of quotation marks. **When a cursor key is pressed following an odd number of quotation marks, the cursor movement is programmed and a symbolic representation (see Table 3-1) of the cursor movement is printed. The PET remains in this program mode until it encounters an even number of quotation marks; then it returns to immediate mode.**

```
100 PRINT"***"
```

When the PRINT statement is subsequently executed, the PET interprets the cursor representations and moves the cursor accordingly.

RUN

* *

To practice simple programmed cursor movement, type in the following program:

```
10 PRINT"<CLEAR SCREEN>";
20 PRINT"<CURSOR>*<CURSOR>*<CURSOR>*<REVERSE><CURSOR>*<CURSOR>*<CURSOR>*"
30 PRINT"<CURSOR><CURSOR><CURSOR><CURSOR>";
```

The program should look like this on your screen:

```
10 PRINT" ";
20 PRINT" * * * * *";
30 PRINT" * * * * *";
40 END
```

Upon execution, the output should appear as follows:

```

          33
        * 33
        * 33
        *
```

This may or may not have been what you expected. If you expected the character sequence:

```
" * * * * *"
```

to print the asterisks in a vertical line:

```

*
*
*
```

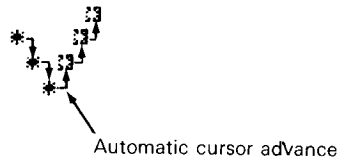
or if you expected the character sequence:

```
" * * * * *"
```

to print the asterisks back up over the original three:

```
  3  
  3  
  3
```

you forgot about the **automatic right movement of the PET cursor**. Although the programmed cursor control informs the PET to move the cursor directly up or directly down, the asterisks will print in a diagonal line due to the cursor's automatic advance. Each time a character is printed, the cursor is automatically advanced one space to the right to prevent overwriting the next character. The following diagram shows the cursor movement of the previous program:



To print a vertical line, you must compensate for the automatic advance by moving the cursor back one space to the left before moving it up or down one space. For example, the following program statement prints a vertical line of three asterisks down and then a vertical line of three asterisks back up next to the original three.

```
20 PRINT"******";
```

If you attempt to program the INSERT/DELETE and the RETURN keys, you will encounter some surprising results.

The INSERT key is not programmable, although it appears to be. If you try to program the INSERT key, a " " will appear between the two sets of quotes. But when the statement is executed the PET simply ignores the INSERT character.

The DELETE and RETURN keys remain in immediate mode. Trying to program the DELETE character in a PRINT statement will merely erase the previous character; the RETURN character in a PRINT statement will immediately move the cursor out of the statement and to the next line.

A word of caution: if you wish to move the cursor with cursor control keys while entering characters between quotes in a PRINT statement, you must be sure to first get into immediate mode. If you remain in program mode, the cursor editing movements will be included as part of the program statement. To get out of program mode, type a second set of quotation marks or press RETURN. Either method will return the PET to immediate mode, allowing you to go back and make the necessary corrections.

THE CHR\$ FUNCTION: PROGRAMMING CHARACTERS IN ASCII

Certain PET characters cannot be programmed by enclosure within quotation marks. It is possible to program these select characters using their ASCII values. ASCII is the code name for "American Standard Code for Information Interchange." ASCII assigns a unique number in the range of 0 to 255 to each alphabetic and numeric character, and to various symbolic characters. Table 6-4 in Chapter 6 lists the standard ASCII codes. Appendix A shows the PET ASCII codes. Notice that the PET ASCII chart assigns numbers to cursor controls and graphic characters peculiar to the PET.

The CHR\$ function translates an ASCII code number into its equivalent character. The format of the CHR\$ function is:

```
CHR$( )  
  ↑  
  |  
  | ASCII number from 0 to 255 of  
  | desired character or control
```

To obtain the correct ASCII code for the desired character, refer to Appendix A. Scan the columns until you find the needed character or cursor control, then note the corresponding ASCII code number. Insert its ASCII code between the two parentheses of the function. For example, to create the symbol "\$" from its ASCII code number, first scan the PET ASCII chart for "\$". "\$" has two ASCII values: 36 and 100. Which value should you use? Actually either number works just as well. But for good programming technique, once you select one number over the other, use that number consistently throughout the program. We will use number 36 and insert it into the function as follows:

```
CHR$(36)
```

Try printing this character in calculator mode on your PET.

```
?CHR$(36)  
$
```

Now, try printing ASCII code 100:

```
?CHR$(100)  
$
```

The result is the same. Experiment in calculator mode using any ASCII code from 0 to 255.

You can use the CHR\$ function in a programmed PRINT statement as follows:

```
READY.  
10 PRINT CHR$(36);CHR$(42);CHR$(166)  
RUN  
$*®
```

Using CHR\$ and ASCII values makes it possible to program or do comparison checking for cursor controls such as RETURN and INSERT/DELETE. Suppose a program must check incoming characters from the keyboard to see if the programmer depressed the RETURN key; this may be needed when using an INPUT or GET statement. You could check for a RETURN which has an ASCII code of 13, as follows:

```
10 GET X#
20 IF X#=CHR$(13) GOTO 340
30 GOTO 10
```

This test would be impossible if you tried to put a RETURN between quotation marks:

```
20 IF X#="RETURN" GOTO 340
```

↑
Impossible

This is impossible because when you depress the RETURN key following a set of quotes, it automatically moves the cursor to the next line:

```
20 IF X#="
* ←
```

As you can see, use of the CHR\$ function and ASCII numbers is extremely helpful because it allows you to program characters and controls which could not otherwise be programmed.

ARITHMETIC

With the PET, you can do addition, subtraction, multiplication, and division, using up to 9-digit numbers in either calculator mode or program mode. A 9-digit length limit can cause problems. For example, integer numbers range from 0 to 999999999; in dollars and cents this gives you a range of \$0.00 to \$9,999,999.99 (or \$999,999,999 if cents are not used). Fractional numbers may range from 0.000000001 to 99999999.9. Although these limits pose no problems in many applications, numbers will frequently exceed these limits in business and scientific applications.

Two programming methods can overcome the PET's numeric length limitations. The first method calculates the numbers as numeric strings. The second method calculates with multiple integer math, where a large number is separated into smaller segments, and each segment is handled separately.

ADDITION

This section will demonstrate how both the numeric string and multiple integer techniques add two integer numbers larger than nine digits in length. Both the augend and addend must be positive integer numbers. The **augend** is the first number in the equation, and the **addend** is the second number, to be added to the first.

Method 1: Addition via Numeric Strings

The steps involved are:

1. Input the augend and addend as two positive numeric strings.
2. Right justify the strings.
3. Add the corresponding digits of the strings separately, including carry.
4. Concatenate the answer into a one-string result.
5. Print the answer string.

Let us examine each step in turn.

Step 1: Input the augend and addend as positive numeric strings with an INPUT statement.

<u>Screen Display</u>	<u>Representation of Memory Contents</u>
10 PRINT "*****ADDITION***":PRINT	A\$ 1234567890123456
20 INPUT A\$,B\$	B\$ 57943572
RUN	
*****ADDITION***	
?1234567890123456	
??57943572	

A\$ is the augend and B\$ is the addend. Both may initially exceed the 9-digit length limit with the INPUT statement. For simplicity we will allow only positive integer numbers to be input. Once you are familiar with the basic concepts of the addition program, you may experiment and alter the program to allow for negative and fractional numbers.

Step 2: Right justify the strings. Before performing arithmetic operations, the numbers should be right-justified because, in BASIC, alphabetic and numeric strings are automatically left-justified. If the contents of numeric strings are added without first being right-justified, the answer will be incorrect, as shown below:

<u>Left Justified</u> Incorrect	<u>Right Justified</u> Correct
1234567890123456 +57943572 ----- 7028925090123456	1234567890123456 + 57943572 ----- 1234565948067028

The following statements right-justify the shorter of the numeric strings A\$ and B\$ and fill it with leading zeros until it equals the length of the longer string. X is assigned to the length of A\$, and Y to the length of B\$:

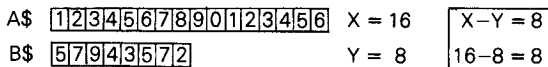
```
30 BLANK$=""
40 X=LEN(A$):Y=LEN(B$)
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$
```

BLANK\$ in line 30 is a buffer string to fill the shorter numeric string with blanks. Lines 50 and 60 use the LEN function to compare X (the length of A\$) to Y (the length of B\$), and subtract the length of the smaller string from the length of the larger string. In our example B\$ is shorter than A\$, so the length of B\$ is subtracted from the length of A\$.

```
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$
```

Length of smaller string subtracted
 from length of larger string

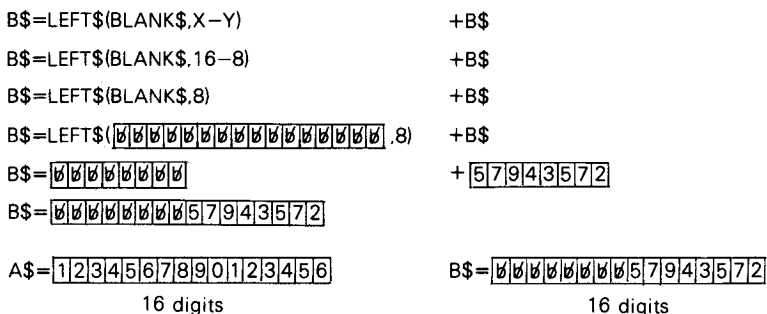
If the length of A\$ is 16 digits and the length of B\$ is 8 digits, the difference is 8 digits:



The number of blanks concatenated onto the front of B\$ is the difference of the two lengths. Since the difference is 8, eight blanks are taken from BLANK\$ to fill the shorter string. Blanks are then concatenated to the front of the shorter string B\$ with the following statement:

```
LEFT$(BLANK$(X-Y))+B$
```

The procedure is as follows:



Step 3: Add the corresponding digits of the strings. At first glance, you might assume that A\$ and B\$ can now be added using the following statement:

C\$=A\$+B\$

This is incorrect. When strings are added with a plus sign they are not added, but are concatenated:

```
C$=A$+B$
C$=1234567890123456 + 57943572
C$=123456789012345657943572
```

We want to add the digits in the strings, not concatenate the strings. **To add the contents of numeric strings, each digit must be extracted separately from the string, converted into a numeric digit, then added to one digit from the other string.** This is done with a routine using the two string functions VAL and MID\$.

```
1020 FOR I=LEN(A$) TO 1 STEP-1
1030 A=VAL(MID$(A$,I,1))
1050 B=VAL(MID$(B$,I,1))
1100 NEXT I
```

where: A = digit extracted from A\$
 B = digit extracted from B\$

“I” is a counter initialized to the length of the INPUT strings (either A\$ or B\$ may be used). With each FOR . . . NEXT loop iteration, the value of I is decremented by 1. As I decrements, it allows the string contents to be extracted one by one, right to left, using the MID\$ function:

I	MID\$(B\$,I,1)
16	5
15	7
14	9
13	4
12	3
11	5
10	7
9	9
8	4
7	3
6	5
5	7
4	9
3	4
2	3
1	5

The VAL function converts each extracted string literal into a numeric value:

When I = 16,

```
B=VAL(MID$(B$,16,1))
B=VAL($1234567890123457943572)
B=2
```

When I = 15,

```
B=VAL(MID$(B$,15,1)) . . .
```

After each digit of the numeric string is converted into a numeric constant, the necessary calculations to add them together are performed before the loop reiterates. The calculations to arrive at C\$ are done in lines 1060 to 1090.

1000 N=1	Initialize string pointer N.
1010 D=0	Initialize carry value.
1020 FOR I=LEN(A\$) TO 1 STEP -1	Initialize decrement counter I.
1030 A=VAL(MID\$(A\$, I, 1))	Extract digits separately. Convert to non-string numeric.
1040 A=A+D: D=0	Add tens value from carry (D) to A.
1050 B=VAL(MID\$(B\$, I, 1))	Extract digits separately. Convert to non-string numeric.
1060 C=A+B	Add extracted digits of A\$ and B\$.
1070 IF C>=10 THEN D=1	Carry tens value into D if C>=10.
1080 IF D=1 AND I=1 THEN N=2	
1090 C\$=RIGHT\$(STR\$(C), N)+C\$	Link sums into string answer.
1100 NEXT I	

Variable D is initialized to zero at line 1010 and used as a carry value in lines 1040, 1070, and 1080. During addition, if the value of C is greater than or equal to ten, a tens value is carried over to the next left position. The tens value carried over is stored in D:

```

  1 2 3 +14 +15 +16 +17 +18 9 0 1 2
+
  5 7 9 4 3 5 7 2
-----
 1 2 3 5 1 4 7 3 2 5 8 4
```

If C is greater than or equal to 10, the carry variable D is incremented to 1 at line 1070; otherwise it remains 0:

1070 IF C >= 10 THEN D=1

$$\begin{array}{r} A \quad \boxed{6} \\ +B \quad \boxed{9} \\ \hline C \quad \boxed{15} \end{array} \longrightarrow 15 \geq 10 \rightarrow D \boxed{1}$$

or

$$\begin{array}{r} A \quad \boxed{3} \\ +B \quad \boxed{0} \\ \hline C \quad \boxed{3} \end{array} \longrightarrow 3 < 10 \rightarrow D \boxed{0} \text{ (no change)}$$

D will be either 0 or 1, but never greater than 1 because the maximum possible sum of any two single-digit numbers is 18, thus the maximum tens value that can be carried over is 1.

To prevent losing the carry in D, line 1040 resets the value of A to A + D on the next loop iteration:

1040 A=A+D: D=0

If this statement were omitted, the carry would never be carried out, and the value of A would be incorrect. When D is added to A, D is reset to 0 in preparation for the next loop iteration.

Step 4: Link the individual sums (C) and convert the total sum into a string. Just as the augend and addend were entered as strings to avoid the 9-digit length limit, the sums must be converted back into a string to avoid the length limit again during printing. This step converts the numbers back to string form.

Line 1090 links the individual sums of C and converts the final answer back into string form.

The STR\$(C) function converts C into a one-digit string. The RIGHT\$ function extracts the rightmost N characters from STR\$(C). N is set to 1 at line 1000 to indicate that we want only the rightmost character to be extracted; the leftmost character of C is unnecessary because it is the sign value ("+" if positive and "-" if negative) and would be concatenated between each number of C\$ if we were not to exclude it.

1000 N=1

N[1]

1060 C=A+B

C[8] = A[6] + B[2]

1090 C\$=RIGHT\$(STR\$(C), N)+C\$

C\$=RIGHT\$(STR\$(C), 1)+C\$

C\$=RIGHT\$([8], 1)+C\$

C\$=[8] + C\$

Even if C is a two-digit number, only the rightmost digit is concatenated onto C\$. The tens value has already been assigned to D and will be added during the next loop iteration.

N is set to 2 to include the last carry only if D=1 and I=1 (signaling a carry on the last loop iteration). This is important, because if both conditions are true the loop will NOT iterate again to add D carry into A in line 1040, thereby losing the last carry value in D. By setting N to 2 on the last loop iteration, both digits of C are included in C\$, and the last carry over is not lost.

```

1070 IF C>=10 THEN D=1
      C[D12]>=10   D[D1]
1080 IF D=1 AND I=1 THEN N=2
      D[D1]   I[I1]   N[N2]
1090 C$=RIGHT$(STR$(C),N)+C$
      C$=RIGHT$(D12,2)+C$
      C$=I12+C$
      C$=I12XXXXXX

```

The entire FOR . . . NEXT loop routine at lines 1020 through 1100 does the following:

1. It extracts individual digits from a numeric string and assigns numeric values to them (statements 1030, 1050).
2. The digits from both strings are added together one digit at a time (statement 1060), checked for a carry value (statement 1070), and the carry is added to A in the next column (line 1040).
3. The individual sums are then linked and converted back into a numeric string for printing (line 1090).

Step 5: Print the answer string. To complete this addition routine, the input and length test commands are inserted at the beginning of the FOR . . . NEXT loop (statements 10 to 1010) and a PRINT and CLEAR string command is added to the end (statements 1110 to 1130). The final program now reads as follows:

```

10 PRINT "D***ADDITITON***":PRINT           Clear screen
20 INPUT A$,B$                               Input numeric strings
30 BLANK$=""                                  "
40 X=LEN(A$):Y=LEN(B$)
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$      } Right justify strings
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$
1000 N=1
1010 D=0
1020 FOR I=LEN(A$) TO 1 STEP-1
1030 A=VAL(MID$(A$,I,1))
1040 A=A+D:D=0
1050 B=VAL(MID$(B$,I,1))
1060 C=A+B
1070 IF C>=10 THEN D=1
1080 IF D=1 AND I=1 THEN N=2
1090 C$=RIGHT$(STR$(C),N)+C$
1100 NEXT I
1110 PRINT:PRINT"ANSWER= ";C$               Print C$
1120 C$="":PRINT:GOTO 20                   Clear C$
1130 END

```

Two sample runs of the program should give the following output:

```

***ADDITION***

?12345
??579

ANSWER=  12924

?1234567890123456
??57943572

ANSWER=  1234567948067028

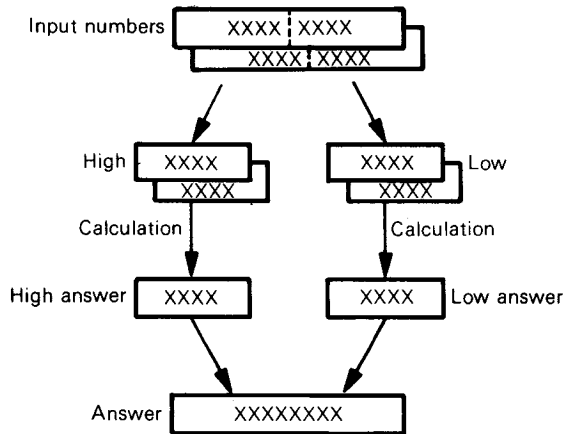
```

This addition routine overcomes the 9-digit numeric length limit. With a few alterations and additions, this routine can also be adapted to print out and round off to dollars and cents.

Method 2: Multiple Integer Addition

Another way to overcome the 9-digit length limit during addition is to use the multiple integer addition method.

Multiple integer math reorganizes a large number into smaller segments. Each segment is handled independently during the addition. The individual answers are joined together into one final answer, as demonstrated below:



The steps involved in multiple integer addition are as follows:

1. Input the augend and addend as two positive numeric strings.
2. Divide the number into two equal parts of high, low.
3. Calculate the sums of high-order digits and low-order digits.
4. Concatenate the sums into one answer string.
5. Print the answer string.

Step 1: Input the augend and addend as two positive numbers in numeric string mode:

```
10 PRINT "D***MULTIPLE INTEGER ADDITION***":PRINT
20 INPUT A$,B$
```

```
RUN
```

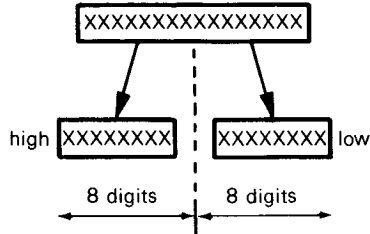
```
***MULTIPLE INTEGER ADDITION***
```

```
?1234567890123456
??57943572
```

A\$ is the augend and B\$ is the addend. The numbers are input as numeric strings because: 1) the numeric length limit is avoided and 2) it allows use of string functions to divide the numbers into small segments.

Step 2: You must establish the maximum length of numeric input in order to determine into how many segments to divide all numeric input. For instance, if the maximum length of numeric input is 16 digits, you may divide large numbers into two segments with a maximum of eight digits each. Any combination is permissible as long as no one segment exceeds a length of nine digits.

To keep our sample program simple, the maximum input length we will allow is 16 digits, to be divided into high and low segments of eight digits each.



For our sample program the divider point variable F is calculated to one-half of the larger input length.

First, determine which input string is longer. The lengths of A\$ and B\$ are assigned to variables X and Y respectively.

```
1000 X=LEN(A$):Y=LEN(B$)
```

Next, the lengths are compared. If $X > Y$ (length of A\$ is larger than length of B\$) then variable F, the divider variable, is set to one-half the length of X. But if X is smaller than Y, the program drops through and F is set to one-half the length of Y.

```
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
```

In this example, A\$="1234567890123456" and B\$="57943572"; let's run this through:

```
1000 X=LEN(A$):Y=LEN(B$)
      X = 16      Y = 8
1002 IF X>Y THEN F=X/2:GOTO 1006
      16>8 true statement, therefore
      F = 16/2
      F = 8
      program continues at line 1006
```

Once the value of F is set, the program continues at line 1006. Statement 1006 compares the value of F to the integer value of F. If F is larger than its integer value, then F is assigned the value of its integer value plus 1. This is basically a method of rounding F up to the nearest integer if F is a fractional number. For example, if the value of F equals 7.5, line 1006 would ensure that F is an integer number so that both A\$ and B\$ may be divided as easily and evenly as possible.

```
1006 IF F>INT(F) THEN F=INT(F)+1
      If 7.5>7 then F = 7+1
      F = 8
```

To obtain the high (H) and low (L) numeric values for both A\$ and B\$, use the following statements:

```

1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
1006 IF F>INT(F) THEN F=INT(F)+1
1008 F$=LEFT$(ZERO$,F)
1010 IF X<F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<F THEN BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))

```

Statements 1010 and 1040 compare the string lengths with the divider point F (8). If the string is shorter than eight, AH (or BH) is assigned a zero value, leaving only AL (or BL) equal to A\$ or B\$. If the string is longer than eight it must be divided into high and low segments; AH or BH — the high segments — are assigned the value of the leftmost LEN(X or Y) minus eight digits at 1020 and 1050.

```

1020 AH=VAL(LEFT$(A$,X-F))
      AH=VAL(LEFT$(A$,16-8))
      AH=VAL(LEFT$(1234567890123456,8))
      AH=VAL(12345678)
      AH=12345678

```

To obtain AL, the rightmost eight digits are extracted from A\$:

```

1030 AL=VAL(RIGHT$(A$,F))
      AL=VAL(RIGHT$(1234567890123456,8))
      AL=VAL(90123456)
      AL=90123456

```

The same procedure is used to extract BH and BL. Notice that **the VAL function converts the strings into numbers.**

Step 3: Once the large strings are divided into segments small enough for the PET to handle, **addition** can begin. Multiple integer addition adds corresponding groups of numbers by adding AH and BH together and AL and BL together. Whole groups of digits are added, instead of individual numbers. When a number is handled as a group of digits and not as a numeric string, the addition of each number does not have to be done digit by digit as with the numeric string method. **The PET can add numbers, whereas it is unable to add numeric strings.**

AH	12345678	AL	90123456
+BH	00000000	+BL	57943572
CH	12345678	CL	148067028

First, the low segments AL and BL are added using the following program statement:

```
1070 CL$=STR$(AL+BL)
      number
      CL$ = converted = (numeric sum of AL+BH)
      to string
```

The sum of AL and BL is converted into numeric string form when assigned to CL\$. It isn't necessary that the sum be in string form, but it is much simpler to test for carry-over using the LEN function at line 1080.

Line 1075 truncates the leading blank from the front of CL\$. Remember that when a number is converted into a string the leading blank is included. We do not want this leading blank as part of CL\$ when we concatenate the high and low segments together, therefore we truncate it with the MID\$ function.

Line 1080 tests the length of sum CL\$ against the segment length F. If the length of CL\$ is greater than F, the leftmost digit is carried over and added to the sum CH\$. (The value of D is equal to either 0 or 1.)

CH\$ is obtained by adding AH, BH, and the carry D. Statements 1070 to 1090 show the process of adding the high and low numbers to obtain CH\$+CL\$.

```
1070 CL$=STR$(AL+BL)
      CL$=STR$(090123456 + 057943572)
      CL$=STR$(0148067028)
      CL$=148067028
1075 CL$=MID$(CL$, 2, LEN(CL$)-1)
      CL$=MID$(148067028, 2, 10-1)
      CL$=MID$(148067028, 2, 9)
      CL$=148067028
1080 IF LEN(CL$)>F THEN D=1
      LEN(CL$)=9 ;F=8
      9>8→D=1
1090 CH$=STR$(AH+BH+D)
      CH$=STR$(012345678 + 000000000 + 0)
      CH$=STR$(012345679)
      CH$=012345679
1095 CH$=MID$(CH$, 2, LEN(CH$)-1)
      CH$=MID$(012345679, 2, 10-1)
      CH$=MID$(012345679, 2, 9)
      CH$=12345679
```

Step 4: The last step before printing is to **concatenate the two sums** into one answer by linking CH\$ to the front of CL\$. The preceding space and carry D are truncated from CL\$ by selecting the rightmost eight digits from that string (see the previous discussion of Numeric Strings for further explanation of this method).

```
1100 C$=CH$+RIGHT$(CL$,F)
      C$=CH$012345679 + RIGHT$(CL$0148067028,8)
      C$=012345679 + 48067028
      C$=01234567948067028
```

Step 5: Print the answer C\$.

```
1110 PRINT:PRINT"ANSWER=";C$:PRINT
```

The program is now complete. **In summary, this Multiple Integer Addition program accepts two positive integer numbers up to 16 digits long, divides them into high and low segments of eight digits each, and adds the corresponding high segments and low segments. The two sums are then concatenated into a single string answer** with a maximum length of 17 digits. This Multiple Integer Addition program allows you eight more digits than the PET's built-in maximum, yielding greater accuracy and much more flexibility.

Below is the listing of the program with a sample run. You may wish to amend this program or create your own to allow for decimal or negative numbers.

```
10 PRINT"***MULTIPLE INTEGER ADDITION***":PRINT
20 INPUT A$,B$                               Input numeric strings
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2                                     Set divider point, F
1006 IF F>INT(F) THEN F=INT(F)+1
1008 F$=LEFT$(ZERO$,F)
1010 IF X<F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<F THEN BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
1070 CL$=STR$(AL+BL)
1075 CL$=MID$(CL$,2,LEN(CL$)-1) }           Add low strings
1080 IF LEN(CL$)>F THEN D=1 }
1090 CH$=STR$(AH+BH+D)                       Add high strings
1095 CH$=MID$(CH$,2,LEN(CH$)-1) }           Add high and low answers
1100 C$=CH$+CL$
1110 PRINT:PRINT"ANSWER=";C$:PRINT           Print answer
1120 AH=0:AL=0:BH=0:BL=0:D=0:CH$="":CL$="":C$="":GOTO 20
1130 END
```

*****MULTIPLE INTEGER ADDITION*****

?1234567890123456
?57943572

ANSWER= 1234567948067028

SUBTRACTION

As with addition, there are two methods of subtraction. The first method uses numeric strings by converting the input strings to numeric constants, then back to strings for printing. The second uses multiple integer math.

Method 1: Subtraction via Numeric Strings

This subtraction program contains many sections of the "Addition via Numeric Strings" (page 190) program with a few changes. The steps involved are as follows:

1. Input the minuend and subtrahend as two positive numeric strings.
2. Right justify the strings.
3. Determine the larger numeric string.
4. Subtract corresponding digits of the strings separately with borrowed carries.
5. Concatenate the answer into a one-string result.
6. Eliminate leading zeros in the answer string.
7. Print the answer string.

At the end of this section is a complete listing of the sample program to be used as demonstration.

Step 1: The first step is to **input the minuend and subtrahend** as two positive numeric strings using an INPUT statement:

Screen Display

Memory Contents

```
10 PRINT"***SUBTRACTION***":PRINT  
20 INPUT A$,B$
```

RUN

*****SUBTRACTION*****

?123456789012
??57943572

A\$123456789012
B\$57943572

A\$ is the minuend (the first or top number entered, from which another number is subtracted). B\$ is the subtrahend (the number subtracted from the minuend).

Step 2: Align the minuend and subtrahend by right-justifying both numeric strings. This process is the same as was presented in step 2 of the "Addition via Numeric Strings" program (page 190).

```

30 BLANK$=""
40 X=LEN(A$):Y=LEN(B$)
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$

```

Step 3: For subtraction, we must **determine which numeric string has a larger value.** Although the input strings may be equal in length, their values can be quite different. For simplicity in our example, the minuend, A\$, is assigned the larger value, and the subtrahend, B\$, is a smaller value. Ideally, the minuend is always the larger value, but this cannot be guaranteed.

The values of A\$ and B\$ are compared using the VAL function in statements 65 and 70:

```

65 IF VAL(A$)=VAL(B$) THEN C$="0":GOTO 1150
70 IF VAL(A$)>VAL(B$) GOTO 1000

```

If A\$ is larger than B\$, we have a simple subtraction problem, and the program drops to line 1000. If B\$ is larger than A\$, or we are subtracting a larger number from a smaller number, the program prepares for a negative answer.

If the subtrahend is larger than the minuend (B\$ is larger than A\$), the answer will be negative. To subtract two numbers that yield a negative answer, program a small routine to switch the contents of A\$ and B\$ so that the value of A\$ is larger than B\$. Subtract B\$ from A\$, and the difference is C\$. To make C\$ negative, a negative sign, "-", is concatenated onto the front of C\$: C\$="-"+C\$.

Let us subtract 5 from 3, for example. This presents a subtraction problem where VAL(B\$)>VAL(A\$), or the subtrahend is larger than the minuend.

```

A$ 3
B$ 5
Switch A$ and B$
  A$ 5  B$ 3  →  A$ 5  B$ 3
Subtract: VAL(A$)-VAL(B$)=C$
  A$ 5 - B$ 3  →  C$ 2
Convert to negative
  C$ = "-" + C$
  "-" + C$ 2  →  C$ -2
Answer:
  C$ -2

```


The variables are switched at line 80.

```
80 X$=A$ : A$=B$ : B$=X$
```

Program Statement	Memory		
	X\$	A\$	B\$
⋮	0	3	5
X\$=A\$	3	3	5
A\$=B\$	3	5	5
B\$=X\$	3	5	3

X\$ acts as a storage string to prevent losing the contents of an entire string. Without X\$, the original contents of A\$ would be written over and the contents of B\$ would be written back into itself:

Program Statement	Memory	
	A\$	B\$
⋮	3	5
A\$=B\$	5	5 Incorrect
B\$=A\$	5	5

Later in the program we will need to know if the variables have been switched. We therefore set a marker to signal that A\$ and B\$ have been switched. Use variable S for this: S remains 0 if the variables have not been switched. If the variables are switched, set S=1. Later, at line 1140, if S=1 it signals that a negative sign must be concatenated to the front of C\$. Line 90 sets S=1 if the values of A\$ and B\$ have been switched.

```
90 S=1
```

Remember that after the strings are properly switched, a value of 1 is assigned to S to signal that the numbers have been switched and a negative answer is needed. The negative answer is obtained by concatenating a negative sign to the front of the answer before it is printed. This occurs at statement 1140, where C\$ equals the answer to the problem.

```
1140 IF S=1 THEN C$="-"+C$
```

Step 4: Whether the final answer is to be negative or positive, the value of A\$ is now larger than B\$. We can now perform simple subtraction. **The subtraction subroutine** occurs at lines 1000 to 1080. The routine is taken directly from lines 1020 to 1100, step 3 of the "Addition via Numeric Strings" program (page 192), because the digits are extracted from the strings in the same manner. However, at line 1050, the carry variable D is now used as a "borrow" variable. If $(A-B) < 0$, then a tens digit must be borrowed from the direct left column to increase the value of A by 10. D is set to -1 because a "1" is being borrowed, therefore decreasing the value of the direct left column. Once A is larger than B, B is subtracted from A, placing the result in C:

```

1000 REM**SUBTRACTION ROUTINE**
1010 FOR I=LEN(A$) TO 1 STEP-1
1020 A=VAL(MID$(A$, I, 1))
1030 A=A+D:D=0
1040 B=VAL(MID$(B$, I, 1))
1050 IF (A-B)<0 THEN D=-1:A=A+10
1060 C=A-B

```

					+10	+10	+10							+9					
A	1	2	3	3	4	5	7	8	8	9	0	1	2						
-B	0	0	0	0	5	7	9	4	3	5	7	2							
C	1	2	3	3	9	8	8	4	5	4	4	0							

Step 5: Concatenate the answer into a one-string result. This function is taken directly from line 1090 of the "Addition via Numeric Strings" (page 194) program, except that the N is replaced with a 1. Since there will never be a final carry as there is in addition, only one digit will be used in every iteration. In our subtraction program, concatenation of the individual answers into one result occurs at line 1070.

```

1070 C$=RIGHT$(STR$(C), 1)+C$

```

Step 6: In subtraction, it is possible to pick up leading zeros in the final answer. We eliminate these leading zeros before printing the answer. The FOR . . . NEXT loop in lines 1090 to 1120 **checks and eliminates all leading zeros**, using the VAL function and variable L as a counter.

```

1090 FOR I=1 TO LEN(C$)
1100 IF VAL(MID$(C$, I, 1))=0 THEN L=L+1
1110 IF VAL(LEFT$(C$, I))>0 THEN I=LEN(C$)
1120 NEXT I

```

The FOR . . . NEXT loop, which iterates from 1 to the length of the answer C\$, searches for leading zeros or blanks by extracting each digit from C\$ and comparing it to zero. It compares digits from left to right. If it identifies a zero or blank, counter variable L is incremented by 1 (statement 1100). As soon as the first non-zero or non-blank character is encountered, loop counter I is set to the length of the string so the program may drop out of the loop immediately.

Once we have determined the number of leading zeros in the answer, we separate the leading zeros from the remainder of the answer C\$. At line 1130, the RIGHT\$ function takes the LEN(C\$)-L rightmost digits and stores them in the answer variable C\$.

C\$ = 0012357			LEN(C\$) = 7
I	MID\$(C\$,I,1)		
1	0	= 0	L = 1
2	0	= 0	L = 2
3	0	< > 0	I = LEN(C\$)
7			I = 7 drop out of loop

```

1130 C$=RIGHT$(C$,LEN(C$)-L)
      C$=RIGHT$(0012357,7-2)
      C$=RIGHT$(0012357,5)
      C$=12357
  
```

Step 7: Print the answer string C\$. But before we print C\$, we check to see if the answer is to be negative by testing variable S at line 1140. If S=1, that means that originally A\$ < B\$, and the final answer is to be negative, so a negative sign is added to C\$. If S=0, the answer is positive, so nothing is added. Line 1150 prints C\$:

```

1140 IF S=1 THEN C$="-"+C$
1150 PRINT:PRINT"ANSWER=";C$:PRINT
  
```

The last lines, 1160 through 1180, clear all strings and variables to zero, or to null mode and return the program to the beginning for the next input numbers. The total program listing, complete with a RUN, is listed below.

```

10 PRINT "D***SUBTRACTION***":PRINT
20 INPUT A$,B$
30 BLANK$=""
40 X=LEN(A$):Y=LEN(B$)
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$
65 IF VAL(A$)=VAL(B$) THEN C$="0":GOTO 1150
70 IF VAL(A$)>VAL(B$) GOTO 1000
80 X$=A$:A$=B$:B$=X$
90 S=1
1000 REM**SUBTRACTION ROUTINE**
1010 FOR I=LEN(A$) TO 1 STEP-1
1020 A=VAL(MID$(A$,I,1))
1030 A=A+D:D=0
1040 B=VAL(MID$(B$,I,1))
1050 IF (A-B)<0 THEN D=-1:A=A+10
1060 C=A-B
1070 C$=RIGHT$(STR$(C),1)+C$
1080 NEXT I
1090 FOR I=1 TO LEN(C$)
1100 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
1110 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
1120 NEXT I
1130 C$=RIGHT$(C$,LEN(C$)-L)
1140 IF S=1 THEN C$="-"+C$
1150 PRINT:PRINT"ANSWER=";C$:PRINT
1160 C$=""A$=""B$=""X$=""
1165 A=0:B=0:C=0:D=0:S=0:X=0:Y=0:
1170 GOTO20
1180 END

```

Clear screen
Input numeric strings
} Right justify strings (from lines 20-60 of the addition program)
} If A\$ < B\$, switch strings
} Subtraction loop (based on lines 1020-1100 of the addition program)
} Truncate leading zeros and blanks
Print answer
} Clear strings and variables

```

***SUBTRACTION***

?123456789012
??57943572

ANSWER= 123398845440

```

Method 2: Multiple Integer Subtraction

The alternative method to numeric string subtraction is multiple integer subtraction. Recall from the previous discussion of multiple integer addition that the multiple integer method divides a large number into smaller segments, calculates the segments separately, and joins the answers into one string. This method evades the 9-digit length limit.

The total process of multiple integer subtraction is as follows:

1. Input the minuend and subtrahend as two positive numeric strings.
2. Determine which string has the larger value.
3. Divide the numbers into parts: high, low.
4. Calculate the difference of low- and high-order digits.
5. Concatenate the differences into a one-string answer.
6. Truncate leading zeros.
7. Print the answer string.

At the end of the subtraction section is the sample program used to demonstrate multiple integer subtraction.

Step 1: Input the minuend and the subtrahend as two positive numeric strings:

```
10 PRINT "*****MULTIPLE INTEGER SUBTRACTION*****":PRINT
20 INPUT A$,B$
```

```
RUN
```

```
*****MULTIPLE INTEGER SUBTRACTION*****
```

```
?123456789012
??57943572
```

A\$, the minuend, and B\$, the subtrahend, are entered as strings to avoid the 9-digit length limit.

Like multiple integer addition, A\$ and B\$ are divided into smaller segments. The maximum input length is arbitrarily set at 16 digits, so that we can divide the largest possible string into equal segments of eight digits each.

Step 2: Determine which input string has the larger value. If A\$ is equal to B\$ then the program drops down to line 1190 to print a zero answer. If B\$ is larger than A\$ the difference is negative and extra steps are needed.

If the answer is to be negative, the contents of the two strings are switched to put the larger value in A\$ and the smaller value in B\$. They are then subtracted, and a negative sign ("–") is concatenated onto the front of the difference (C\$) as was demonstrated in line 70 of "Numeric String Subtraction" (page 204). Line 30 is used here to direct the program past the switching routine if switching is not needed.

```
30 IF VAL(A$)>VAL(B$) THEN 1000
40 X#=A$:A#=B$:B#=X$
50 S=1
```

If the value of B\$ is larger than the value of A\$, the contents of A\$ and B\$ are switched at lines 40 to 50 before B\$ is subtracted from A\$, to ensure that the smaller number is subtracted from the larger one. A marker is set to indicate that the variables have been switched.

For a detailed explanation of this routine, refer to step 3 of "Numeric String Subtraction" (page 203).

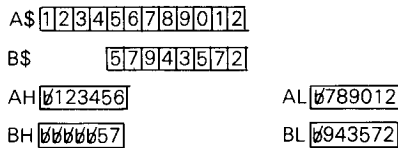
Step 3: Divide A\$ and B\$ into two smaller segments, high and low.

```

1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
1006 IF F>INT(F) THEN F=INT(F)+1
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))

```

Statements 1010 and 1040 compare the string lengths with the divider point F. F is determined at lines 1002 and 1006. These lines are identical to lines 1002 and 1006 of the "Multiple Integer Addition" program (page 198). If the string is shorter than F, AH (or BH) is assigned a zero value, leaving AL (or BL) with the entire string as its value. If the string is longer than F it must be divided into high and low segments. AH is assigned the leftmost LEN(AH) minus F digits.



Lines 1000 through 1060 are also similar to lines 1000 through 1060 of the "Multiple Integer Addition" program, which divides A\$ and B\$ into AH, AL, BH, and BL. Refer to step 2 of "Multiple Integer Addition" (page 199) for review and further explanation.

Step 4: Calculate the differences for the high-order and low-order segments. BL is subtracted from AL, and BH is subtracted from AH:



Before the segments are subtracted, the minuend and subtrahend must be compared. If the value of BL is larger than AL the difference is negative. This creates problems because a negative CL cannot be concatenated onto CH:

$$CH \boxed{0xxxxxx} \ominus CL \boxed{-xxxxxx} = C \boxed{0xxxxxx-xxxxxx} \text{ Incorrect}$$

Therefore, **we must borrow from AH to increase the value of AL so that the difference will be positive.** Lines 1070 to 1090 borrow from AH and increase AL before BL is subtracted from AL:

```
1070 IF AL >= BL THEN 1100
1080 AL=AL+10↑F
1090 AH=AH-1
```

If AL is larger than BL we bypass 1080 and 1090 and jump directly to the subtraction. But if BL is larger than AL we must borrow a one million value from AH to increase the value of AL:

-1	→	+1000000
AH	xxxxxx	AL
-BH	xxxxxx	-BL
CH	xxxxxx	CL
	xxxxxx	xxxxxx

A ten is added to the leftmost digit of AL. The easiest way to add the ten in the correct position is to raise 10 to the Fth power.

AL=AL+10↑F

In our sample program, AL is smaller than BL, as tested in line 1070.

AL 0789012 < BL 0943572

Therefore we must borrow 1000000 (10↑F=10↑6 = 1000000) from AH to increase the value of AL:

```
1080 AL=AL+10↑F
AL=AL+10↑6
AL=AL+1000000
AL=789012 + 1000000
AL=1789012
```

After AL is increased, AH must be decremented by 1, since we borrowed from it.

```
1090 AH=AH-1
AH=0123456 - 01
AH=0123455
```

Once AH, AL, BH, and BL have been set up properly, the subtraction of the segments takes place. CL\$ is the difference between AL and BL, and CH\$ is the difference between AH and BH.

Lines 1100 through 1102 determine CL\$. Line 1000 changes the integer value of AL-BL into string form.

```
1100 CL$=STR$(INT(AL-BL))
CL$=STR$(01789012-0943572)
CL$=STR$(0845340)
CL$=0845340
```

Then, using the MID\$ function at line 1101, the leftmost character (a blank representing a positive sign value) is truncated:

```
1101 CL$=MID$(CL$,2,LEN(CL$)-1)
      CL$=MID$( 845440,2,6)
      CL$= 845440
```

At 1102, if the length of CL\$ is shorter than F, zeros from ZERO\$ are concatenated onto the front of CL\$. You will need to add an assignment statement for a string of 0s for variable ZERO\$; we have done so at line 15. In this case, the length of CL\$ is equal to F, therefore no leading zeros are needed.

```
15 ZERO$="0000000000000000"
1102 CL$=LEFT$(ZERO$,F-LEN(CL$))+CL$
      CL$=LEFT$(ZERO$,6-6)+CL$
      CL$=LEFT$(ZERO$,0)+CL$
```

Then, at line 1110, CH\$ is assigned the string integer value of AH-BH:

```
1110 CH$=STR$(INT(AH-BH))
      CH$=STR$( 123455- 57)
      CH$=STR$( 123398)
      CH$= 12339 5
```

Then, using the MID\$ function, the leftmost blank character is truncated off:

```
1111 CH$=MID$(CH$,2,LEN(CH$)-1)
      CH$=MID$( 12339 8,2,6)
      CH$=123398
```


The subtraction routine looks like this:

```
1070 IF AL>=BL GOTO 1100
      789012 >= 943572 -----> False statement
      Program continues at next line
1080 AL=AL+10↑F
      AL=789012+1000000
      AL=1789012
1090 AH=AH-1
      AH=123456-1
      AH=123455
1100 CL#=STR$(INT(AL-BL))
      CL#=STR$(1789012-943572)
      CL#=STR$(845540)
      CL#=[845540]
1101 CL#=MID$(CL#,2,LEN(CL#)-1)
      CL#=MID$(845540,2,7-1)
      CL#=MID$(845540,2,6)
      CL#=[845540]
1102 CL#=LEFT$(ZERO$,F-LEN(CL#))+CL#
      CL#=LEFT$(ZERO$,6-6)+CL#
      CL#=LEFT$(ZERO$,0)+[845540]
      CL#=[845540]
1110 CH#=STR$(INT(AH-BH))
      CH#=STR$(123455-57)
      CH#=STR$(123398)
      CH#=[123398]
1111 CH#=MID$(CH#,2,LEN(CH#)-1)
      CH#=MID$(123398,2,7-1)
      CH#=MID$(123398,2,6)
      CH#=[123398]
```

Step 5: Concatenate the answer strings, CH\$ and CL\$, together by numeric string concatenation. They are concatenated in statement 1120:

```
1120 C$=CH$+CL$
      C$=CH$[ ]+CL$[ ]
      C$=[ ]
```

Only the rightmost "F" numbers from CL\$ are concatenated to CH\$ to avoid concatenating any leading blanks in CL\$ (see the earlier "Numeric Strings" section for further discussion).

Step 6: Truncate leading zeros in C\$ before C\$ is printed. Leading zeros are subtracted in the same way for Multiple Integer Subtraction as for Numeric String Subtraction (see step 5, "Numeric String Subtraction," page 205). Lines 1130 through 1170 truncate leading zeros just prior to printing C\$:

```
1130 FOR I=1 TO LEN(C$)
1140 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
1150 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
1160 NEXT I
1170 C$=RIGHT$(C$,LEN(C$)-L)
1180 IF S=1 THEN C$="-"+C$
```

If A\$ and B\$ had been switched, S would have been set to 1, signaling a negative answer, and thus a negative sign would be concatenated to the front of C\$ at 1180.

Step 7: Print the answer and clear out variable strings before allowing another problem to be input:

```
1190 PRINT:PRINT"ANSWER=";C$:PRINT
1200 A$="":B$="":C$="":CH$="":CL$=""
1205 AH=0:AL=0:BH=0:BL=0:F=0:S=0:X=0:Y=0
1210 GOTO 20
1220 END
```

The finished program appears as follows:

```
10 PRINT"***MULTIPLE INTEGER SUBTRACTION***":PRINT
15 ZERO$="0000000000000000"
20 INPUT A$,B$
25 IF VAL(A$)=VAL(B$) THEN C$="0":GOTO 1190
30 IF VAL(A$)>VAL(B$) GOTO 1000
40 X$=A$:A$=B$:B$=X$
50 S=1
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
1006 IF F>INT(F) THEN F=INT(F)+1
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN B=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
1070 IF AL>=BL GOTO 1100
1080 AL=AL+10#F
1090 AH=AH-1
1100 CL$=STR$(INT(AL-BL))
1101 CL$=MID$(CL$,2,LEN(CL$)-1)
1102 CL$=LEFT$(ZERO$,F-LEN(CL$))+CL$
1110 CH$=STR$(INT(AH-BH))
1111 CH$=MID$(CH$,2,LEN(CH$)-1)
1120 C$=CH$+CL$
1130 FOR I=1 TO LEN(C$)
1140 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
1150 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
1160 NEXT I
1170 C$=RIGHT$(C$,LEN(C$)-L)
1180 IF S=1 THEN C$="-"+C$
1190 PRINT:PRINT"ANSWER= ";C$:PRINT
1200 A$="":B$="":C$="":CH$="":CL$=""
1205 AH=0:AL=0:BH=0:BL=0:F=0:S=0:X=0:Y=0
1210 GOTO 20
1220 END
```

*****MULTIPLE INTEGER SUBTRACTION*****

?123456789012
??57943572

ANSWER= 123398845440

?1234567890123456
??57943572

ANSWER= 1234567832179884

?9999999999999999
??1234567890

ANSWER= 9999998765432109

You now know two methods of subtraction. The first method used numeric strings. The second uses multiple integer math. By comparing their outputs, you can see that both methods work equally well at getting around the 9-digit length limit. You, the programmer, may now choose the method which most specifically meets your needs.

MULTIPLICATION

The PET's 9-digit length limit may be easily exceeded because a product may be very large even when the multiplier and multiplicand are small. This numeric length limit prohibits products longer than nine digits from being printed without exponential notation. You can get around this limitation by writing a program that print products larger than nine digits of precision. Printing products exceeding nine digits without exponential notation is most easily done using multiple integer multiplication. **The following program and discussion will enable you to print out products up to 16 digits in length without exponential notation.**

Multiple Integer Multiplication

Using virtually the same steps as multiple integer addition and subtraction, multiple integer multiplication separates the multiplicand and multiplier into smaller segments, multiplies all segments, and adds the multiple products together into one final product, ranging from 1 to 16 digits in length.

The steps for multiple integer multiplication are as follows:

1. Input the multiplicand and the multiplier as two positive numeric strings.
2. Divide the strings into segments: high and low.
3. Multiply the corresponding segments.
4. Add the segment products to create one product string. Truncate any leading zeros.
5. Print the product string.

The listing of the sample program is shown at the end of this section.

Step 1: Input the multiplicand and the multiplier as two positive numeric strings, where A\$ is the multiplicand and B\$ is the multiplier. As with the other math programs, the numbers are input as strings to avoid the 9-digit length limit. However, the input numbers are limited somewhat by the program supplied here.

This program limits the length of the product to 16 digits. Since the maximum product length equals the sum of the lengths of the multiplicand and multiplier, the sum of the lengths of the input numbers cannot exceed 16. To change the program to accept larger numbers requires several alterations in the program which will not be discussed, but which you should be able to do yourself.

$$(\text{length of A\$}) + (\text{length of B\$}) \leq 16$$

Examples: $12 + 4 \leq 16$
 $2 + 3 \leq 16$
 $8 + 8 \leq 16$

The example program will multiply two input numbers with equal lengths of eight digits: 99999999 and 99999999, to give us a 16-digit product.

$$\begin{array}{r} 99999999 \text{---} \\ \times 99999999 \text{---} \\ \hline 9999999800000001 \text{---} \end{array} \quad \begin{array}{l} 8 \text{ digits} \\ + 8 \text{ digits} \\ 16 \text{ digits} \end{array}$$

Input the multiplier and multiplicand as two positive numeric strings, A\$ and B\$:

```
10 PRINT "*****MULTIPLE INTEGER MULTIPLICATION*****":PRINT
20 INPUT A$,B$
```

RUN

```
*****MULTIPLE INTEGER MULTIPLICATION*****
```

```
?99999999
??99999999
```

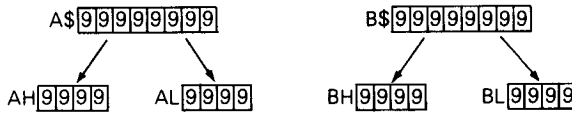
Step 2: Separate both input strings into two segments: high (H) for the leftmost digits and low (L) for the rightmost digits. The dividing point, variable F, tells the PET where to divide A\$ and B\$ into segments. The value of F is set at lines 1002 and 1006 (for explanation refer to "Multiple Integer Addition," page 198).

```
1000 X=LEN(A$):Y=LEN(B$)
      X=8      Y=8
1002 IF X>Y THEN F=X/2:GOTO 1008
1004 F=Y/2
      F=8/2
      F=4
1006 IF F>INT(F) THEN F=INT(F)+1
```

Once F is set, the program divides the numbers into high and low segments. This routine was presented in the "Multiple Integer Addition" program (see page 199). Lines 1010 through 1060 divide the two strings into high and low segments.

```
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
```

The routine above divides A\$ into AH and AL (4 digits) and B\$ into BH and BL (4 digits):



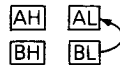
Step 3: Multiply AH, AL, BH, and BL into four product strings: P1\$, P2\$, P3\$, and P4\$. The rules of algebraic multiplication multiply each variable as if it were a single number. A\$ and B\$ are multiplied as follows:

$$\begin{array}{r} \boxed{AH} \quad \boxed{AL} \\ \times \boxed{BH} \quad \boxed{BL} \\ \hline \end{array}$$

Think of A\$ and B\$ as two sets of 4-digit numbers (H and L) joined in the middle, and not as eight individual digits: A\$ is not eight 9s, but two sets of four 9s each. Thus AL and BL are multiplied as:

$$\begin{array}{r} AL \quad \boxed{9999} \\ \times BL \quad \boxed{9999} \\ \hline \end{array}$$

Multiplying A\$ and B\$ is a four-step process. To begin, multiply BL × AL:



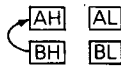
and then BL × AH:



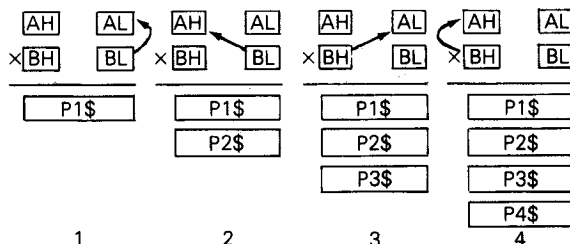
Next, move over to BH and multiply BH by AL:



and finally BH × AH:



When the PET multiplies BL × AL, etc., it internally multiplies the values of BL and AL. Consequently, the 4-digit values of BL and AL are multiplied together, producing the 8-digit product P1. Below is the four-step process:



Let's look step by step at how the multiplication works, using the values of AH, AL, BH, and BL from our example:

AH	9999	AL	9999
BH	9999	BL	9999

The first multiplication is BL × AL;

AH	9999	↗
×	BH	9999
<div style="text-align: right; margin-right: 10px;">89991</div> <div style="text-align: right; margin-right: 10px;">89991</div> <div style="text-align: right; margin-right: 10px;">89991</div> <div style="text-align: right; margin-right: 10px;">89991</div> <div style="text-align: right; margin-right: 10px; border-top: 1px solid black;">99980001</div>		

The second multiplication is BL × AH, as shown in the diagram below:

AH	9999	↘	AL
×	BH	BL	9999
<div style="text-align: right; margin-right: 10px;">99980001</div> <div style="text-align: right; margin-right: 10px;">999800010000</div>			

Notice that P2 is not directly beneath P1, but four spaces to the left (recall the rules for lining up the products of 2-digit multiplication problems). To continue in the same manner, the third multiplication should be as follows:

AH	9999	↘	AL	9999
BH	9999	BL	9999	9999
<div style="text-align: right; margin-right: 10px;">999800010000</div>				

The fourth and final multiplication should be as follows:

AH	9999	↘	AL	9999
BH	9999	BL	9999	9999
<div style="text-align: right; margin-right: 10px;">9998000100000000</div>				

Remember that only the values of the four segments are multiplied; this means that the actual multiplication done by AL × BH, etc. yields the same number, 99980001, for all four products. So, in the program the products are aligned by converting the products into strings and concatenating the necessary number of zeros onto the end of the strings to align them properly. This occurs in statements 1070 through 1100.

```

1070 P1$=STR$(BL*AL)
1080 P2$=STR$(BL*AH)+F$
1090 P3$=STR$(BH*AL)+F$
1100 P4$=STR$(BH*AH)+F$+F$

```

Without the zeros to hold the positions, the answers would be aligned incorrectly as:

```

P1  99980001
P2  99980001  Incorrect
P3  99980001
P4  99980001
-----

```

instead of:

```

          99980001
    99980001 0000
    99980001 0000  Correct
    99980001 00000000

```

The number of zeros to be concatenated onto the end of the product strings is assigned to F\$. The constant variable "F" is set to the dividing number, in this example F = 4. F\$ is assigned the leftmost "F" zeros in ZERO\$.

```

40 ZERO$="0000000000000000"
1008 F$=LEFT$(ZERO$,F)
      F$=LEFT$(ZERO$,4)
      F$=00000000000000
      F$='0000'

```

When P1\$, P2\$, P3\$, and P4\$ are calculated (lines 1070 through 1100), the correct number of zeros are simultaneously concatenated to the end of the string to align the products correctly. The products are now aligned and ready to be added:

```

      AH 999999  AL 999999
× BH 999999  BL 999999
-----
          99980001  P1
          999800010000  P2
          999800010000  P3
          9998000100000000  P4
              F$  F$

```


At the end of step 3, the program looks like this:

<pre> 20 INPUT A\$, B\$ 30 IF VAL(A\$)=0 OR VAL(B\$)=0 THEN C\$="0":GOTO 1190 40 ZERO\$="0000000000000000" 1000 X=LEN(A\$):Y=LEN(B\$) 1002 IF X>Y THEN F=X/2:GOTO 1008 1004 F=Y/2 1006 IF F>INT(F) THEN F=INT(F)+1 1008 F\$=LEFT\$(ZERO\$,F) 1010 IF X<=F THEN AH=0:AL=VAL(A\$):GOTO 1040 1020 AH=VAL(LEFT\$(A\$,X-F)) 1030 AL=VAL(RIGHT\$(A\$,F)) 1040 IF Y<=F THEN BH=0:BL=VAL(B\$):GOTO 1070 1050 BH=VAL(LEFT\$(B\$,Y-F)) 1060 BL=VAL(RIGHT\$(B\$,F)) 1070 P1\$=STR\$(BL*AL) 1080 P2\$=STR\$(BL*AH)+F\$ 1090 P3\$=STR\$(BH*AL)+F\$ 1100 P4\$=STR\$(BH*AH)+F\$+F\$ </pre>	<div style="display: flex; align-items: center;"> <div style="font-size: 3em; margin-right: 10px;">}</div> <div> <p>Input values for A\$, B\$</p> <p>If multiplicand or multiplier = 0 then answer (C\$) = 0</p> <p>Set divider point, F</p> <p>Divide A\$ and B\$ into parts: high and low</p> <p>Multiply A\$ and B\$ and align products</p> </div> </div>
--	--

Step 4: Add the four products together. This is the most complicated part of the "Multiple Integer Multiplication" program because parameters are passed back and forth from the main program to an addition subroutine. We will use a portion of the "Addition via Numeric Strings" program as a subroutine to add the products together (page 190). Below is the portion of the addition program we will convert into a subroutine:

```

2000 REM**ADD PRODUCTS**
2010 BLANK$=" "
2020 X=LEN(A$):Y=LEN(B$)
2030 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
2040 IF X>Y THEN B$=LEFT$(BLANK$,X-Y)+B$
2050 D=0:N=1:C$=""
2060 FOR I=LEN(A$) TO 1 STEP-1
2070 A=VAL(MID$(A$,I,1))
2080 B=VAL(MID$(B$,I,1))
2090 C=A+B
2100 C=C+D
2110 IF C>=10 THEN D=1
2120 IF D=1 AND I=1 THEN N=2
2130 C$=RIGHT$(STR$(C),N)+C$
2140 NEXT I

```

At line 1110 the contents of P1\$ and P2\$ are passed to the parameters A\$ and B\$, which are used in the addition subroutine (lines 2000 to 2140).

```

1110 A$=P1$:B$=P2$
      A$ 99980001
      B$ 999800010000

```

(Notice that the contents of A\$ and B\$ are not the same as those input at line 20. The same variable names are used to allow program compatibility between all four math programs.) Only two parameters are passed at a time because the addition subroutine adds only two numbers at a time. After P1\$ and P2\$ are added, the addition subroutine will be called again to add the contents of P3\$ and P4\$.

Within the subroutine, the variable C will be assigned the sum of A\$ + B\$ and converted into C\$.

Once the values for P1\$ and P2\$ are passed to A\$ and B\$ the addition subroutine is called:

```
1120 GOSUB 2000
```

A\$ and B\$ are right-justified and equated in length for addition by adding blanks from BLANK\$ to the shorter string (if there is one) in lines 2010-2040:

```
2010 BLANK$=""
2020 X=LEN(A$):Y=LEN(B$)
2030 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
2040 IF X>Y THEN B$=LEFT$(BLANK$,X-Y)+B$
```

Statements 2050 to 2140 add the corresponding digits of A\$ and B\$ and convert the sum C into the numeric string C\$. (A full explanation of this process is offered in the "Addition via Numeric Strings" section, page 191.)

```
2050 D=0:N=1:C$=""
2060 FOR I=LEN(A$) TO 1 STEP-1
2070 A=VAL(MID$(A$,I,1))
2080 A=A+D:D=0
2090 B=VAL(MID$(B$,I,1))
2100 C=A+B
2110 IF C>=10 THEN D=1
2120 IF D=1 AND I=1 THEN N=2
2130 C$=RIGHT$(STR$(C),N)+C$
2140 NEXT I
```

The sum, C\$, is passed through a FOR . . . NEXT loop to truncate any leading blanks or zeros at lines 3000 to 3060. This truncation routine is from "Subtraction via Numeric Strings" (page 205).

```
3000 REM***TRUNCATE LEAD ZEROS***
3001 L=0
3010 FOR I=1 TO LEN(C$)
3020 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
3030 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
3040 NEXT I
3050 C$=RIGHT$(C$,LEN(C$)-L)
3060 RETURN
```

C\$, the sum of P1\$ and P2\$, is returned to the main program and converted to M1\$:

```
1130 M1$=C$
```

The contents of C\$ must be transferred to M1\$ because C\$ must be cleared before the addition subroutine is called again at line 1150 to add P3\$ and P4\$.

To add P3\$ and P4\$ together, the values of P3\$ and P4\$ are passed to the parameters A\$ and B\$ before calling the addition subroutine 2000:

```
1132 A$=P3$:B$=P4$:GOSUB 2000
```

```
A$  99980000000000
```

```
B$  999800001000000000
```

The addition subroutine adds the corresponding digits of P3\$ and P4\$, truncates any leading zeros, and returns sum C\$ to the main program, where C\$ is converted to M2\$:

```
1135 M2$=C$
```

At this point, M1\$ is the sum of P1\$ and P2\$, and M2\$ is the sum of P3\$ and P4\$. To get the sum of all four products, M1\$ and M2\$ must be added together:

AH	AL		
×	BH	BL	
		P1\$	} M1\$
	+	P2\$	
	+	P3\$	} + M2\$
	+	P4\$	
		C\$	

The addition subroutine is called a third time to add M1\$ and M2\$ together to get the final answer, C\$.

```
1140 A$=M1$:B$=M2$
```

```
A$  99989999900001
```

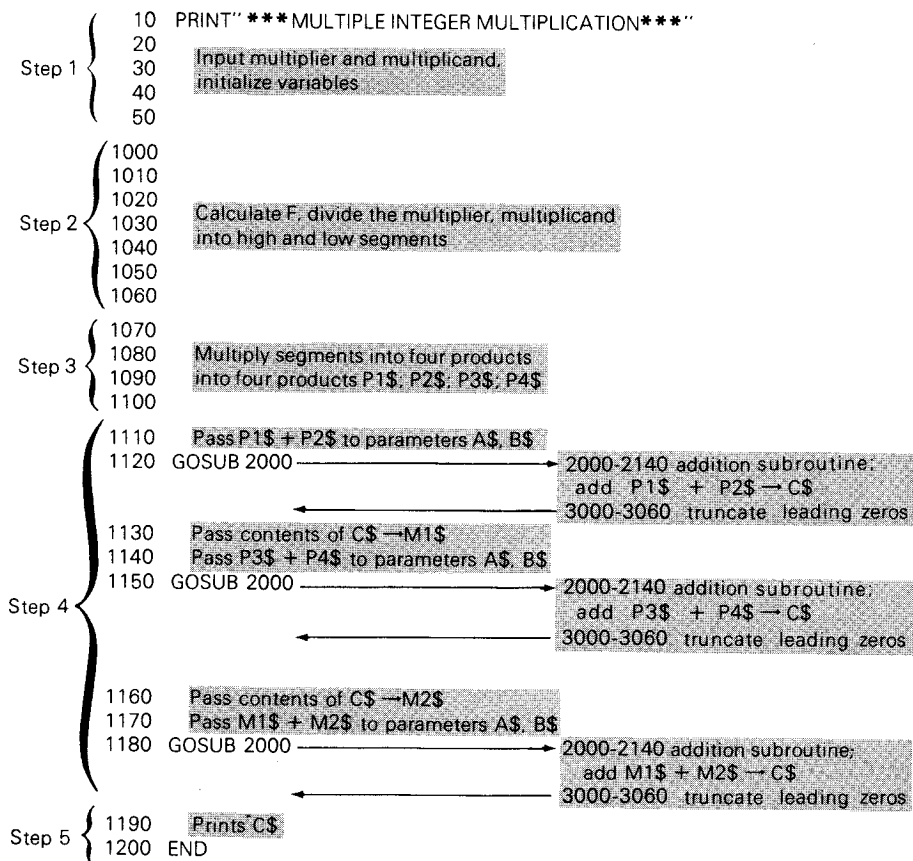
```
B$  99989999900010000
```

```
1150 GOSUB 2000
```

Step 5: After the third return from the addition subroutine, C\$ equals the sum of all four products. Step 5 **prints the answer**. Then return to the input statement to solve another multiplication problem.

```
1190 PRINT:PRINT"ANSWER=";C$:PRINT:GOTO 20
1200 END
```

The flow of the program looks like this:



Following is the listing and sample run of the program:

```
10 PRINT "*****MULTIPLE INTEGER MULTIPLICATION*****":PRINT
20 INPUT A$,B$
30 IF VAL(A$)=0 OR VAL(B$)=0 THEN C$="0":GOTO 1190
40 ZERO$="000000000000000000"
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1008
1004 F=Y/2
1006 IF F>INT(F)THEN F=INT(F)+1
1008 F$=LEFT$(ZERO$,F)
1010 IF X<F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
1070 P1$=STR$(BL*AL)
1080 P2$=STR$(BL*AH)+F$
1090 P3$=STR$(BH*AL)+F$
1100 P4$=STR$(BH*AH)+F$+F$
1110 A$=P1$:B$=P2$
1120 GOSUB 2000
1130 M1$=C$
1132 A$=P3$:B$=P4$:GOSUB 2000
1135 M2$=C$
1140 A$=M1$:B$=M2$
1150 GOSUB 2000
1190 PRINT:PRINT"ANSWER=":C$:PRINT:GOTO 20
1200 END
2000 REM***ADD PRODUCTS**
2010 BLANK$=" "
2020 X=LEN(A$):Y=LEN(B$)
2030 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
2040 IF X>Y THEN B$=LEFT$(BLANK$,X-Y)+B$
2050 D=0:N=1:C$=""
2060 FOR I=LEN(A$) TO 1 STEP-1
2070 A=VAL(MID$(A$,I,1))
2080 B=VAL(MID$(B$,I,1))
2090 C=A+B
2100 C=STR$(C)
2110 IF C>=10 THEN D=1
2120 IF D=1 AND I=1 THEN N=2
2130 C$=RIGHT$(STR$(C),N)+C$
2140 NEXT I
3000 REM***TRUNCATE LEAD ZEROS***
3001 L=0
3010 FOR I=1 TO LEN(C$)
3020 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
3030 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
3040 NEXT I
3050 C$=RIGHT$(C$,LEN(C$)-L)
3060 RETURN
```

*** MULTIPLE INTEGER MULTIPLICATION***

?99999999
??99999999

ANSWER= 9999999800000001

GRAPHICS

The PET graphic character set consists of 64 graphic symbols. Graphics are available on all PET models. On the CBM, select graphics by issuing a POKE 59468,12.

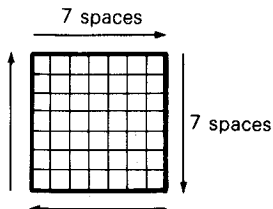
The graphic characters are all located in the upper case positions on the keys, so they must be entered in shifted mode. On the full size keyboard, the SHIFT LOCK key may be engaged to remain in "graphic mode" until you release the SHIFT LOCK.

Graphic symbols are entered onto the screen in either calculator mode or program mode.

Many graphic characters are referenced and illustrated on the following pages. Refer to Table 2-2 for easy reference to graphic character keys, names, and symbols.

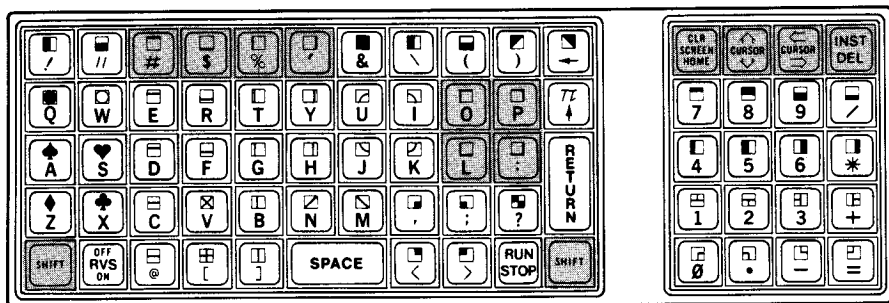
GRAPHICS IN CALCULATOR MODE

Sketching in calculator, or immediate, mode requires no line numbers, no PRINT statements, and no quotation marks. In calculator mode, the cursor may be moved freely up, down, right, or left to any spot on the screen without pressing the RETURN key after each directional change. Below is an example of a square drawn in calculator mode. Starting with the cursor in home position, the square was drawn left to right, top to bottom, right to left, and bottom to top, in one continuous movement. No line numbers, program statements, or line returns were needed.



We will use the square shown above as the basic graphic design to illustrate elementary graphics. Though simple in its design, sketching this square involves the major techniques used in graphic drawing on the PET.


Drawing a square on the PET is more complex than it may first appear. The eight graphic keys and three cursor movements which are used to draw the four thin lines and four corners of the square are shaded in the diagram below:




Draw the Square

There are nine steps to drawing the seven-space square discussed above. Let us now draw the square, step by step.

Step 1: HOME the cursor. The top left corner of the HOME position space should be the top left corner of the square (Figure 5-3a).

Step 2: Type the upper left corner of the square. This is done by using the TOP LEFT CORNER  (Figure 5-3b).

Step 3: Draw the top line of the square. Because we will use a CORNER key for the top right corner, type five TOP LINE HORIZONTAL  in this step (Figure 5-3c).




Step 4: Type the upper right corner of the square using the TOP RIGHT CORNER  (Figure 5-3d).

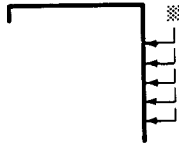
Step 5: Draw the vertical right side of the square. To allow space for the CORNER key, type five RIGHT LINE VERTICAL  .




We all know what this part of the square should look like, but does your screen look like this instead?





If so, this happened because the cursor is automatically moved one space to the right after any character is printed. To enter characters vertically, the cursor must be repositioned both vertically and horizontally to compensate for the automatic cursor movement to the right.

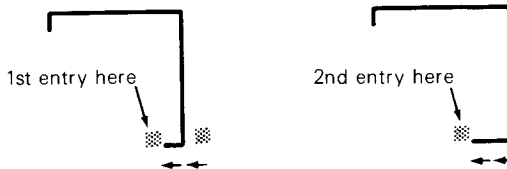
To print the vertical line of the square, then, repeat the sequence of CURSOR DOWN , CURSOR LEFT , and RIGHT LINE VERTICAL . Do this five times, and you should have printed the right side of the square (Figure 5-3e).





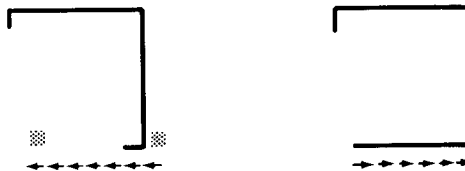
Step 6: Type the bottom right corner of the square using the BOTTOM RIGHT CORNER . Before you type this, look to see where your cursor is; if you haven't already done so, use CURSOR DOWN  and CURSOR LEFT  to position the cursor at the corner of the square; then press the corner key (Figure 5-3f).



Step 7: Draw the bottom line. Because we are using CORNER keys, we need just five BOTTOM LINE HORIZONTAL  (Figure 5-3g).




One method is to enter the line from right to left. After each character entry on the bottom line, two CURSOR LEFT  movements will be needed to correctly position the cursor for the next entry.



A second, and possibly more natural, method of drawing the bottom line is from left to right. To do this, position the cursor to the leftmost space of the bottom line (one space to the right of the left edge of the screen); this can be done using six CURSOR LEFT . You can then easily enter five BOTTOM LINE HORIZONTAL  to create the bottom line of the square.



Step 8: Type the bottom left corner. Depending on which method you used to enter the bottom line, you will need to use `CURSOR LEFT`  two times (method 1) or six times (method 2) to position the cursor at the bottom left corner, then use `BOTTOM LEFT CORNER`  to complete this step (Figure 5-3h).

Step 9: Complete the square by drawing the left vertical side. You should be able to type five `LEFT LINE VERTICAL`  to complete the square (Figure 5-3i). You will need to position the cursor before each entry, using `CURSOR LEFT`  and `CURSOR UP` .

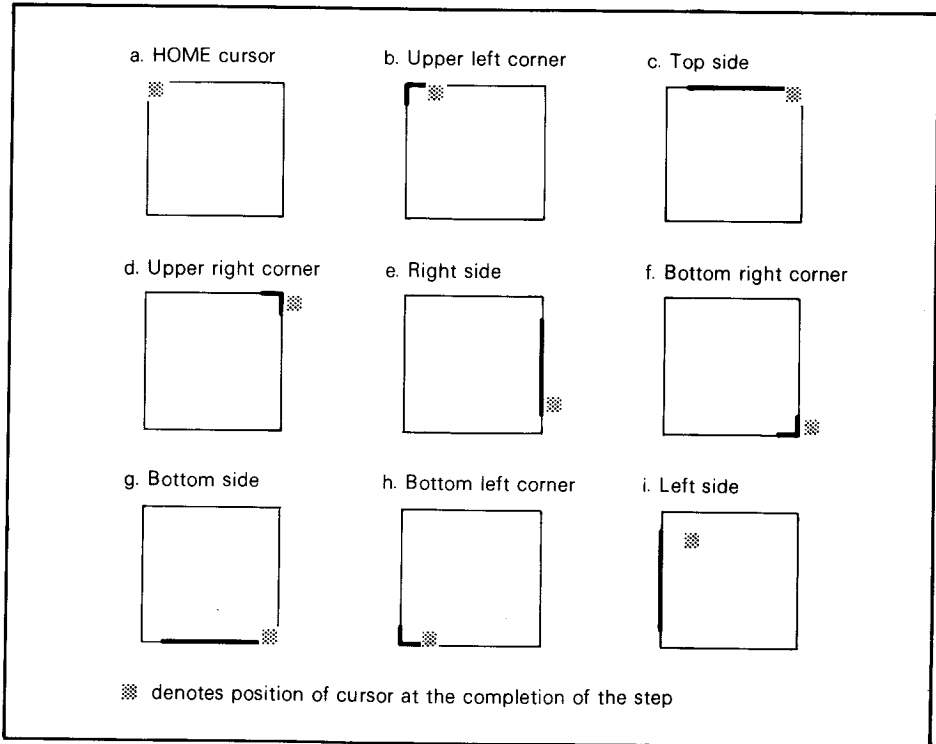


Figure 5-3. Draw the Square

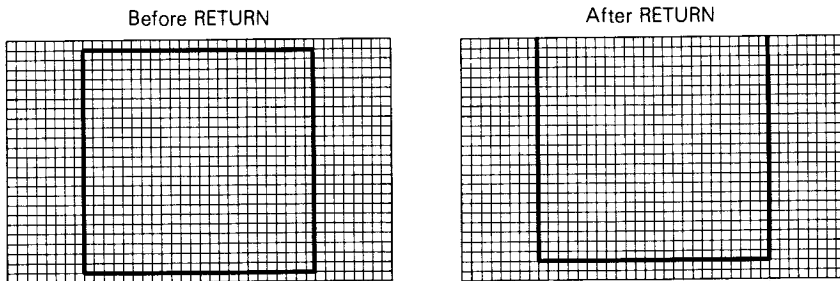
PROGRAMMING GRAPHICS

Any graphics sketched by you directly onto the screen will be instantly lost when you execute a NEW statement or turn the power off unless you first convert the graphics into a program. **You can convert any design sketched onto the screen into a program simply by making each line on the screen a string which is to be printed as part of a program.**

After you have sketched the square, move the cursor to the HOME position. Do not press CLEAR or RETURN. If you press CLEAR you will lose your picture forever. If you press RETURN, "READY." will be written through the middle of the square as shown below:



Or, if you had made your square so large that the horizontal lines of the square were printed on the top and bottom rows of the screen, and the cursor was positioned on the bottom line, a RETURN would cause the display to roll up one line in order to write the READY message on the next line, losing the top of the picture.



For this reason, pictures larger than 39 characters wide or 24 characters long should never be drawn in calculator mode.

Once the cursor is HOMEd, the next step is to move each line of the picture to the right in order to insert line numbers, question marks (shorthand for PRINT) and quotes. This converts each line from calculator to program mode so it may be saved on a tape file.

When the cursor has been HOMEd, it should be at the upper left corner of the square (Figure 5-4a). Key INSERT five times so that the top line of the square is shifted five spaces to the right (Figure 5-4b). Now there is enough room to type a line number (100), a ?, and opening string quotes (Figure 5-4c). Then press RETURN (Figure 5-4d). The top line of the square is now a programmed statement. Continue doing this for each line, incrementing each line number by one hundred until the entire square has been converted into program statements (Figure 5-4e-f).

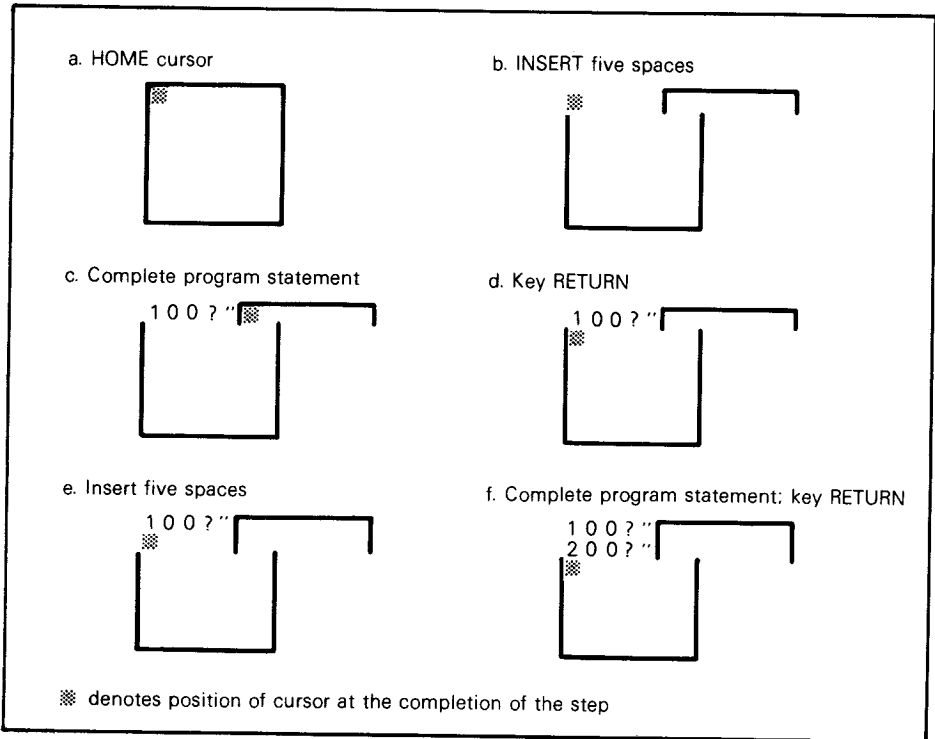


Figure 5-4. Make Program Statement from Graphics

Be sure to number the lines in sequential order to avoid distorting the picture. Also, you do not need to move the cursor past the graphics to insert a second set of quotation marks at the end of each line. After the first set of quotes is typed, merely key RETURN. Your final program listing should appear as follows:

```

100 PRINT"
200 PRINT" |
300 PRINT" |
400 PRINT" |
500 PRINT" |
600 PRINT" |
700 PRINT" |

```

Program Mode

Instead of creating graphics in calculator mode and converting it to a program in program mode, you can skip calculator mode completely. To draw the picture in program mode, each line of the picture is typed individually, with the line number, PRINT statement, and quotation marks typed in before the line of the picture.

```
100 ?"┌───┐
200 ?" │   │
300 ?" │   │
```

The space directly to the right of the quotation marks becomes column number 1 on the screen. If you do not program with this in mind, your picture may end up towards the left-hand side of the screen.

If you program a PRINT string of exactly 40 characters in length, you must include a second set of quotes and a semicolon at the end of the line. If you do not include the semicolon, an extra line will be printed in your display due to the cursor's automatic positioning to the next line after printing in column 40.

SUMMARY

Let us briefly review the drawing and programming of a graphics picture.

Graphic symbols are printed only when the keyboard is in shifted mode.

Graphic symbols are printed in either calculator or program mode. To program the picture, two methods are available:

1. Sketch the picture in calculator mode. Move the picture towards the right side of the screen. Add the line number, PRINT statement, and quotes to the left of each pictorial line to program statements.
2. In program mode, type one line at a time with the line number, PRINT statement, and quotes to the left of each picture segment.

A hint before moving to the next aspect of graphics. It is advisable to draw your picture or diagram on a piece of paper before drawing it on the screen. Map out on a piece of graph paper an area 40 squares wide by 25 squares long, using one square on paper for each space on the PET screen. Then draw your picture on the paper. Be sure to include space for the line numbers if you are going to convert the picture to program mode. Once everything is ready, type the program from the paper onto the screen.

ANIMATION

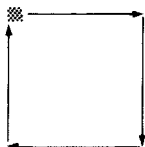
Animation on the PET is done in program mode. Any graph, number, design, word, or picture may be programmed to move sideways, up, down, or diagonally, flash on and off, print faster or slower, in almost any combination. Animation is a capability that distinguishes the PET from many other computers, and it will probably be the most fun aspect for you.

To demonstrate animation, we will begin by animating the sketching of the small square programmed in the previous section (see page 5-95 for final listing). **Instead of seeing the square appear instantaneously on the screen, animation will allow the viewer to watch each element of the square slowly appear on the screen.**

The program to animate the square looks very different from the previous program because the line segments are programmed in BASIC code and not as picture segments. There is no large square within quotation marks. The square is broken down into individual graphic characters.

TIME DELAY

The purpose of the animated program is to slowly move the cursor so we can see the lines of the square being printed clockwise from the top left corner of the screen.



The first step, as always, is to clear the screen. This also puts the cursor in the home position.

```
5 PRINT" ";
```

The second step is to type the top left corner. However, do not draw the whole top line as you did in the previous program, just the corner.

```
10 PRINT"┌";
```

In order to see each element of the square being printed, it is necessary to slow down the normal PET print speed. This can be done by using a time delay loop, similar to this statement 100:

```
100 FOR J=1 TO 10:NEXT J:RETURN
```

The FOR...NEXT loop increases the time between the printing of each character. It forces the PET to count from 1 to 10 each time the subroutine is called. While the PET is counting it cannot execute other statements or print, thus it is delayed from printing the next character and its print speed appears slower. The TO index for J can be increased or decreased to lengthen or shorten the delay. The larger the TO index for J is set, the longer the time between printing each character.

For our animation program, then, we must include this time delay loop after printing the next element. Since the programmed time delay loop remains the same for each element, we call it as a subroutine. Therefore, after printing the upper left corner of the square, call the time delay loop, subroutine 100.

PROGRAMMING CHARACTER PLACEMENT

The third phase is to print the top line of the square. Instead of programming PRINT "———" we will use a FOR...NEXT loop:

```
15 FOR I=1 TO 5:PRINT"~":GOSUB 100:NEXT I
```

Statement 15 uses a FOR...NEXT loop so that the subroutine time delay can be called between each printing of "~". To program the PET to sketch the square slowly, the time delay must be called after each character is printed. It would be useless to program:

```
15 PRINT"———" :GOSUB 100 ← incorrect
```

because the whole line would be printed instantaneously without any time delay.

To complete the top line, type the upper right corner. Again, include the time delay subroutine call.

```
20 PRINT"~":GOSUB 100
```

So far, the program looks like this:

```
5 PRINT"⌈";
10 PRINT"┌":GOSUB 100
15 FOR I=1 TO 5:PRINT"~":GOSUB 100:NEXT I
20 PRINT"~":GOSUB 100
100 FOR J=1 TO 10:NEXT J:RETURN
```

Run the program. You should see the following progression appear on the screen:

```
⌈
┌
┌~
┌~~
┌~~~
┌~~~~
┌~~~~~
┌~~~~~~
┌~~~~~~
┌~~~~~~
┌~~~~~~
*
```

Hopefully, this is what you saw. If not, go back and find out what went wrong. Did you forget the semicolons after each PRINT statement?

End all PRINT statements in this program with a semicolon; the semicolon concatenates graphic strings together when printed. This allows the "┌" and the top line "———" to be concatenated together on the same line:

```
┌———
```

Without the semicolons, the PET performs a carriage return after each statement and the top line will look like this:

```

┌
|
|
|
|
|
└
  
```

The other three sides are drawn in a sequence similar to lines 10 and 15. Line 20 begins the next sequence, then converts the linear segment to FOR...NEXT/PRINT statements, calling the time delay subroutine in that PRINT loop. Don't forget to include the cursor controls to compensate for the automatic right cursor movement inside the FOR...NEXT loops for the vertical sides of the square:

```

PRINT" ███" right side PRINT <RIGHT LINE VERT. > <CURSOR L. >
                                <CURSOR DOWN >
PRINT" _███" bottom PRINT <BOTTOM LINE HORIZ. > <CURSOR L. >
                                <CURSOR L. >
PRINT" |███" left side PRINT <LEFT LINE VERT. > <CURSOR L. >
                                <CURSOR UP >
  
```

The program listing looks like this:

```

5 PRINT"□";
10 PRINT"┌";GOSUB 100
15 FOR I=1TO 5:PRINT"─";GOSUB 100:NEXT I
20 PRINT"└";GOSUB 100
25 FOR I=1TO 5:PRINT" ███";GOSUB 100:NEXT I
30 PRINT"┐";GOSUB 100
35 FOR I=1TO 5:PRINT" _███";GOSUB 100:NEXT I
40 PRINT"┌";GOSUB 100
45 FOR I=1TO 5:PRINT" |███";GOSUB 100:NEXT I
50 END
100 FOR J=1 TO 10:NEXT J:RETURN
  
```

Now, try a trial run. Does your square look like this?

```

┌      |
READY.  |
      | |
      | |
      | |
      | |
└      ─
  
```

If this design appeared instead of a perfect square, some of the cursor controls were left out. The PET did exactly what it was programmed to do, so where is the problem? Take a closer look at the program. We included cursor controls within the FOR...NEXT loops for all four sides of the square. Now look at the screen. The problem is not with the sides. The problem, therefore, must be in the corners. Look at statements 20, 30, and 40. We forgot the cursor controls after each corner

position. Make the proper changes, and the program should look like this:

```
5 PRINT"□";
10 PRINT"Γ";:GOSUB 100
15 FOR I=1 TO 5:PRINT"┌";:GOSUB 100:NEXT I
20 PRINT"┌□";:GOSUB 100
25 FOR I=1 TO 5:PRINT"┌□□";:GOSUB 100:NEXT I
30 PRINT"┌□□□";:GOSUB 100
35 FOR I=1 TO 5:PRINT"┌□□□";:GOSUB 100:NEXT I
40 PRINT"┌□□□";:GOSUB 100
45 FOR I=1 TO 5:PRINT"┌□□□";:GOSUB 100:NEXT I
50 END
100 FOR J=1 TO 10:NEXT J:RETURN
```

Now try another trial run. Your picture should look like this:



You should have been able to watch the PET slowly sketch the square on the screen in a clockwise direction. Remember, you may change the print speed by changing the TO index value for variable J in the time delay loop.

One last problem: How to avoid destroying the square with the READY message.

When drawing the square is completed, the cursor is on line 2; when the program ends, the READY message is printed on the next line, which happens to be within the square. Therefore, before ending the program, you must compensate for this by moving the cursor below the square; the READY message will be written underneath the square and not across it. This is done by printing several CURSOR DOWNS before the END statement.

```
50 PRINT"████████":END
```

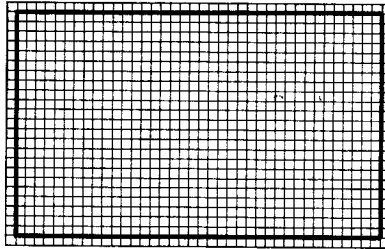
This will move the cursor down below the square and the square will not be destroyed:



```
READY.
█
```


Enlarging the Square

Let's take the small square we just animated and enlarge it so that it forms a boundary one space from the perimeter of the screen:



The PET screen is 40 spaces wide by 25 spaces long. If the rectangle's sides are to be one space from the perimeter, then the square should be 38 spaces wide by 23 spaces long:

$$40 - 2 \text{ (1 space for each side)} = 38$$

$$25 - 2 \text{ (1 space for each side)} = 23$$

With just a few changes to the animated small square program we can draw the larger rectangle to form the screen boundary. FOR...NEXT loops were used in the previous animation program to print a string of graphics for each side, FOR I=1 to 5. To enlarge the square, change the value of the TO index to 35 for the horizontal sides and 21 for the vertical sides, leaving spaces for the corners.

$$40 - 2 \text{ (1 space for each side)} - 2 \text{ (corners)} = 36$$

$$25 - 2 \text{ (1 space for each side)} - 2 \text{ (corners)} = 21$$

```

15 FOR I=1 TO 36:?"~";
25 FOR I=1 TO 21:?" 00";
35 FOR I=1 TO 36:?"_|||";
45 FOR I=1 TO 21:?"| ||";

```

← Horizontal sides
 ← Vertical sides

Make these changes in your program and try a trial RUN.

That was simple. Because you have just created a boundary around the edge of the screen, the last statement of the program to move the cursor out of the square is unwanted. Instead, delete line 50 and program the cursor to move inside of the box and print something; you do not want a boundary surrounding an empty screen. Be sure not to program the cursor to go beneath the square, because the screen will scroll up, and you will lose the top of the square. Program something inside the box, type RUN and watch it go!

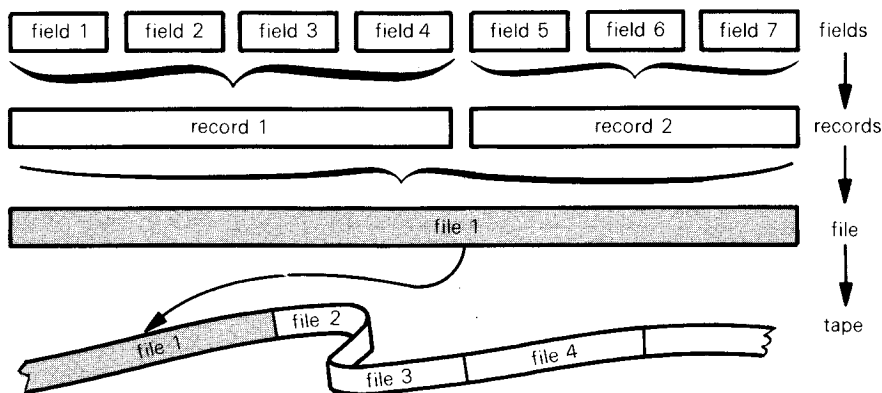
SUMMARY

Here are some important points to remember when animating:

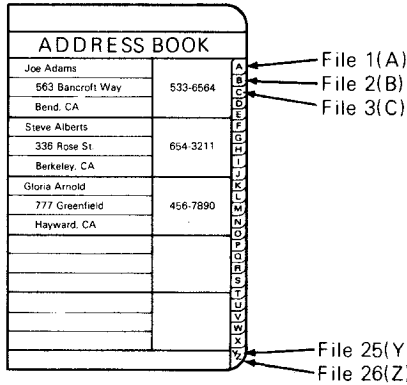
1. Map the picture on graph paper first.
2. Use time delay loops to control print speed.
3. Do not forget semicolons or cursor controls.
4. Before the end of the program, move the cursor out of the picture before the program END or STOP writes the READY message.

FILES

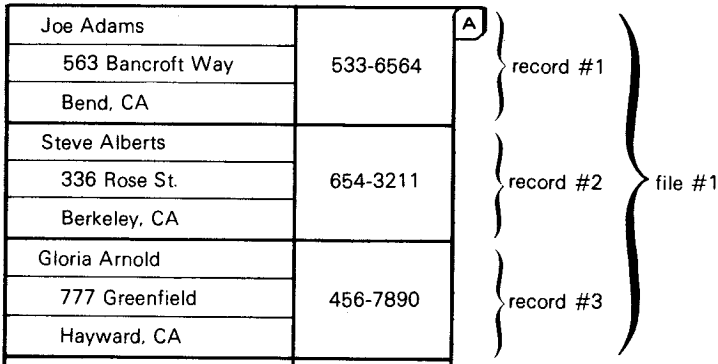
Before attempting to use files, you must have a good understanding of what a file is. A **“file”** is a collection of information. Files appear in many forms, but the ones we are interested in are the groups of information on a cassette tape. Each group of information found on a PET cassette is either a **“program”** or **“data”** file. A file is divided up into smaller segments called **“records”** and **“fields.”** A **“field”** is a unit of data. A unit of data may be of any type and any length. These units of data are grouped together into larger groups called **“records.”** Records contain at least one field, but usually contain several fields. The records are then grouped together into a file. Each file contains at least one record, which may be as long as the file itself, or several records variable in length. Files, records, and fields have no fixed length unless your program sets a maximum length. The format of a file is shown below:



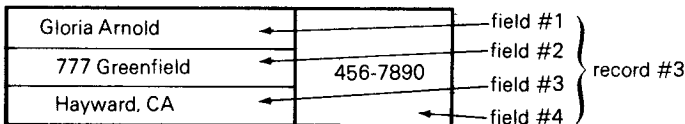
A good way to visualize what constitutes a file is to picture an address book. The address book may be a book, file cabinet, or cassette tape. The address book is then divided into 26 sections by letter: A through Z. Each section contains a collection of names and addresses of people whose last name starts with the particular letter. We will call these alphabetic sections files. Therefore, the address book is divided into 26 files, each file referenced by a letter:



Each file is further divided into records. The name, address, and telephone number of each person in the address book constitutes a record:



Each record in the address file is divided into four fields. Field #1 contains the name; field #2 the street address; field #3 the city and state; and field #4 the telephone number.



To get to a field, the file and record numbers must be accessed first. In other words, to find out in what city Gloria Arnold resides, File #1 and Record #2 must be referenced first before Field #3 may be accessed. This structuring is very important because it enables large quantities of data to be separated into smaller units, making it more organized and more easily accessible.

The PET uses two types of files: program files and data files.

PROGRAM FILES

A program is written onto the tape as an entire file. Its organization of records and fields is not determined by the programmer, but is done internally within the PET. The user need not worry about a program file's records and fields, since they are not important to the programmer. **A program file should have a unique name** because a name makes it much easier to find the program when searching the tape. The program file is named with a SAVE command; in other words, **the program name becomes the program file name.**

Since you already know how to create a program file, we will quickly review the procedure (see Chapter 3). There are three steps in writing a program file:

1. Code the program and type it into memory.
2. SAVE the program on tape.
3. VERIFY the SAVED program.

The first step is to code and then enter the program into the PET's memory. The second step uses the SAVE command:

```
SAVE"filename",device,s
```

where:

filename	is the program name enclosed in quotes.
device	(optional) is a predefined number (1-255).
s	(optional) is a secondary address code (0-5). See Table 4-2.

The third step VERIFIES the program just SAVED on tape by comparing it to memory to make sure everything is written correctly. Rewind the tape and type:

```
VERIFY"filename"
```

where:

filename	is the program filename just SAVED.
----------	-------------------------------------

DATA FILES

The other type of files are data files. **The data file**, as its name implies, **contains data to be read from a program file.** Unlike a program file, **the programmer may define the organization of a data file into records and fields.** The programmer may establish the number of fields in each record, and the number of records in each file. He/she may also set maximum lengths or allow records and fields to be variable in length. The program file may be programmed to read or write any type of data to or from the data tape, whether numeric, alphabetic, constant, or string.

The emphasis of this section is on data files — how to create and manipulate them with program files. Therefore, throughout the remainder of this chapter the term "file" refers to "data file," unless "program file" is clearly stated. Further, the discussion of data files concentrates on files stored on cassette tape.

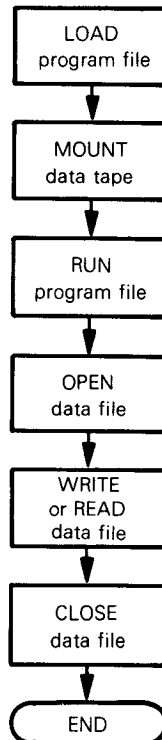
Before beginning file operations it is advisable, if you have not already installed the new ROMs (Read-Only Memory chips) in the PET, that you do so before attempting to write files. With the old ROMs it is very difficult, if not impossible, to read and write data tapes, as the tapes will not move without several extra commands to force the PET to move the tapes. File attributes of the old ROMs are discussed further in Appendix H. **The following discussion assumes that you have the new ROMs.**

FILE HANDLING

Being able to handle your files correctly is extremely important for successful programming. The programmer must know when to open or close a file, and when and how a file can be read from or written to. **Basically, there are four steps in file handling:**

1. **LOAD** the program file.
2. **OPEN** the data file.
3. **Read/write** to the data file.
4. **CLOSE** the data file.

Program LOAD is done in immediate mode. OPEN, read/write and CLOSE of the data file are programmed within the program file. The flow of the procedure appears as follows:



Step 1: LOAD the program file with the LOAD command:

```
LOAD"PROGRAM"
```

When you execute the program, it may write data to a data tape, or read data from a data tape. Therefore, once the program file is loaded, mount a tape. If the program will write a data tape, mount a blank tape. If the program will read a data tape, mount the data tape specified in Step 2. Then execute the program using the RUN command.

Step 2: OPEN the data file. The data file is opened in program mode within the program file as illustrated:

```
OPEN 1,1,2,"DATA"
```

The diagram shows the OPEN statement with four arrows pointing to its components: the first arrow points to '1' (file number), the second to the first '1' (physical device number), the third to '2' (secondary address), and the fourth to '"DATA"' (file name).

The OPEN statement above opens file number 1 on physical device number 1. (Physical device #1 is the cassette unit; refer to Table 4-1 for physical device codes.) Secondary address #2 is specified, which indicates an OPEN for WRITE with an End of Tape (EOT) mark to be set when the file is closed (refer to Table 4-2 for secondary address codes). The file name specified is DATA.

When writing a data file to a cassette tape, a secondary address code, #1 or #2, must be specified in the OPEN statement. A secondary address code #1 indicates the file to be opened to record data, but no EOT mark is set when the file is closed. Secondary address code #2 indicates that the file is opened to record data, and an EOT mark is to be written when the file is closed. When the PET encounters an EOT mark when reading a data tape, it displays END OF TAPE and reads no further; this can be used to assure that no attempt is made to read large amounts of blank tape. Therefore, a secondary address code 2 should be used when the data file is the only file on one side of a cassette, or the data file is the last file on one side of a cassette.

If no secondary address code is specified in the OPEN statement, the secondary address defaults to 0.

Step 3: Read or write to the data file. This will be discussed in the sections "Writing a Data File" and "Reading a Data File."

Step 4: After a data file has been read from or written to, it must be closed. Like OPEN, **closing a file is programmed within the program file with a CLOSE command.** Be careful to CLOSE the same logical field that was OPENed.

```
OPEN 1,1,2,"DATA"  
:  
:  
:  
CLOSE 1
```

The diagram shows the CLOSE statement with an arrow pointing from the '1' in 'CLOSE 1' up to the first '1' in the 'OPEN 1,1,2,DATA' statement, indicating that the file opened by the first '1' is being closed.

When the data file is closed, the PET writes a special mark, called an End of File (EOF) mark, on the data tape following the last data item. This mark signals that the end of the data file has been reached. Later, when the PET is reading a file, it will read no further when it encounters an EOF mark, knowing that it has reached the end of the file. An EOF mark may be searched for with the STATUS WORD command (ST), which checks the file status. ST equals 64 when an EOF mark has been found. The status word will be useful later when reading back data tapes. (For further discussion of the status word, refer to Chapter 4, "Basic System Functions.")

Mounting the Data Tape

Mount the data file cassette into the tape drive after you have loaded but before you have executed the program. If you run a program which is to read to or write from an external data file (such as a cassette) and there is no cassette to read to or write from, the PET will search forever looking for a file on a cassette that isn't even there! **When the data cassette is mounted, the tape must be positioned so that the data file can be located.** Certainly if the cassette is mounted but the tape is wound to the end, the PET will not be able to locate, read, or write any data file.

When writing to a data tape, the PET begins writing on the tape exactly where it is positioned. Therefore, be sure that there is nothing at that position on the cassette which you don't want to lose if the PET writes over it. Also, be sure there is plenty of empty space on the cassette following the initial position, so that as the PET continues to write to the tape no further information gets written over.

When reading from a tape, the PET searches forward for the data file name you have specified in your OPEN statement. Your data cassette should therefore be positioned at some point prior to the beginning of the data file you wish to read from. If the PET searches forward and never finds the data file specified in the OPEN statement, a FILE NOT FOUND ERROR message is printed on the screen.

It is a good practice to put only one data file on each side of a cassette, always beginning the file at or near the start of the tape. By doing this, you will always position your tape at the beginning of the tape for both reading and writing. Furthermore, when writing to the tape, you will never need to worry about writing over any other file.

WRITING A DATA FILE

Creating a data file is an important programming technique because it expands the PET's capabilities to allow processing of large quantities of data. This is the emphasis of this section.

To create a data file, LOAD the program file. To use an external data file, you must have three statements within the program file: an OPEN command, a read or write command, and a CLOSE command.

To write from a program to a data tape use the PRINT# command:

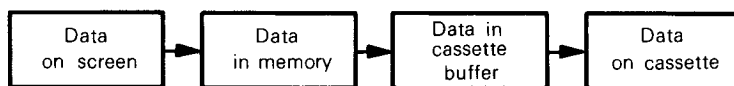
PRINT#f,data

where:

- f is the logical file number (1-255, must match the f in the OPEN and CLOSE commands).
- data is the data to be written.

PRINT# cannot be typed as PRINT # or ?#. PRINT# must be completely spelled out when referencing files.

PRINT# transfers data one character at a time to a cassette buffer before sending the data to the tape. When the cassette buffer reaches its maximum capacity of 191 bytes of data, the cassette buffer stops accepting data, and sends the data in the buffer to the data tape as a "block" of data. A block may contain a partial record, a single record, or several records of data.



Either numeric constants or strings may be transferred with the PRINT# statement. Figure 5-5 shows the flow of the PRINT# process.

Writing Numbers

Writing numbers to a data tape is similar to printing numbers on the screen. Let's write a program called NUM.PRINT#, to write the numbers 1 through 10 on a data tape, where each number is a single record.

The first action of the program is a display of the program function and load instructions:

NUM.PRINT#

```
10 PRINT":◆◆ CREATE NUMERIC DATA TAPE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE; PRESS <RETURN> WHEN READY ◆◆":PRINT
30 GET A$:IF A$="" THEN 30
```

Line 20 instructs the user to insert the data tape in the cassette unit and rewind it to the beginning of the tape. Once the data tape is ready for writing, press RETURN. Line 30 is a wait loop which waits for RETURN or any key to be pressed. If no keystroke is entered, it waits.

This wait loop is important because it doesn't let the PET do anything until you have loaded a data tape. If line 30 were deleted and there were no wait loops, the PET would give the user no time to mount and rewind the data tape. As soon as any cassette button was depressed, the PET would attempt to open the file, even if you were trying to rewind the data tape.

Once a key is pressed, A\$ ≠ "" and the program drops down to the next line to OPEN the mounted data file:

```
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN1,1,2,"NUMBERS"
```

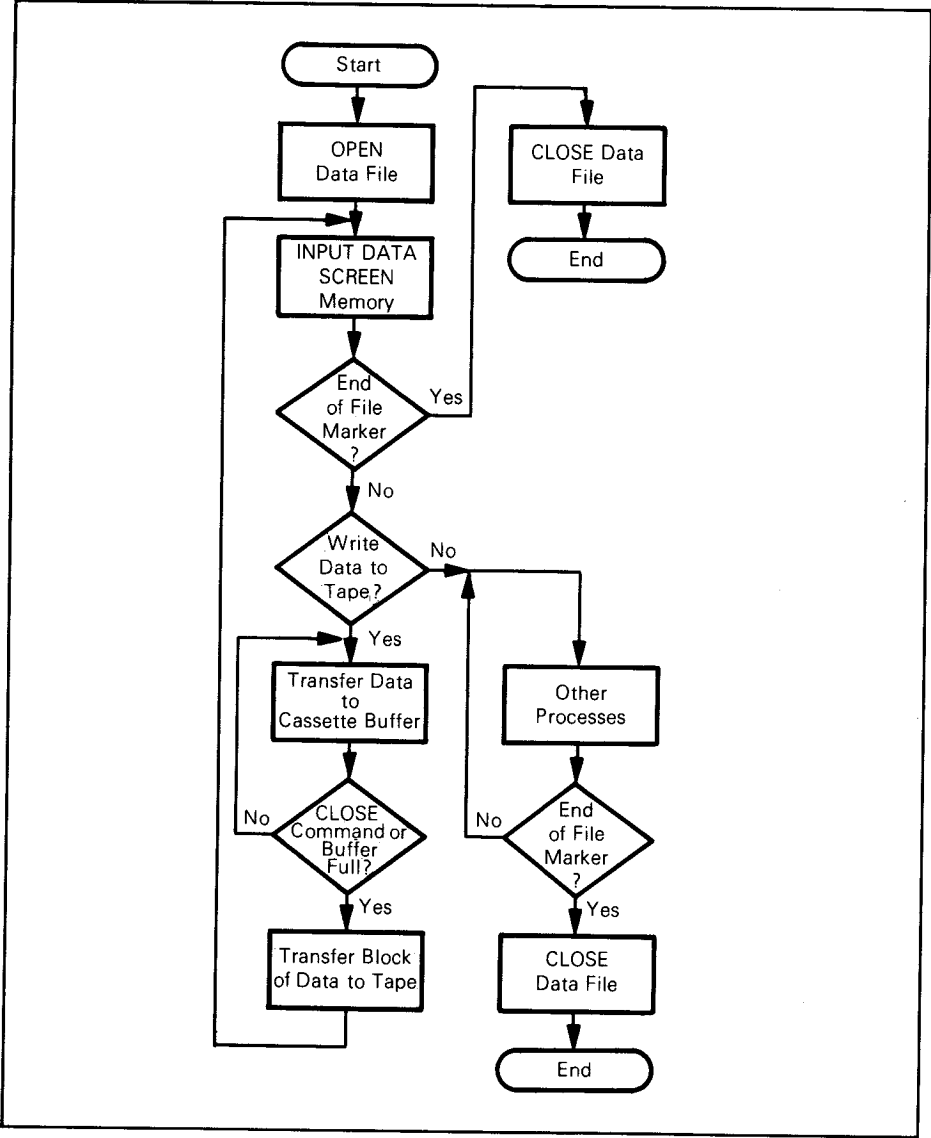



Figure 5-5. Using PRINT#

This OPEN command opens logical file #1, physical device #1 (the cassette tape unit), secondary address 2 (OPEN for write and EOT mark at close of file), assigning the new data file the name NUMBERS.

Next, we set up a FOR . . . NEXT loop to print the numbers 1 through ¹⁰20 on the screen and on the data tape:

```
50 FOR N=1 TO 10
60 PRINT N           ← Print N on screen
70 PRINT#1,N        ← Print N on data file #1 (NUMBERS)
80 NEXT N
```

Printing to the screen and to the tape require separate commands. PRINT #1,N will print only to the tape. Remember that PRINT # cannot be typed in as ?#. PRINT must be spelled out completely, with the number sign, file number, comma, and variable following respectively.

<u>Incorrect</u>	<u>Correct</u>
?#1,N	PRINT#1,N
PRINT N	
PRINT #1,N	
PRINT#1N	
PRINT1,N	

Any of the above incorrect methods will result in a syntax error, except PRINT N, which will print to the screen but not to the tape.

If everything works correctly, lines 50 through 80 write to the tape and screen:

PET screen

```
1
2
3
4
5
6
7
8
9
10
```

Representation of Data Tape



<CR> = carriage return


Because data cannot be written on separate "vertical lines" on a tape, carriage returns are automatically inserted on the tape in relation to the carriage returns on the screen.

After all data is written to the tape, the file is closed. To ensure that all the data is written to the tape you must CLOSE the file.

```
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END
```

Be sure that the logical file CLOSEd is the same one that was previously OPENed.

```
OPEN 1,1,2,"NUMBERS"  
.  
.  
.  
CLOSE 1
```



The listing of NUM.PRINT# looks like this:

```
10 PRINT"◆◆ CREATE NUMERIC DATA TAPE ◆◆":PRINT  
20 PRINT"◆◆ MOUNT TAPE; PRESS <RETURN> WHEN READY ◆◆":PRINT  
30 GET A$: IF A$="" THEN 30  
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN1,1,2,"NUMBERS"  
50 FOR N=1 TO 10  
60 PRINT N  
70 PRINT#1,N  
80 NEXT N  
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1  
100 END
```

The main statements that differentiate this program from a program that prints only to the screen are at lines 40, 70, and 90: the OPEN, PRINT#, and CLOSE commands.

Below is a RUN of the program:

```
◆◆ CREATE NUMERIC DATA TAPE ◆◆  
  
◆◆ MOUNT TAPE; PRESS <RETURN> WHEN READY◆◆  
  
◆◆ OPENING DATA FILE ◆◆  
  
PRESS PLAY & RECORD ON TAPE #1  
OK  
  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
◆◆CLOSING DATA FILE ◆◆
```

Writing Strings

The PRINT# statement also prints strings. Taking the previous program NUM.PRINT# and altering it somewhat, we can print the words "ONE" through "TEN" as strings. The new program will be called WORD.PRINT#. The words are supplied through either an INPUT or READ/DATA statement. Our sample program uses a READ/DATA statement. The READ statement is inserted in the FOR . . . NEXT loop at line 60 and a DATA statement is added to the end of the program. The final program is listed below, followed by a sample run of the program.

WORD.PRINT#

```
10 PRINT"♦♦CREATE WORD DATA FILE♦♦":PRINT
20 PRINT"♦♦MOUNT DATA TAPE; PRESS <RETURN> WHEN READY♦♦"
30 GET A$:IF A$="" THEN 30
40 PRINT"♦♦OPENING DATA FILE♦♦":OPEN1,1,2,"NUMWORD":PRINT
50 FOR N=1 TO 10
60 READ N$
70 PRINT N$
80 PRINT#1,N$
90 NEXT N
100 PRINT"♦♦CLOSING DATA FILE♦♦":CLOSE1
110 DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
120 END
```

♦♦CREATE WORD DATA FILE♦♦

♦♦MOUNT TAPE; PRESS <RETURN> WHEN READY♦♦

♦♦OPENING DATA FILE♦♦

PRESS PLAY & RECORD ON TAPE #1
OK

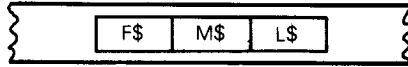
ONE
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
TEN

♦♦CLOSING DATA FILE♦♦

Writing one string per data item is simple, but writing several strings per data item is a little more difficult. To put several numbers or strings on each line, the item separators (commas or semicolons) must be "forced." These separators are otherwise automatically deleted, which results in a garbled mess when reading the file. For instance, if the following statement is executed:

```
PRINT#1,F$.M$.L$
```

it is sent to the tape with no commas separating the words.



The separators must be forced by one of the following two methods:

1. Enclose the separator within quotes: PRINT#1,F\$;"M\$";L\$
 2. Use the CHR\$() statement: PRINT#1,F\$;CHR\$(44);M\$;CHR\$(44);L\$
- Item
Item
Separator
Separator

where:

CHR\$(32) = ␣
 CHR\$(44) = ,
 CHR\$(59) = ;

This second method will force commas or any other separators to be written on the tape, thus ensuring your strings will stay separated:



The program below, called NAMES.PRINT#, forces separators to keep F\$, M\$, L\$ name strings (first, middle, last) from running together:

NAMES.PRINT#

```

10 PRINT"♦♦CREATE NAME DATA FILE♦♦":PRINT
20 PRINT"♦♦MOUNT DATA TAPE; PRESS <RETURN> WHEN READY♦♦"
30 GET A$: IF A$="" THEN 30
40 PRINT"♦♦OPENING DATA FILE♦♦":OPEN1,1,2,"NAME":PRINT
50 FOR J=1 TO 4
60 INPUT F$,M$,L$
70 PRINT F$,M$,L$
80 PRINT#1,F$;CHR$(44);M$;CHR$(44);L$
90 NEXT J
100 PRINT"♦♦CLOSING DATA FILE♦♦":CLOSE1
110 END
  
```

Sample Program

This sample program will demonstrate how to write complex records of data to a data tape, where the format of the data is specified by the programmer. The new program MAIL.PRINT# will print a mailing list named MAIL onto a data tape. MAIL will later be read by another program called MAIL.INPUT#. MAIL is similar to the address book previously discussed, except that MAIL contains all the names and addresses from A to Z. Each name and address, along with the record number, makes up a record of the mailing list.

```

♦♦ RECORD #6 ♦♦
WIDGETS SUPPLY CO.
555 BOGUS AVE.
GERTIE
TENNESSEE 38901
    
```

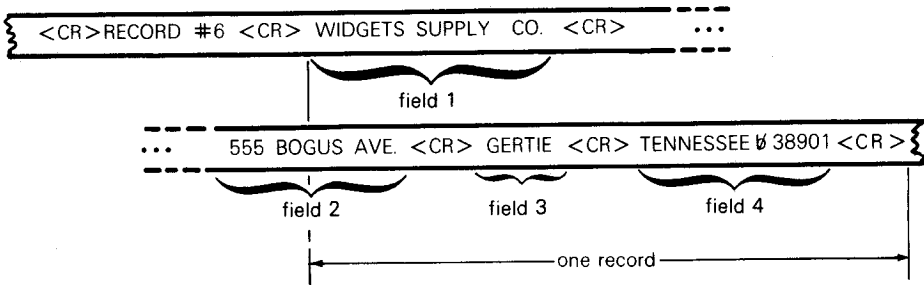
} one record

At this point, the total number of records in the file is unknown. Each record is formatted to contain five fields: 1) record number, 2) name, 3) street address, 4) city, 5) state and ZIP code.

```

♦♦ RECORD #6 ♦♦      field 1
WIDGETS SUPPLY CO.  field 2
555 BOGUS AVE.      field 3
GERTIE               field 4
TENNESSEE 38901     field 5
    
```

Of course, this is not what the data actually looks like on the data tape, but only how it has been arranged on the screen. The data on the tape is not written in English, but for purposes of demonstration it will be pictured as such. Below is an illustration of how record #6 appears on the tape.



The program statements to read and print the data read from the tape determine how the data appears on the screen.

Following is a program listing of MAIL.PRINT#. Type MAIL.PRINT# into PET memory, and save it on a cassette tape. Then list the program to follow the step-by-step discussion to see how it writes MAIL.

MAIL.PRINT#

```

10 PRINT"*****"
20 PRINT"  "
30 PRINT"  MAILING LIST ENTRY  "
40 PRINT"  "
50 PRINT"*****"
60 PRINT"  MOUNT TAPE; <RETURN> WHEN READY  "
70 GET A$:IF A#="" THEN GOTO 70
80 PRINT"  OPENING MAIL FILE  ":OPEN 1,1,2,"MAIL"
90 I=I+1
100 PRINT"  MAILING LIST ENTRY ITEM";I;"  "
110 PRINT"
120 PRINT"  (IF NO MORE ENTRIES, ENTER ");CHR$(34);"END
    ";CHR$(34);")"
130 PRINT"  INPUT "1) NAME          ";NM$
140 IF NM#="END" THEN CLOSE 1:PRINT "  ");"  END OF PROG
    RAM  ":END
150 INPUT "2) ADDR LINE 1";A1$
160 INPUT "3) ADDR LINE 2";A2$
170 INPUT "4) ADDR LINE 3";A3$
180 INPUT "  ENTER FIELD # TO CHANGE (0=SAVE)";X
190 IF X=0 THEN 220
200 IF X>=1 AND X<=4 THEN GOSUB 280
210 GOTO 180
220 PRINT#1,I
230 PRINT#1,NM$
240 PRINT#1,A1$
250 PRINT#1,A2$
260 PRINT#1,A3$
270 GOTO 90
280 PRINT"  ":ON X GOTO 290,300,310,320
290 INPUT "1) NAME          ";NM$:RETURN
300 PRINT:INPUT "2) ADDR LINE 1";A1$:RETURN
310 PRINT"  ":INPUT "3) ADDR LINE 2";A2$:RETURN
320 PRINT"  ":INPUT "4) ADDR LINE 3";A3$:RETURN

```

Recall the procedure to create a data file:

1. Create and/or LOAD the program file.
2. OPEN the data file.
3. WRITE to the data file.
4. CLOSE the data file.

The first step is to LOAD the program file. MAIL.PRINT# should be loaded if it is not already in memory. List the program. The first five lines (10 to 50) display a brief description of the function of the program. It's a good idea to describe the program because it helps to ensure that the user knows the function of the program. The next segment instructs the user to mount the data tape (lines 60 and 70).

The second step is to OPEN the data file:

```
80 PRINT"*** OPENING MAIL FILE ***":OPEN 1,1,2,"MAIL"
```

MAIL is opened as logical file #1 on the cassette unit, with an EOT mark to be written at the CLOSE of the file. The message " OPENING MAIL FILE " is displayed on the screen immediately prior to the actual OPEN command so the user knows that the file is being opened (it takes a few seconds to open the file).

Now the tape is ready to accept data. As data is input to the tape it should also be input to the screen so the data may be checked for mistakes. Lines 130 through 170 input the data to the screen.

Variable "I" in line 90 is the incrementing record counter, displayed at line 100. Lines 130 to 170 accept each variable, NM\$ (name) and A1\$, A2\$, and A3\$ (addresses) as separate fields by keyboard input; the end of each field is signaled by a carriage return. After all four fields have been entered, line 180 instructs the user to either change a field or save the record. If a field is incorrect, the user types the field number (1-4) and the program jumps to a correction routine at line 280 to change that field.

Depending on the field number input (variable X) the cursor is placed at the specified field, allowing the user to change the field, and the program returns to line 180 for the user to specify another field change. When all the fields are correct and 0 is input, the program continues at lines 220 through 270 to write the record on the data file. Each data item is written as a separate field by writing only one data item with a carriage return per PRINT# command. The data is written to the data tape as:

```
65 <CR> WIDGET SUPPLY CO. <CR> 555 BOGUS AVE. <CR> GERTIE
```

Be sure the file number referenced by the PRINT# is the same one specified in the OPEN statement.

After the record is saved, the program returns to line 90 to prepare for input of another record. If you have no more records to enter, enter "END" for NM\$. Line 140 closes the data file and writes an EOT mark (specified in the OPEN command) when NM\$="END".

You may have noticed that **the tape does not move after each record is saved. The PET stores all the data to be printed in an "input buffer" until it is written to the tape. When the input buffer is full, it writes a "block" of data to the tape.** A block may contain a partial record, a single record, or several records. The PET leaves interblock gaps between each block of data. However, you need not be concerned with blocks except to understand that the PET reads and writes data in blocks.

```
block gap block gap
```

Figure 5-6 shows a flowchart of the MAIL.PRINT# program.

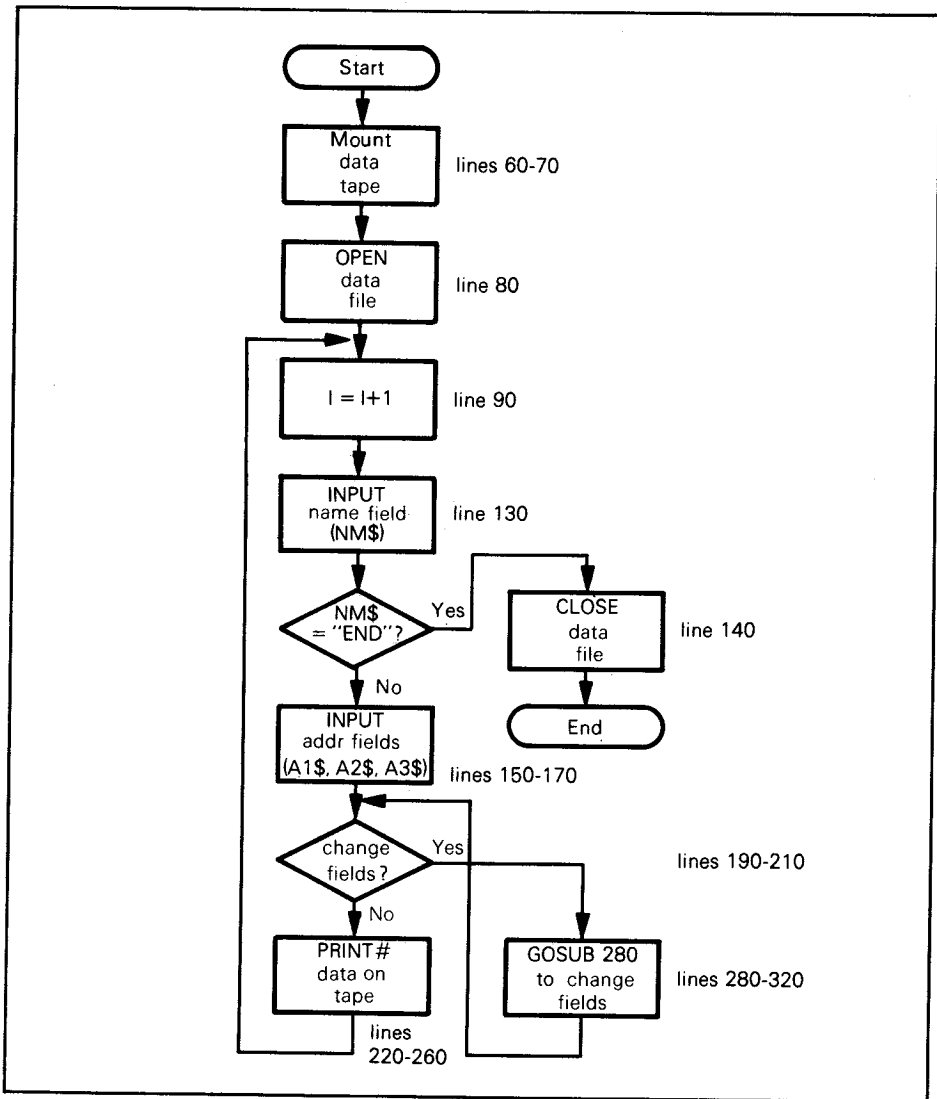


Figure 5-6. MAIL.PRINT#

Let's try a sample run of the program. Prepare a data tape and run the program to write your mailing list onto the data tape. A sample run may look like this:

```
*****  
*  
* MAILING LIST ENTRY *  
*  
*****
```

** MOUNT TAPE; PRESS <RETURN> WHEN READY **

** OPENING MAIL FILE **

PRESS PLAY & RECORD ON TAPE #1

OK

** MAILING LIST ENTRY ITEM 1 **

<IF NO MORE ENTRIES, ENTER "END">
ENTER FIELD # TO CHANGE (0=SAVE)

1) NAME ACME MANUFACTURING CO.
2) ADDR LINE 1 1235 MAIN ST.
3) ADDR LINE 2 DOWNTOWN
4) ADDR LINE 3 IL 62501

** MAILING LIST ENTRY ITEM 2 **

<IF NO MORE ENTRIES, ENTER "END">
ENTER FIELD # TO CHANGE (0=SAVE)

1) NAME BENJAMIN FRANKLIN
2) ADDR LINE 1 12 LIBERTY TOWER
3) ADDR LINE 2 PHILADELPHIA
4) ADDR LINE 3 PA 16524

** MAILING LIST ENTRY ITEM 3 **

<IF NO MORE ENTRIES, ENTER "END">
ENTER FIELD # TO CHANGE (0=SAVE)

1) NAME NEIL ARMSTRONG
2) ADDR LINE 1 597 SEA OF TRANQUILITY AVE.
3) ADDR LINE 2 EARTHVIEW
4) ADDR LINE 3 LUNAR 000000

◆◆ MAILING LIST ENTRY ITEM 4 ◆◆

<IF NO MORE ENTRIES, ENTER "END">
ENTER FIELD # TO CHANGE (0=SAVE)

1) NAME MAMMOTH DISTRIBUTION CO.
2) ADDR LINE 1 INDUSTRIAL PARK
3) ADDR LINE 2 CITY OF INDUSTRY
4) ADDR LINE 3 CA 92425

◆◆ MAILING LIST ENTRY ITEM 5 ◆◆

<IF NO MORE ENTRIES, ENTER "END">
ENTER FIELD # TO CHANGE (0=SAVE)

1) NAME HENRY MUSCATEL
2) ADDR LINE 1 819 OAK ST.
3) ADDR LINE 2 NAPA
4) ADDR LINE 3 CA 95303

◆◆ MAILING LIST ENTRY ITEM 6 ◆◆

<IF NO MORE ENTRIES, ENTER "END">
ENTER FIELD # TO CHANGE (0=SAVE)

1) NAME WIDGET SUPPLY CO.
2) ADDR LINE 1 555 BOGUS
3) ADDR LINE 2 GERTIE
4) ADDR LINE 3 TENNESSEE 38901

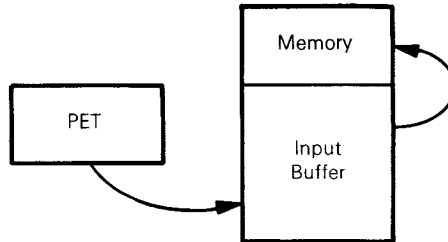
◆◆ MAILING LIST ENTRY ITEM 7 ◆◆

<IF NO MORE ENTRIES, ENTER "END">
ENTER FIELD # TO CHANGE (0=SAVE)

1) NAME END
◆◆ END OF PROGRAM ◆◆

READING A DATA FILE

After you have written information to a data file, in order to use it in another program you must be able to read the information off the tape and put it into memory. When the PET reads a data tape it transfers the data to an "input buffer" before sending the data on to memory:



The input buffer holds up to 191 bytes of data. **The data is read off the tape until the input buffer is full. When full, the PET temporarily stops the tape, and transfers the data to its proper location in memory.**

You can see this happen if you read large quantities of data at a time: the tape stops for several seconds while the input buffer "dumps" the data into memory. Once emptied, the tape will move again as more data is read. A flowchart of this process is shown in Figure 5-7.

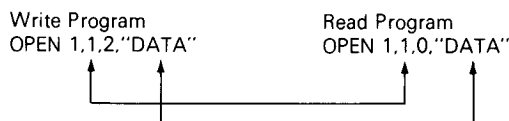
Reading a data file is similar to writing one. The four main steps are the same (except that the file is read instead of written):

1. **LOAD the program file**
2. **OPEN the data file**
3. **Read the data file**
4. **CLOSE the data file**

Step 1: Like writing to the tape, you must code your program to read the data file. **Code and type the program into PET memory and save it on tape.** Later, the program may be loaded when needed. Remember that the commands to open, read, and close the data file are programmed within the program file.

Once the program file is loaded, mount the data tape. Then execute the program using the RUN command.

Step 2: When the data file is opened it must be referenced with the same physical device number and file name it was assigned in the write program. A different logical file may be assigned, and the secondary address code must be 0 for the READ option.



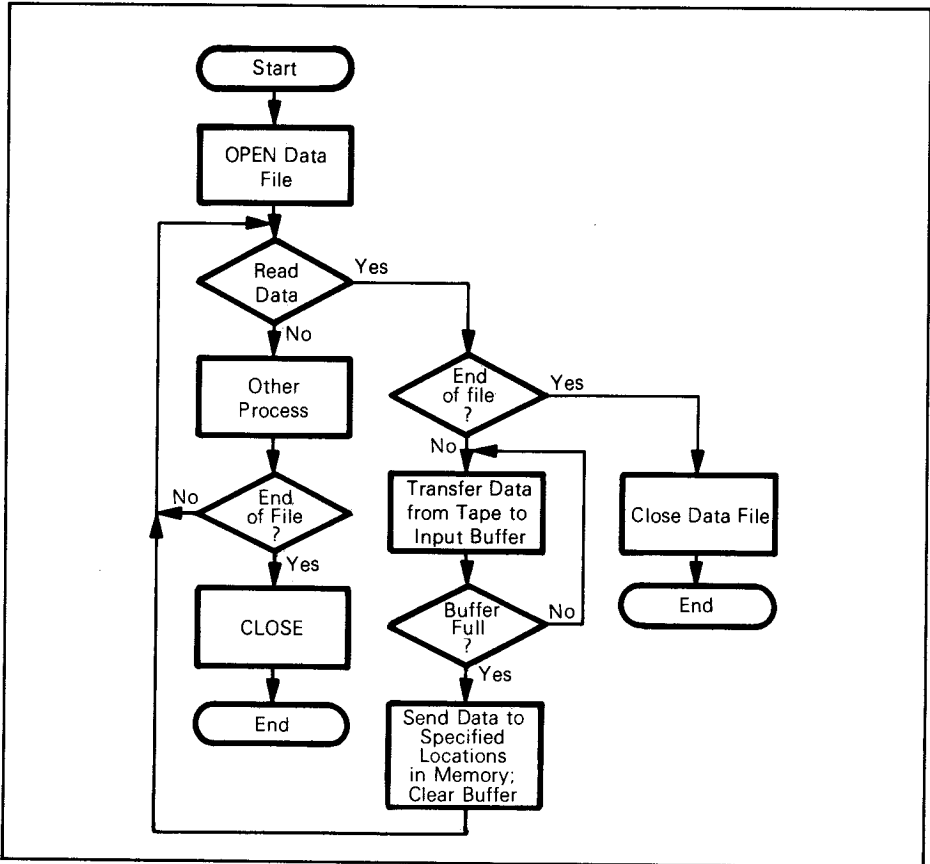


Figure 5-7. Read a Data File

Step 3: To read a data file, **load the program file**. The program that you load should include statements that will open, read, and close the data file.

There are two commands to read data from a tape: INPUT# and GET#. To read numbers and strings from a data file the INPUT# statement is used. The GET# statement reads one character at a time.

Reading data from a data file using INPUT# and GET# is discussed in the following sections.

Step 4: The last step **closes the file after the data is read**. CLOSE the same logical file that you OPENed.

```

OPEN 1,1,0,"DATA"
.
.
.
CLOSE 1
  
```

A good way to CLOSE a file that is being read is to test for an End of File (EOF) with the STATUS WORD (ST). When the data file was written, an EOF mark was written at the end of the file. When an EOF mark has been found, the file status equals 64 and the file may be closed. You may test for an EOF mark and close the file in one statement:

```
IF ST=64 THEN CLOSE 1
```

When ST equals 64, the file will automatically CLOSE.

Reading Numbers with INPUT#

Previously we wrote the program NUM.PRINT# to write the numbers 1 through 10 on the data tape NUMBERS, where each number was written as a single record. Now we will write a program called NUM.INPUT# to read the ten numbers from the NUMBERS data file, and print them on the screen.

To read numbers and strings from a data tape the INPUT# command is used. INPUT# reads one data item at a time, the data items being separated by a semicolon, comma, or carriage return.

```
INPUT#f,var
```

where:

f	is the logical file number (1-255, matching the file number in the OPEN and CLOSE command).
var	is the variable name(s) of the data to be read.

The first few lines of NUM.INPUT# instruct the user to load the data tape. These lines are identical to the first three lines of NUM.PRINT#. At line 30 is the wait loop allowing the user time to mount the data tape. After mounting the tape, key RETURN; the program continues at the next line.

```
10 PRINT"◆◆ READ NUMERIC DATA TAPE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE ; PRESS <RETURN> WHEN READY ◆◆":PRINT
30 GET A$:IF A$="" THEN 30
```

Before we can read any data, we must open the data file. Line 40 opens file #1, physical device #1, and secondary address 0 (OPEN for read) with filename NUMBERS.

```
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN 1,1,0,"NUMBERS":PRINT
```

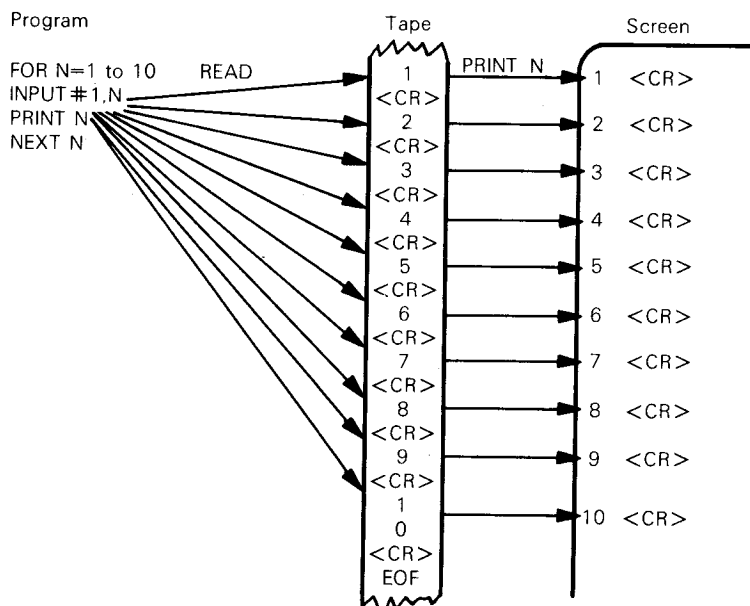
Next, a FOR...NEXT loop is set up to read the first ten data items from the tape and print to the screen:

```
50 FOR I=1 TO 10
60 INPUT#1,N           read N from tape
70 PRINT N             print N on screen
80 NEXT I
```

The INPUT# command at line 60 reads one number at a time. **The PET knows where the numbers are separated because carriage returns were written between each number input to the tape.** The FOR...NEXT loop (or a similar command such as GOTO) is necessary because **a single INPUT# command stops reading when a carriage return is encountered.** Without a loop or GOTO command to return to the INPUT# statement, only one number would be read. The PET reads both the number and the carriage return and sends them to memory so that the carriage return will be printed on the screen.

After the data is read, the file must be closed before ending the program.

```
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END
```



A complete listing of NUM.PRINT# is printed below, followed by a sample run of the program. The only file manipulation commands in this program are at lines 40, 60, and 90: OPEN, INPUT# , and CLOSE.

NUM.INPUT#

```
10 PRINT"◆◆ READ NUMERIC DATA FILE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE;PRESS <RETURN> WHEN READY":PRINT
30 GET A$:IF A$="" THEN 30
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN 1,1,0,"NUMBERS":PRINT
50 FOR I=1 TO 10
60 INPUT#1,N
70 PRINT N
80 NEXT I
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END
```

```

◆◆ READ NUMERIC DATA TAPE ◆◆

◆◆ MOUNT TAPE; PRESS <RETURN> WHEN READY ◆◆

◆◆ OPENING DATA FILE ◆◆

PRESS PLAY ON TAPE #1
OK

1
2
3
4
5
6
7
8
9
10

◆◆ CLOSING DATA FILE ◆◆

```

Reading Strings

The INPUT# statement also reads strings. In the "Writing Strings" section (page 247), we wrote the program WORD.PRINT# to write onto tape ten strings read from a DATA statement. The data file created was named NUMWORD. The data from NUMWORD looks like this on tape:

```

} <CR> ONE <CR> TWO <CR> ..... <CR> NINE <CR> TEN <CR> }

```

To read NUMWORD, use INPUT# in your program to read the data. The difference from the previous program is that the input variable must be able to accept strings. With only slight modification, you can change the READ NUMERIC DATA TAPE program to read NUMWORD. The changes occur at line 40 (name the data file), and line 60 (INPUT variable). The complete changed listing appears below, followed by a sample run of the program.

To read data in which several strings have been written for each carriage return is no different from reading single strings nor as difficult as writing several strings per line, except for screen formatting.

```

10 PRINT"◆◆ READ NUMWORD DATA FILE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE;PRESS <RETURN> WHEN READY":PRINT
30 GET A$:IF A$="" THEN 30
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN 1,1,0,"NUMWORD":PRINT
50 FOR I=1 TO 10
60 INPUT#1,N$
70 PRINT N$
80 NEXT I
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END

```



```

10 PRINT"◆◆ READ NAME DATA FILE ◆◆":PRINT
20 PRINT"◆◆ MOUNT TAPE;PRESS <RETURN> WHEN READY":PRINT
30 GET A$:IF A$="" THEN 30
40 PRINT"◆◆ OPENING DATA FILE ◆◆":OPEN 1,1,0,"NAME":PRINT
50 FOR J=1 TO 4
60 INPUT#1,F$,M$,L$
70 PRINT F$;" ";M$;" ";L$
80 NEXT J
90 PRINT"◆◆ CLOSING DATA FILE ◆◆":CLOSE1
100 END

```

◆◆ READ NAME DATA FILE ◆◆

◆◆ MOUNT TAPE; PRESS <RETURN> WHEN READY ◆◆

◆◆ OPENING DATA FILE ◆◆

PRESS PLAY ON TAPE #1
OK

ARNOLD J. SIMPSON
BETTY S. CLARK
HEADLY GEORGE JOYCE
CAROL ANNE SMITH

◆◆ CLOSING DATA FILE ◆◆

Sample Program

This sample program demonstrates how to read complex records of data from a data tape. This program, called MAIL.INPUT#, reads the mailing list data, which was written to data file MAIL by a program titled MAIL.PRINT# (MAIL.PRINT# listing, page 250). Each MAIL file record contains five fields; these fields are reserved for the record number, customer name, street, city, and state and ZIP code. Below is an example of a MAIL file record:

◆◆ RECORD #6 ◆◆	field 1
WIDGETS SUPPLY CO.	field 2
555 BOGUS AVE.	field 3
GERTIE	field 4
TENNESSEE 38901	field 5

Following is a program listing of MAIL.INPUT#. Type in MAIL.INPUT# and save it on a cassette tape. Then list the program to follow the step-by-step discussion.

MAIL.INPUT#

```
10 PRINT"*****"  
20 PRINT"  
30 PRINT" READ MAIL FILE W/ INPUT# "  
40 PRINT"  
50 PRINT"*****":PRINT:PRINT  
60 PRINT"  
70 GET A$:IF A#="" THEN 70  
80 PRINT"  
90 PRINT"  
100 IF ST=64 THEN 9999  
110 INPUT#1,I$  
120 INPUT#1,NM$  
130 INPUT#1,A1$  
140 INPUT#1,A2$  
150 INPUT#1,A3$  
160 PRINT"  
170 PRINT"  
180 PRINT"ADDR:";TAB(9);A1$  
190 PRINTTAB(9);A2$  
200 PRINTTAB(9);A3$  
210 PRINT"  
220 INPUT"ENTER 'Y' TO READ NEXT RECORD";A$:IF A#="Y" GOTO 100  
9999 PRINT"  
CLOSE1:END
```

The procedure for reading from data files is:

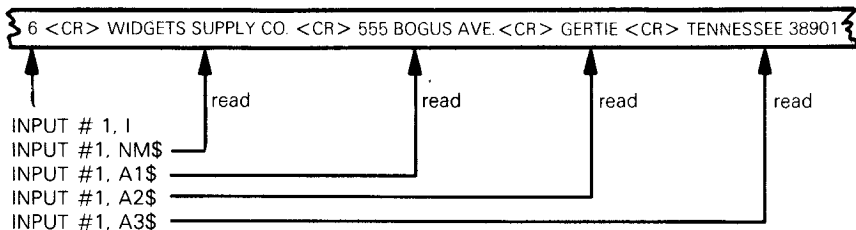
1. Create and/or LOAD the program file.
2. OPEN the data file.
3. READ the data file.
4. CLOSE the data file.

The first five lines display a brief program description. The next two lines, 60 and 70, instruct the user to mount the data tape. We are now ready to begin reading customer addresses. First the data file must be opened. MAIL is opened as logical file #1 on the cassette unit #1. This correlates to how it was saved in MAIL.PRINT#. The secondary address must be 0 for READ.

```
80 PRINT"  
OPEN1,1,0,"MAIL"
```

Next, read the data. Line 100 uses the status word (ST) to check for the End of File mark. If ST=64 (indicating an End of File mark is found), then the file is closed at line 9999. The ST should be checked before data is read so that you do not attempt to do any unnecessary reading when there is no more data.

Lines 110 to 150 read the data using INPUT#. Each field was printed to the tape with an individual PRINT#; the fields are separated by carriage returns so each field will be read in turn with an individual INPUT#. The variable or string names may be different when reading the data than when writing the data. For instance, data may be written to the tape as X\$ and read back from the tape as A\$, and the PET will not know the difference because the data variable name is not saved.



When the PET reads data it stores it in the input buffer (memory), but nothing is written to the screen unless you tell it to. Therefore, you must tell the PET to print the data on the screen. Once the data is in the input buffer it may be printed to the screen in any format, as you can see in lines 160 to 200, where tabs and leaders were inserted. Line 210 moves the cursor down four lines.

```

160 PRINT"◆◆ RECORD #";I$;" ◆◆"
170 PRINT"NAME:";TAB(9);NM$
180 PRINT"ADDR:";TAB(9);A1$
190 PRINTTAB(9);A2$
200 PRINTTAB(9);A3$
210 PRINT"◆◆◆◆":

```

The screen output looks like this:

```

◆◆ RECORD #6 ◆◆

NAME:   WIDGETS SUPPLY CO.
ADDR:   555 BOGUS AVE.
        GERTIE
        TENNESSEE 38901

```

After all four fields are printed, PET asks the user if the next record is desired:

```

220 INPUT"ENTER 'Y' TO READ NEXT RECORD";A$:IF A$="Y" GOTO 100

```

If the user wants the next record, the program goes to 100 and repeats the process until the status word (ST) signals an EOF. If the user does not wish to continue, or if an EOF is encountered, the file is closed and the program ends.

Figure 5-8 shows a flowchart of the MAIL.INPUT# program. A sample run of the program follows.

As you run your own MAIL.INPUT# program, do not panic if the PET appears to shut off for a few seconds. If you watch the cassette until you will see that the cassette tape is moving when it appears that the PET is stopped. What is happening is that the PET is reading the next 191 bytes of data into the input buffer before continuing on with the program, as was previously explained. Once the buffer is full, program control will return.

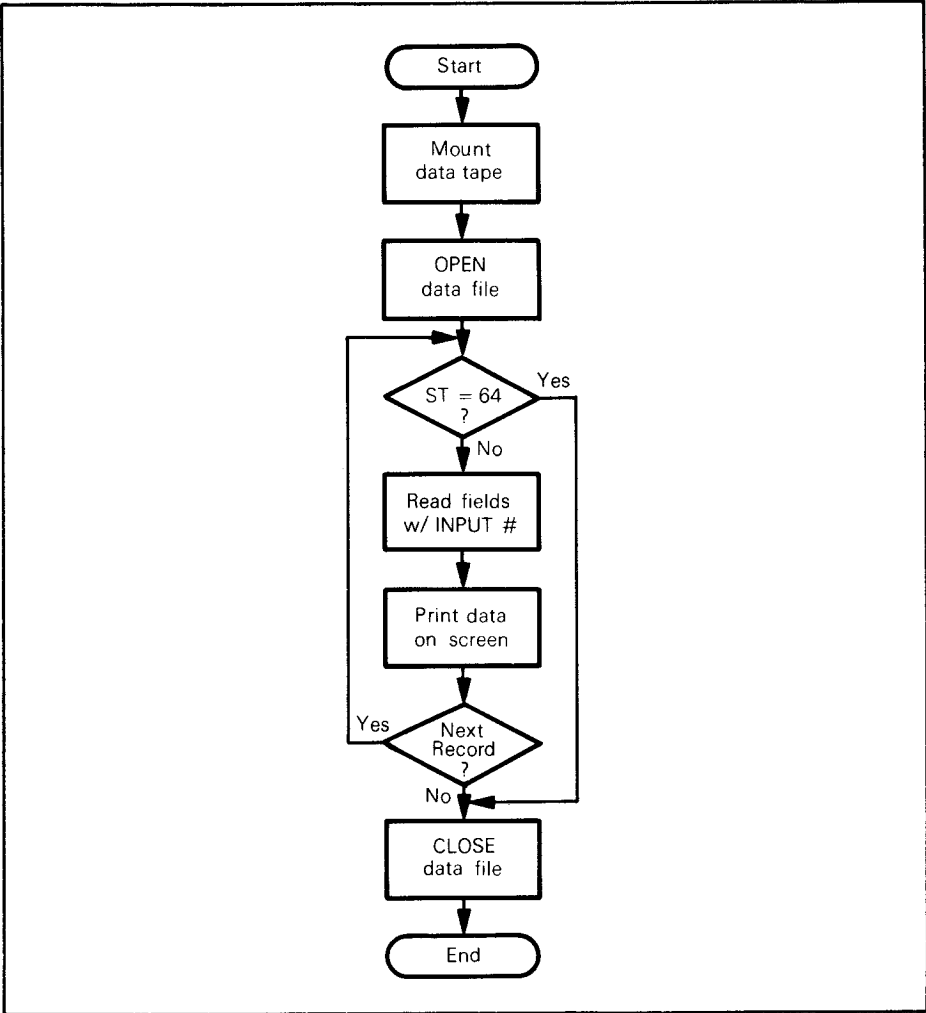


Figure 5-8. MAIL.INPUT#

*
* READ MAIL FILE W/ INPUT# *
*

** PRESS <RETURN> WHEN TAPE IS LOADED **

** OPENING MAIL FILE **

PRESS PLAY ON TAPE #1
OK

** READING MAIL FILE **

** RECORD # 1 **

NAME: ACME MANUFACTURING CO.
ADDR: 1235 MAIN ST.
DOWNTOWN
IL 62501

ENTER 'Y' TO READ NEXT RECORD

** RECORD # 2 **

NAME: BENJAMIN FRANKLIN
ADDR: 12 LIBERTY TOWER
PHILADELPHIA
PA 16524

ENTER 'Y' TO READ NEXT RECORD

** RECORD # 3 **

NAME: NEIL ARMSTRONG
ADDR: 597 SEA OF TRANQUILITY AVE.
EARTHVIEW
LUNAR 000000

ENTER 'Y' TO READ NEXT RECORD

** RECORD # 4 **

NAME: MAMMOTH DISTRIBUTION CO.
ADDR: INDUSTRIAL PARK
CITY OF INDUSTRY
CA 92425

ENTER 'Y' TO READ NEXT RECORD

◆◆ RECORD # 5 ◆◆

NAME: HENRY MUSCATEL
ADDR: 819 OAK ST.
NAPA
CA 95303

ENTER 'Y' TO READ NEXT RECORD

◆◆ RECORD # 6 ◆◆

NAME: WIDGET SUPPLY CO.
ADDR: 555 BOGUS
GERTIE
TENNESSEE 38901

ENTER 'Y' TO READ NEXT RECORD

◆◆ END OF MAIL FILE--PROGRAM TERMINATED ◆◆

Reading Numbers with GET#

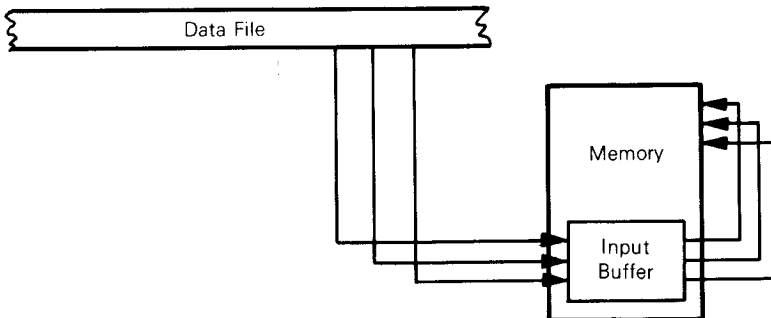
Another method of reading data files uses the GET# command.

GET#f,var

where:

- f is the logical file number (1-255, matching the file number in the OPEN and CLOSE commands).
- var is the variable name of the data to be read.

GET# reads one character at a time from the data file and transfers it through the input buffer to memory. It is similar to GET, which accepts one character at a time from the keyboard and transfers it to the screen.



Because **GET#** reads one character at a time and not a 191-byte block at a time like **INPUT#**, it **is able to read all file delimiters and anything else on the tape**. This feature is especially helpful when you want to read everything that is written on a bad data tape to find the problem. **GET#** also allows a single character to be compared to a status or character value within programs. This will be demonstrated in the sample programs.

The two sample programs in this section will demonstrate how to print out an entire file including all file delimiters, and how to print out the MAIL data file separated into records.

Sample Program 1

The following program, MAIL.GET#1, reads data file MAIL one character at a time and prints the contents of MAIL on the screen:

```

MAIL.GET#1
10 PRINT"#####"
20 PRINT"▲"
30 PRINT"▲ READ MAIL FILE W/ GET# ▲"
40 PRINT"▲"
50 PRINT"#####":PRINT:PRINT
60 PRINT"▲▲ PRESS <RETURN> WHEN TAPE IS LOADED ▲▲"
70 GET A$:IF A$="" THEN 70
80 PRINT"▲▲ OPENING MAIL FILE ▲▲":PRINT:OPEN1,1,0,"MAIL"
90 PRINT"▲▲ MAIL FILE ▲▲"
100 IF ST=64 THEN 9999
110 GET#1,X$
120 IF X$=CHR$(13) THEN X$="█"
130 PRINT X$;
140 GOTO 100
9999 PRINT"▲▲▲ END OF MAIL FILE--PROGRAM TERMINATED▲▲"
:CLOSE1:END

```

Lines 10 through 90 are similar to the first lines of MAIL.INPUT# (page 262). These lines introduce the program, give instructions for mounting the data tape, and then open the data file.

Lines 100 through 140 control the reading of the MAIL data file and the printout of its contents on the screen.

Line 100 checks for an End of File (EOF) status. If an EOF is not encountered, the data is read at line 110. In line 110, #1, is the file number and X\$ is the variable name assigned to the data strings. This statement will read the first character encountered and transfer it through the input buffer to memory.

Then the program drops through to line 120. Line 120 compares the current value of X\$ to a carriage return (CHR\$(13)). If the value of X\$ is CHR\$(13), then the value of X\$ is changed to a FULL GRID █. This change avoids printing a carriage return, which would push the cursor to the next line; with the FULL GRID being printed, the whole file appears as one continuous line, representing how it is written on the data tape. An example of this will be shown in the sample run.

Next, the character is printed on the screen. You will see in the sample run that a FULL GRID ■ appears for each carriage return. Also, make sure that a semicolon follows the variable in the PRINT statement, otherwise the data will only be printed down the first column of the screen.

After one character is read from the tape and printed on the screen, the program is directed back to check the status and GET# another character. This process should repeat itself until ST=64 (the End of File). When the End of File is encountered at line 100, the job of MAIL.GET#1 is complete. At line 9999 the program closes the data file and ends.

Following is a sample run of MAIL.GET#1, using MAIL as the data file.

MAIL.GET#1

```
*****
*
*   READ MAIL FILE W/ GET#   *
*
*****

** PRESS <RETURN> WHEN TAPE IS LOADED **

** OPENING MAIL FILE **

PRESS PLAY ON TAPE #1
OK

** MAIL FILE **

  1 **ACME MANUFACTURING CO.**1235 MAIN ST.
  **DOWNTOWN**IL  62501** 2 **BENJAMIN FRANKL
  IN**12 LIBERTY TOWER**PHILADELPHIA  16524
  ** 3 **NEIL ARMSTRONG**597 SEA OF TRANQUILI
  TY**EARTHVIEW**LUNAR  00000** 4 **MAMMOTH D
  ISTRIBUTION CO.**INDUSTRIAL PARK**CITY OF
  INDUSTRY**CA  92425** 5 **HENRY MUSCATEL**8
  19 OAK ST.**NAPA**CA  95303** 6 **WIDGET SU
  PPLY CO.**555 BOGUS AVE.**GERTIE**TENNESSEE
  38901**

** END OF MAIL FILE--PROGRAM TERMINATED**
```

Sample Program 2

Next is the sample program MAIL.GET#2, to read MAIL and print the data on the screen divided into records. Below is a program listing of MAIL.GET#2.

MAIL.GET#2

```
10 PRINT"*****"
20 PRINT"@"
30 PRINT"@ READ MAIL FILE W/ GET# @"
40 PRINT"@"
50 PRINT"*****":PRINT:PRINT
65 PRINT"@@ PRESS <RETURN> WHEN TAPE IS LOADED @@":PRINT:
70 GET A$:IF A$="" THEN 70
80 PRINT"@@ OPENING MAIL FILE @@":PRINT:OPEN 1,1,0,"MAIL"
90 PRINT:PRINT"@@ MAIL FILE @@":PRINT:
100 IF ST=64 THEN 9999
110 GET#1,X$
120 IF X$=CHR$(13) THEN F=F+1
130 PRINT X$:
140 IF F>=5 THEN GOSUB 160
150 GOTO 100
160 PRINT:
170 R=R+1
180 IF R>2 THEN PRINT "PRESS 'Y' FOR NEXT SET OF RECORDS"
:INPUT A$
185 IF A$="Y" THEN R=0
190 F=0:PRINT:RETURN
9999 PRINT"***** END OF MAIL FILE--PROGRAM TERMINATED*****":
CLOSE1:END
```

Type in MAIL.GET#2 and SAVE and VERIFY it on a cassette tape. LIST the program.

The first ten lines (10 through 100) of MAIL.GET#2 are identical to the first ten lines of MAIL.GET#1. This part of the program informs the user of the program's functions and procedures, and opens the MAIL data file in preparation for reading the data.

The real difference between MAIL.GET#2 and MAIL.GET#1 is at line 120. If X\$=CHR\$(13), instead of changing the value of X\$ from a carriage return to FULL GRID ■, variable F — a carriage return counter — is incremented by one. When MAIL.PRINT# wrote to the data file, a carriage return marked the end of each of the five fields in each record. By keeping track of the number of carriage returns, MAIL.GET#2 counts the number of fields read to know when an entire record has been read. This can only be done by programming a conditional statement into the read program where GET# is used instead of INPUT#. Line 140 conditionally calls a subroutine if the specified condition is true.

Statement 160 prints blank lines between records. In statement 170, variable R serves as a record counter. Line 180 tests to see if more than two records have been read. When three records have been read, the screen is full, and the user is asked if a new set of records is desired. If yes, the record counter R and the field counter F are initialized to zero before returning to read the next set of records at line 100. If more records are to be read, the program is sent to line 100 to GET# the next character. This continues until the user inputs something other than a Y character or ST=64; at that time the file is closed and the program ends. Figure 5-9 shows a flowchart of the process of lines 100 to 190.

Although GET# is similar to INPUT# in some ways, it is more difficult to format the printout when using GET# if titles and indentation or spacing is desired. Like comparing X\$ to CHR\$(13), other field delimiters or characters would have to be conditionally tested if one desires more than a simple printout.

At this time you should feel comfortable manipulating program and data files. The three file commands PRINT#, INPUT#, and GET# are surprisingly similar. When using files, make sure that the same logical file numbers are used throughout the program, and that each logical file is opened before it is called and closed before the program ends. It is also good practice to display messages on the screen explaining what the PET is doing so the user always knows what is happening.

Following is a sample RUN of MAIL. GET#2 reading MAIL.

↑

↑ READ MAIL FILE W/ GET# ↑

↑

↑↑ PRESS <RETURN> WHEN TAPE IS LOADED ↑↑

↑↑ OPENING MAIL FILE ↑↑

PRESS PLAY ON TAPE #1

OK

↑↑ MAIL FILE ↑↑

1

ACME MANUFACTURING CO.
1235 MAIN ST.
DOWNTOWN
IL 62501

2

BENJAMIN FRANKLIN
12 LIBERTY TOWER
PHILADELPHIA
PA 16524

3

NEIL ARMSTRONG
597 SEA OF TRANQUILITY
EARTHVIEW
LUNAR 00000

PRESS 'Y' FOR NEXT SET OF RECORDS?Y

4

MAMMOTH DISTRIBUTION CO.
INDUSTRIAL PARK
CITY OF INDUSTRY
CA 92425

5

HENRY MUSCATEL
819 OAK ST.
NAPA
CA 95303

6

WIDGET SUPPLY CO.
555 BOGUS AVE.
GERTIE
TENNESSEE 38901

PRESS 'Y' FOR NEXT SET OF RECORDS?Y

↑↑ END OF MAIL FILE--PROGRAM TERMINATED↑↑

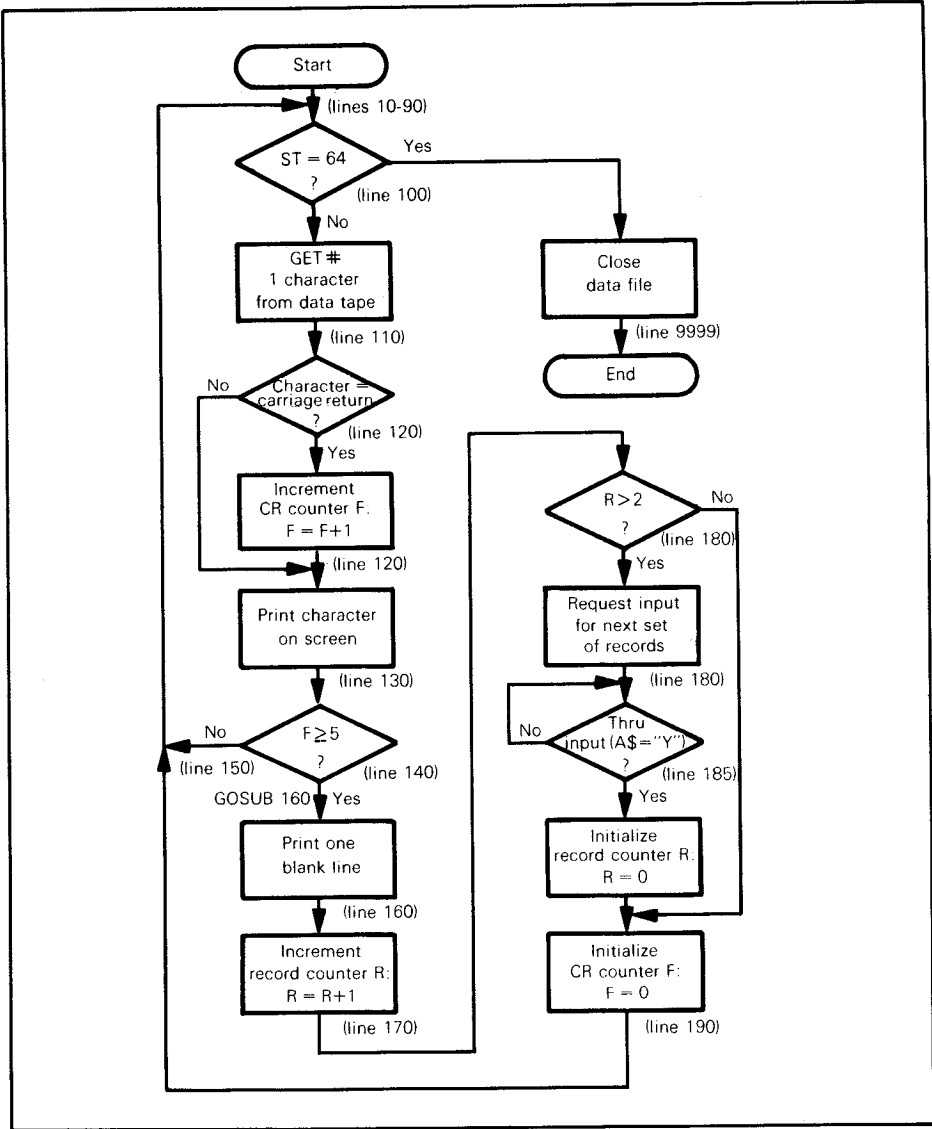


Figure 5-9. Format Printing Using GET#

ALTERNATE CHARACTER SET

Another PET feature is its dual character set, made up of a "standard" and an "alternate" character set. **For most keyboards the Standard Character Set consists of capital, or upper-case alphabetic characters, and graphic characters. The Alternate Character Set is upper case alphabetic characters, shifted lower-case alphabets and about half of the graphic characters. On the CBM, standard and alternate character sets are reversed.** The difference between the two character sets occurs in shifted mode:

	Standard Character Set	Alternate Character Set
Unshifted mode	Upper-case alphabets	Upper-case alphabets
Shifted mode	Graphics	Lower-case alphabets some graphics

The Alternate Character Set is put into effect by POKEing a special value into a specified location in memory. **The contents of memory address 59468 determines which character set is in operation.** When memory location 59468 contains the value 12 or 13, the Standard Character Set is in use. If location 59468 contains any value from 0 to 11, or 14 to 255, the Alternate Character Set is in effect:

<u>Contents of Memory Location 59468</u>	<u>Character Set in Operation</u>
0,1,2,3,4,5,6,7,8,9,10,11	alternate
12,13	standard
14,15...255	alternate

For the purpose of simplicity, we will always use value 12 to designate the Standard Character Set and value 14 for the Alternate Character Set.

Figure 5-10 is a flowchart of the Standard/Alternate Character Set decision.

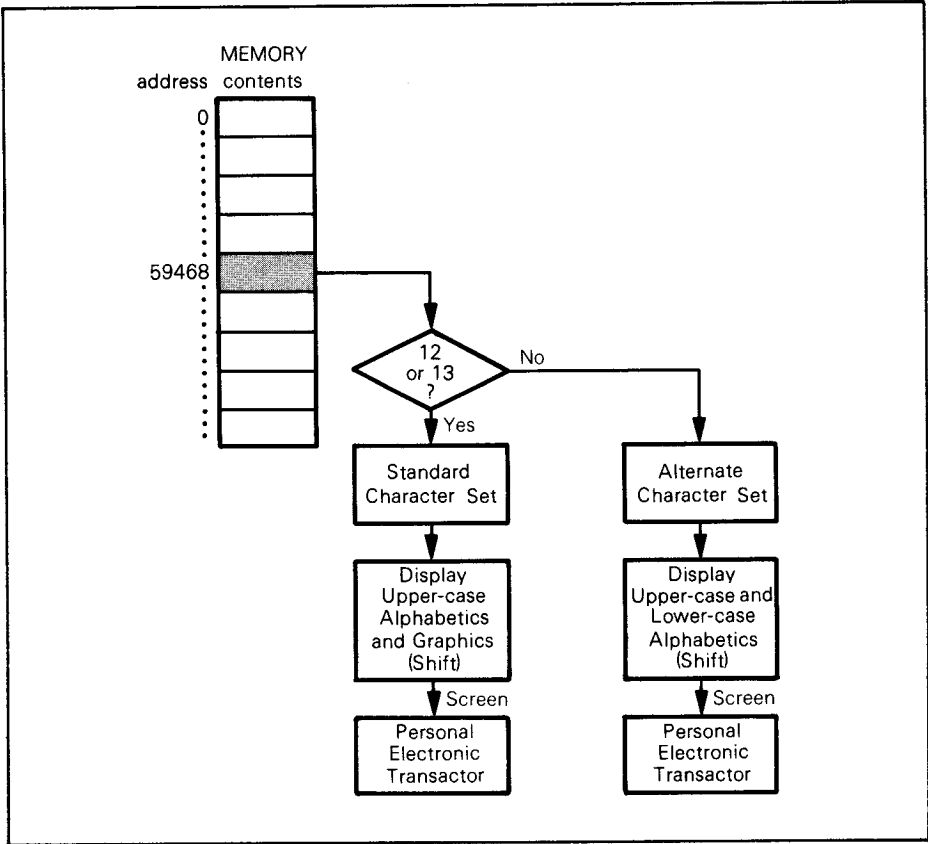
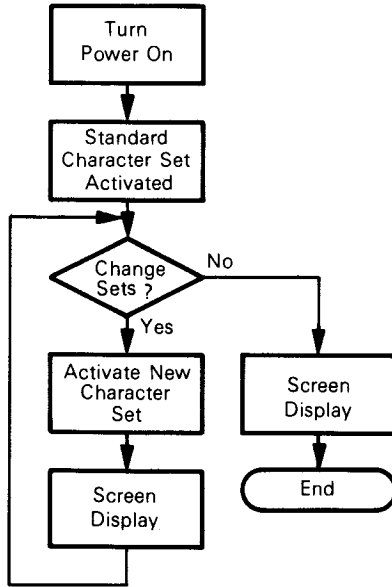


Figure 5-10. Standard/Alternate Character Set Decision

On PET power-up, value 12 is automatically POKEd into memory address 59468 (value 14 for the CBM), activating the Standard Character Set. The PET begins and remains in this mode until programmer intervention changes the character mode. However, once the character sets are switched, the new character set is displayed until either 1) the PET is turned off; whereas on power-up the Standard Character Set is activated, or 2) the other character set is POKEd into effect.



Changing character sets is done by POKing a numeric value into memory location 59468. POKE a numeric value from 0 to 11 or 14 to 255 into location 59468:

- POKE 59468,12 activate the Standard Character Set
- POKE 59468,14 activate the Alternate Character Set

To change from Standard to Alternate Character Set, type the following statement in calculator mode:

```
POKE 59468,14
```


This will push value 12 out of location 59468 and POKE value 14 in.



Character sets may be changed in either calculator or program mode. In calculator mode the change is immediate, whereas a programmed conversion requires the program statement to be run first. **Whenever the character set is changed, the whole screen image is affected.** For example, draw a graphic picture of a spiral on the screen, as shown below. In calculator mode, POKE value 14 into memory location 59468. The picture will instantly change to lower-case alphabets and graphics. Figure 5-11 illustrates this sequence.

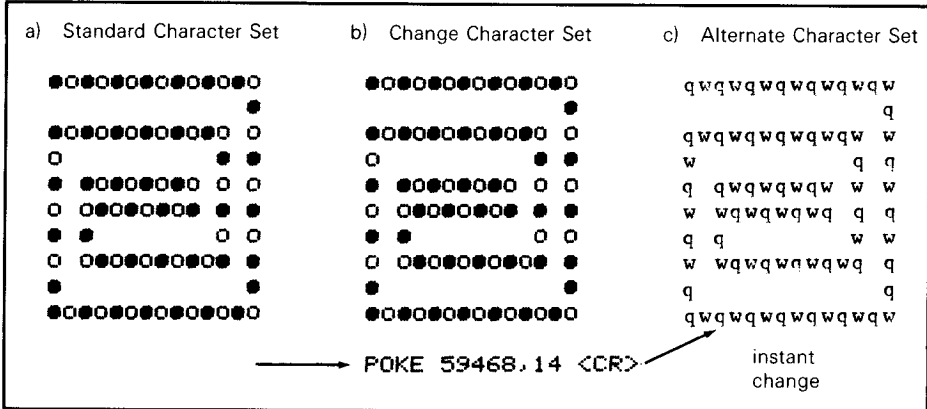
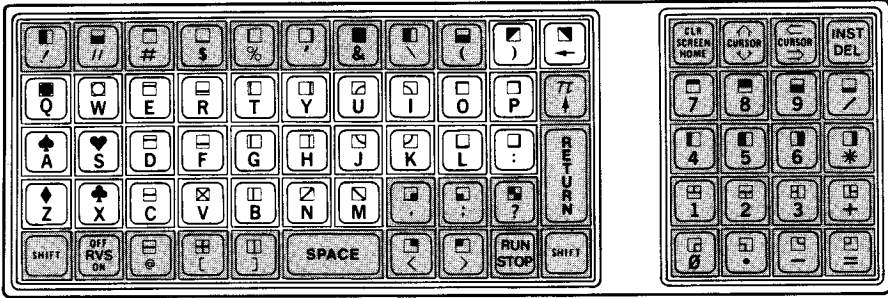


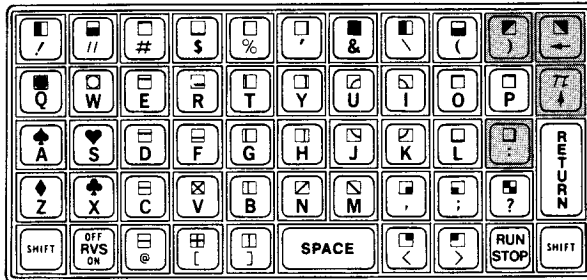
Figure 5-11. Standard to Alternate Character Set

FEATURES OF THE ALTERNATE CHARACTER SET

In alternate character mode, the 26 alphabetic keys will print lower-case alphabets. Shifted punctuation and special symbol keys print the graphic symbols because there are no lower-case punctuation marks. The shaded keys in the compact keyboard diagram below are the keys which remain unchanged; some special symbols are rearranged on the full size keyboards. Notice that nearly the entire top and bottom rows and the entire numeric keyboard are not affected.










Four keys in the upper right side of the keyboard print a unique special symbol when shifted, instead of the standard graphic character. These four are shaded in the diagram below:



In shifted alternate mode, these keys print:



The cursor controls perform their normal movements in the alternate mode. However, some of the shifted cursor controls display a lower-case alphabetic symbol instead of their normal symbol. For instance, CLEAR is normally displayed as a , but in alternate mode it is displayed as a lower-case s. The chart below shows the change in the cursor symbols.

	s		q
	t		(unaltered)
	r		(unaltered)

Summary

With a few exceptions, the Alternate Character Set may be summed up as follows:

1. Value 14 is POKEd into memory location 59468 to activate the Alternate Character Set.
2. The PET must be in shifted mode to print the lower-case alphabetic or special symbols.
3. The Alternate Character Set may be divided into three categories:
 - a. Upper-case alphabetic: keys A through Z
 - b. Graphic symbols: all punctuation (except the four unique keys in the top right corner, as previously explained)
 - c. Lower-case alphabetic and cursor control keys

BASIC WORD ABBREVIATIONS

You learned early in this book that the BASIC command PRINT could always be entered from the keyboard by the abbreviation `?`, the question mark character. This was expanded by PET BASIC to the full word PRINT at the first and all subsequent listings.

Most of the BASIC commands, statements, and functions can similarly be abbreviated by the first two characters of the keyword, with the second character entered in shifted mode. With the standard character set, the second character appears as a graphic character. For example, the abbreviation for LIST appears as:

L 

Select the Alternate Character Set with a POKE 59468,14 to have the second character appear as a lower-case letter, e.g.:

Li


This markedly improves the readability of the line being entered. PET BASIC makes no distinction between the two of them, however. Either one is expanded to the word LIST.

Where a two-letter abbreviation is ambiguous (does St mean STEP or STOP?) the two-letter abbreviation is assigned to the most frequently used keyword, and the other word or words are either not abbreviated or are abbreviated by the first three characters, the third entered in shifted mode. For STEP/STOP, STOP is abbreviated:

St

STEP is abbreviated:

STe

To abbreviate STEP, type unshifted S (capital S), unshifted T (capital T), and shifted E (lower-case e or graphic 3/4 TOP LINE HORIZONTAL ).

Following are a few sample input lines showing use of the two- and three-letter abbreviations wherever possible. All the abbreviated words are expanded to the full spelling when you list the programs.

```

Po 59468,14 ← (after RETURN) Abbreviation for POKE 59468,14
10 Le A=10
20 B=A An 14+Ex(2)
30 Di C(5)
40 Fo I=0 TO 5
50 Re C(I)
60 Ne
70 Da 1,6,2,4,10,5,16
80 REs
90 En
Li ← Abbreviation for LIST
10 LET A=10
20 B=A AND 14+EXP(2)
30 DIM C(5)
40 FOR I=0 TO 5
50 READ C(I)
60 NEXT
70 DATA 1,6,2,4,10,5,16
80 RESTORE
90 END
Po 59468,12 ← (before RETURN) Abbreviation for POKE 59468,12

```

After keying RETURN at the last POKE command line (return to Standard Character Set), you will see the abbreviations show with graphics as the shifted characters.

A list of reserved words and their abbreviations, if any, is given in Table 5-2. Note that the expansions from abbreviations for the two functions SPC and TAB include the left parenthesis. This means that if you use the abbreviation for either of these, you must not type in the left parentheses. For example:

10?Sp(5)

expands to:

10 PRINT SPC((5)

syntax error results from two
left parentheses

The correct key-in is:

10? Sp5)

This parenthesis rule applies only to the SPC and TAB functions and is a format inconsistency you will have to watch for when abbreviating these function names. For all other functions, you key in both parenthesis. For example:

10? Rn(1)

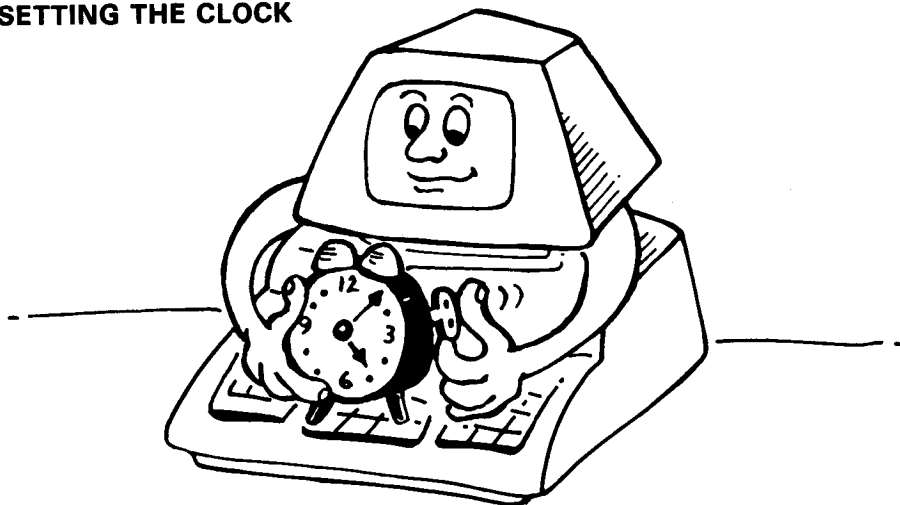
expands to:

10 PRINT RND(1)

Table 5-2. Keyword Abbreviations

BASIC Keyword	Abbreviation	BASIC Keyword	Abbreviation	BASIC Keyword	Abbreviation	BASIC Keyword	Abbreviation
ABS	Ab	FRE	Br	NOT	No	SIN	Si
AND	An	GET	Ge	ON	--	SPC(Sp
ASC	As	GET#	--	OPEN	Op	SQR	Sq
ATN	At	GOSUB	GOs	OR	--	ST	--
CHR\$	Ch	GOTO	GO	PEEK	Pe	STEP	STe
CLOSE	CLo	IF	--	POKE	Po	STOP	St
CLR	Cl	INPUT	--	POS	--	STR\$	STr
CMD	Cm	INPUT#	In	PRINT	?	SYS	Sy
CONT	Co	INT	--	PRINT#	Pr	TAB(Ta
COS	--	LEFT\$	LEf	READ	Re	TAN	--
DATA	Da	LEN	--	REM	--	THEN	Th
DEF	De	LET	Le	RESTORE	REs	TI	--
DIM	Di	LIST	Li	RETURN	REt	TI\$	--
END	En	LOAD	Lo	RIGHT\$	Ri	TO	--
EXP	Ex	LOG	--	RND	Rn	USR	Us
FN	--	MID\$	Mi	RUN	Ru	VAL	Va
FOR	Fo	NEW	--	SAVE	Sa	VERIFY	Ve
		NEXT	Ne	SGN	Sg	WAIT	Wa

SETTING THE CLOCK



Another PET feature is the real-time clock. The PET clock keeps real time in a 24-hour cycle by hours, minutes, and seconds. The reserved string variable TIME\$ or TI\$ keeps track of the time. **To set the clock, use the following format:**

TIME\$ = "hhmmss"

where:

hh is the hour between 0 and 23.
mm is the minutes between 0 and 59.
ss is the seconds between 0 and 59.

For hh, enter the hour of the day from 00 (12 am) to 23 (11 PM). The PET is on a 24-hour cycle so that you can distinguish between AM and PM, unlike 12-hour clocks. The hours from 00 to 11 designate AM, and the hours from 12 to 23 designate PM, returning to 00 at midnight. At midnight, when one 24-hour cycle ends and another begins, hh, mm, and ss are all equal to zero.

When initializing TIME\$ to the actual time, type in a time a few seconds in the future. When that actual time is reached, press RETURN and the PET clock is set.

```
TIME$=" 120150<CR>
```

To retrieve the time any time after the initialization of TIME\$, type the following in calculator or program mode:

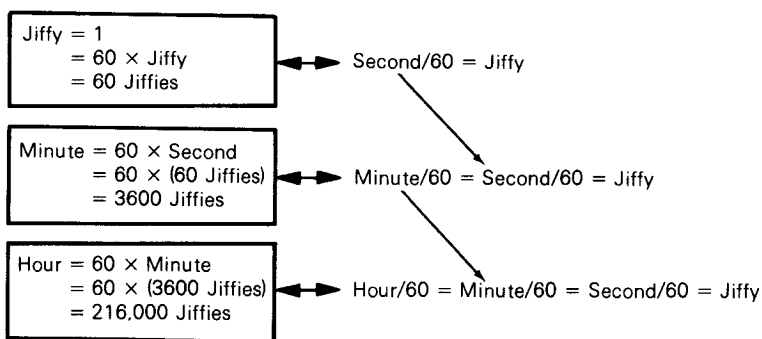
```
?TIME$ <CR>
```

and the PET will display the time in hhmmss:

```
?TIME$  
120200
```

The PET clock keeps time until the PET is turned off. The clock needs to be reset when the PET is powered up again.

The PET also keeps track of time in "jiffies". A "jiffy" is 1/60 of a second. Reserved numeric variable TIME, or TI, is a numeric variable which is automatically incremented every 1/60 of a second. TIME is initialized to zero on start-up, and is reset back to zero after 51,839,999 jiffies. TIME\$ (hhmmss) is a string variable that is generated from TIME. **To ensure greater accuracy the PET keeps track of time in jiffies by incrementing every 1/60 of a second, instead of once every second. When TIME\$ is called to print the time in hours, minutes, and seconds, the memory converts jiffy time to real time.** Notice that TIME\$ and TI\$ are not the string conversion of TIME and TI but are numbers representing real time calculated from jiffy time (TIME, TI). The conversion is done as follows. Each second is divided into 60 jiffies. One minute is composed of 60 seconds. One hour is made up of 60 minutes. In summary, one second is 60 jiffies, one minute is 3600 jiffies, and one hour is 216,000 jiffies, as illustrated below:



The following equations convert jiffy time (J) into real time divided into hours (H), minutes (M), and seconds (S). We can write a program to print the time from jiffies.

<pre>10 J=TI 20 H=INT(J/216000)</pre>	<p>Calculate hours. Integer function takes only whole number.</p>
<pre>30 IF H<>0 THEN J=J-H*21600</pre>	<p>If any hours, subtract number of jiffies in one hour by H to leave remaining jiffies.</p>
<pre>40 M=INT(J/3600)</pre>	<p>Calculate minutes. Integer function takes only whole number.</p>
<pre>50 IF M<>0 THEN J=J-M*3600</pre>	<p>If any minutes, subtract number of jiffies in minutes by M to leave remaining jiffies.</p>
<pre>60 S=INT(J/60)</pre>	<p>Calculate seconds. Integer function takes only whole number.</p>



```

5 PRINT "REAL TIME":PRINT:PRINT:
10 J=TI
15 T%=TIME$
20 H=INT(J/21600)
30 IF H<>0 THEN J=J-H*21600
40 M=INT(J/3600)
50 IF M<>0 THEN J=J-M*3600
60 S=INT(J/60)
70 H%=RIGHT$(STR$(H),2)
80 M%=RIGHT$(STR$(M),2)
90 S%=RIGHT$(STR$(S),2)
100 PRINT "H:M:S: ";H%";";M%";";S%,"TIME%: ";T%
110 PRINT "0000";:GOTO10

```

Statements 70 through 90 convert the numeric answers into proper string form for tidy printing. Statement 100 prints both the real time calculated from the program, and TIME\$, the real time calculated automatically by the PET. Notice that the result is the same in both cases.

To get an idea of jiffy speed and the conversion from the jiffy to the standard clock, type the following program into your PET. This program prints out the running time of both TIME\$ and TIME (TI):

```

5 REM **RUNNING CLOCKS**
10 PRINT "REAL TIME: ":PRINT:PRINT "JIFFY TIME: "
20 FOR I=1 TO 235959
30 PRINT " ";TAB(13);TIME$
40 FOR J=1 TO 60 STEP 2
50 PRINT "000";TAB(12);TI
60 NEXT J
70 NEXT I

```

The FOR...NEXT loop to print TIME (TI) is nested inside the FOR...NEXT loop in TIME\$ because multiple jiffies are incremented and printed for each increment and print of a second shown in TIME\$.

```

20 FOR I=1 TO 235959
30 PRINT " ";TAB(13);TIME$
40 FOR J=1 TO 60 STEP 2
50 PRINT "000";TAB(12);TI
60 NEXT J
70 NEXT I

```

The FOR...NEXT loop for TIME increments by STEP 2 (every two jiffies) for two reasons: 1) the printing of 60 jiffies a second is too fast to read, and 2) the printing of each jiffy takes longer than its incrementations; this would delay the loop, so the printing of TIME\$ is slower than it should be. By incrementing and printing every other jiffy we can minimize this delay problem. Run this program and you will see that jiffies increment to 60 within each second. Run this program without STEP 2 in line 40 and see the time delay when printing TIME\$.

```

REAL TIME: 006704
JIFFY TIME: 25500

```


Keeping time in jiffies is useful for timing program speed to test the efficiency of a program. A short program is listed below:

```

10 PRINT"***KEYBOARD TEST**":PRINT
20 FOR I=32 TO 127
30 PRINT CHR$(I);
40 NEXT I
50 FOR J=161 TO 255
60 PRINT CHR$(J);
70 NEXT J
80 PRINT:PRINT:PRINT"***END TEST**"

```

If we were to time the execution of this program, it would take three steps:

1. TI (or TIME\$) is assigned to a variable constant near the start (or wherever you wish timing to begin).
2. TI (or TIME\$) is reassigned to a different variable constant near the end (or wherever you wish timing to end).
3. Subtract the first TI assigned variable from the second. This will give you the amount of jiffy time it took to process the steps in between.

The listing below shows the three added steps:

```

Step 1 10 PRINT"***KEYBOARD TEST**":PRINT
        15 A=TI
        20 FOR I=32 TO 127
        30 PRINT CHR$(I);
        40 NEXT I
        50 FOR J=161 TO 255
        60 PRINT CHR$(J);
        70 NEXT J
        75 B=TI
Step 2 80 PRINT:PRINT:PRINT"***END TEST**"
Step 3 100 PRINT:PRINT"TI = ";B-A

```

At line 15, variable A is set to the current value of TI.

```

15 A=TI
      A = TI [6001762]
      A [6001762]

```

Then, as the program is processed, TI increments 60 times every second. At line 75, B is set to the current value of TI.

```

75 B=TI
      B TI [6001953]
      B [6001953]

```

Line 100 subtracts the first value of TI (A) from the second (B).

```

100 PRINT:PRINT"TI=";B-A
      B [6001953]
      - A [6001762]
      -----
      191

```

The example shows that it took 191 jiffies to print the keyboard characters on the screen. Dividing jiffy time by 60 (the number of jiffies in a second):

$$191 \div 60 = 3.1833$$

it took 3.1833 seconds or 191 jiffies to process the program. Below is a sample run of the program.

```

**KEYBOARD TEST**

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G
H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` { | } ~ ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾
**END TEST**

TI = 191

```

DIGITAL DISPLAY CLOCK

The following program is a fun program. It is a variation of the PET digital clock using enlarged numbers 0 through 9, created with the graphic characters. It prints out only the hour and minutes due to the size of the clock and the screen, but is very accurate. The program is long, as you can see, but it is made up almost entirely of PRINT statements to print the numbers. After keying in the program, watch it run.

```

100 PRINT "00000000";
110 S=INT(TIME/60)
120 M=INT(S/60)
130 H=INT(M/60)
140 M=M-H*60
150 T=H
160 GOSUB500
170 PRINT "0000 0000 0000 0000 0000 ";
180 T=M
190 GOSUB500
200 PRINT "0000";
210 GOTO110
500 U=T-10*INT(T/10)
510 T=INT(T/10)
520 D=T+1
530 GOSUB600
540 D=U+1
550 GOSUB600
560 RETURN
600 ON D GOSUB 1000,1100,1200,1300,1400,1500,1600,
1700,1800,1900
610 RETURN

```

105 A=T
110 B=T: S=INT((B-A)/60)

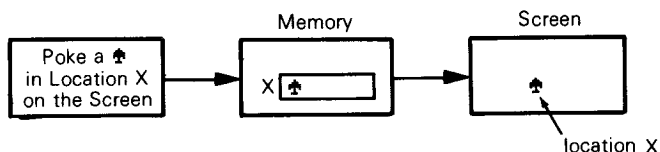

```

1500 PRINT"█";
1501 PRINT"█";
1502 PRINT"█";
1503 PRINT"█";
1504 PRINT"█";
1505 PRINT"█";
1506 PRINT"█";
1507 PRINT"█";
1508 PRINT"█";
1509 PRINT"█";
1510 RETURN
1600 PRINT"█";
1601 PRINT"█";
1602 PRINT"█";
1603 PRINT"█";
1604 PRINT"█";
1605 PRINT"█";
1606 PRINT"█";
1607 PRINT"█";
1608 PRINT"█";
1609 PRINT"█";
1610 RETURN
1700 PRINT"█";
1701 PRINT"█";
1702 PRINT"█";
1703 PRINT"█";
1704 PRINT"█";
1705 PRINT"█";
1706 PRINT"█";
1707 PRINT"█";
1708 PRINT"█";
1709 PRINT"█";
1710 RETURN
1800 PRINT"█";
1801 PRINT"█";
1802 PRINT"█";
1803 PRINT"█";
1804 PRINT"█";
1805 PRINT"█";
1806 PRINT"█";
1807 PRINT"█";
1808 PRINT"█";
1809 PRINT"█";
1810 RETURN
1900 PRINT"█";
1901 PRINT"█";
1902 PRINT"█";
1903 PRINT"█";
1904 PRINT"█";
1905 PRINT"█";
1906 PRINT"█";
1907 PRINT"█";
1908 PRINT"█";
1909 PRINT"█";
1910 RETURN

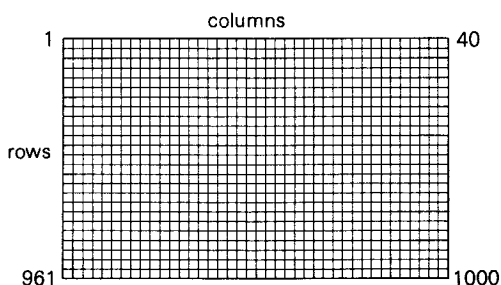
```

POKE TO THE SCREEN

The programming of individual characters anywhere on the screen is most effectively done with the POKE statement. **POKE allows a character to be placed anywhere on the screen by POKEing its character value into a screen location in memory, which POKES it onto the screen.**



The PET screen is like a grid of 1000 squares; 40 columns by 25 rows:



One character may be displayed in each square. **All 1000 screen locations are assigned an address and space in memory. Memory screen space begins at address 32768 for square 1 (row 1, column 1) and ends at address 33767 for square 1000 (row 25, column 40).** Memory address 32768 is screen location (1,1), address 32769 is screen location (1,2), etc. Figure 5-12 shows the correlation between PET screen locations and their corresponding memory space and addresses.

CALCULATING THE SCREEN ADDRESS

To find the screen address in memory for each screen location, manual counting of each square is unnecessary. The following equation is a simpler and more efficient method to calculate the memory address:

$$\text{Memory Address of Screen Location} = 32768 + (\text{column} - 1) + (40 (\text{row} - 1))$$

Enter the column and row numbers of any screen position into the equation to find its memory address. To demonstrate, enter the values 5 and 3 to find the

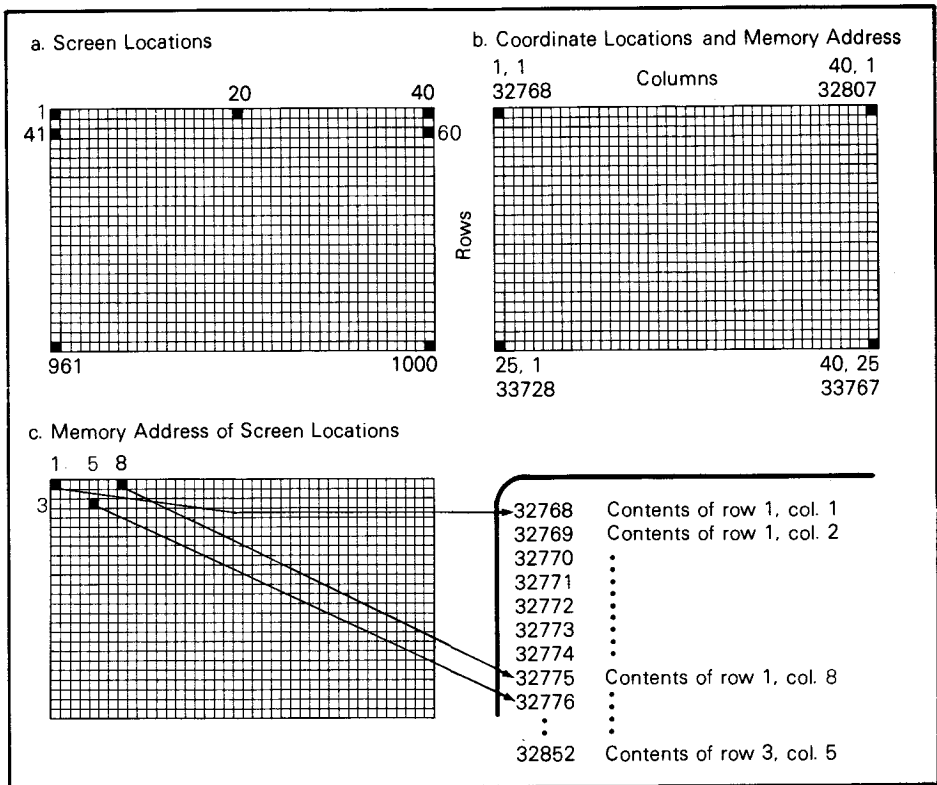
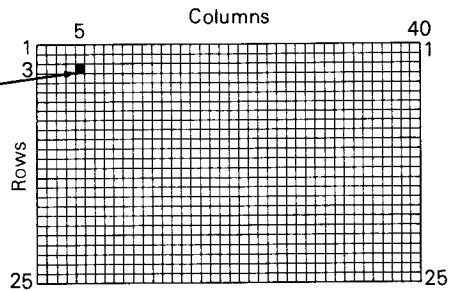


Figure 5-12. Screen Locations and Memory Addresses

memory address for the screen location at column 5, row 3 (5,3):

$$\begin{aligned}
 &= 32768 + (\text{COL} - 1) + (40 \cdot (\text{ROW} - 1)) \\
 &= 32768 + (5 - 1) + (40 \cdot (3 - 1)) \\
 &= 32768 + 4 + (40 \cdot 2) \\
 &= 32768 + 4 + 80 \\
 &= 32852
 \end{aligned}$$



The memory address for screen location (5,3) is 32852.

This equation makes it possible to POKE characters to the screen without knowing any more than the column and row number of the location to be POKEd. Recall the format of the POKE statement:

POKE A,X

where:

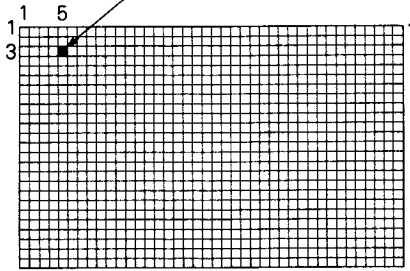
- A is the screen address.
- X is the character or variable to be POKEd into A.

By replacing A with this equation, the PET will calculate the screen address itself.

$$\text{POKE} \underbrace{32768 + (\text{COL} - 1) + (40 * \text{ROW} - 1)}_A, X$$

For instance, if COL (C) and ROW (R) were input as 5,3, and X as ♠ then a spade would be POKEd at screen location (3,5), address 32852.

```
10 INPUT C,R,X
20 POKE 32768+(C-1)+(40*(R-1)),X
    = 32768+(COL-1)+(40*(ROW-1)),X
    = 32768+(5-1)+(40*(3-1)),X
    = 32768+4+(40*2),X
    = 32768+4+80,X
    = 32852,X
```



Although the above equation is useful when you do not know the exact screen address, you should not always incorporate that long equation into your programs. It is much neater and more concise if you use the aforementioned formula to calculate the screen address outside of the program, and then use the screen address using POKE inside of the program:

```
POKE 32852,X
or
A=32852
POKE A,X
```

Variables are acceptable in the POKE command, as long as the variables stay within the proper limits:

1. POKE 32768+A,X where A is a number between 0 and 999 inclusive (32768+999=33767).
2. POKE A,X where A is a number between 32768 and 33767 inclusive.

Using variables to represent the screen address is practicable when POKEing to a repeating sequence of screen spaces. For example, the two programs below POKE the value of X ten spaces apart across the screen. This is a much more efficient method of printing than programming several PRINT statements and cursor movements.

```
10 A=32768
20 POKE A,X
30 A=A+10
40 IF A<=33767 GOTO 20

or: 10 POKE 32768,X
    20 A=A+10
    30 POKE 32768+A,X
    40 IF A<=1000 GOTO 30
```

GENERATING RANDOM NUMBERS

Random numbers are generated on the PET using an algorithm on a starting number, or seed, to begin a sequence of numbers. The same seed always gives the same sequence of numbers.

When the PET is powered up, the initial seed is always the same so that the random number sequence is likewise always the same. The display below shows a typical first five numbers that will appear on referencing the RANDOM function RND(arg) after power-up.

```
### COMMODORE BASIC ###  
  
7167 BYTES FREE  
  
READY.  
  
FOR I=1 TO 5:RND(I):NEXT  
RUN  
.880969862  
.355265655  
.659512252  
.803285178  
.546991144  
READY.
```

You can have the same sequence generated each time (this is very helpful in debugging situations) by starting from the same seed, or you can program the PET to generate random numbers from different sequences by giving different seeds.

To generate the same sequence, select a negative number and all RND with this number before beginning to fetch the random numbers. The following program prints the first five random numbers for seeds -1, -2, -3,... -100.

```

10 FOR I=-1 TO -100 STEP -1
20 X=RND(I):PRINT I
30 FOR J=1 TO 5
40 PRINT RND(1)
50 NEXT J:PRINT:FOR K=0 TO 1000:NEXT K
60 NEXT I
RUN
-1
.735030872
.354388983
.747932106
.16562769
.62863439

-2
.271819872
.14311354
.511223365
.367604656
.148456903

-3
.235981913
.365113894
.905614705
.916307965
.987458745

```

⋮

Note here that the "1" in RND(1) on line 40 can be any positive, non-zero number; it doesn't matter which one. On line 50, the PRINT is to give a blank line; the FOR K loop is to pause between number sets.

Suppose you select -40 as a seed; your program might have statements like the following:

```

10 X=RND(-40): REM START SEED
20 A=RND(1): REM FETCH A RANDOM NUMBER
⋮
150 MX=RND(1): REM FETCH ANOTHER RANDOM NUMBER

```

You probably don't have any use for the seed value assigned to X, but an assignment statement is the easiest way to include a legitimate reference for RND (-arg). Each time statement 10 is executed, it will restart the random number sequence at the same beginning.

To generate different random numbers at each RND use, you need to have some way of generating a random seed. (You can't call RND to do it!) A convenient way of getting a pseudo-random seed on the PET is to use the current jiffy count, TI:

```
10 X=RND(-TI): REM START SEED
```

Here you will get a different random sequence started each time statement 10 is executed.

A more nearly pure random seed* can be obtained by using RND (-RND(0)). For example:

```
10 X=RND(-RND(0)): REM START SEED
```

Here again you will get a different random sequence started each time statement 10 is executed.

In the programs that follow, -TI is used, as it is compatible with both old and new ROMs. If you have the new ROMs, you can use -RND(0) in place of -TI.

You will have to convert the random number, which is returned as a number between 0 and 1, to whatever range you want the random number to be in. Say you want numbers from 1 to 6 (as in one die number of a dice game). You will need to multiply the random number by 6:

```
6*RND(1)
```

This gives a floating point number in a range just greater than 0 but just less than 6 ($0 < n < 6$). Add 1 to get a number in the range $1 < n < 7$:

```
6*RND(1)+1
```

Then convert the number to integer, which discards any fractional part of a number, returning the number to the range 1 to 6 but in integer form:

```
INT(6*RND(1)+1)
```

or

```
RN%=6*RND(1)+1
```

The general cases for converting the RND fraction to whole number ranges are shown below. Note that with the INT function, these formulas handle numbers in the integer range ± 32767 .

$\text{INT}((n+1) \cdot \text{RND}(1))$	range 0 to n
$\text{INT}(n \cdot \text{RND}(1)+1)$	range 1 to n
$\text{INT}((n-m+1) \cdot \text{RND}(1)+m)$	range m to n

The formulas for ranges beginning at 0 or 1 are just simplified versions derived directly from the general case of range m to n:

where m=0	$\text{INT}((n-m+1) \cdot \text{RND}(1)+m)$	
	$\text{INT}((n-0+1) \cdot \text{RND}(1)+0)$	
	$\text{INT}((n+1) \cdot \text{RND}(1))$	as shown above

where m=1	$\text{INT}((n-m+1) \cdot \text{RND}(1)+m)$	
	$\text{INT}(n-1+1) \cdot \text{RND}(1)+1)$	
	$\text{INT}(n \cdot \text{RND}(1)+1)$	as shown above

*according to the PET manufacturer

The -1 to -100 display program shown earlier, which displayed five numbers in the range 0 to 1 for each seed, can be changed so that the random numbers returned by RND will be put into a selected range. The program below is the same as the previous one except that line 40 is changed to put the random numbers into the range 1 to 6; the loop at line 30 has a TO index of 10 to print the first ten numbers of each series.

```

10 FOR I=-1 TO -100 STEP -1
20 X=RND(I):PRINT I
30 FOR J=1 TO 10
40 PRINT INT(6*RND(1)+1)
50 NEXT J:PRINT:FOR K=0 TO 1000:NEXT K
60 NEXT I
RUN
-1
 5
 3
 5
 1
 4
 5
 6
 1
 1
 4
-2
 2
 1
 4
 3
 1
 2
 5
 4
 2
 3
-3
 2
 3
 6
 6
 6
 6
 6
 5
 1
 5

```

As in the original program, the sequence for -1 will always be the same, the sequence for -2 will always be the same, etc. To adjust this program to start at a random seed, change I in line 20 to -TI (or, for the new ROMs, -RND(0) may be used).

The program below shows the use of -TI to generate a random seed. It is the same genre of program as before, but it has been modified to calculate numbers

in the range m to n (as shown by the previous formula). In this program, the values of m and n are set in line 10 for a given program run. Note that these values can be negative. In the following example, the printout is an unending sequence of random numbers between -50 and $+50$. (Press STOP to end the program.) A different sequence of numbers will be printed each time the program runs, since $-TI$ provides a random seed.

Two other changes have been made for the previous program: 1) the X value returned from $RND(-TI)$ is printed instead of the TI value, and 2) the need for the K -variable wait loop is eliminated by printing sets of numbers across the line in continuous line format, which slows the display scrolling compared to single-column printing along the left-hand side of the screen.

```

10 M=-50:N=50
20 X=RND(-TI):PRINT X
30 FOR I=1 TO 8
40 CX=(N-M+1)*RND(1)+M
50 PRINT CX: NEXT I
60 PRINT:GOTO 30
RUN
8.27633085E-06
-14 9 -34 -35 -47 -44 28 31
 29 -8 -36 -28 -42 -28 15 14
 7 -13 3 -8 8 41 19 -43
 35 12 24 -7 -7 -21 -47 1
-32 -49 7 -49 28 -22 -17 -24
-12 7 27 1 11 9 -18 35
 48 49 1 34 -46 -29 -43 29
-18 5 -30 2 8 -28 -13 -23
 48 -15 -12 -45 26 44 -25 2
-9 4 27 50 33 -16 -43 -15
 20 20 17 43 -18 -48 -38 24
-16 43 -50 36 -38 5 11 25
-30 6 -25 -47 32 10 42 -21
-47 -38 -28 -8 16 -20 42 -4
-34 36 -17 27 -8 -49 -6 -35
-19 19 -35 48 -42 36 -25 2
-49 37 47 38 -20 -25 32 -50
-5 -35 -35 17 -41 36 -19 4
 33 -20 45 -7 48 -4 -33 -10
 1 27 -39 -14 -38 -6 4 10
-5 17 2 49 0 -40 -5 32
-50 32 -24 -37 -38 22 -13 -27
-24 -30 35 10 6 16 -50 49
-49 50 43 38 -21 47 -43 28
 32 -35 -18 -5 27 -46 -14 23
-49 -45 27 7 -35 1 46 -25
-8 20 -8 -12 -46 -31 -17 -18
-47 47 -49 18 47 17 40 -13
-40 48 -41 -33 5 -14 -46 45
-29 -37 22 17 42 33 -31 49
 8 -4 36 37 11 18 29 25
 0 -1 2 -16 32 -29 -31 33
-9 -41 -4 47 12 -22 9 -48
-40 32 15 32 -50 3 -9 19

```

To illustrate different number ranges, change the values of M and N in line 10. For example, make M=1 and N=6; this will generate an unending sequence of random numbers between 1 and 6.

A quick scan of the display above shows that numbers repeat within the first 100 generated. That is, every 101 numbers will not sort from -50 to +50 with every number present and no duplications. This is fine in, say, a dice game where you take the rolls as they come. For other random number uses, however, **you may need to develop random numbers in a certain range where every number is accounted for and there are no duplications. An example is sorting a deck of cards.** Here you need to pick a card, and when that card has been picked it cannot be picked again during the same deal.

The program below shows one way to program shuffling a deck of cards on the PET. It fills a 52-element table D%* with the numbers 1 through 52 in a random sequence. The cards can be pegged to the random numbers in any way, such as:

A=1, 2=2, 3=3, ..., Q=12, K=13
Spades=0, Hearts=13, Diamonds=26, Clubs=42

With this scheme the Ace of Spades = 1+0=1, the Queen of Spades=12+0=12, The Three of Hearts=3+13=16, etc.

In the shuffle program, a 52-element flag table FL keeps track of whether a card has been picked or not. PRINT statements are inserted to print out the seed value, followed by the numbers in a continuous-line format. Note that exactly 52 numbers are printed out and that no number is repeated. Each program run will produce a new random sequence.

```

10 DIM FL(52),D%(52)
20 X=RND(-TI):PRINT X
30 FOR I=1 TO 52
40 C%=52*RND(1)+1
50 IF FL(C%)<>0,GOTO 40
60 D%(I)=C%:FL(C%)=1
70 PRINT C%:
80 NEXT I
RUN
  1.18586613E-05
  48 40 13 37 50 43 46 31 49 44
  23 38 25 11 9 35 32 30 24 41
  26 5 6 1 45 10 21 14 42 20 15
  34 18 52 47 7 16 8 19 33 36 4
  17 3 22 27 29 28 39 2 51 12
RUN
  1.01154728E-06
  14 35 52 50 26 48 27 36 34 25
  18 20 41 33 39 7 46 24 23 28 1
  9 3 12 43 2 31 44 4 1 32 37 3
  0 40 22 45 48 42 49 16 11 6 10
  29 9 51 17 8 15 38 5 21 13

```

*Element 0 is not used.

Note that the program runs more slowly as it nears the 52nd number and is especially slow on the last card; this is because it has to fetch more and more random numbers to find one that has not already been picked. A simple routine such as this has much room for improvement, of course. It can be speeded up just by finding the last number in the program from the table rather than randomly.

A further example of RND use illustrates the difference in having a faster selection routine. This program also handles random numbers larger than the integer range allows.

The following program is a modification of program BLANKET, the display program developed in Chapter 3. Instead of printing the display character in continuous-line format, this program fills the screen by randomly POKEing the character into the 1000 positions of the screen.

Here is the first version.

```

10 REM ***** B L A N K E T *****
20 REM RANDOM DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C#="" GOTO 100
105 IF C#=CHR$(13) GOTO 170
110 PRINT" "; :REM CLEAR SCREEN
120 X=RND(-TI) :REM START NEW SEED
125 C=(ASC(C#)AND128)/2 OR (ASC(C#)AND63)
127 A=1000*RND(1)+32768
130 POKE A,C :REM DISPLAY CHAR
140 GET D$:IF D#="" GOTO 127
150 C#=D#
160 GOTO 105
170 END

```

The program is the standard BLANKET program through line 110, inputting a new character and clearing the screen. Line 120 stores a new seed in preparation for a random display sequence on the screen. Line 125 converts C\$ to its equivalent POKE number as in BLANKET Program 10 (see Appendix C). Line 127 calculates a random screen address in the range 32768 to 33767; using the RND range formula with $m=32768$ and $n=33767$, line 127 was derived as follows:

$(n-m+1) \cdot \text{RND}(1) + m$	Range formula
$(33767-32768+1) \cdot \text{RND}(1) + 32768$	as used in line 127
$=1000 \cdot \text{RND}(1) + 32768$	

Neither the INT function nor an integer variable (which would have been A%) can be used, because the screen addresses begin just beyond the maximum integer value of 32767. Fortunately the POKE function, which is where the screen addresses will be used, simply discards any fractional portion of a real number address presented to it. (For other applications when you are dealing with random numbers outside the integer range, you will have to check that the floating point equivalent provides the intended range.)

The first version of the program above randomly fills the screen with the keyed-in character. It does this by simply POKEing to random screen locations. It may POKE many times to the same location when other locations are not yet filled, and it continues to POKE even after the screen is filled until a new character is keyed in.

When the program is run, about half the screen positions quickly fill with the character. Then character placement slows down more and more until at the end, when the screen is almost filled, and remaining positions are filled very slowly. It takes about three minutes to completely fill the screen with this version of the program.

The program is operating at the same speed throughout, but it doesn't get much work done towards the end because many of the positions that it POKES to are already filled. The program appears to slow down because a POKE (or PRINT) of a character over the same character has no visible effect.

The program can be speeded up a good deal by eliminating the superfluous POKES to screen positions that are already filled. A new version of program BLANKET does this.

Rather than calculating a number in the same range all the time and discarding, or in this case re-POKEing, the duplicate numbers, the new program decreases the range of numbers generated to correspond with the number of items left to operate on. It does this by keeping track of the screen positions remaining to be filled in a table and generating a random number within the range of table indexes yet to be POKEd to. The POKE address itself is gotten from the contents at the table index.

```

5 REM RANDOM VERSION 2
10 REM ***** B L A N K E T *****
20 REM RANDOM DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
70 DIM T(999)
80 GOSUB 200 :REM INITIALIZE TABLE
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"C"; :REM CLEAR SCREEN
120 X=RND(-TI) :REM START NEW SEED
125 C=(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
126 FOR N=999 TO 0 STEP -1
127 A=(N+1)*RND(1) :REM PICK AN ELEM
128 A=T(A)+32768 :REM FORM POKE ADDR
129 TP=T(A):T(A)=T(N):T(N)=TP:REM SWAP ELEMENTS
130 POKE A,C :REM DISPLAY CHAR
140 NEXT N
150 GOTO 100
170 END
199 REM **SUBR TO INITIALIZE TABLE**
200 FOR I=0 TO 999:T(I)=I:NEXT
210 RETURN

```

In this program, lines 5 through 50 are the standard header. The Table T, used to hold the 1000 screen position indicators, is dimensioned in line 70.

At line 80 an initialization subroutine is called that places the numbers 0 through 999 into corresponding elements of Table T. T(0) will contain 0, T(1) will contain 1, ... T(999) will contain 999. The elements would not have to be filled with consecutive numbers since they are to be picked randomly, but this is the easiest way to program the fill loop. In fact, the table will be in order only the first time the program is run after loading. Lines 90 through 125 are exactly the same as in the earlier version.

Lines 126 through 140 form a FOR...NEXT loop for filling the 1000 screen locations with the keyed-in character. Line 127 picks a random table index A% from the remaining unfilled range of 0 to N. The formula $(N+1)*\text{RND}(1)$ is taken from the boxed formulas given earlier. Line 128 forms the POKE address A as the sum of the T table element whose index was picked by line 127, plus the beginning screen memory address of 32768. Line 129 exchanges the picked table element T(A%) with the highest active table element T(N) via a temporary location TP. Line 130 displays the character at the random screen location. The NEXT N at line 140 decrements the pointer N so that the used screen address just swapped into T(N) is not picked again during the current program run.

```

5 REM RANDOM VERSION 2A
10 REM ***** B L A N K E T *****
20 REM RANDOM DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
70 DIM T1(249),T2(249),T3(249),T4(249)
75 T=4 :REM NUMBER OF TABLES
76 N=250 :REM NO OF ELEMENTS
80 GOSUB 200 :REM INITIALIZE TABLES
90 PRINT"HIT A KEY OR <R> TO END";
95 N1=N:N2=N:N3=N:N4=N
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"J"; :REM CLEAR SCREEN
120 X=RND(-TI) :REM START NEW SEED
125 C=(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
126 FOR L=1TO1000 :REM 1 FOR EACH SPOT
127 T%=T*RND(1)+1 :REM PICK A TABLE
128 ON T% GOSUB 300,400,500,600 :REM GO PICK AN ELEMENT
130 POKE A,C :REM DISPLAY CHAR
140 NEXT L
160 GOTO 95
170 END
199 REM **SUBR TO INITIALIZE TABLES**
200 FOR I=0 TO N-1:T1(I)=I:NEXT
210 FOR I=0 TO N-1:T2(I)=I+250:NEXT
220 FOR I=0 TO N-1:T3(I)=I+500:NEXT
230 FOR I=0 TO N-1:T4(I)=I+750:NEXT
240 RETURN
299 REM **SUBROUTINE FOR TABLE T1**

```



```

300 N1=N1-1
305 REM IF EMPTY, GO TO ANOTHER TABLE
310 IF N1<0 THEN ON INT(3*RND(1)+1) GOTO 400,500,600
320 A2=(N1+1)*RND(1) :REM PICK AN ELEM
330 A=T1(A2)+32768 :REM FORM POKE ADDR
340 TP=T1(A2):T1(A2)=T1(N1):T1(N1)=TP :REM SWAP ELEMENTS
350 RETURN
399 REM **SUBROUTINE FOR TABLE T2**
400 N2=N2-1
410 IF N2<0 THEN ON INT(3*RND(1)+1) GOTO 300,500,600
420 A2=(N2+1)*RND(1)
430 A=T2(A2)+32768
440 TP=T2(A2):T2(A2)=T2(N2):T2(N2)=TP
450 RETURN
499 REM **SUBROUTINE FOR TABLE T3**
500 N3=N3-1
510 IF N3<0 THEN ON INT(3*RND(1)+1) GOTO 300,400,600
520 A2=(N3+1)*RND(1)
530 A=T3(A2)+32768
540 TP=T3(A2):T3(A2)=T3(N3):T3(N3)=TP
550 RETURN
599 REM **SUBROUTINE FOR TABLE T4**
600 N4=N4-1
610 IF N4<0 THEN ON INT(3*RND(1)+1) GOTO 300,400,500
620 A2=(N4+1)*RND(1)
630 A=T4(A2)+32768
640 TP=T4(A2):T4(A2)=T4(N4):T4(N4)=TP
650 RETURN

```

This program is not much longer than the first version, yet it is a good deal faster. It fills the screen in approximately 45 seconds (compared to a minimum of three minutes for the first version).

A remaining difficulty, which you will have already noticed if you ran the program, is that there is a noticeable wait while the initialization subroutine runs, before the HIT A KEY message is displayed.

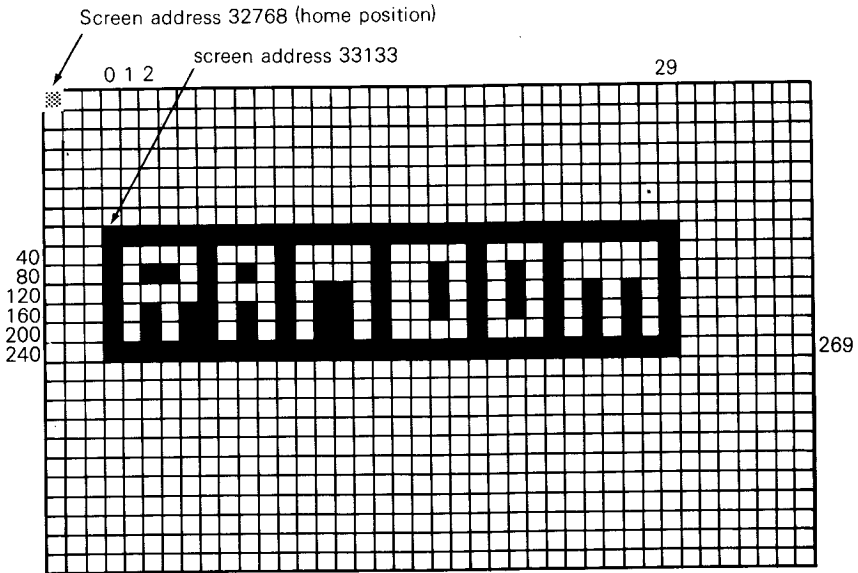
A final modification to the program will show how this kind of program activity can be hidden in a screen display. The following program intersperses displaying a header on the screen with the initialization procedure.

```

5 REM RANDOM VERSION 3
10 REM ***** B L A N K E T *****
20 REM RANDOM DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
70 DIM T(999),H(121)
75 FOR I=0 TO 121:READ H(I):NEXT
76 PRINT"J" :REM CLEAR SCREEN
80 GOSUB 200 :REM INITIALIZE TABLE
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"J"; :REM CLEAR SCREEN
120 X=RND(-TI) :REM START NEW SEED
125 C=(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
126 FOR N=999 TO 0 STEP -1
127 A%=(N+1)*RND(1) :REM PICK AN ELEM
128 A=T(A%)+32768 :REM FORM POKE ADDR
129 TP=T(A%):T(A%)=T(N):T(N)=TP :REM SWAP ELEMENTS
130 POKE A,C :REM DISPLAY CHAR
140 NEXT N
160 GOTO 100
170 END
199 REM **SUBR TO INITIALIZE TABLE**
200 FOR I=0 TO 999 STEP 8
210 FOR J=I TO I+7:T(J)=J:NEXT
220 IF K>121 GOTO 250
230 POKE H(K)+33133,160
240 K=K+1
250 NEXT I
260 RETURN
300 DATA 28,171,7,165,245,0,223,249,54,97,16,19,262,109
,160,183,167,99,248,14
310 DATA 267,264,185,120,189,17,269,29,172,247,9,145,26
5,204,162,25,261,266,137
320 DATA 251,45,207,243,10,257,26,22,85,254,225,242,149
,20,11,18,205,263,129
330 DATA 200,103,229,27,15,12,21,174,268,139,125,101,20
2,24,5,141,1,132,169,63
340 DATA 212,4,8,164,219,256,181,253,23,6,214,3,187,255
,131,177,83,179,240,246
350 DATA 87,143,241,209,82,211,89,258,69,59,134,80,147,
94,2,13,259,260,250,227
360 DATA 244,252,49,40

```

The header is made up of the word RANDOM as blanks outlined by reverse blanks and situated in the middle of the screen. The display is created by clearing the screen and then POKEing the reverse blanks to the appropriate screen positions. There are 122 reverse blanks used in the display. The display is shown below with some of the screen positions relative to the first reverse blank shown.



In the program, at line 70 a second table, H, is dimensioned to hold the relative positions of the 122 reverse blanks. Line 75 fills Table H with the relative position numbers from DATA statements at lines 30 through 360. The order of the numbers in the DATA statements is random but fixed; the header appears to fill randomly to the viewer, but it in fact fills the same way each time. (This is in contrast to the random filling of the 1000 screen positions, which is different each time.) The initialization routine at lines 200 through 260 fills Table T as before, but rather than doing so continuously it breaks every eight elements to POKE a reverse blank from Table H to the screen for the header ($1000 \div 122 = 8$ plus a small remainder). The header POKE address is formed as the sum of the screen position relative to the first reverse blank and the actual memory address of the first reverse blank (33133, where 32768 is the home position address). The reverse blank value for POKEing is 160, as shown in Appendix A.

You can see by running this amended version of the program that there is no longer any apparent delay during program initialization.

System Information

ORGANIZATION OF THE PET SYSTEM

The organization of the PET system is shown in Figure 6-1. The PET is built around the 6502 microprocessor. The display screen, cassette tape unit, and keyboard are physical devices that have been described in Chapter 2. The three external I/O ports are interfaced through the 2K block of memory-mapped I/O. On 4K/8K PETs, the cassette tape unit connects directly to the I/O block, and the Cassette Tape Interface is available for connecting a second cassette unit. On 16K/32K PETs the cassette tape unit is connected through the Cassette Tape Interface; additional tape units, if any are desired, must be interfaced through the IEEE 488 port. The six ROM, RAM, and I/O blocks are allocated from the total 65K memory space available on the PET.

The memory allocation by 4K blocks is shown in Table 6-1. Each portion of the memory is described in more detail in the following text.

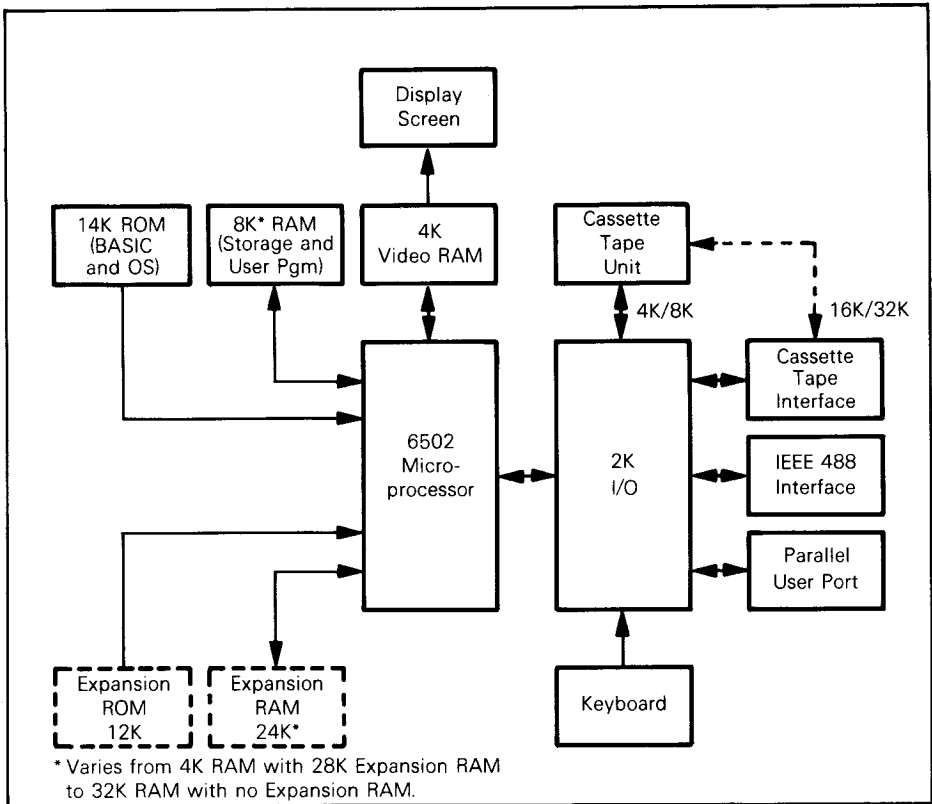


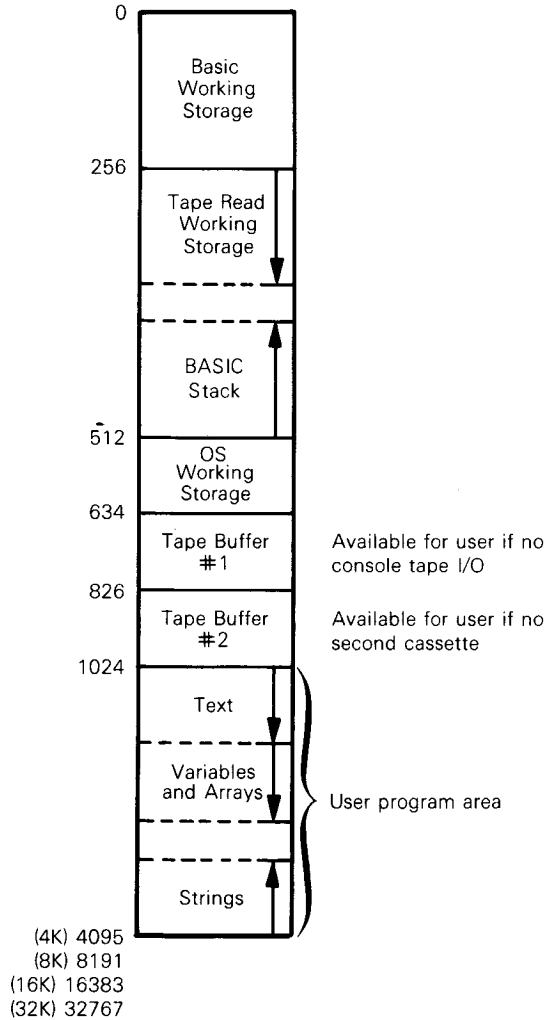
Figure 6-1. PET Block Diagram

Table 6-1. PET Memory Allocation by 4K Blocks

Block	Memory Type	Start Address		Description
		Decimal	Hexadecimal	
0	RAM	0	0000	Working storage, start of text
1	RAM	4096	1000	Text and variable storage (8K only)
2	—	8192	2000	Expansion RAM
3	—	12288	3000	
4	—	16384	4000	
5	—	20480	5000	
6	—	24576	6000	
7	—	28672	7000	
8	RAM	32768	8000	
9	—	36864	9000	Expansion ROM
10	—	40960	A000	
11	—	45056	B000	
12	ROM	49152	C000	BASIC (principally statement interpreter)
13	ROM	53248	D000	BASIC (principally math package)
14	ROM	57344	E000	Screen Editor (2K)
	I/O	59392	E800	I/O Memory (2K)
15	ROM	61440	F000	Operating System (OS)

Addresses 0-8191: 8K RAM (Storage and User Program)

The first block of RAM is allocated to working storage, the stack, tape buffers, and storage of user programs. The amount of active RAM may be 4K (addresses 0-4095), 8K (addresses 0-8191), 16K (addresses 0-16384), or 32K (addresses 0-32767). The first 1K allocation (to 1024) is fixed; the larger the memory size, the more space is available in the user program area.



Locations 0 through 255 are used by the BASIC interpreter as working storage locations. This area is detailed in Table 6-2.

Locations 256 through 511 are comprised mainly of the BASIC Stack. A portion of the area beginning at location 256 and proceeding upward is used by the Tape Read routine for error correction and by BASIC as an expansion buffer. The stack begins at location 511 and proceeds downward. Storage is allocated dynamically as needed by BASIC and the hardware. An OUT OF MEMORY error occurs if the stack pointer reaches the end of available space in this area.

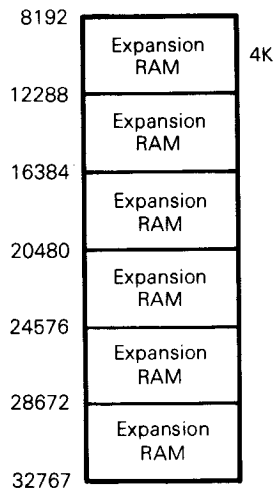
Locations 512 through 633 are used by the OS as working storage locations. This area is detailed in Table 6-2.

Locations 634 through 825 form a 192-byte tape buffer for the console tape cassette. Locations 826 through 1023 form a second 192-byte tape buffer for the optional second cassette unit. User-written assembly language programs can be stored in tape buffer #2 if there is no second cassette used on the system.

Locations 1024 through the end of available RAM are used for storage of the user program and variables. The program begins at location 1024 and is stored upward toward the end of memory. Variable storage begins after the end of the program. Array storage begins at the end of variable storage. Strings are stored beginning at the end of memory and working downward. An OUT OF MEMORY error occurs if an upgoing pointer meets the downgoing pointer.

Addresses 8192-32767: Expansion RAM 24K

Memory addresses 8192 through 32767 are allocated for expansion of RAM to 32K.

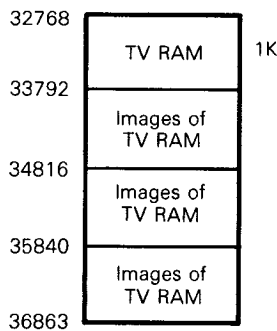


Each PET has 32K of RAM address space, allocated between active RAM and Expansion RAM, as follows:

<u>Active RAM</u>	<u>Expansion RAM</u>
4K (0-4095)	28K (4096-32767)
8K (0-8191)	24K (8191-32767)
16K (0-1638)	

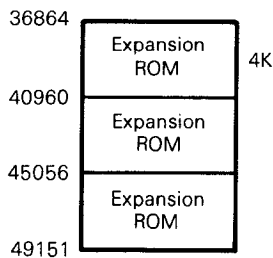
Addresses 32768-36863: 4K Video RAM

The first thousand locations of this block, from addresses 32768 through 33767, are allocated to screen memory. A POKE to any of these locations displays the character in the appropriate screen position.



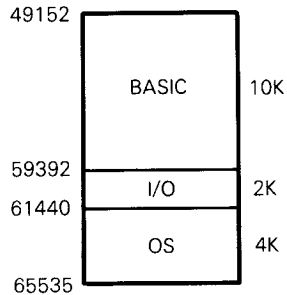
Addresses 36864-49151: Expansion ROM 12K

Memory addresses 36864 through 49151 are allocated for optional expansion of ROM to 26K.



Addresses 49152-65535: 14K ROM and 2K I/O

Locations 49152 through 59391 and locations 61440 through 65535 comprise the BASIC interpreter and OS diagnostics. Memory-mapped I/O locations are from 59392 through 61439.



Location 65535 is the end of PET memory.

MEMORY MAP

The detailed PET memory map is shown in Table 6-2.

The table shows the memory address in decimal and hexadecimal. The decimal value is the PEEK/POKE address (0 to 65535).

A sample value in decimal and its hexadecimal equivalent are also given. With the exception of pointers, these sample values are typical of what you might see if you PEEKed at the location; these are all byte values, in the range 0 to 255 (0-FF₁₆). **A pointer is a two-byte address, in the range 0 to 65535 (0-FFFF₁₆), that is stored in the PET in low-byte, high-byte order.** All two-byte locations in the table contain values stored in low-high order. Consider the first such location in the table:

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
1-2	0001-0002	826	033A	User address jump vector

If you PEEKed at these locations, the 16-bit address would be presented in two parts, first the low-order byte:

```
?PEEK(1)
58
```

and then the high-order byte:

```
?PEEK(2)
3
```

To convert the two values to the appropriate address, you can convert them separately to hexadecimal and then convert the hexadecimal address to decimal:

<u>Low</u>	<u>High</u>	<u>Address</u>
58 ₁₀ =3A ₁₆	31 ₀ =03 ₁₆	→ 033A ₁₆ =826 ₁₀

Note carefully that the sample value 033A means that the first memory byte = 3A and the second (higher) memory byte = 03.

Or you can multiply the high-order byte by 256 and add it to the low-order byte. The following is a PEEK statement that will do this for you:

$$\begin{aligned} &?PEEK(1)+256*PEEK(2) \\ &826 \end{aligned}$$

Conversely, to convert a 16-bit memory address into two separate bytes for POKEing (in low-byte, high-byte order), you can convert the decimal value to hexadecimal and then convert the separated byte digit pairs to decimal. e.g., to convert the address 59409:

<u>High</u>	<u>Low</u>
59409 ₁₀ =E811 ₁₆	→ E8 ₁₆ =232 ₁₀ and 11 ₁₆ =17 ₁₀

Or you can convert using decimal arithmetic by first dividing the address value by 256 and discarding any fractional remainder:

$$\begin{aligned} &\text{High} \\ &59409/256=232.06641=232 \end{aligned}$$

Then subtract "High"*256 from the original value (59409 in this case) to get the remainder, which is the low-order byte value:

$$\begin{aligned} &232*256=59392 \\ &59409 - 59392=17 \\ &\text{Low} \end{aligned}$$

(Of course, if you do the division by longhand, the remainder is directly available.)

For a block of byte locations, only the first byte value is shown in the table.

The column labeled Description in the table gives a short description of the location's use. There are multiple uses for some locations, in which case the primary one is indicated.

While not exhaustive, the table illustrates the overall makeup of the PET memory.

PET BASIC INTERPRETER

The PET BASIC interpreter executes a user program by decoding each source line stored in memory in a compacted form. When you enter a line from the keyboard, the PET Line Editor has control, allowing you to do any editing of the line until you press the RETURN key. Program lines are stored in memory in ascending line number order. When the RETURN key is struck, the BASIC interpreter searches memory for the same line number. If there is one, it replaces the current line with the new line. If there isn't one, the next higher line number is encountered. The BASIC interpreter then inserts the new line into memory and moves the rest of the program up.

Program lines are stored at the beginning of the user program area of memory, which starts at memory location 1024. Variables are stored in memory above the program lines, and arrays are stored above the variables. All three areas begin at lower addresses and build upwards to higher addresses. Strings are stored beginning at the top of memory and work downwards. The BASIC interpreter builds all four areas, moving them as necessary and adjusting pointers for insertions and deletions. Eight pairs of memory locations contain pointers to the division points in the user program area of memory. These are shown in Figure 6-2. (They are also listed in Table 6-2.)

The formats in which BASIC statements, variables, arrays, and strings are stored in their respective areas are discussed next.

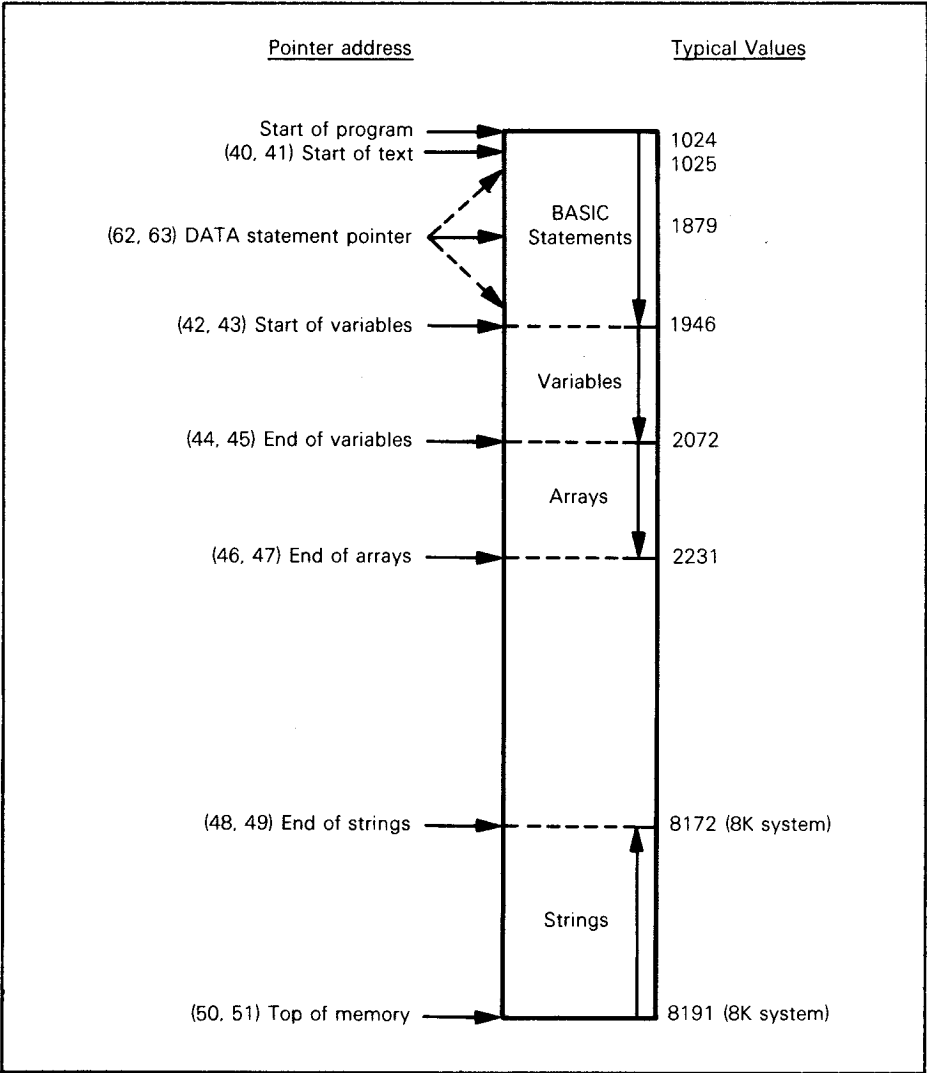


Figure 6-2. Principal Pointers in User Program Area

BASIC STATEMENT STORAGE

BASIC statements are stored in the format shown in Figure 6-3.

Memory location 1024 always contains a zero byte.

The next two bytes contain a pointer to the beginning of the first BASIC statement. The pointer, like all addresses in the PET, is stored in low-byte, high-byte order. The pointer is a link to the memory address of the next link. **A link address of zero denotes the end of the text;** i.e., there are no more links and no more statements. BASIC statements are stored in order of ascending line numbers, even though there are links to the next statement. Links are used to quickly search through line numbers.

Following the link address is the line number of the statement, stored in low-byte, high-byte order. Line numbers go from 1 (stored as 1 and 0) to 63999 (stored as 255 and 249).

After the line number, the BASIC statement text begins. Keywords are comprised of reserved words (listed in Table 3-2) and operators (listed in Table 3-4). Reserved word and logical operator keywords are stored in a compressed format. A one-byte token is used to represent a keyword. All keywords are encoded such that the high-order bit is set to 1. Other elements of the BASIC text are represented by their stored ASCII code. Other elements are comprised of constants, variable and array names, and special symbols other than operators and are coded just as they appear in the original BASIC statement. **Table 6-3 shows the byte codes for all values from 0 to 255 that may appear in the compressed BASIC text. Codes are interpreted according to this table except after an odd number of double quotation marks enclosing a character string; within a character string the standard ASCII codes prevail (see Appendix A).**

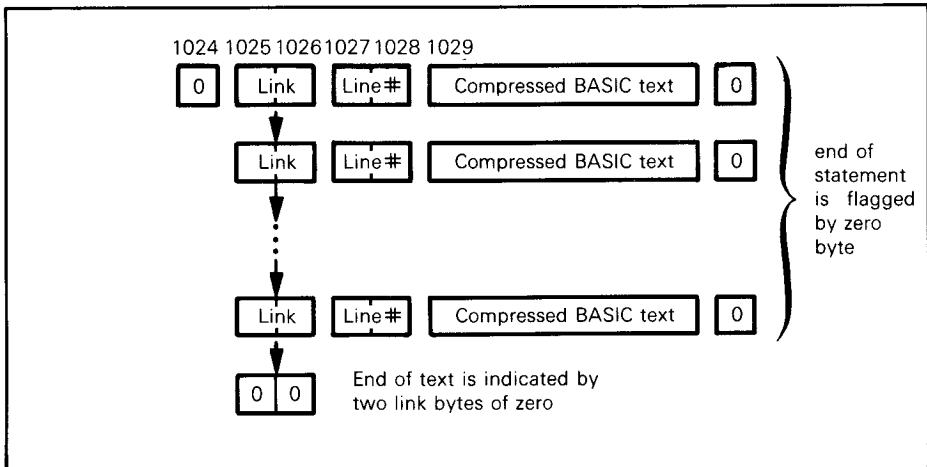


Figure 6-3. BASIC Statement Storage

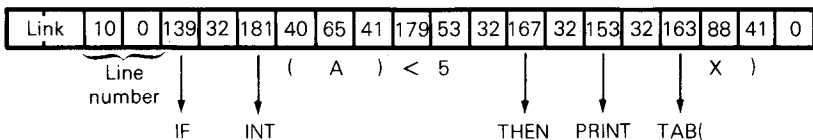
Table 6-3. PET BASIC Keyword Codes

Code (decimal)	Character/Keyword	Code (decimal)	Character/Keyword	Code (decimal)	Character/Keyword	Code (decimal)	Character/Keyword
0	End of line	66	B	133	INPUT	169	STEP
1-31	Unused	67	C	134	DIM	170	+
32	space	68	D	135	READ	171	-
33	!	69	E	136	LET	172	*
34	"	70	F	137	GOTO	173	/
35	#	71	G	138	RUN	174	↑
36	\$	72	H	139	IF	175	AND
37	%	73	I	140	RESTORE	176	OR
38	&	74	J	141	GOSUB	177	>
39	'	75	K	142	RETURN	178	=
40	(76	L	143	REM	179	<
41)	77	M	144	STOP	180	SGN
42	*	78	N	145	ON	181	INT
43	+	79	O	146	WAIT	182	ABS
44	,	80	P	147	LOAD	183	USR
45	-	81	Q	148	SAVE	184	FRE
46	.	82	R	149	VERIFY	185	POS
47	/	83	S	150	DEF	186	SQR
48	0	84	T	151	POKE	187	RND
49	1	85	U	152	PRINT #	188	LOG
50	2	86	V	153	PRINT	189	EXP
51	3	87	W	154	CONT	190	COS
52	4	88	X	155	LIST	191	SIN
53	5	89	Y	156	CLR	192	TAN
54	6	90	Z	157	CMD	193	ATN
55	7	91	[158	SYS	194	PEEK
56	8	92	\	159	OPEN	195	LEN
57	9	93]	160	CLOSE	196	STR\$
58	:	94	↑	161	GET	197	VAL
59	;	95	←	162	NEW	198	ASC
60	<	96-127	Unused	163	TAB(199	CHR\$
61	=	128	END	164	TO	200	LEFT\$
62	>	129	FOR	165	FN	201	RIGHT\$
63	?	130	NEXT	166	SPC(202	MID\$
64	@	131	DATA	167	THEN	203-254	Unused
65	A	132	INPUT#	168	NOT	255	π

Note that the left parenthesis is stored as part of the one-byte token for the functions TAB and SPC, but that the other functions use a separate byte for this symbol. For example, the line

10 IF INT(A) < 5 THEN PRINT TAB(X)

would be coded as the following bytes (in decimal):



The operators (the symbols +*/|<=> as well as the words AND, OR, and NOT) are given keyword codes (high-order bit set) since they "drive" the BASIC interpreter just as reserved words do (e.g., 179 for <). The standard ASCII codes for these symbols (e.g., 60 for <) appear only in the text of a string.

Spaces in the source line are stored except for the space between the line number and first keyword. This space is supplied on LISTing when a stored statement is expanded to its original form. **You can conserve memory storage space by eliminating blanks (but this makes the program harder to read). You can also conserve space by putting more than one statement on a line, since the five bytes of link, line number, and 0 end byte are stored only once.**

The size of each statement is variable and is terminated by a byte of zero to indicate the end of the statement. (A value of zero anywhere within the text is stored as 48.) Zero byte flags are used by the BASIC interpreter in executing a program when it goes through the compressed BASIC text from left to right picking out keywords and performing the indicated operations. A zero byte indicates the end of the statement; the next four bytes are the link and the line number of the next statement. In contrast to searching through the text and using 0 byte indicators to locate the next statement, links are used when searching the statements for their line numbers. Three consecutive bytes of zero (the last statement's 0 byte followed by two zero link bytes) flag the end of text when executing the program.

A program is stored onto cassette tape in the same format as shown in Figure 6-3 for memory storage. Thus, it is basically "dumped" onto tape in a continuous block, including link addresses and 0 end bytes.

The use of tokens in place of keywords is not unique to the PET, but there is no standard coding from one interpreter to another. Thus, **a BASIC source program SAVED on tape by PET BASIC is not compatible with other BASICs, nor can BASIC programs generated on other (non-PET) machines normally be loaded by the PET BASIC interpreter.**

USER PROGRAM AREA INITIALIZATION

On power-up, the user program area of memory is initialized to “+” characters (code 170) except for the first few beginning locations at 1024 on. Location 1024 is zero. The initial link in locations 1025 and 1026 is zero. The pointers into the user area are initialized as shown in Figure 6-4.

As lines are entered and edited and new programs loaded, the contents of memory locations throughout the user program area change. They change, however, **only as necessary for the current program.** The user area is not continuously reinitialized (to “+” or any other code). It is the pointers into the user area that determine the extents of the current program, if any. The action of a NEW command is simply to readjust the pointers to the initial values shown in Figure 6-4. A CLR does the same thing except that it adjusts the variable and array pointers from the end of the program rather than the start of the program as NEW does. (In fact, **if you have accidentally cleared the program or variables, you can reinstate them by “reading” through the user program area as needed and restoring the pointer values.**)

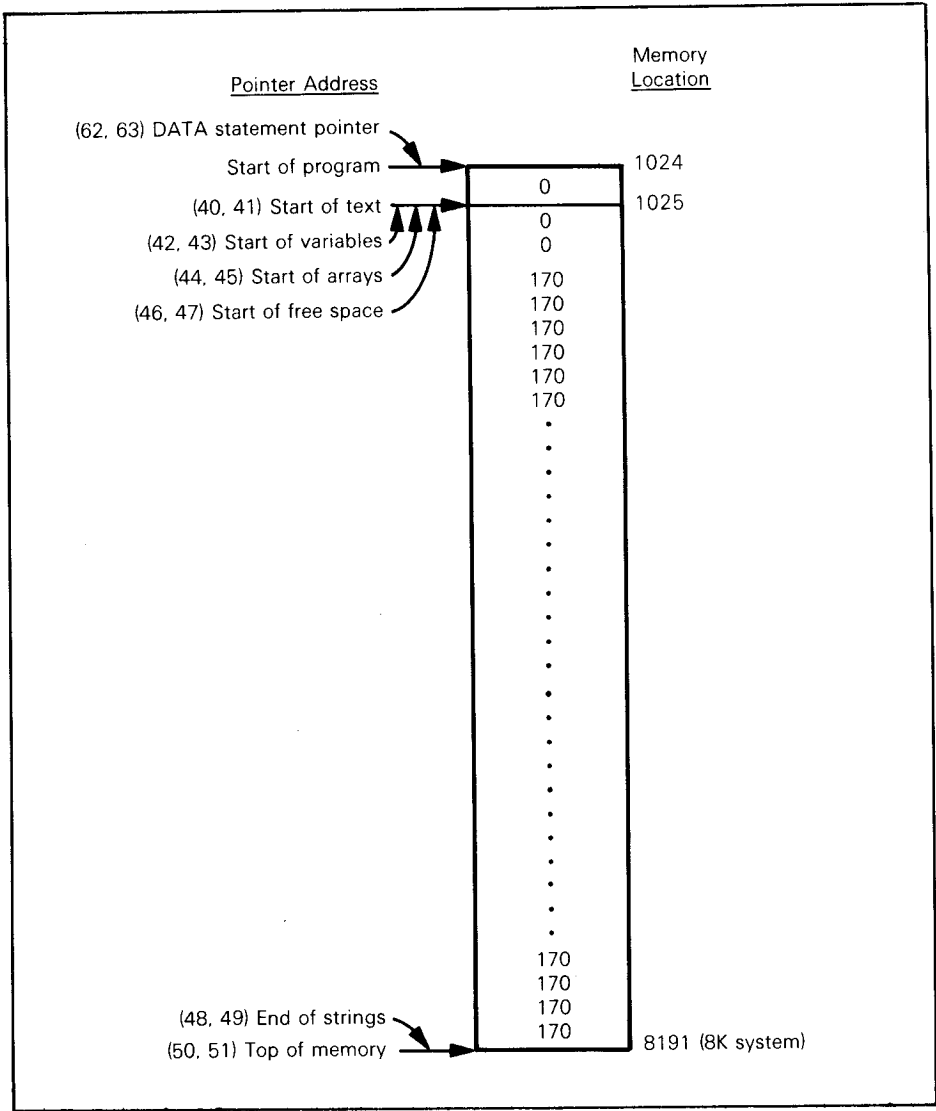


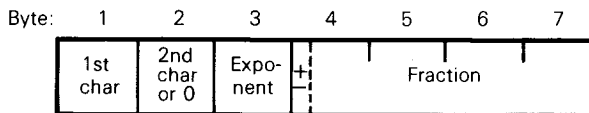
Figure 6-4. User Program Area on Power-Up

DATA FORMATS

Variables

Variables are stored in the Variable Area of user program memory (see Figure 6-2). These are simple (unsubscripted) variables; arrays are stored in a separate area. The variables may be floating point, integer, or string and are freely intermixed in the Variable Area. Each variable, regardless of its type, occupies seven bytes of memory. The first two bytes contain the variable name, and the remaining five bytes further define the variable. Variables are entered into the variable table as they are encountered during execution of the user program. A variable that is not in the table is assumed to have a value of zero for numeric variables or null for a string variable.

Floating Point Variable Format



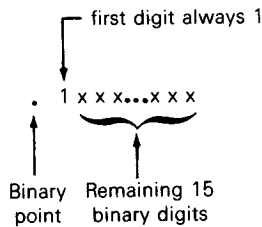
Byte 1 contains the first character of the variable name. Byte 2 contains the second character of the variable name or, if there is no second character, byte 2 contains a zero. The characters are stored in standard ASCII codes (see Appendix A). For example, the name A is stored as 65, 0 whereas the name A0 is stored as 65, 48. A floating point variable is denoted by variable names having stored ASCII values of 90 or below.

Bytes 3 through 7 contain the value of the floating point variable in standard normalized, excess 128 format. Byte 3 contains the exponent in excess 128 format. The exponent determines the magnitude of the number. In excess 128 format, 128 is added to the true exponent (after normalization of the significant digits) so that the smallest exponent representation contains all zeros. The largest exponent representable contains all ones. A true exponent of zero is represented by an exponent value of 128 (0+128). Excess 128 format eliminates having to consider a sign in the exponent.

Examples:

<u>Exponent</u>	<u>Approximate Value</u>
255	10^{38} (maximum exponent)
162	10^{10}
127	10^{-1}
2	10^{-38}
0	10^{-39} (minimum exponent—number is zero)

Bytes 4 through 7 contain the significant digits of the number. The number is normalized such that the binary point is to the immediate left of the first non-zero binary digit. That is, it is represented as a fraction in the form:



The binary point is always assumed and is not stored. Further, **the most significant 1 digit is always assumed** (since it is always 1) **and** is not stored either. **Its bit position is used to hold the sign of the number**, 0=positive and 1=negative. To normalize a number, the point is moved to the left and the exponent decremented (smaller numbers), or the point is moved to the right and the exponent incremented (larger numbers), until the number is a fraction in the form shown above. **The number zero is generally represented by** all zeros in bytes 3 through 7, but the fraction may contain roundoff errors; **an exponent of zero** is sufficient to make the number zero.

Some examples of floating point number representations stored in the Variable Area follow. 1E+38 has the maximum exponent of 255. This decreases down to zero as the numbers decrease to zero. Fractional floating point numbers (e.g., .5, .01, .006) have exponents below 129. For negative numbers, the exponent increases from 0 to 255 as the absolute value of the numbers increases. In byte 4 the high-order bit is the sign bit. In this column, decimal numbers ≤ 127 have bit 7=0 (positive numbers), and decimal numbers higher than this have bit 7=1 (negative numbers).

Byte:	3	4	5	6	7
Number	Exponent	±MSB	Fraction		LSB
1E+38	255	22	118	153	83
1E+10	162	21	2	249	0
1000	138	122	0	0	0
1	129	0	0	00	
.01	122	35	215	10	62
1E-4	115	81	183	23	90
1E	62	60	229	8	101
1E-39	0	32	0	0	0
0	0	0	0	0	0
-1	129	128	0	0	0
-1000	138	250	0	0	0
-1E+10	162	149	2	249	0
-1E+38	255	150	118	153	83

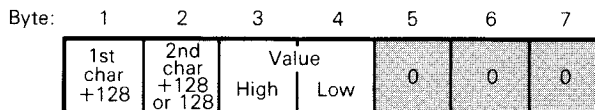
The following short program allows you to examine floating point representations for any numbers. Line 10 inputs a number that you enter from the keyboard, terminating with a RETURN key. Line 20 points to the beginning of variables +2 to go past the two-byte variable name. Line 30 prints the number that was input, followed by the five bytes PEEKed from the variable table. The program is continuous; to end, enter a null line (RETURN key only).

```

10 INPUT A
20 X=PEEK(43)*256+PEEK(42)+2
30 PRINT A;"="";PEEK(X);PEEK(X+1);PEEK(X+2);PEEK(X+3);PEEK(X+4)
40 GOTO 10

```

Integer Variable Format

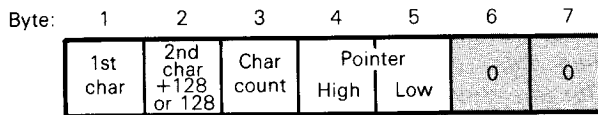


Byte 1 contains the first character of the variable name shifted (+128). Byte 2 contains the second character of the variable name shifted (+128), or if there is no second character, byte 2 contains 128. An integer variable is denoted by variable names having ASCII values of 176 or higher. The % notation is dropped from the variable name. Bytes 3 and 4 contain the value of the integer in high-byte, low-byte order. (Note that this value is not an address and does not conform to the reverse standard for pointers). The value is stored in **twos complement format** so that the high-order bit (bit 7 of byte 3) represents the sign, 0=positive, and 1=negative. The remaining three bytes are not used and are set to zeros.

The following are some examples of integer representations stored in the Variable Area. You can use the same 4-line program to look at integer number representations after changing A to A% in lines 10 and 30.

Byte:	3	4	
Number	32767	127 255	(256*127+255 = 32767)
	32766	127 254	
	14000	54 176	
	256	1 0	
	255	0 255	
	1	0 1	
	-1	255 255	(FFFF ₁₆)+1 = 1
	-2	255 254	
	-32766	128 2	
	-32767	1281 1	

String Variable Format



Byte 1 contains the first character of the variable name. **Byte 2** contains the second character of the variable name shifted (+128), or if there is no second character, the second byte contains 128. This combination of ASCII ranges denotes a string variable entry. The \$ notation is dropped from the variable name. **Byte 3** contains a count of the number of characters in the string (1 to 255). This is the value fetched for the LEN function. **Bytes 4 and 5** contain a pointer to the beginning of the string itself, stored elsewhere in memory. This pointer is in the standard 6502's low-byte, high-byte order. The remaining two bytes are not used and are set to zeros.

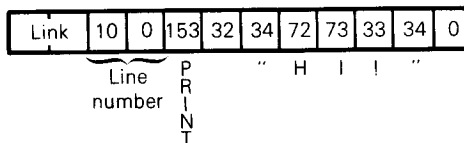
String shortage is optimized by using the copy of the string already in memory if there is one. If there is not, a string is created and stored in the String Area in upper core. A few examples are given below.

Constants

Constants are stored in the BASIC statement itself. They are not placed into a separate area of memory, and they are not stored in the Variable Area. **Floating point and integer and string constants are stored as ASCII character source codes**, as described previously under BASIC Statement Storage. For example, the line:

```
10 PRINT "HII"
```

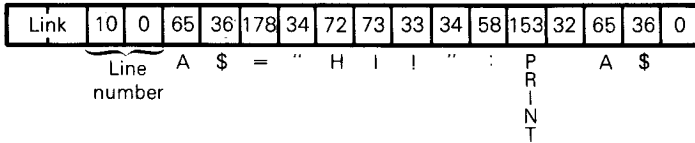
is stored **entirely in the BASIC Statement Area**, in the form:



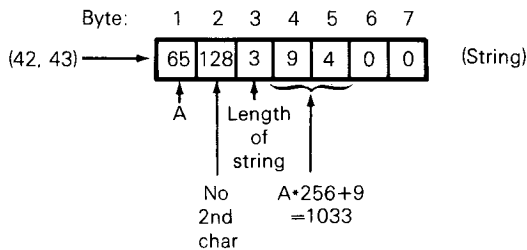
whereas the statement

```
10 A$="HI!":PRINT A$
```

is stored in two areas. The original statement is stored in the BASIC Statement Area:



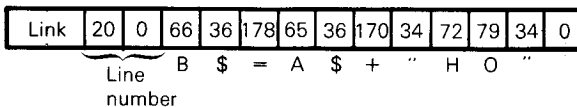
In addition, when this statement is executed the following entry is made in the Variable Area:



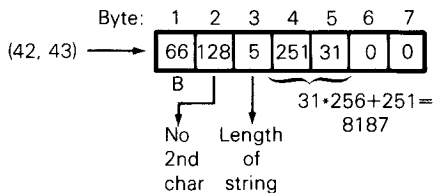
The string in the BASIC Statement Area is pointed to (beginning at memory location 1033 in this program) rather than storing a copy of it in uppercore. However, when you create a new string, as in:

```
20 B$=A$+"HO"
```

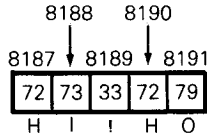
the BASIC Program Area entry is:



and the entry in the Variable Area is:

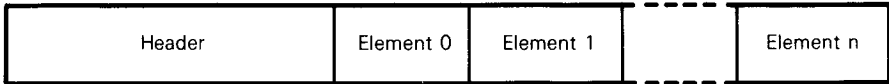


This time the pointer addresses a location in upper core (8187 in this program) that contains the string:



Array Storage Format

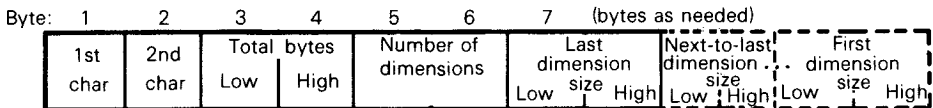
Arrays are stored in the Array Area of user program memory (see Figure 6-2). Arrays may be floating point, integer, or string and are stored in the order in which they are created by the program. The type of array is distinguished by the way in which the two-character array name is stored, exactly the same as for variable names. An array is stored with a header followed by the elements of the array, in the general form:



(Elements are stored in reverse order for strings.)

All types of arrays have the same header format. The header contains seven bytes plus two bytes for each array dimension beyond 1.

ARRAY HEADER

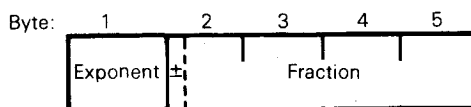


Like floating point variables, floating point array elements each have five bytes. However, for integers the unused three bytes, and for strings the unused two bytes, are dropped for array elements.

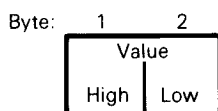
In the array header, bytes 1 and 2 contain the array name. Bytes 3 and 4 contain a count of the number of memory locations that the array occupies. For example, A(0) would occupy 12 bytes: 7 for the header and 5 for the single element. The byte count is stored in low-byte, high-byte order. Byte 5 contains a count of the number of dimensions in the array. As examples, A(5) has one dimension (byte 5 = 1) and A(10,10,2) has three dimensions (byte 5 = 3). For a one-dimensional array (or vector), bytes 6 and 7 contain the dimension size — that is, the number specified between parentheses in the DIM statement + 1. For example, the dimension size = 61 for DIM A(60), = 101 for DIM A(100), etc. If the array does not appear in a Dimension statement, the dimension size defaults to 11. The dimension size is stored in low-byte, high-byte order. For a multiple dimension array, the header contains additional bytes in which are stored all the dimension sizes. Two additional bytes are used for each additional dimension. The dimension sizes are stored in reverse order from the order in which they appear in the DIM statement. For example, for DIM A(10,5) the dimension sizes are stored as bytes 6,7=6 and bytes 8,9=11. For DIM X(2,1,3) the dimension sizes are stored as bytes 6,7=4, bytes 8,9=2 and bytes 10,11=3.

The formats of array elements for each type of array are shown below. They are represented exactly as described for variables (without the unused bytes for integers and strings).

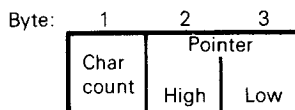
Floating Point Array Element Format



Integer Array Element Format



String Array Element Format



The size of the header may be calculated as five bytes plus two times the number of dimensions in the array. The total size of the elements may be calculated as the number of bytes per element (5 for floating point, 2 for integer, 3 for string) times the number of elements (the dimensions multiplied together + 1). The total size of the array, header plus elements, is stored in byte 4 of the array header.

The following is a program for viewing sample Array Area entries:

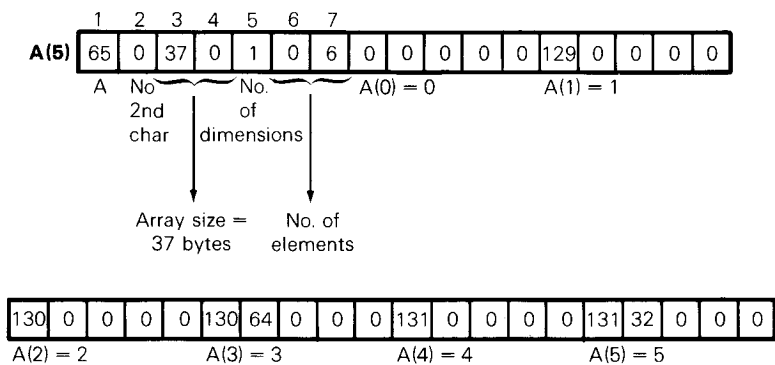
```

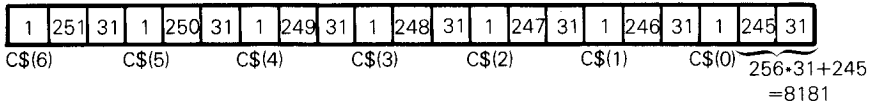
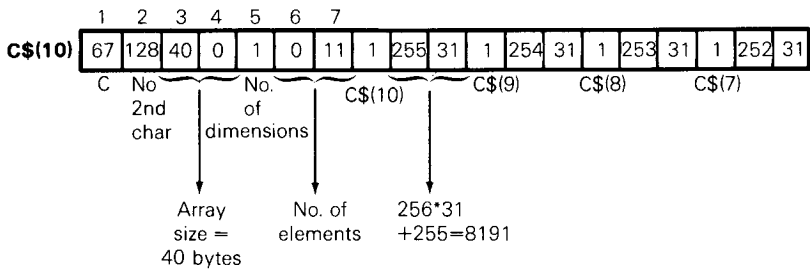
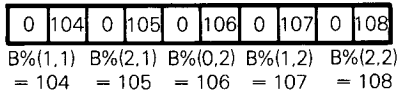
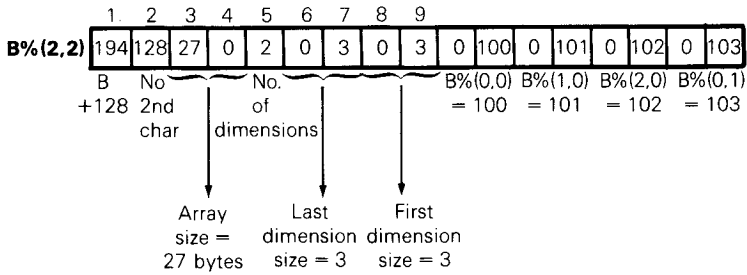
10 DIM A(5), B%(2,2), C$(10):REM SAMPLE ARRAYS
20 FOR I=0 TO 5: A(I)=I:NEXT
30 FOR I=0 TO 2:FOR J=0 TO 2:B%(J,I)=100+3*I+J:NEXT J,I
40 FOR I=0 TO 10:C$(I)=CHR$(ASC("A")+I):NEXT
50 X=PEEK(45)*256+PEEK(44):REM POINT TO ARRAY AREA
60 Y=PEEK(47)*256+PEEK(46):REM END OF ARRAYS
70 FOR I=X TO Y
80 PRINT I, PEEK(I)
90 GET D$:IF D$=""GOTO 90
100 NEXT

```

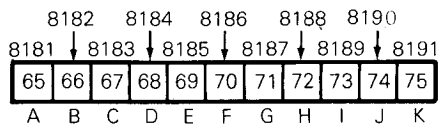
Each of the three types of arrays is dimensioned. Line 20 fills the floating point array A with the numbers 0 through 5. Line 30 fills the integer array B% with the numbers 100 through 108. Line 40 fills the character array C\$ with the single strings A through K. Lines 50 and 60 fetch the pointers to the end of the Variable Area and the end of the Array Area. Lines 70 to 100 print the address followed by the byte value in that address within the Array Area. Printing shops at each memory location; to print the next location, press any key (e.g., the RETURN key). You will need to locate the beginning of the arrays by the sequence for the first array shown below (the pointer addresses the end variable). **The memory locations will appear as shown below.**

Array Area



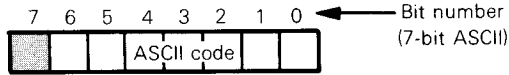


String Area



Character Representation

ASCII (American Standard Code for Information Interchange) is a widely used code for representing character data. It is nominally a 7-bit code, allowing 128 characters ($7F_{16}=128_{10}$) to be represented. The standard ASCII 7-bit character set is shown in Table 6-4. Bits are numbered from 0 (least significant bit) to 6 (most significant bit):



The first 32 codes are reserved for non-printable control characters, intended for message formatting and print format control.

On the PET, characters are stored in ASCII format but in an extended version using eight bits. With eight bits normally available, rather than just seven, up to 255 characters can be represented. Within compressed BASIC text the 8-bit character codes are interpreted as shown in Table 6-3, where the eighth bit signifies a keyword. Elsewhere in main memory the 8-bit character codes are interpreted as shown in Appendix A.

The screen memory, occupying memory locations 32768 through 33767, uses a different ASCII character representation from main memory. It is a 7-bit code as shown in Figure 6-5. The eighth bit is a normal/reverse field indicator. Note that the characters are arranged such that bits 0 through 5 represent one key on the PET keyboard, with bit 6=0 being the unshifted character and bit 6=1 being the shifted character of the same key:

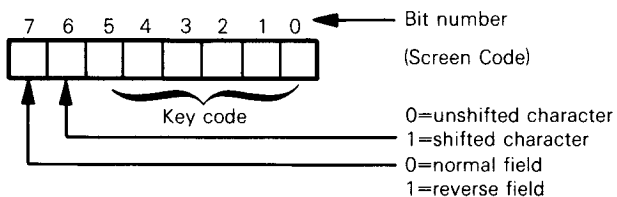
Table 6-4. ASCII Standard 7-Bit Codes

Bit →				6	0	0	0	0	1	1	1	1
3	2	1	0	5	4	3	2	1	0	0	1	1
0	0	0	0	NUL	DLE	SP	0	@	P	.	p	
0	0	0	1	SOH	DC1	!	1	A	Q	a	q	
0	0	1	0	STX	DC2	"	2	B	R	b	r	
0	0	1	1	ETX	DC3	#	3	C	S	c	s	
0	1	0	0	EOT	DC4	\$	4	D	T	d	t	
0	1	0	1	ENQ	NAK	%	5	E	U	e	u	
0	1	1	0	ACK	SYN	&	6	F	V	f	v	
0	1	1	1	BEL	ETB	'	7	G	W	g	w	
1	0	0	0	BS	CAN	(8	H	X	h	x	
1	0	0	1	HT	EM)	9	I	Y	i	y	
1	0	1	0	LF	SUB	*	:	J	Z	j	z	
1	0	1	1	VT	ESC	+	;	K	[k	{	
1	1	0	0	FF	FS	,	<	L	\	l		
1	1	0	1	CR	GS	-	=	M]	m	~	
1	1	1	0	SO	RS	.	>	N	^	n	~	
1	1	1	1	SI	US	/	?	O	_	o	~	DEL

NUL	Null	DC1	Device control 1
SOH	Start of heading	DC2	Device control 2
STX	Start of text	DC3	Device control 3
ETX	End of text	DC4	Device control 4
EOT	End of transmission	NAK	Negative acknowledge
ENQ	Enquiry	STN	Synchronous idle
ACK	Acknowledge	ETB	End of transmission block
BEL	Bell, or alarm	CAN	Cancel
BS	Backspace	EM	End of medium
HT	Horizontal tabulation	SUB	Substitute
LF	Line feed	ESC	Escape
VT	Vertical tabulation	FS	File separator
FF	Form feed	GS	Group separator
CR	Carriage return	RS	Record separator
SO	Shift out	US	Unit separator
SI	Shift in	SP	Space
DLE	Data line escape	DEL	Delete

Table 6-5. PET Screen Memory 7-Bit Codes

B	01	0	0	0	0	1	1	1	1
1	51	0	0	1	1	0	0	1	1
1	41	0	1	0	1	0	1	0	1
3210	1								
0000	1	@	P		0	-			┌
0001	1	A	Q	!	1	↑	•	■	└
0010	1	B	R	"	2	┌	▼	■	┐
0011	1	C	S	#	3	└	▼	■	┘
0100	1	D	T	\$	4	┌		└	
0101	1	E	U	%	5	└		┐	
0110	1	F	V	&	6	┌	/	└	
0111	1	G	W	'	7	└	X	┐	
1000	1	H	X	(8	┌	U	└	┐
1001	1	I	Y)	9	└	U	┐	┘
1010	1	J	Z	*	:	┌	+	└	┐
1011	1	K	L	+	;	└	+	┐	┘
1100	1	L	\	,	<	┌	+	└	┐
1101	1	M]	-	=	└	+	┐	┘
1110	1	N	↑	.	>	┌	/	└	┐
1111	1	O	←	/	?	└	/	┐	┘



The complete character set for screen memory is shown in Appendix A under the PEEK/POKE column.

The screen memory ASCII code may be derived from the main PET ASCII code by moving bit 7 of the main code over into bit 6 and dropping the previous value of bit 6. The examples below illustrate the four cases of a 0 or 1 in bit 7 going into a 0 or 1 in bit 6:

<u>Character</u>	<u>Main Memory Representation</u>	<u>Screen Memory Representation</u>
A	01000001	00000001
Shifted A (♠)	11000001	01000001
1	00110001	00110001
Shifted 1 (☒)	10110001	01110001

When PRINTing to the screen, the PET automatically makes the conversion to screen codes. Only when you are PEEKing and POKEing in screen memory do you need to be concerned about the character set differences.

Screen memory can be considered to have an additional "bit" that represents the alternate character set by a POKE 59468,14 as described in Chapter 5. POKE 59468,12 restores the standard set. The alternate set is also shown in Appendix A.

ASSEMBLY LANGUAGE PROGRAMMING

PET BASIC provides minimal assistance to the user who wishes to write and execute programs written in 6502 assembly language. Assembly language programs execute faster and require less memory space for a given function than the equivalent BASIC program. **You might want to write an assembly language program to be run on the PET if:**

1. The operation is not fast enough using a BASIC program
2. The operation cannot be implemented in PET BASIC
3. The operation takes up too much memory space as BASIC program
4. Assembly language lends itself better to the task than the BASIC language. Some I/O operations probably fall into this category.

An assembly language program can be loaded into PET memory by POKEing the decimal values of the 6502 instructions that make up the program. There is no area set aside for use by assembly language programs. You have to make space, either by taking otherwise unused system locations or by setting up a space in the user program area of memory. The following are possible locations:

1. **Cassette Buffers.** If you do not have a second cassette unit, then the 192-byte tape buffer for cassette #2 can be used to store an assembly language program. The buffer #2 extents are locations 826 to 1017 (see Table 6-2). In addition, if the console cassette unit is not going to be used while the assembly language program is operating, then the other 192-byte tape buffer for cassette #1, at memory locations 634-825, is also available. No LOADs, SAVEs or other tape I/O can be performed to the particular cassette while the cassette buffer is used by an assembly language program.
2. **Top of Memory.** Memory locations 52 and 53 contain the pointer to the top of memory. On 8K PETs this value is 8192. You can temporarily set the top of memory pointer to a lower address, thereby reserving a number of bytes from the new pointer value to the actual top of memory for storage of an assembly language program. To set the pointer, say, down 1000 bytes, you will need to store the value 7192 (8192-1000) converted into low, high address order, e.g.:

$$7192_{10} = 1C18_{16} \rightarrow \begin{array}{cc} \text{High} & \text{Low} \\ 1C_{16} = 28_{10} & \text{and } 18_{16} = 24_{10} \end{array}$$

So 24 is to be stored at location 52 (low byte), and 28 is to be stored at location 53 (high byte). the following instructions can be used:

```
10 AL=PEEK(52);AH=PEEK(53):REM SAVE CURRENT POINTER
20 POKE 52,24:POKE 53,28:REM TOP OF CORE = 7192
.
.
.
100 POKE 52,AL:POKE 53,AH:REM RESTORE POINTER
110 END
```

3. You may find usable locations in the **BASIC Statement Area**. You may create a block of dummy DATA statements and use those locations. There are generally a few locations free between the end of the program and the beginning of the Variable Area. **You must be very careful when trying these types of approaches** that your assembly language program and the PET BASIC interpreter do not get in each other's way.

The PET BASIC interpreter can be used to load an assembly language program into the selected area of memory. The process is a rudimentary one, consisting of POKEing the decimal equivalents of the 6502 machine language instructions. To get the instructions in decimal, you can write your program in 6502 assembly language (reference manuals are listed in an appendix), hand code it into hexadecimal, and then convert the hexadecimal codes to decimal. Commodore's Terminal Interface Monitor (available on cassette from Commodore if it is not built into your PET) stores the hexadecimal codes directly. However, with the Monitor you must load the assembly language routine separately from the BASIC program, whereas by POKEing you can load the assembly language routine as part of executing the main program written in BASIC. (The PET is suitable for implementing only small portions of the program in assembly language.) DATA statements are used to define the machine language codes, which can be subsequently READ into the program and passed to a POKE loop.

Control is transferred to an assembly language program in one of two ways: the SYS command or the USR function. They are more or less interchangeable, but SYS is geared to turning control over to an assembly language program, whereas USR is a true function reference that allows a value to be sent to the called assembly language routine and a value returned by it to the main program.

The assembly language program must return control back to BASIC via a Return-From-Subroutine (RTS) instruction.

SYS

SYS is a system function that transfers program control to an independent subsystem.

Format:

SYS(address)

where:

address is a numeric constant, variable, or expression representing the starting address at which execution of the subsystem is to begin. The value must be in the range $0 \leq \text{address} \leq 65535$.

Unlike other functions, SYS can be specified alone in a direct or program statement.

Examples:

- SYS(826) In immediate mode transfer control of the system to the 6502 machine language program beginning at memory location 826 (the 2nd cassette buffer).
- 55 SYS(826) Same as above but executed in program mode. On return, execution proceeds with the first statement in sequence following the SYS statement.
- 126 SYS(A+14) Transfer control of the system to the computed address A+14.

SYS is the assembly language subroutine equivalent to a GOSUB but with the important difference that the safeguards built in to PET BASIC to protect the system from user program errors are no longer operable once control has been released by execution of a SYS (or USR). The system will tend to crash even more frequently debugging assembly language programs than it does debugging BASIC program.

Values can be passed between the BASIC program and the SYS subroutine using PEEKs and POKEs (for the BASIC program) to known memory locations.

USR

USR is a system function that passes a parameter to a user-written assembly language subroutine whose address is contained in memory locations 1 and 2 and fetches a return parameter from the subroutine.

Format:

USR(arg)

where:

arg is the parameter value passed to the subroutine.

Examples:

?USR(60) Prints in immediate mode the value returned by the USR subroutine when passed a value of 60.

105 A=USR(60) Same as above but in program mode.

210 IF USR(X) < 4 GOTO 50

510 SM=USR(XA)+USR(3.4)+SQR(Y)+π

Before making a USR reference, the beginning address of the assembly language subroutine must be placed into memory locations 1 and 2. For example, if the subroutine is located in the cassette #2 area, you would include the instructions:

10 POKE 1,58

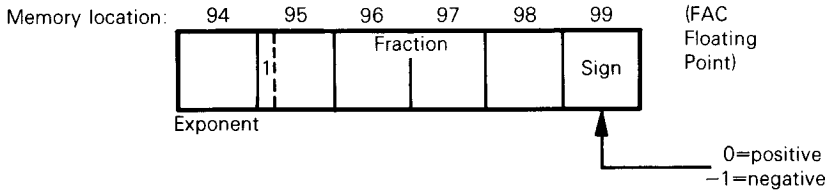
20 POKE 2,3

Low

High

since $826_{10} = 033A_{16}$, $3A_{16} = 58_{10}$ and $03_{16} = 3_{10}$.

The parameter value is passed to the USR subroutine in system locations that function as a floating point accumulator (FAC) for all functions. The FAC resides in six bytes from memory locations 94 to 99 (5E₁₆–63₁₆). The FAC has the following format:



Like floating variables, the exponent is stored in excess 128 format and the fraction is normalized with the high-order bit of byte 95 (the high-order byte of the fraction) set to 1. The difference between this format and the variable format is that the high-order 1 bit is present in byte 95 of the FAC. An extra byte (99) is used to hold the sign of the fraction. (This is done for ease of manipulation by the functions that use the FAC.)

The USR subroutine must fetch the value passed to it from the FAC locations. It must deposit the value being returned into the FAC before terminating. If the USR subroutine does not alter the FAC, then the same value is returned to the program as was passed from it.

Table 6-2. PET Memory Map (Rev. 3 ROMs)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
Page 0 (0-255)				
USR Function Locations				
0	0000	76	4C	Constant 6502 JMP instruction
1-2	0001-0002	826	033A	User address jump vector
Evaluation of Variables and Terminal I/O Maintenance				
3	0003	0	00	Search character
4	0004	0	00	Delimiter flag for quote mode scan
5	0005	255	FF	Input buffer pointer, general counter
6	0006	0	00	Flag for dimensioned variables
7	0007	0	00	Flag for variable type: 00=numeric FF=string
8	0008	0	00	Flag for numeric variable type: 00=floating point 80=integer
9	0009	0	00	Flag for DATA scan; LIST quote; memory
10	000A	0	00	Flag to allow subscripted variable; FNx flag
11	000B	0	00	Flag for input type: 0=INPUT 64=GET 152=READ
12	000C	0	00	Flag for ATN sign; comparison evaluation
13	000D	0	00	Flag to suppress output: + normal -- suppressed
14	000E	0	00	Current I/O device for prompt-suppress
15	000F	40	28	Terminal width (unused)
16	0010	30	1E	Limit for scanning source columns (unused)
17-18	0011-0012	828	033C	Basic integer address (for SYS, GOTO, etc.)
19	0013	22	16	Index to next available descriptor
20-21	0014-0015	19	13	Pointer to last string temporary
22-29	0016-001D	2	0002	Table of double-byte descriptions that point to variables (8 bytes)
30-31	001E-001F	16451	4043	Indirect index #1
32-33	0020-0021	26119	6607	Indirect index #2
34	0022	1	01	Pseudo-register for function operands (6 bytes)
35	0023	140	8C	
36	0024	0	00	
37	0025	0	00	
38	0026	0	00	
39	0027	0	00	

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
Data Storage Maintenance				
40-41	0028-0029	1025	0401	Pointer to start of BASIC text
42-43	002A-002B	1920	0780	Pointer to start of variables
44-45	002C-002D	2032	07F0	Pointer to end of variables
46-47	002E-002F	2191	088F	Pointer to end of arrays
48-49	0030-0031	8192	2000	Pointer to start of strings (moving down)
50-51	0032-0033	8191	1FFF	Pointer to end of strings (top of available RAM)
52-53	0034-0035	8192	2000	Pointer to limit of BASIC memory
54-55	0036-0037	2000	07D0	Current line number. Loc. 55=2 if no program yet executed
56-57	0038-0039	110	006E	Previous line number
58-59	003A-003B	1897	0769	Pointer to next line to be executed (for CONT)
60-61	003C-003D	200	00C8	Line number of current DATA line
62-63	003E-003F	1855	073F	Pointer to current DATA item
Expression Evaluation				
64-65	0040-0041	514	0202	INPUT vector
66-67	0042-0043	89	0059	Current variable name.
68-69	0044-0045	2006	07D6	Pointer to current variable
70-71	0046-0047	2006	07D6	Pointer to current FOR...NEXT variable
72-73	0048-0049	1279	04FF	Pointer to current operator in ROM table
74	004A	0	00	Mask for current logical operator
75-76	004B-004C	62268	F33C	Pointer to user function FN definition
77-78	0040-004E	26531	67A3	Pointer to a string description
79	004F	243	F3	Length of string
80	0050	3	03	Constant used by garbage collection routine
81	0051	76	4C	Constant 6502 JMP instruction
82-83	0052-0053	0	00	Jump vector for functions
84-89	0054-0059	211	D3	Floating point accumulator #3 (6 bytes)
90-91	005A-005B	0	0000	Block transfer pointer #1
92-93	005C-005D	0	0000	Block transfer pointer #2
94-99	005E-0063			Floating point accumulator (FAC) #1 (6 bytes)
		0	00	94 005E Exponent +128
		0	00	95 005F Fraction MSB Floating Point
		0	00	96 0060 Fraction
		0	00	97 0061 Fraction MSB Integer
		0	00	98 0062 Fraction LSB
		0	00	99 0063 Sign of fraction (0 if zero or positive, 1 if negative)
100	0064	0	00	Copy of FAC #1 sign of fraction
101	0065	0	00	Counter for number of bits to shift to normalize FAC #1
102-107	0066-006B	0	00	Floating point accumulator #2 (6 bytes)
108	006C	0	00	Overflow byte for floating argument
109	006D	0	00	Copy of FAC #2 sign of fraction
110-111	006E-006F	258	0102	Conversion pointer

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
RAM Subroutines				
112-135	0070-0087	230	E6	Routine to fetch next BASIC character
		173	AD	118 76 Entry to refetch current character
		1904	0770	119-120 77-78 Pointer into source text
136-140	0088-008C	128	80	Next random no. in storage and RND work area
OS Page Zero Storage				
141-143	008D-008F	398710	061576	24-hour clock incremented every 1/60 second (jiffy). Resets every 5,184,000 jiffies (24 hours). Stored in high to low order
144-145	0090-0091	58926	E62E	Hardware interrupt vector
146-147	0092-0093	64791	FD17	6502 BRK instruction interrupt vector
148-149	0094-0095	50057	C389	NMI interrupt vector
150	0096	0	00	Status word ST (1 byte)
151	0097	255	FF	Matrix coordinate of key depressed at current jiffy. 1-80=key, 255=no key
152	0098	0	00	Status of SHIFT key: 0=unshifted (up) 1=shifted (down)
153-154	0099-009A	65282	FF02	Correction factor for clock
155	009B	255	FF	Keyswitch PIA: STOP and RVS flags
156	009C	0	00	Timing constant buffer
157	009D	0	00	I/O flag: 0=LOAD 1=VERIFY
158	009E	0	00	Number of characters in keyboard buffer (0 to 9)
159	009F	0	00	Flag to indicate reverse field on (0=normal)
160	00A0	0	00	IEEE 488 output flag FF=character waiting
161	00A1	13	0D	Byte pointer to end of line for input
162	00A2	0	00	Utility
163-164	00A3-00A4	11, 13	0B, 0D	Cursor log (row, column)
165	00A5	63	3F	IEEE 488 output character buffer
166	00A6	255	FF	Key image
167	00A7	1	01	Flag for cursor enable: 0=Enable 1=Disable
168	00A8	17	11	Counter to flip cursor (20 to 1)
169	00A9	32	20	Copy of character at current cursor position
170	00AA	0	00	Flag for cursor on/off: 0=cursor moved 1=blink started
171	00AB	0	00	Flag for tape write
172	00AC	0	00	Flag for input source: 0=keyboard buffer 1=screen memory

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
OS Page Zero Storage (Continued)				
173	00AD	0	00	I/O utility; X save flag
174	00AE	1	01	Number of open files (index into tables)
175	00AF	0	00	Default input device number (0=keyboard)
176	00B0	3	03	Default output device number (3=screen)
177	00B1	0	00	Tape parity byte
178	00B2	0	00	Flag for byte received
179	00B3	0	00	I/O utility
180	00B4	0	00	Tape buffer character
181	00B5	0	00	Byte pointer in filename transfer
182	00B6	0	00	I/O utility
183	00B7	0	00	Serial bit count
184	00B8	0	00	Tape utility
185	00B9	0	00	Cycle counter — flip for each bit read from tape
186	00BA	0	00	Countdown synchronization on tape write
187	00BB	0	00	Tape buffer 1 index to next character
188	00BC	0	00	Tape buffer 2 index to next character
189	00BD	0	00	Countdown synchronization on tape read
190	00BE	0	00	Flag to indicate bit/byte tape error
191	00BF	0	00	Flag to indicate tape error 0=first half-byte marker not written
192	00C0	0	00	Flag to indicate tape error 0=2nd half-byte marker not written
193	00C1	0	00	Tape dropout counter
194	00C2	0	00	Flag for cassette read current function 0=scan, 1-15=count, 40 ₁₆ =load, 80 ₁₆ =end
195	00C3	0	00	Checksum utility
196-197	00C4-00C5	33728	83CD	Pointer to start of line where cursor is flashing
198	00C6	0	00	Column position where cursor is flashing (0- 79)
199-200	00C7-00C8	33792	8400	Load start address; utility pointer
201-202	00C9-00CA	0	0000	Load end address
203-204	00CB-00CC	0	00	Tape timing constants
205	00CD	0	00	Flag for quote mode 0=not quote mode
206	00CE	0	00	Flag for tape read timer enable 0=disabled
207	00CF	0	00	Flag for EOT received from tape
208	00D0	0	00	Read character error
209	00D1	0	00	No. of characters in current file name
210	00D2	4	04	Current logical file number
211	00D3	255	FF	Current secondary address
212	00D4	4	04	Current device number
213	00D5	39	27	Current screen line length (39, 79)
214-215	00D6-00D7	0	0000	Pointer to start of current tape buffer (634 or 826)

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
216	00D8	24	18	Line number where cursor is flashing (0-24)
217	00D9	10	0A	I/O storage: last key input, buffer checksum, bit buffer
218-219	00DA-00DB	0	0000	Pointer to current file name
220	00DC	0	00	Number of Insert keys pushed to go
221	00DD	0	00	Serial bit shift word
222	00DE	0	00	Number of blocks remaining to read/write
223	00DF	0	00	Serial word buffer
224-248	00E0-00F8			High byte of screen line addresses
		128	80	224-230=128 (lines 1-7)
		129	81	231-236=129 (lines 8-13)
		130	82	237-243=130 (lines 14-20)
		131	83	244-248=131 (lines 21-25)
249	00F9	0	00	Cassette #1 status switch
250	00FA	0	00	Cassette #2 status switch
251-252	00FB-00FC	54144	D380	Tape start address
253-255	00FD-00FF	243	F3	Utility
Page 1 (256-511)				
256-up	0100-up	32	20	Tape read working storage (up to 511) and conversion storage
				256-318 For error correction in tape reads (62 bytes)
				256-266 Binary to ASCII conversion (11 bytes)
511-down	01FF-down	44	2C	Stack (down to 256)
Page 2-3 (512-1023)				
512-592	0200-0250			BASIC input line buffer (80 bytes)
		12597	3135	512-513 0200-0201 Program Counter
		50	32	514 0202 Processor status
		0	00	515 0203 Accumulator
		171	AB	516 0204 X index
		0	00	517 0205 Y index
		0	00	518 0206 Stack pointer
		15104	3B00	519-520 0207-0208 User modifiable IRQ
593-602	0251-025A	4	04	Table of logical numbers of open files
603-612	025B-0264	4	04	Table of device numbers of open files
613-622	0265-026E	255	FF	Table of secondary address modes of open files
623-632	026F-0278	3	03	Keyboard buffer (10 bytes)
633	0279	28	1C	Keyboard utility
634-825	027A-0339	28	1C	Tape buffer for cassette #1 (192 bytes)
826-1017	033A-03F9	173	AD	Tape buffer for cassette #2 (192 bytes)
1018-1019	03FA-03FB	59383	E7F7	Vector for Machine Language Monitor
1020-1023	03FC-03FF	195	C3	Utility space/unused

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
OS Page Zero Storage (Continued)				
Page 4-128 (1024-32767)				
1024-32767	0400-7FFF	0	00	User program area and Expansion RAM 4K PET: 1024-4095 0400-0FFF User program area 4096-32767 1000-7FFF Expansion RAM 8K PET: 1024-8191 0400-1FFF User program area 8192-32767 2000-7FFF Expansion RAM 16K PET: 1024-16383 0400-3FFF User program area 16384-32767 4000-7FFF Expansion RAM 32K PET: 1024-32767 0400-7FFF User program area
Page 129-144 (32768-36863)				
32768-36863	8000-8FFF	32	20	TV RAM 32768-33767 Display memory (1000 bytes)
Page 145-192 (36864-49151)				
36864-49151	9000-BFFF	144	90	Expansion ROM
Page 193-232 BASIC (49152-59391)				
Pointers to BASIC Routines				
49152-49153	C000-C001	51008	C740	Pointer --1 to END*
49154-49155	C002-C003	50775	C657	Pointer --1 to FOR
49156-49157	C004-C005	52255	CC1F	Pointer --1 to NEXT
49158-49159	C006-C007	51199	C7FF	Pointer --1 to DATA
49160-49161	C008-C009	51878	CAA6	Pointer --1 to INPUT#
49162-49163	C00A-C00B	51904	CAC0	Pointer --1 to INPUT
49164-49165	C00C-C00D	53090	CF62	Pointer --1 to DIM
49166-49167	C00E-C00F	51974	CB06	Pointer --1 to READ
49168-49169	C010-C011	51372	C8AC	Pointer --1 to LET
49170-49171	C012-C013	51116	C7AC	Pointer --1 to GOTO
49172-49173	C014-C015	51076	C784	Pointer --1 to RUN
49174-49175	C016-C017	51247	C82F	Pointer --1 to IF
49176-49177	C018-C019	50991	C72F	Pointer --1 to RESTORE
49178-49179	C01A-C01B	51087	C78F	Pointer --1 to GOSUB
49180-49181	C01C-C01D	51161	C7D9	Pointer --1 to RETURN
49182-49183	C01E-C01F	51266	C842	Pointer --1 to REM
49184-49185	C020-C021	51006	C73E	Pointer --1 to STOP
49186-49187	C022-C023	51282	C852	Pointer --1 to ON
49188-49189	C024-C025	55055	D70F	Pointer --1 to WAIT
49190-49191	C026-C027	65492	FFD4	Pointer --1 to LOAD
49192-49193	C028-C029	65495	FFD7	Pointer --1 to SAVE
49194-49195	C02A-C02B	65498	FFDA	Pointer --1 to VERIFY
49196-49197	C02C-C02D	53900	D28C	Pointer --1 to DEF
49198-49199	C02E-C02F	55046	D706	Pointer --1 to POKE
49200-49201	C030-C031	51594	C98A	Pointer --1 to PRINT#

* These memory locations contain the address of the byte preceding the specified BASIC routines.

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
Pointers to BASIC Routines (Continued)				
49202-49203	C032-C033	51626	C9AA	Pointer --1 to PRINT
49204-49205	C034-C035	51050	C76A	Pointer --1 to CONT
49206-49207	C036-C037	50612	C5B4	Pointer --1 to LIST
49208-49209	C038-C039	50550	C576	Pointer --1 to CLR
49210-49211	C03A-C03B	51600	C990	Pointer --1 to CMD
49212-49213	C03C-C03D	65501	FFDD	Pointer --1 to SYS
49214-49215	C03E-C03F	65471	FFBF	Pointer --1 to OPEN
49216-49217	C040-C041	65474	FFC2	Pointer --1 to CLOSE
49218-49219	C042-C043	51836	CA7C	Pointer --1 to GET
49220-49221	C044-C045	50522	C55A	Pointer --1 to NEW
49222-49223	C046-C047	56133	DB45	Pointer to SGN **
49224-49225	C048-C049	56280	DBD8	Pointer to INT
49226-49227	C04A-C04B	56164	DB64	Pointer to ABS
49228-49229	C04C-C04D	0	0000	Pointer to USR pointer
49230-49231	C04E-C04F	53849	D259	Pointer to FRE
49232-49233	C050-C051	53882	D27A	Pointer to POS
49234-49235	C052-C053	56926	DE5E	Pointer to SQR
49236-49237	C054-C055	57215	DF7F	Pointer to RND
49238-49239	C056-C057	55542	D8F6	Pointer to LOG
49240-49241	C058-C059	57050	DEDA	Pointer to EXP
49242-49243	C05A-C05B	57304	DFD8	Pointer to COS
49244-49245	C05C-C05D	57311	DFDF	Pointer to SIN
49246-49247	C05E-C05F	57384	E028	Pointer to TAN
49248-49249	C060-C061	57484	E08C	Pointer to ATN
49250-49251	C062-C063	55016	D6E8	Pointer to PEEK
49252-49253	C064-C065	54870	D656	Pointer to LEN
49254-49255	C066-C067	54079	D33F	Pointer to STR\$
49256-49257	C068-C069	54919	D687	Pointer to VAL
49258-49259	C06A-C06B	54885	D664	Pointer to ASC
49260-49261	C06C-C06D	54726	D5C6	Pointer to CHR\$
49262-49263	C06E-C06F	54746	D5DA	Pointer to LEFT\$
49264-49265	C070-C071	54790	D606	Pointer to RIGHT\$
49266-49267	C072-C073	54801	D611	Pointer to MID\$
49268-49297	C074-C091			Hierarchy and action addresses for operators
49298-49553	C092-C191			Table of BASIC keywords
49554-49833	C192-C2A9			BASIC error messages
BASIC Routines				
				Starting Address Function
49834-59343	C2AA-DFFF			49834 C2AA FOR...NEXT stack check
				49880 C2D8 Insert line space marker
				49947 C31B Stack overflow check
				49960 C328 Error message abort
				50057 C389 READY
				50091 C3AB Handle new line

** These memory locations contain the address of the first byte of the specified BASIC routines.

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
BASIC Routines (Continued)				
				Starting Address Function
				50242 C442 Rechain lines after insert/delete
				50287 C46F Input line
				50325 C495 Keyword encoder
				50476 C52C Line number search
				50523 C55B NEW
				50551 C577 CLR
				50599 C5A7 Set pointer to start of program
				50613 C5B5 LIST
				50776 C658 FOR
				50944 C700 Statement execute
				50992 C730 RESTORE
				51007 C73F STOP
				51009 C741 END
				51051 C76B CONT
				51077 C785 RUN
				51088 C790 GOSUB
				51117 C7AD GOTO
				51162 C7DA RETURN
				51200 C800 DATA
				51214 C80E Scan for next BASIC statement
				51217 C811 Scan for next BASIC line
				51248 C830 IF
				51267 C843 REM
				51283 C853 ON
				51315 C873 Number fetch
				51373 C8AD LET =
				51496 C928 Add ASCII digit to Accumulator #1
				51595 C98B PRINT #
				51601 C991 CMD
				51627 C9AB PRINT
				51740 CA1C Print string
				51769 CA39 Print character
				51791 CA4F Input data error
				51837 CA7D GET
				51879 CAA7 INPUT #
				51962 CAFA Input prompt
				51975 CB07 READ
				52220 CBFC Error messages
				52256 CC20 NEXT
				52345 CC79 Format checker
				52383 CC9F Expression evaluator
				53091 CF63 DIM
				53101 CF6D Variable table lookup
				53249 D001 Create new variable
				53420 D0AC Array table search/ create array

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description		
Decimal	Hexadecimal	Decimal	Hexadecimal			
BASIC Routines (Continued)						
				<table border="0" style="width: 100%;"> <tr> <td style="text-align: left;">Starting Address</td> <td style="text-align: left;">Function</td> </tr> </table>	Starting Address	Function
Starting Address	Function					
				53849 D259 FRE		
				53869 D26D Integer-to-floating		
				53882 D27A POS		
				53888 D280 Valid direct check		
				53901 D28D DEF		
				54079 D33F STR\$		
				54726 D5C6 CHR\$		
				54746 D5DA LEFT\$		
				54790 D606 RIGHT\$		
				54801 D611 MID\$		
				54870 D656 LEN		
				54885 D665 ASC		
				54919 D687 VAL		
				54994 D6D2 Floating-to-integer		
				55016 D6E8 PEEK		
				55047 D707 POKE		
				55056 D710 WAIT		
				55091 D733 Subtraction		
				55150 D76E Addition		
				55542 D8F6 LOG		
				55607 D937 Multiplication		
				55704 D998 Load number to AFAC		
				55818 DA0A Division		
				55982 DAAE Load Accumulator (FAC) <i>va. Basic</i>		
				56030 DADE Store FAC <i>Basic</i>		
				56072 DB08 Copy AFAC to FAC		
				56088 DB18 Copy FAC to AFAC		
				56133 DB45 SGN		
				56164 DB64 ABS		
				56280 DBD8 INT		
				56526 DCCE IN line message		
				56553 DCE9 Numeric-to-ASCII		
				56319 DBFF String-to-floating		
				56926 DE5E SQR		
				56936 DE68 Power function		
				57050 DEDA EXP		
				57215 DF7F RND		
				57304 DFD8 COS		
				57311 DFDF SIN		

va. Basic

ACC 0050-0000

*AFAC 0466 - 046B
FAC 045E - 0463*

va. Basic

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
57344-5391	E000-E7FF			<p>Screen Editor</p> <p>Starting Address Function</p> <p>57384 E028 TAN</p> <p>57484 E08C ATN</p> <p>57593 E0F9 Subroutine to be moved to page 0 (\$70-\$87)</p> <p>57617 E111 Initial RND seed (5 bytes)</p> <p>57622 E116 Initialize BASIC system</p> <p>57897 E229 Clear screen</p> <p>57943 E257 Home cursor</p> <p>57989 E285 Character fetch</p> <p>Video driver</p> <p>58100 E2F4 Input from screen</p> <p>58175 E33F Quote mode (\$CD) switcher</p> <p>58188 E34C Print character</p> <p>58687 E53F Scroll 1 line</p> <p>Interrupt Handler</p> <p>Keyboard Scan</p> <p>Keyboard Encoding Table</p> <p>Subroutines for Machine Language Monitor</p>
58100-58906	E2F4-E61A			
58907-59113	E61B-E6E9			
59114-59127	E6EA-E6F7			
59128-59241	E6F8-E769			
59242-59391	E76A-E7FF			
Page 233-240 I/O Ports and Expansion I/O (PIA's and VIA) (59392-61439)				
				<p>Keyboard PIA (59408-59411)</p> <p>I/O Port A and Data Direction register</p> <p>Control Register A — screen blanking</p> <p>52=Screen off (blanked)</p> <p>60=Screen on</p> <p>I/O Port B and Data Direction register</p> <p>255=all keys except:</p> <p>254=RVS key</p> <p>253=[key</p> <p>251=SPACE key</p> <p>247=< key</p> <p>Control Registers B — #1 cassette motor</p> <p>53=motor on</p> <p>61=motor off</p>
59408	E810	249	F9	
59409	E811	60	3C	
59410	E812	255	FF	
59411	E813	61	3D	
				<p>IEEE Port PIA (59424-59427)</p> <p>I/O Port A and Data Direction register</p> <p>PEEK (59424) reads input data</p> <p>Control Register A — set output line CA2</p> <p>POKE 59425,52=low</p> <p>POKE 59425,60=high</p> <p>I/O Port B and Data Direction registers</p> <p>POKE 59426, data writes output data</p> <p>POKE 59426,255 before a read to Port A</p> <p>Control Register B — set output line CB2</p> <p>POKE 59427,52=low</p> <p>POKE 59427,60=high</p>
59424	E820	255	FF	
59425	E821	188	BC	
59426	E822	255	FF	
59427	E823	60	3C	

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
Parallel User Port VIA (59456-59471)				
59456	E840	223	DF	I/O Port B 207=#2 cassette motor on 223=#2 cassette motor off WAIT 59456,23,23 waits for vertical retrace of display Bit 1=PB1 (NFRD on IEEE connector) output line Bit 3=PB3 (ATN on IEEE connector) output line
59457	E841	255	FF	I/O Port A with handshaking
59458	E842	30	1E	Data Direction register for I/O Port B
59459	E843	0	00	Data Direction register for I/O Port A For each bit 1=output, 0=input =0 all input =255 all output
59460-59461	E844-E845	29241	7239	(Low, high order) Read Timer 1, Counter; write to Timer 1 Latch and (high byte) initiate count
59462-59463	E846-E847	65535	FFFF	(Low, high order) Read Timer 1 Latch
59464	E848	147	93	Read Timer 2 Counter low byte and reset interrupt; write to Timer 2 low byte PEEK (59464) Clock decrements every microsecond POKE 59454,n sets SR rate of shift from high (n=0) to low (n=255) for music from User Port
59465	E849	217	D9	Read Timer 2 Counter high byte; write to Timer 2 high byte and reset interrupt PEEK (59465) Clock decrements every millisecond
59466	E84A	0	00	Serial I/O Shift register (SR) POKE 59466, 15 or 85 to generate Square wave output at CB2 for playing music from User Port.
59467	E84B	0	00	Auxiliary Control register =16 Sets SR to free-running mode for music from User Port =0 for proper operation of tape drive
59468	E84C	14	0E	Peripheral Control register =12 for graphics on shifted characters =14 for lower-case letters on shifted characters
59469	E84D	0	00	Interrupt Flag register
59470	E84E	128	80	Interrupt enable register
59471	E84F	255	FF	I/O Port A without handshaking
Page 241-256 Operating System (61440-65535)				
61440-61621	F000-F0B5			Monitor messages

Table 6-2. PET Memory Map (Rev. 3 ROMs) (Continued)

Memory Address		Sample Value		Description	
Decimal	Hexadecimal	Decimal	Hexadecimal		
61622-61904	F0B6-F1D0			GPIB Handler (IEEE 488 Bus)	
				Starting Address Function	
				61622 F0B6 Setup for Listen, Talk, etc.	
				61678 F0EE Send character	
				61736 F128 Output character immediate mode	
				61750 F136 Error messages	
				61796 F164 Send immediate Listen command, then secondary address	
				61807 F16F Output characters	
				61823 F17F Send Unlisten/ Untalk	
				61836 F18C Input character	
					File Control
				61905-63493	F1D1-F805
61921 F1E1 Input a character (with cursor)					
62002 F232 Output a character to any device					
62062 F26E Close all files					
62066 F272 Restore default I/O devices					
62121 F2A9 CLOSE					
62209 F301 STOP search					
62223 F30F STOP key					
62229 F315 Direct mode test					
62402 F3C2 LOAD					
62474 F40A Display filename/ fetch file number					
62526 F43E Fetch LOAD/SAVE parameters					
62560 F460 Fetch byte paramter					
62566 F466 Send program name to GPIB					
62612 F494 Tape header search					
62647 F4B7 VERIFY					
62670 F4CE Fetch OPEN/CLOSE parameters					
62753 F521 OPEN					
62886 F5A6 Find any tape header					
62938 F5DA Write tape header					
63036 F63C Process tape header					
63108 F684 SYS					
63134 F69E SAVE					
63273 F729 Clock update					
63344 F770 Set input device					
63420 F7BC Set output device					

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
Tape Control				
63494-64720	F806-FCD0			63494 F806 Advance tape buffer pointer 63541 F835 Check for cassette on 63573 F855 Tape read to buffer 63622 F886 Write block to tape 63716 F8E6 Interrupt wait
Power-on Diagnostics				
64721-64784	FCD1-FD10			64721 FCD1 System reset SYS(64721) simulates power-on reset. 64766 FCFE NMI interrupt entry point 64769 FD01 Table of interrupt vectors
64785-65471	FD11-FFBF			
Machine Language Monitor				
Jump Vectors				
65472-65474	FFC0-FFC2	76 62753	4C F521	JMP OPEN
65475-65477	FFC3-FFC5	76 62121	4C F2A9	JMP CLOSE
65478-65480	FFC6-FFC8	76 63344	4C F770	JMP Set Input Device
65481-65483	FFC9-FFCB	76 63420	4C F7BC	JMP Set Output Device
65484-65486	FFCC-FFCE	76 62066	4C F272	JMP Restore Default I/O Devices
65487-65489	FFCF-FFD1	76 61921	4C F1E1	JMP Input Character — RDT
65490-65492	FFD2-FFD4	76 62002	4C F232	JMP Output Character — WRT
65493-65495	FFD5-FFD7	76 62402	4C F3C2	JMP LOAD
65496-65498	FFD8-FFDA	76 63134	4C F69E	JMP SAVE
65499-65501	FFDB-FFDD	76 62647	4C F4B7	JMP VERIFY
65502-65504	FFDE-FFED	76 63108	4C F684	JMP SYS
65505-65507	FFE1-FFE3	76 62223	4C F30F	JMP Test STOP Key
65508-65510	FFE4-FFE6	76 61905	4C F1D1	JMP Get Character
65511-65513	FFE7-FFE9	76 62062	4C F26E	JMP Close all files
65514-65516	FFEA-FFEC	76 63273	4C F729	JMP Clock Update
6502 Interrupt Vectors				
65530-65531	FFFA-FFFB	65766	FCFE	Non-maskable interrupt (NMI)
65532-65533	FFFC-FFFD	64721	FCD1	System reset (RESET)
65534-65535	FFFE-FFFF	58907	E61B	Interrupt request break (IRQ+BRK)

APPENDIX A

PET ASCII Codes

Appendix A is a detailed chart illustrating the characters and cursor controls of the PET and CBM computer. The left side of the chart shows the standard and alternate character sets for the PET with a graphic keyboard and the CBM with a full-size keyboard. The right three columns show each character's corresponding ASCII and PEEK/POKE. The characters are arranged in ascending sequence by their PET ASCII number, or by their PEEK/POKE number if the character does not have a PET ASCII number. Many characters appear twice because they have two PET ASCII numbers.

STANDARD CHARACTER SET: The Standard Character Set is made up of the characters normally displayed when the PET is powered up.

GRAPHIC KEYBOARD: This column represents the Standard Character Set of all PET's with graphic symbols displayed upon the keyboard. This set is in effect when the PET is powered up or when value 12 is poked into memory location 59468 by a POKE 59468,12. When in effect, all the characters shown on the keyboard, the upper-case letters, numbers, and graphic characters are available.

CBM KEYBOARD: This column represents the Standard Character Set of the CBM version of the PET without graphics displayed on the keyboard. This set is in effect when the CBM is powered up or when value 14 is poked into memory loca-

tion 59468 by a POKE 59468,14. When in effect, all the characters shown on the keyboard, upper (with shift) and lower-case letters, numbers and some symbols are available. The CBM standard character is only represented if it differs from the standard graphic keyboard character. Otherwise, if the column is blank, you may assume that the graphic and typewriter standard character is the same.

ALTERNATE CHARACTER SET: The Alternate Character Set is all the characters displayed only after poking the alternate set into effect with a POKE 59468,xx.

GRAPHIC KEYBOARD: This column represents the alternate character set of all PETs with graphic symbols displayed upon the keyboard. This set is activated only after poking value 14 into memory location 59468 by a POKE 59468,14. When in effect, upper AND lower-case letters, numbers and most of the graphic characters are available, along with some special graphic characters not illustrated on the keyboard.

CBM KEYBOARD: This column represents the Alternate Character Set of the CBM version of the PET without graphics displayed on the keyboard. This set is activated only after poking value 12 into memory location 59468 by a POKE 59468,12. When in effect, upper and lower-case letters, numbers, AND the graphic characters are available. Again, the CBM character is represented only if the character differs from the graphic keyboard character. If the column is blank, you may assume that the graphic and typewriter alternate character is the same.

PET ASCII: As you recall from the discussion of the CHR\$ function in Chapter 5, ASCII stands for the "American Standard Code for Information Interchange." Commodore developed its own ASCII code for the PET (and CBM) computer to include its unique graphic characters. We will call this special ASCII code PET ASCII. Two PET functions, ASC() and CHR\$() use the PET ASCII decimal numbers to reference its characters. The PET ASCII code column enables you to find a character's PET ASCII number quickly by finding the desired character and looking across the chart for its PET ASCII number. Although both the PET ASCII decimal (DEC) number from 0 to 255, and the hexadecimal (HEX) number from 00 to FF are given, when using the ASC() or CHR\$() functions, use only the decimal ASCII number. The last portion of the chart, the reverse characters, do not have PET ASCII numbers, and thus are arranged by their PEEK/POKE numbers, as explained in the next section.

PEEK/POKE: The PEEK/POKE code is the number that must be used when either POKEing a character to the screen, or when PEEKING into memory to see what character is contained in a specified memory location; the number returned will be the PEEK/POKE number representing that character. Notice that in most cases the PEEK/POKE number is not the same as the decimal ASCII number. This code number CANNOT be used with ASCII functions, only with PEEK and POKE. The PEEK/POKE code numbers do not appear in strict ascending sequence until the reverse characters portion of the chart. At this point, the chart is arranged in ascending PEEK/POKE order because the reverse PET characters lack PET ASCII

numbers, and without PRINT statements can only be referenced with PEEKs or POKEs.

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/POKE
PET	CBM	PET	CBM	DEC	HEX	
				0	00	
				1	01	
				2	02	
STOP		STOP		3	03	
				4	04	
				5	05	
				6	06	
				7	07	
				8	08	
				9	09	
				10	0A	
				11	0B	
RETURN		RETURN		12	0C	
				13	0D	
				14	0E	
				15	0F	
CRSR↓		CRSR↓		16	10	
RVS		RVS		17	11	
HOME		HOME		18	12	
DELETE		DELETE		19	13	
				20	14	
				21	15	
				22	16	
				23	17	
				24	18	
				25	19	
				26	1A	
				27	1B	
				28	1C	
CRSR→		CRSR→		29	1D	
				30	1E	
				31	1F	
␣		␣		32	20	32

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/ POKE
PET	CBM	PET	CBM	DEC	HEX	
!		!		33	21	33
"		"		34	22	34
#		#		35	23	35
\$		\$		36	24	36
%		%		37	25	37
&		&		38	26	38
/		/		39	27	39
<		<		40	28	40
>		>		41	29	41
*		*		42	2A	42
+		+		43	2B	43
,		,		44	2C	44
-		-		45	2D	45
.		.		46	2E	46
/		/		47	2F	47
0		0		48	30	48
1		1		49	31	49
2		2		50	32	50
3		3		51	33	51
4		4		52	34	52
5		5		53	35	53
6		6		54	36	54
7		7		55	37	55
8		8		56	38	56
9		9		57	39	57
:		:		58	3A	58
;		;		59	3B	59
<		<		60	3C	60
=		=		61	3D	61
>		>		62	3E	62
?		?		63	3F	63
@		@		64	40	0

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/POKE
PET	CBM	PET	CBM	DEC	HEX	
A	a	A	A	65	41	1
B	b	B	B	66	42	2
C	c	C	C	67	43	3
D	d	D	D	68	44	4
E	e	E	E	69	45	5
F	f	F	F	70	46	6
G	g	G	G	71	47	7
H	h	H	H	72	48	8
I	i	I	I	73	49	9
J	j	J	J	74	4A	10
K	k	K	K	75	4B	11
L	l	L	L	76	4C	12
M	m	M	M	77	4D	13
N	n	N	N	78	4E	14
O	o	O	O	79	4F	15
P	p	P	P	80	50	16
Q	q	Q	Q	81	51	17
R	r	R	R	82	52	18
S	s	S	S	83	53	19
T	t	T	T	84	54	20
U	u	U	U	85	55	21
V	v	V	V	86	56	22
W	w	W	W	87	57	23
X	x	X	X	88	58	24
Y	y	Y	Y	89	59	25
Z	z	Z	Z	90	5A	26
[[[91	5B	27
\		\	\	92	5C	28
]]]	93	5D	29
↑		↑	↑	94	5E	30
←		←	←	95	5F	31
				96	60	32

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/ POKE
PET	CBM	PET	CBM	DEC	HEX	
!		!		97	61	33
"		"		98	62	34
#		#		99	63	35
\$		\$		100	64	36
%		%		101	65	37
&		&		102	66	38
>		>		103	67	39
<		<		104	68	40
))		105	69	41
*		*		106	6A	42
+		+		107	6B	43
,		,		108	6C	44
-		-		109	6D	45
.		.		110	6E	46
/		/		111	6F	47
0		0		112	70	48
1		1		113	71	49
2		2		114	72	50
3		3		115	73	51
4		4		116	74	52
5		5		117	75	53
6		6		118	76	54
7		7		119	77	55
8		8		120	78	56
9		9		121	79	57
:		:		122	7A	58
;		;		123	7B	59
<		<		124	7C	60
=		=		125	7D	61
>		>		126	7E	62
?		?		127	7F	63
				128	80	64

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/ POKE
PET	CBM	PET	CBM	DEC	HEX	
				129	81	65
				130	82	66
RUN		RUN		131	83	67
				132	84	68
				133	85	69
				134	86	70
				135	87	71
				136	88	72
				137	89	73
				138	8A	74
				139	8B	75
				140	8C	76
Shifted RETURN		Shifted RETURN		141	8D	77
				142	8E	78
				143	8F	79
				144	90	80
CRSR↑		CRSR↑		145	91	81
RVS Off		RVS Off		146	92	82
CLR Screen		CLR Screen		147	93	83
INSERT		INSERT		148	94	84
				149	95	85
				150	96	86
				151	97	87
				152	98	88
				153	99	89
				154	9A	90
				155	9B	91
				156	9C	92
CRSR←		CRSR←		157	9D	93
				158	9E	94
				159	9F	95
Shifted ␣		Shifted ␣		160	A0	96

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/POKE
PET	CBM	PET	CBM	DEC	HEX	
█		█		161	A1	97
▀		▀		162	A2	98
▄		▄		163	A3	99
▁		▁		164	A4	100
				165	A5	101
⌘		⌘		166	A6	102
				167	A7	103
⌘		⌘		168	A8	104
▀	▀	▀	▀	169	A9	105
				170	AA	106
┌		┌		171	AB	107
█		█		172	AC	108
└		└		173	AD	109
┌		┌		174	AE	110
▀		▀		175	AF	111
┌		┌		176	B0	112
└		└		177	B1	113
┌		┌		178	B2	114
└		└		179	B3	115
				180	B4	116
				181	B5	117
				182	B6	118
▀		▀		183	B7	119
▄		▄		184	B8	120
█		█		185	B9	121
┌	✓	┌	┌	186	BA	122
█		█		187	BB	123
█		█		188	BC	124
┌		┌		189	BD	125
█		█		190	BE	126
█		█		191	BF	127
▀		▀		192	C0	64

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/ POKE
PET	CBM	PET	CBM	DEC	HEX	
█		█		225	E1	97
■		■		226	E2	98
—		—		227	E3	99
—		—		228	E4	100
				229	E5	101
⊗		⊗		230	E6	102
				231	E7	103
⊗		⊗		232	E8	104
▀	▀	▀	▀	233	E9	105
				234		106
┆		┆		235		107
■		■		236		108
L		L		237		109
⌒		⌒		238		110
—		—		239		111
r		r		240		112
⊥		⊥		241		113
⌒		⌒		242		114
⊥		⊥		243		115
				244		116
█		█		245		117
				246		118
—		—		247		119
—		—		248		120
■		■		249		121
└	✓	✓	└	250		122
■		■		251		123
■		■		252		124
└		└		253		125
■		■		254		126
π	⊗	⊗	π	255		127

**Standard
Character Set**

PET CBM

00	
01	
02	a
03	b
04	c
05	d
06	e
07	f
08	g
09	h
0A	i
0B	j
0C	k
0D	l
0E	m
0F	n
10	o
11	p
12	q
13	r
14	s
15	t
16	u
17	v
18	w
19	x
1A	y
1B	z
1C	
1D	
1E	
1F	

} Reverse

**Alternate
Character Set**

PET CBM

00	
01	
02	
03	
04	
05	
06	
07	
08	
09	
0A	
0B	
0C	
0D	
0E	
0F	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	

PET ASCII

DEC HEX

**PEEK/
POKE**

128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/ POKE
PET	CBM	PET	CBM	DEC	HEX	
█		█				160
▣		▣				161
▤		▤				162
▥		▥				163
▦		▦				164
▧		▧				165
▨		▨				166
▩		▩				167
▪		▪				168
▫		▫				169
▬		▬				170
▭		▭				171
▮		▮				172
▯		▯				173
▰		▰				174
▱		▱				175
▲		▲				176
△		△				177
▴		▴				178
▵		▵				179
▶		▶				180
▷		▷				181
▸		▸				182
▹		▹				183
►		►				184
▻		▻				185
▼		▼				186
▽		▽				187
▾		▾				188
▿		▿				189
◀		◀				190
▶		▶				191

Standard Character Set		Alternate Character Set		PET ASCII		PEEK/ POKE
PET	CBM	PET	CBM	DEC	HEX	
█		█				224
┆		┆				225
▬		▬				226
█		█				227
█		█				228
█		█				229
▒		▒				230
▒		▒				231
▒		▒				232
▒	▒	▒	▒			233
█		█				234
█		█				235
█		█				236
█		█				237
█		█				238
█		█				239
█		█				240
█		█				241
█		█				242
█		█				243
█		█				244
█		█				245
█		█				246
█		█				247
█		█				248
█		█				249
█		█	█			250
█		█				251
█		█				252
█		█				253
█		█				254
█		█				255

APPENDIX B

PET Error Messages

Error messages may be displayed in response to just about anything you may key in at the PET keyboard or when your program is running. Both the PET BASIC interpreter and the operating system issue error messages, listed separately below.

Whenever the PET BASIC interpreter detects an error, it displays a diagnostic message, headed by a question mark, in the general form:

?message ERROR IN LINE number

where message is the type of error (listed alphabetically below) and number is the line number in the program where the error occurred (not present in immediate mode). Following any error message, BASIC returns to immediate mode and gives the prompt:

READY.

PET BASIC error messages are listed below, with two descriptive paragraphs: the first describes the cause of the error, and the second discusses possible ways of correcting the error.

BASIC Error Message

<u>Error Message</u>	<u>Cause and Suggested Remedies</u>
BAD SUBSCRIPT	<p>An attempt was made to reference an array element that is outside the dimensions of the array. This may happen by specifying the wrong number of dimensions (different from the DIM statement), using a subscript larger than specified in the DIM statement or using a subscript larger than 10 for a non-dimensioned array.</p> <p>Correct the array element number to remain within the original dimensions, or change the array size to allow more elements.</p>
CAN'T CONTINUE	<p>A CONT command was issued, but program execution cannot be resumed because the program has been altered, added to or cleared in immediate mode, or execution was stopped by an error. Program execution cannot be continued past an error message.</p> <p>Correct the error. The most prudent course is to type RUN and start over. However, you can attempt to reenter the program at the point of interruption by a directed GOTO.</p>
DIVISION BY ZERO	<p>An attempt was made to perform a division operation with a divisor of zero. Dividing by zero is not allowed.</p> <p>Check the values of variables (or constants!) in the indicated line number. Change the program so that the divisor can never be evaluated to zero or add a check for zero before performing the division.</p>
FORMULA TOO COMPLEX	<p>This is not a program error but indicates that a string expression in the program is too intricate for PET BASIC to handle.</p> <p>Break the indicated expression into two or more parts and rerun the program (this will also tend to improve program readability).</p>

<u>Error Message</u>	<u>Cause and Suggested Remedies</u>
ILLEGAL DIRECT	<p>A command was given in immediate (direct) mode that is valid only in program mode. The following are invalid in immediate mode: DATA, DEF FN, GET, GET#, INPUT, INPUT#.</p> <p>Enter the desired operation as a (short) program and RUN it.</p>
ILLEGAL QUANTITY	<p>A function is passed one or more parameters that are out of range. This message also occurs if the USR function is referenced before storing the subroutine address at memory locations 1 and 2.</p> <p>Check the ranges given in Chapter 4 for the function in question. Change the program to be sure that the argument will always be within range, or add a check before the function reference to make sure that the argument is allowed. If USR error, insert statements to POKE the subroutine address before the USR reference.</p>
NEXT WITHOUT FOR	<p>A NEXT statement is encountered that is not tied to a preceding FOR statement. Either there is no FOR statement or the variable in the NEXT statement is not in a corresponding FOR statement.</p> <p>The FOR part of a FOR . . . NEXT loop must be inserted or the offending NEXT statement deleted. Be sure that the index variables are the same at both ends of the loop.</p>
OUT OF DATA	<p>A READ statement is executed but all of the DATA statements in the program have already been read. For each variable in a READ statement, there must be a corresponding DATA element.</p> <p>Add more DATA elements or restrict the number of READs to the current number of DATA elements. Insert a RESTORE statement to reread the existing data. Or add a flag at the end of the last DATA statement (any value not used as a DATA element may be used for the flag value) and stop READING when the flag has been read.</p>

Error Message

Cause and Suggested Remedies

OUT OF MEMORY

The user program area of memory has been filled and a request is given to put more in, e.g., add a line to the program. This message may also be caused by multiple FOR . . . NEXT and/or GOSUB nestings that fill up the Stack; this is the case if ?FRE(0) shows considerable program area storage left.

Simplify the program. Pay particular attention to reducing array sizes. It may be necessary to restructure the program into overlays.

OVERFLOW

A calculation has resulted in a number outside the allowable range, i.e., the number is too big. The largest number allowed is 1.70141184E+38.

Check your calculations. It may be possible to eliminate this error just by changing the order in which the calculations are programmed.

REDIM'D ARRAY

An array name appears in more than one DIM statement. This error also occurs if an array name is used (given a default size of 11) and later appears in a DIM statement.

Place DIM statements near the beginning of the program. Check to see that each DIM statement is executed only once. DIM must not appear inside a FOR . . . NEXT loop or in a subroutine where either may be executed more than once.

REDO FROM START

This is a diagnostic message during an INPUT statement operation and is not a fatal error. It indicates that the wrong type of data (string for numeric or vice versa) was entered in response to an INPUT request.

Reenter the correct type data. INPUT will continue prompting until an acceptable response is entered.

Error Message
RETURN WITHOUT GOSUB

Cause and Suggested Remedies

A RETURN statement was encountered without a previous matching GOSUB statement being executed.

Insert a GOSUB statement or delete the RETURN statement. The error may be caused by dropping into the subroutine code inadvertently. In this case correct the program flow. An END or STOP statement placed just ahead of the subroutine serves as a debugging aid.

STRING TOO LONG

An attempt was made by use of the concatenation operator (+) to create a string longer than 255 characters.

Break the string into two or more shorter strings as part of the program operation. Use the LEN function to check string lengths before concatenating them.

SYNTAX

There is a syntax error in the line just entered (immediate mode) or scanned for execution (program mode). This is the most common error message, and is caused by such things as misspellings, incorrect punctuation, unmatched parentheses, extraneous characters, etc.

Examine the line carefully and make corrections. Note that syntax errors in a program are diagnosed at run time, not at the time the lines are entered from the keyboard. You can eliminate many syntax error messages by carefully scrutinizing newly entered program lines before running the program.

TYPE MISMATCH

An attempt was made to enter a string into a numeric Assignment variable or vice versa, or an incorrect type was given as a function parameter.

Change the offending item to correct type. Refer to Chapter 4 for acceptable parameter types.

UNDEF'D STATEMENT

An attempt was made to branch to a nonexistent line number.

Insert a statement with the necessary line number or branch to another line number.

Error Message

Cause and Suggested Remedies

UNDEF'D FUNCTION

Reference was made to a user defined function that has not previously been defined by appearing in a DEF FN statement. The definition must precede the function reference.

Define the function. Place DEF FN statements near the beginning of the program.

OPERATING SYSTEM ERROR MESSAGES

BAD DATA

String data was input when numeric data was expected.

Correct the input data to numeric, or change the program to accept string input.

DEVICE NOT PRESENT

No device on the IEEE 488 Bus was present to handshake an attention sequence. The Status function will have a value of 2, indicating a timeout. This message may occur for any I/O command.

If the device identification is in error, correct the OPEN (or other) statement. If the statement is correct, especially if it has worked before, check the addressed device for malfunction, misconnection, or power off.

FILE NOT FOUND

The filename given in the LOAD or OPEN statement was not found on the tape (or other specified device).

Check that you have the correct tape in the cassette. Check the filenames on the tape (or other medium) for possible spelling error in the program statement.

FILE NOT OPEN

An attempt was made to access a file that was not opened via the OPEN statement.

Open the file.

FILE OPEN

An attempt was made to open a file that has already been opened via a previous OPEN statement.

Check logical file numbers (first parameter in the OPEN statement) to be sure a different number is used for each file. Insert a CLOSE statement if you want to reopen the same file for a different I/O operation.

Error Message

Cause and Suggested Remedies

LOAD

An unacceptable number of tape errors were accumulated on a tape load (more than 31) that were not cleared on reading the redundant block. This message is issued in connection with the LOAD command (see Chapter 4).

NOT INPUT FILE

An attempt was made to read from a file that has been opened for output only.

Check the READ# and OPEN statement parameters for correctness. Reading requires a zero as the third parameter of the OPEN statement (this is the default option).

NOT OUTPUT FILE

An attempt was made to write to a file that has been opened for input only.

Check the PRINT# and OPEN statement parameters for correctness. Writing to a file requires a 1 (or a 2 if you want an EOT at the end of the file) as the third parameter in the OPEN statement.

VERIFY

The program in memory and the specified file do not compare. This message is issued in connection with the VERIFY command (see Chapter 4).

APPENDIX C

Program Examples Solved

This appendix contains the solutions to the programming problems presented at the conclusion of Chapter 3. The original program BLANKET as developed in Chapter 3 is listed first. The program variations that follow are based on modifying this original program according to the tasks presented in Chapter 3. For each program modification the program name is shown, followed by a program listing of the modified program. The Changes identify those lines in the original program that have been added, changed, or deleted. Under Comments is a description of the modifications that have been made.

BLANKET

```
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT "HIT A KEY OR <CR> TO END";
100 GET C$:IF C#="" GOTO 100
105 IF C#=CHR$(13) GOTO 170
110 PRINT "J"; :REM CLEAR SCREEN
120 FOR I=1 TO 920 :REM 920/40=23 LINES
130 PRINT C#;
140 NEXT
150 PRINT "PHEW!"
160 GOTO 90
170 END
```

Program 1, CLEAN SCREEN.

```
5 REM PROGRAM 1 CLEAN SCREEN
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
80 PRINT "J";
90 PRINT "HIT A KEY OR <CR> TO END";
100 GET C$:IF C#="" GOTO 100
105 IF C#=CHR$(13) GOTO 170
110 PRINT "J"; :REM CLEAR SCREEN
120 FOR I=1 TO 920 :REM 920/40=23 LINES
130 PRINT C#;
140 NEXT
150 PRINT "PHEW!"
160 GOTO 90
170 END
```

Changes

Add:	Lines 5, 80
Change:	None
Delete:	None

Comments

This is an easy change. All you have to do is print a Clear Screen literal just before the HIT A KEY message. The reset of the program stays the same.

Program 2, MESSAGES.

```
5 REM PROGRAM 2  MESSAGES
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT" "; REM CLEAR SCREEN
120 FOR I=1 TO 920 REM 920/40=23 LINES
130 PRINT C$;
140 NEXT
150 PRINT"PHEW!"
155 PRINT"HIT A KEY OR <R> TO END";
160 GOTO 100
170 END
```

Changes

Add:	Lines 5, 155
Change:	Lines 90, 160
Delete:	None

Comments

This is also an easy change. First change the original message at line 90 to the shortened, initial message. Then add the repeating message to print at the end of the sequence just before the branch back at line 160. You also have to change the GOTO at line 160 to branch below the initial message line.

Program 3, SINGLE.

```
5 REM PROGRAM 3 SINGLE
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"J"; :REM CLEAR SCREEN
120 FOR I=1 TO 960 :REM 960/40=24 LINES
130 PRINT C$;
140 NEXT
150 PRINT"PHEW! ";
160 GOTO 90
170 END
```

Changes

Add:	Line 5
Change:	Lines 120, 150
Delete:	None

Comments

To combine the messages, add a semicolon (for continuous-line format) following the PHEW! message. The ten spaces were added following this word, but they could just as well have been put ahead of the second message. The FOR . . . NEXT ending count of 920 gives $920/80=23$ lines. To get 24 lines, change the 920 to 960.

Program 4, CALLING YOU.

```
5 REM PROGRAM 4 CALLING YOU
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
60 GOSUB 90
70 END
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT" "; :REM CLEAR SCREEN
120 FOR I=1 TO 920 :REM 920/40=23 LINES
130 PRINT C$;
140 NEXT
150 PRINT"PHEW!"
160 GOTO 90
170 RETURN
```

Changes

Add:	Lines 5, 60, 70
Change:	Line 170
Delete:	None

Comments

The program from line 90 through line 170 can be structured as a subroutine simply by changing line 170 from an END to a RETURN. Add a two-line main program just ahead of the subroutine to call it by a GOSUB 90 and then END. Note that the subroutine maintains control until the RETURN key is struck.

Program 5, SLOW DOWN.

```
5 REM PROGRAM 5 SLOW DOWN
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C#="" GOTO 100
105 IF C#=CHR$(13) GOTO 170
110 PRINT"C"; :REM CLEAR SCREEN
120 FOR I=1 TO 920 :REM 920/40=23 LINES
130 PRINT C$;
135 FOR J=1 TO 10:NEXT J :REM DELAY
140 NEXT I
150 PRINT"PHEW!"
160 GOTO 90
170 END
```

Changes

Add: Lines 5, 135
Change: Line 140
Delete: None

Comments

A "do-nothing" FOR . . . NEXT loop acts as a simple delay loop. Looping from 1 to 10 after each character print approximately doubles the processing time. You can change the delay time just by changing the end index value (in this case 10). Optionally add the index variable I to the NEXT in line 140. Using index names on NEXT statements is a good idea, especially when you have nested FOR . . . NEXT loops.

Program 6, SPEED UP.

```
5 REM PROGRAM 6 SPEED UP
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT "HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C#=CHR$(13) GOTO 170
106 FOR I=1TO7
107 C#=C#+C$
108 NEXT :REM C#=128 C#'S
109 C#=C#+LEFT$(C$,102) :REM C#=230 C#'S
110 PRINT " "; :REM CLEAR SCREEN
120 FOR I=1TO4 :REM 230*4=23 LINES
130 PRINT C$;
140 NEXT
150 PRINT "PHEW!"
160 GOTO 90
170 END
```

Changes

Add:	Lines 5, 106-109
Change:	Line 120
Delete:	None

Comments

Here you have to realize that one long string prints more quickly than a number of shorter strings. Ideally we would print one 920-character string. However, on the PET, strings are limited to a maximum of 255 characters. The program breaks the 920 positions into four 230-character quadrants. At lines 106 to 109 it concatenates the keyed-in character into a 230-character string, which is a quarter of the screen area to be covered by the display character. The FOR . . . NEXT loop at lines 106 to 108 uses a pyramided concatenation to quickly "grow" the string to 128 characters. Then at line 109 the 128-character string is concatenated to the leftmost 102 characters of the string to give a 230-character string of the display character. The FOR . . . NEXT print loop at lines 120 to 140 needs to print this longer string only four times to fill the screen.

Program 7, INDIAN BLANKET.

```
5 REM PROGRAM 7 INDIAN BLANKET
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF
30 REM CHARACTERS ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT ANY KEYS";
100 GET C$:IF C$="" GOTO 100
110 PRINT" "; :REM CLEAR SCREEN
120 FOR I=1 TO 920 :REM 920/40=23 LINES
130 PRINT C$;
134 FOR J=1 TO 40:NEXT J :REM DELAY
135 GET D$:IF D$<>"" THEN C$=D$
137 IF D$=CHR$(13) GOTO 170
140 NEXT
150 PRINT"PHEW!"
155 PRINT"HIT ANY KEYS OR <R> TO END";
160 GOTO 100
170 END
```

Changes

Add:	Lines 5, 134, 135, 137, 155
Change:	Lines 20, 30, 90, 160
Delete:	Line 105

Comments

Line 135 allows different characters to be keyed in during the display. If a key has been struck, it is fetched as D\$, and C\$ is assigned the new character. Line 137 is added to handle the case where the RETURN key is struck during a display sequence. The messages were modified by changing lines 90 and 160 and adding line 155 just as described for Program 2. The program is slowed down by a delay loop at line 134 exactly the same as in Program 5 (line 135) except that the delay time is lengthened by having an end value of 40 (instead of 10 as in Program 5).

Program 8, PRINTOVER I.

```
5 REM PROGRAM 8 PRINTOVER I
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"J"; :REM CLEAR SCREEN
120 FOR I=1 TO 4 :REM 230*4=23 LINES
130 PRINT C$;
140 NEXT
150 PRINT"PHEW!"
160 GOTO 90
170 END
```

Changes

Add:	Line 5
Change:	Line 110
Delete:	None

Comments

This is the easiest program modification so far — just change the Clear Screen literal at line 110 to a Home Cursor!

However, the program display suffers from two things caused by not blanking the screen. First, there may be extraneous characters left in the bottom two rows that are not blanked out or overwritten by the program; this is because the PHEW! and HIT A KEY messages don't cover the entire lines, nor does the PET blank out the rest of the lines. Second, the character display appears not to move if the same character is keyed in again; this also holds for the two messages printed at the bottom of the screen. For this program's purposes, you would prefer to see some movement at the cursor position when printing a character over the same character.

Program 9, PRINTOVER II.

```
5 REM PROGRAM 9 PRINTOVER II
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
85 Q$="" " :REM 39 BLANKS
86 R$=" "+Q$ :REM 40 BLANKS
87 S$=R$+Q$ :REM BLANKS FOR 2 LAST LINES
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
107 IF C$>0 THEN GOSUB 600 :REM TO BLANK 2 LINES
110 PRINT"␣"; :REM HOME CURSOR
120 FOR I=1 TO 919 :REM 919/40=23 LINES
130 PRINT C$;
135 PRINT "␣";C$;"␣"; :REM PRINT REV
140 NEXT
141 PRINT C$;
142 GOSUB 610 :REM TO BLANK 2 LINES
150 PRINT"PHEW!"
155 FOR I=1 TO 1000:NEXT :REM PAUSE
157 F=1
160 GOTO 90
170 END
600 PRINT"XXXXXXXXXXXXXXXXXXXXXXXXXXXX"; :REM TO NEXT-TO-LAST
610 PRINT S$;"␣"; :REM TO NEXT-TO-LAST
620 RETURN
```

Changes

Add:	Lines 5, 85-87, 107, 135, 141-142, 155, 157, 600-620
Change:	Lines 110, 120
Delete:	None

Comments

The extensive changes made here illustrate the lengths you have to go to in order to smooth the rough edges of a simple program modification. (Recall that in PRINTOVER I you had to change just one character of the program to get the typeover effect!)

To show movement at the printing of each character, the character is printed (line 130) and then in the next position the reverse of the character is printed (line 135). At line 135 the cursor is also backed up one so that the next loop iteration will print the character in the same position where the reverse character was just printed. The end loop index in line 120 is changed from 920 to 919 so that the reverse character, which runs one ahead of C\$, will not print beyond the 920th position. Line 141 has to be added to print C\$ at this last character display position.

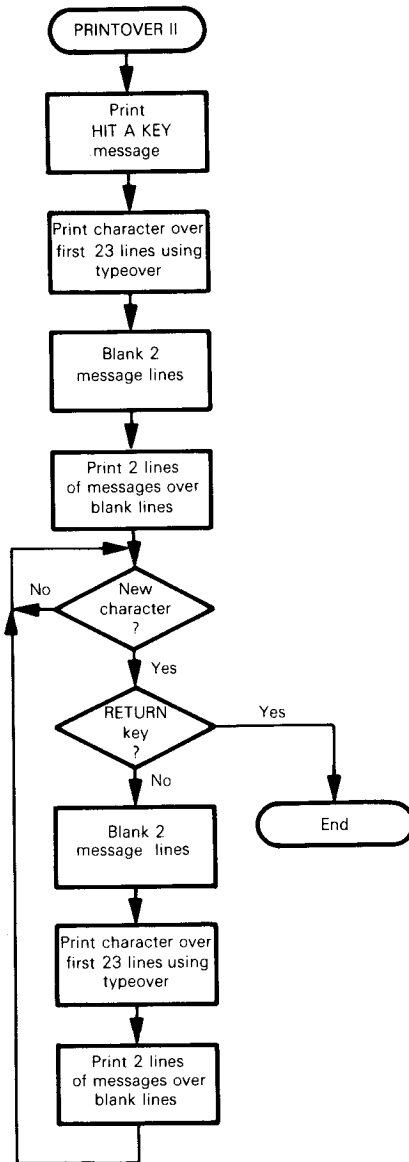
The remaining changes are for the message display lines. Lines 85 to 87 create S\$, a string of 79 blanks. This could have been done with a single statement (S\$=" <79 blanks >") but the three lines are more informative and make use of the PET concatenation feature. Why 79 instead of 80 blanks (since the two lines are each 40 characters long)? You will find, if you have tried this, that printing into the last character position of the screen always causes scrolling up one line. Therefore: first, you will never find a character that needs to be blanked sitting in the last position, so that blanking up to but not including the last position is far enough. And second, to perform the indicated task, you need to blank the bottom two lines of the screen but you do not want to scroll the screen up, thereby losing the top line of the character display and making the program look amateurish.

After the character display has completed (at line 141), line 142 calls a subroutine beginning at line 610 to print the blank string S\$, followed by a continuous-line format indicator (;) to leave the cursor at the last position on the screen. Now you want to return the cursor to the beginning of the first blank line, which is done by cursor control characters to move the cursor up two lines and right one character position. The subroutine terminates and returns control to the main program, and at line 150 the program prints PHEW!. Then, as stipulated by the task, a delay loop causes the program to pause a few seconds by doing nothing for 1000 iterations. This completes the program's processing of the first display character.

The task specifies that the program blank the two message lines before starting the new character display. This also has to be done for succeeding display characters, i.e., all but the first one. A flag F is used to indicate the "first time/not first time" condition. Initially F is zero. After completing the first display character, at line 157 F is set to 1 so that at line 107 the Blank 2 Lines subroutine is called before beginning the display of the next character. The subroutine beginning at line 600 is called this time, which moves the cursor to the beginning of the message area by homing the cursor and then moving the cursor down 23 lines. Execution proceeds to line 610 to print the blank string S\$ and return. (The cursor movements at line 610 are irrelevant now, since the next operation on returning from the subroutine to line 110 is to home the cursor.) After the cursor is homed, the display loop executes (lines 120 to 141). Now the program again calls the Blank 2 Lines subroutine (line 142) before printing the two messages. This is a redundant call, since the two lines are being blanked before the display loop for the second and subsequent display characters. The only time the lines need to be blanked at this point is for the first character. This line is probably better written:

```
142 IF F=0 THEN GOSUB 610: REM TO BLANK 2 LINES
```


The simplified flowchart below summarizes the program flow.



Program 10, ALL THE WAY.

```
5 REM PROGRAM 10 ALL THE WAY
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"J"; REM CLEAR SCREEN
120 FOR I=1 TO 999 REM 999/40=25 LINES
130 PRINT C$;
140 NEXT
150 POKE 33767,(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
160 GOTO 100
170 END
```

Changes

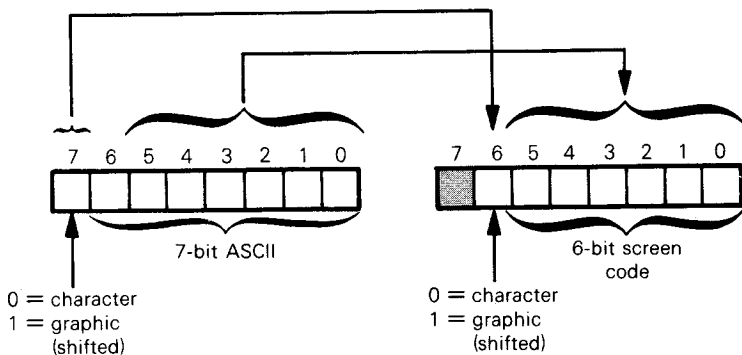
Add:	Line 5
Change:	Lines 120, 150, 160
Delete:	None

Comments

Following the hint given with the problem in Chapter 3, we can get the character to print in 999 positions simply by changing the value of the FOR . . . NEXT ending index in line 120 from the value 920 to the value 999. However, the program cannot PRINT into the 1000th screen position without scrolling the display up one line. Recall that the previous program, PRINTOVER II, had to stop short of printing in the final position to avoid scrolling — but that program just wanted to blank the position, and it was found that the position would always be blank anyway.* The current program needs to display a character at that position. The only way to accomplish this with the PET is to POKE the character's screen value into address 33767, which corresponds to the last screen position. POKE does not use the cursor and thus does not produce scrolling.

*The final position will be blank unless you POKEd a character there before running the program. If you wanted to be absolutely sure about blanking this last position, you could do so by POKeing a blank to it with a POKE 33767,32.

The tough part of this exercise is that the screen display code needed to POKE is different from the PET ASCII code that C\$ is assigned. As covered in Chapter 6, the screen code is equivalent to a 6-bit ASCII subset with ASCII bit 7 moved over into screen bit 6:



This is what line 150 in the program does with the logic operation:

```
(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
```

We'll go over this operation piece by piece.

ASC(C\$) provides the numerical equivalent of the character C\$. As a number, it can be manipulated arithmetically (and logically).

AND 128 $128_{10}=80_{16}$. This AND operation masks off (zeroes) all but the high-order bit (bit 7) of the ASCII code to isolate it. Regardless of the previous value of C\$, after this operation the term has a value of:

$$10000000_2 \text{ or } 00000000_2.$$

/2 Divide by 2. In binary, division by 2 is equivalent to a logical right shift one bit. It admittedly requires a little background to come up with this step. After this operation, the term has a value of:

$$01000000_2 \text{ or } 00000000_2.$$

Thus, we have moved bit 7 of the ASCII code into bit 6 and replaced bit 7 with a zero.

OR This operation will combine the bit configuration already developed (01000000_2 or 00000000_2) with the bit configuration yet to be discussed, $(ASC(C$)AND63)$. The OR operation is performed last (see Table 3-4).

ASC(C\$) again provides the numerical equivalent of C\$.
AND 63 $63_{10}=3F_{16}$. This AND operation zeroes the two high-order bits of the ASCII code (which were developed in the previous expression), leaving the five low-order bits ready to be ORed in. After this operation, this term has a value of:

$00xxxxx_2$

where $x=0$ or 1 .

Now the OR operation is performed, producing the screen memory code equivalent to the ASCII code of the character C\$. After the OR operation, the combined term has a value of:

$01xxxxx_2$ or $00xxxxx_2$

This is the value POKEd into the last screen position at line 150. Since no further messages are to be printed, line 160 branches past the HIT A KEY message.

Program 11, BOTTOM UP.

```
5 REM PROGRAM 11 BOTTOM UP
10 REM ***** B L A N K E T *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 PRINT"J"; REM CLEAR SCREEN
110 IF C#=CHR$(13) GOTO 170
115 PRINT"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
117 POKE 33767,(ASC(C$)ANS128)/2 OR (ASC(C$)AND 63)
120 FOR I=1 TO 999 REM 999/40=25 LINES
130 PRINT C$;"■";
140 NEXT
150 POKE 33767,(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
160 GOTO 100
170 END
```

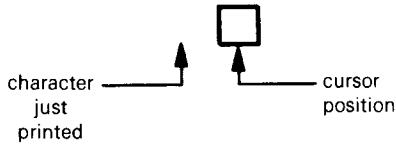
Changes

Add:	Lines 5, 115, 117
Change:	Lines 120, 130, 160
Delete:	Line 150
Exchange:	Lines 105 and 110

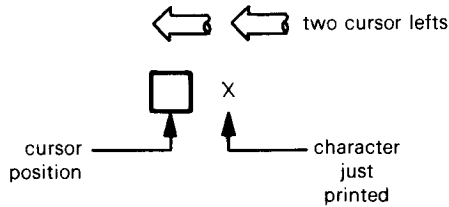
Comments

To print in reverse order, the program first needs to display the character at the last screen position (lower right-hand corner). We saw in Program 10 that this could be done only by POKEing the screen character equivalent of the C\$ character code; the mechanics were all worked out in Program 10. In the current program, line 117 is an exact duplicate of the Program 10 POKE line; it displays the C\$ character at the last screen position.

The next task is to print the character at the next-to-last position. At line 115 the cursor is moved from its home position to position 1 of the last line by 24 cursor downs and then over to the next-to-last position by 38 cursor rights. After POKEing the last character (line 117), the program begins printing C\$ to the other 999 positions of the screen. After each PRINT (line 130) the cursor rests at the position following the position where the last character was printed.



The cursor is moved left two positions (also at line 130) so that the next PRINT will print into the position to the left of the character just printed.



As in Program 10, line 120 is changed to print 999 characters instead of 920; the PHEW! message at line 150 is deleted; and at line 160 the program branches past the HIT A KEY message, which is printed only once at the beginning of the program.

The last task, to clear the screen before ending, is accomplished simply by switching lines 105 and 110 so that the check for a RETURN key is done after clearing the screen.

APPENDIX D

BASIC Bibliography

- Advanced BASIC. James S. Coan, Hayden Book Co., Rochelle Park, New Jersey.
- BASIC. Albrecht, Finkle, and Brown, Peoples Computer Company, Menlo Park, California, 1967.
- BASIC: A Computer Programming Language. C. Pegels, Holden-Day, Inc., 1973.
- Basic BASIC. James S. Coan, Hayden Book Company, Rochelle Park, New Jersey.
- BASIC Programming. J. Kemeny and T. Kurtz, Peoples Computer Company, Menlo Park, California, 1967.
- Entering BASIC. J. Sack and J. Meadows, Science Research Associates, 1973.
- A Guided Tour of Computer Programming in BASIC. T. Dwyer, Houghton Mifflin Company, 1973.
- Hands-On BASIC with a PET. Herbert D. Peckham, McGraw-Hill Book Company, New York, 1979.
- Programming Time Shared Computers in BASIC. Eugene H. Barnett, Wiley-Interscience, Library of Congress #72-175789.
- What to Do After You Hit Return. Peoples Computer Company, Menlo Park, California.

APPENDIX E

PET Newsletters and References

This appendix contains a listing of PET-related publications for PET users who want to seek out continuing sources of information on the PET. Many of these sources contain notices of PET user groups and activities. No endorsement of these publications is implied.

Newsletters

Calculators/Computers Magazine, Box 310, Menlo Park, California 94025. Bimonthly. \$10.00 year. A magazine that has several PET articles in each issue.

Commodore PET Users Club Newsletter, Commodore Business Machines, Inc., 3330 Scott Blvd., Santa Clara, California 95050. Monthly. \$15.00 year U.S., \$25.00 year foreign. Official Commodore newsletter in U.S.

Commodore PET Users Club Newsletter, Commodore Systems, 360 Eusten Rd., London, England NW1 3BL. Bimonthly. £10. Official Commodore newsletter in Europe.

CURSOR, P.O. Box 550, Goleta, California 93017. Monthly. \$33.00 year. A cassette magazine — you receive a tape cassette of programs that can be loaded into the PET. Each cassette comes with a 2-page newsletter/program description.

MICRO, The 6502 Journal, 8 Fourth Lane, South Chelmsford, Massachusetts 01824. Bimonthly. Single copies \$1.50, \$6.00 year. A magazine that has several PET articles in each issue. For the experienced PET user.

People's Computers, 1263 El Camino Real, Box E, Menlo Park, California 94025. Bimonthly. Single copies \$1.50, \$8.00 year. A magazine that has several PET articles in each issue.

PET Gazette, 1929 Northport Drive, #6, Madison, Wisconsin 53704. Bimonthly. Free (donations appreciated). Of great value.

Best of the PET Gazette, 1979. 96 pages. Free with donation to PET Gazette (see above). Retail value \$10.00.

The PET Paper, P.O. Box 43, Audubon, Pennsylvania 19407. 10 issues/year (monthly except July and December). Single copies \$2.00, \$15.00 per calendar year.

PET User Notes, P.O. Box 371, Montgomeryville, Pennsylvania 18936. Bimonthly. \$6.00 for 6 issues in U.S. and Canada, \$12.00 for airmail to other countries. A good newsletter.

PET Users Group Newsletter, Lawrence Hall of Science, University of California, Berkeley, California 94720. Monthly. \$4.50 for 6 integral issues, checks payable to Regents of the University of California. Highly recommended.

Purser's Reference List of Computer Cassettes. Quarterly. Single copy \$4.00 domestic, \$5.00 foreign. \$12.00 year domestic, \$16.00 year foreign. Extensive list of PET programs available on cassette.

Reference Manuals

MCS6500 Microcomputer Family Programming Manual, MOS Technology, Inc., 950 Rittenhouse Road, Norristown, Pennsylvania 19401. \$10.00 (price may vary with location). By the manufacturers of the 6502 microprocessor.

MCS6500 Microcomputer Family Hardware Manual, MOS Technology, Inc., 950 Rittenhouse Road, Norristown, Pennsylvania 19401. \$10.00 (price may vary with location). By the manufacturers of the 6502 microprocessor.

PET and the IEEE 488 Bus (GPIB), E. Fisher and C. W. Jensen, Osborne/McGraw-Hill, 630 Bancroft Way, Berkeley, California 94710. 1980. \$15.00.

PET 2001-8 Personal Computer User Manual, Commodore Business Machines, Inc., 3330 Scott Blvd., Santa Clara, California 95050. (8K system). \$9.95. By the manufacturers of the PET computer.

PET 2001-16, 16N, 32, 32N Personal Computer User Manual, Commodore Business Machines, Inc., 3330 Scott Blvd., Santa Clara, California 95050. (16K and 32K systems.) \$9.95. By the manufacturers of the PET computer.

6502 Assembly Language Programming, Lance Leventhal, Osborne/McGraw-Hill, 630 Bancroft Way, Berkeley, California 94710. 1979. \$12.50.

APPENDIX F

Conversion Tables

This appendix contains the following reference tables:

- Hexadecimal-Decimal Integer Conversion
- Powers of Two
- Mathematical Constants
- Powers of Sixteen
- Powers of Ten

HEXADECIMAL-DECIMAL INTEGER CONVERSION

The table below provides for direct conversions between hexadecimal integers in the range 0-FFF and decimal integers in the range 0-4095. For conversion of larger integers, the table values may be added to the following figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	800 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

Hexadecimal fractions may be converted to decimal fractions as follows:

- Express the hexadecimal fraction as an integer times 16^{-n} , where n is the number of significant hexadecimal places to the right of the hexadecimal point.

$$0. CA9BF3_{16} = CA9BF3_{16} \times 16^{-6}$$

- Find the decimal equivalent of the hexadecimal integer

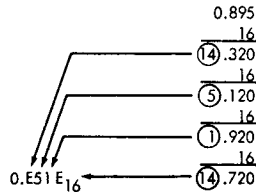
$$CA9BF3_{16} = 13\,278\,195_{10}$$

- Multiply the decimal equivalent by 16^{-n}

$$\begin{array}{r} 13\,278\,195 \\ \times 596\,046\,448 \times 10^{-16} \\ \hline 0.791\,442\,096_{10} \end{array}$$

Decimal fractions may be converted to hexadecimal fractions by successively multiplying the decimal fraction by 16_{10} . After each multiplication, the integer portion is removed to form a hexadecimal fraction by building to the right of the hexadecimal point. However, since decimal arithmetic is used in this conversion, the integer portion of each product must be converted to hexadecimal numbers.

Example: Convert 0.895_{10} to its hexadecimal equivalent



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
20	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
27	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
30	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4C	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
50	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
60	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
70	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
80	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A0	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B0	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C0	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D0	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E0	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 396 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 428 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 661 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 831 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 986 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 775 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 81 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

MATHEMATICAL CONSTANTS

Constant	Decimal Value	Hexadecimal Value
π	3.14159 26535 89793	3.243F 6AB9
π^{-1}	0.31830 98861 83790	0.517C C187
$\sqrt{\pi}$	1.77245 38509 05516	1.C58F 891C
$\ln \pi$	1.14472 98858 49400	1.250D 048F
e	2.71828 18284 59045	2.87E1 5163
e^{-1}	0.36787 94411 71442	0.5E2D 58D9
\sqrt{e}	1.64872 12707 00128	1.A612 98E2
$\log_{10} e$	0.43429 44819 03252	0.6F2D EC55
$\log_2 e$	1.44269 50408 88963	1.7154 7653
γ	0.57721 56649 01533	0.93C4 67E4
$\ln \gamma$	-0.54953 93129 81645	-0.8CAE 98C1
$\sqrt{2}$	1.41421 35623 73095	1.6A09 E668
$\ln 2$	0.69314 71805 59945	0.8172 17F8
$\log_{10} 2$	0.30102 99956 63981	0.4D10 4D42
$\sqrt{10}$	3.16227 76601 68379	3.2988 075C
$\ln 10$	2.30258 40929 94046	2.4D75 3777

POWERS OF SIXTEEN₁₀

16^n		n	16^{-n}									
1		0	0.10000	00000	00000	00000	x 10 ⁰					
16		1	0.62500	00000	00000	00000	x 10 ⁻¹					
256		2	0.39062	50000	00000	00000	x 10 ⁻²					
4	096	3	0.24414	06250	00000	00000	x 10 ⁻³					
65	536	4	0.15258	78906	25000	00000	x 10 ⁻⁴					
1	048	576	5	0.95367	43164	06250	00000	x 10 ⁻⁵				
16	777	216	6	0.59604	64477	53906	25000	x 10 ⁻⁶				
268	435	456	7	0.37252	90298	46191	40625	x 10 ⁻⁷				
4	294	967	296	8	0.23283	06436	53869	62891	x 10 ⁻⁸			
68	719	476	736	9	0.14551	91522	83668	51807	x 10 ⁻⁹			
1	099	511	627	776	10	0.90949	47017	72928	23792	x 10 ⁻¹⁰		
17	592	186	044	416	11	0.56843	41886	08080	14870	x 10 ⁻¹¹		
281	474	976	710	656	12	0.35527	13678	80050	09294	x 10 ⁻¹²		
4	503	599	627	370	496	13	0.22204	46049	25031	30808	x 10 ⁻¹³	
72	057	594	037	927	936	14	0.13877	78780	78144	56755	x 10 ⁻¹⁴	
1	152	921	504	606	846	976	15	0.86736	17379	88403	54721	x 10 ⁻¹⁵

POWERS OF TEN₁₆

10^n		n	10^{-n}						
1		0	1.0000	0000	0000	0000			
A		1	0.1999	9999	9999	999A			
64		2	0.28F5	C28F	5C28	F5C3	x 16 ⁻¹		
3E8		3	0.4189	374B	C6A7	EF9E	x 16 ⁻²		
2710		4	0.68DB	8BAC	710C	B296	x 16 ⁻³		
1	86A0	5	0.A7C5	AC47	1B47	8423	x 16 ⁻⁴		
F	4240	6	0.10C6	F7A0	85ED	8D37	x 16 ⁻⁴		
98	9680	7	0.1AD7	F29A	8CAF	4858	x 16 ⁻⁵		
5F5	E100	8	0.2AF3	1DC4	6118	73BF	x 16 ⁻⁶		
389A	CA00	9	0.44B8	2FA0	9B5A	52CC	x 16 ⁻⁷		
2	540B	E400	10	0.6DF3	7F67	5EF6	EADF	x 16 ⁻⁸	
17	4876	E800	11	0.AFEB	FF0B	CB24	AAFF	x 16 ⁻⁹	
E8	D4A5	1000	12	0.1197	9981	2DEA	1119	x 16 ⁻⁹	
916	4E72	A000	13	0.1C25	C268	4976	81C2	x 16 ⁻¹⁰	
5AF3	107A	4000	14	0.2D09	370D	4257	3604	x 16 ⁻¹¹	
3	8D7E	A4C6	8000	15	0.480E	BE7B	9D58	566D	x 16 ⁻¹²
23	8652	6FC1	0000	16	0.734A	CA5F	6226	F0AE	x 16 ⁻¹³
163	4578	5D8A	0000	17	0.8877	AA32	36A4	B449	x 16 ⁻¹⁴
DE0	B6B3	A764	0000	18	0.1272	5DD1	D243	ABA1	x 16 ⁻¹⁴
8AC7	2304	89E8	0000	19	0.1D83	C94F	86D2	AC35	x 16 ⁻¹⁵

Variations for Original ROMs (Revision Level 2)

This appendix describes the differences between the new ROMs, as presented in the chapter material, and the old ROMs.

Chapter 2: STARTUP

Asterisks (*) appear in place of the pound signs (#) in the initial display line of the original ROMs:

```
***COMMODORE BASIC***
```

You can use this as an indicator of which ROMs your PET has.

Chapter 3: ARRAYS

On the old ROMs, the total number of array elements in any one array is limited to 256. For example, for a one-dimensional array, elements may go from 0 to 255. For a two-dimensional array with dimension 2 in the second subscript, elements may go from (0,0), (1,0). . .(127,0), (01), (1,1). . .(127,1).

An example of programming within this restriction is given below under "Chapter 5."

Chapter 3: DEVELOPING A PROGRAM, Interactive Programming

In the old ROMs, the system location that enables the cursor to blink is location 548. To enable the cursor, you would use the statement:

```
80 POKE 548,0           Enable cursor (old ROMs)
```

instead of

```
80 POKE 167,0          Enable cursor (new ROMs)
```

Chapter 4: RND

RND(0) is non-functional. An argument of zero returns a value that is constant, or nearly constant, and that may vary from PET to PET.

You will have to use -TI to generate random seeds. This is the method used in all of the examples in Chapter 5 under "Generating Random Numbers."

Chapter 5: FILES

This section is for PET users who are having problems reading data files using the old ROMs. If your PET has the old ROMs and you intend to use data files frequently, you should seriously consider replacing the old ROMs with the new, as the new ROMs ensure greater reliability when reading and writing data files.

If you do plan to use the old ROMs, you must do a little extra programming to get around these problems. When writing data to the data tape, the old ROMs neglect to initialize the pointer to the start address of the cassette tape buffer, and also fail to leave enough blank space on the tape between physical records. Consequently, when the PET attempts to read the data back from the data tape, the problems may result in lost or garbled data. Here are a few precautions you can take to overcome these obstacles.

1. Initialize the pointer of the cassette buffer start address. Because the old ROMs fail to initialize the start address to the cassette tape buffer before a file is OPENed, you must be sure to do so before opening a file with a series of POKES:

```
cassette #1: POKE 243,122:POKE 244,2:OPEN 1,1,1
cassette #2: POKE 243,58 :POKE 244,3:OPEN 2,2,1
```

Memory address locations 243 and 244 point to the start address of the current tape buffer. By POKeing in the above values the pointer will be initialized properly.

2. Force interrecord gaps. The old ROMs do not leave enough blank space on the tape between physical records. When the PET attempts to read back the data with an INPUT# or GET#, if the physical records are too close together the data cannot be read, resulting in read errors and lost data. To prevent this, you can force larger interrecord gaps to be written between records by calling a routine to advance the tape each time the cassette buffer is emptied.

Before forcing an interrecord gap you must detect when the cassette buffer has written out a "physical record" or "block" of data to the tape. The buffer holds 191 characters (or 191 bytes). A full buffer is a signal that a block of data was just written to the tape since the contents of the buffer are dumped only after it has reached its capacity. By detecting a full buffer, you can infer that a block of data was just written to the tape and an interrecord gap is needed.

How to Detect a Full Buffer

When writing data out to a tape, following each PRINT# statement the length of each data item is calculated and kept in an accumulator, which is then compared to the buffer limit (191 characters). When the accumulator equals 191 the writing to the tape is stopped until an interrecord gap is written on the tape. Below is a sample program:

```
10 POKE 243,122:POKE 244,2:OPEN1,1,1
20 FOR X=1 TO 100
30 PRINT#1,X
40 A=LEN(STR$(X))+1
50 IF (QT+A)>=191 GOSUB 1000 :REM *IF BUFFER FULL CALL SUB.
60 QT=QT+A :TO ADVANCE TAPE*
70 NEXT X
80 CLOSE1
90 END
```

Line 20 prints a variable. If the variable printed (in this case, X) is numeric it must be converted to string form so the LEN function may be used to determine X's length, as shown in line 40:

```
40 A=LEN(STR$(X))+1
```

One is added on to the lengths of the strings to include the carriage returns that are written on the tape following each data item. Line 50 accumulates the number of characters in the previous strings, (QT), plus A, and compares the total to 191 (the buffer limit). If the number of characters written to the tape (QT + A) is greater than or equal to 191 the entire buffer is written to the tape, and it is time to force an interrecord gap by calling the subroutine at 1000. However, if QT+A is less than 191 (QT+A < 191), the buffer is not yet full. Line 60 increments QT by A, and the process keeps repeating until the buffer is full, and all the data is written from the buffer to the tape, interspersed with the interrecord gaps.

Advancing the Cassette Tape

There are three necessary steps in the routine to advance the tape:

1. Turn on the cassette tape motor (POKE 59411,53).
2. Use a wait loop to stall program while tape is advancing.
3. Turn off the cassette tape motor (POKE 59411,61).

POKE 59411,53 pokes "53" into memory address location 59411 which controls the cassette motor. Value 53 turns on the motor to advance the tape. Once the motor is on, a wait loop lets the tape advance for a few jiffies. The wait loop will be discussed shortly. To stop the tape, a POKE 59411,61 turns off the cassette motor. The length of the wait loop may be varied or altered, but these two POKES are absolutely necessary to turn the cassette motor on and off.

Following is a sample wait loop inserted between the two POKE statements:

```
1000 POKE 59411,53          :REM *START TAPE MOTOR*
1010 T=TI
1020 IF (TI-T)<10 GOTO 1020 :REM *WAIT 10 JIFFIES*
1030 POKE 59411,61        :REM *STOP TAPE MOTOR*
1040 QT=0
1050 RETURN
```

Lines 1010 to 1020 make up the wait loop. Line 1010 sets variable T to the current value of TI. TI is the number of jiffies since the PET was powered up or the clock was zeroed. (A jiffy is 1/60 of a second.) TI is incremented once every jiffy, or 60 times a second. By subtracting T from TI, the elapsed time is calculated. The program must wait until ten jiffies (1/60 of a second) has elapsed before the program can continue. While TI increments, until the difference between TI and T equals ten jiffies the program is stalled, letting the cassette tape advance. This blank space on the tape is the interrecord gap. Once (TI-T) equals ten, the next statement turns off the cassette motor with a POKE 59411,61.

The routine calculates the space between each record. The tape is advanced exactly the same amount between each physical record because the time between POKEing on and off the cassette motor will always be ten jiffies:

The length of the wait loop may be adjusted by changing the constant of the condition expression:

$$TI - T < X$$

The larger the value of X, the larger the interrecord gap will be. If you're unsure how long the interrecord gap should be, keep the wait loop between 5 and 30 jiffies. It is always better to have the interrecord gap too long than too short.

There is one potential problem with this routine, though it is doubtful you will ever encounter the problem. If the PET has been powered up for close to twenty-four hours, or you have set the internal clock close to the twenty-fourth hour, the routine might hang up during the wait loop. At 24:00:00 the jiffy clock is reset from 5184000 jiffies to zero. If T is assigned within a few jiffies of 5184000 both TI and the jiffy clock will be reset to zero. The result is that the condition $TI - T < 10$ will always be true ($0000008 - 5183998 < 10$) and the wait loop will hang up infinitely because $TI - T$ will never be greater than nine. It is very improbable that this will ever happen to you, but you should use caution if the jiffy clock is nearing the twenty-fourth hour.

Here is another way to advance the tape:

```
POKE 59411,53      :REM *START TAPE MOTOR*
POKE 514,0         :REM *ZERO JIFFY CLOCK*
WAIT 514,16       :REM *WAITS 16 JIFFIES*
POKE 59411,61     :REM *STOP TAPE MOTOR*
```

POKE 514,0 pokes a zero into the low-order byte of the internal clock at memory address 514, wiping out the current jiffy time and resetting the clock to zero. The WAIT 514,16 inhibits further program action until the clock has incremented 16 jiffies. Meanwhile, the tape advances until memory address location 514 contains 16 and the following POKE turns the cassette motor off.

There is one drawback with this wait loop. Every time the jiffy clock is reset to zero the PET loses track of time. Therefore, this routine should NOT be used if it is important within the program that real time be kept or used in any way.

Here is yet another way to implement a wait loop during the data tape advance:

```
POKE 59411,53
FOR I=1 TO 60:NEXT I
POKE 59411,61
```

This method is simple but less accurate than the previous two. Using a FOR ...NEXT loop, the program is stalled as the loop increments to the maximum value of I before turning off the motor. However, the time it takes to increment through a FOR. . . NEXT loop cannot be measured as accurately as time measured in jiffies, and thus the interrecord gaps cannot be precise. One advantage with this method is that it does not alter or inhibit the use of the jiffy clock in any way.

Let's go back to the original wait loop and combine it with the routine that detects a full buffer. Below is a sample program which writes 100 numbers to a data tape with a FOR...NEXT loop. Within the loop is a check for a full buffer. If the buffer is full the data is written to the tape, and the subroutine at 1000 is called to create an interrecord gap:

```

10 POKE 243,122:POKE 244,2:OPEN1,1,1
20 FOR X=1 TO 100
30 PRINT#1,X
40 A=LEN(STR$(X))+1
50 IF (QT+A)>=191 GOSUB 1000      :REM *IF BUFFER FULL CALL
60 QT=QT+A                        SUB. TO ADVANCE TAPE
70 NEXT X
80 CLOSE1
90 END
1000 POKE 59411,53                :REM *START TAPE MOTOR*
1010 T=TI
1020 IF (TI-T)<10 GOTO 1020       :REM *WAIT 10 JIFFIES*
1030 POKE 59411,61                :REM *STOP TAPE MOTOR
1040 QT=0                          :REM *RESET ACCUMULATOR
1050 RETURN

```

where:

- A is the length of the printed string plus 1 for carriage return
- QT is the accumulator to add lengths of printed strings.

If you follow these suggestions and routines you should have little or no troubles writing and reading data files. But, if you find that you cannot get the files to work even with these routines, you should install the new ROMs in your PET.

Chapter 5: GENERATING RANDOM NUMBERS

Do not try to use RND(-RND(0)) to generate random seeds; it will not work. Instead, use -TI as shown in all of the examples.

The RANDOM VERSION 2A sample program in Chapter 5 will not work on the original ROMs because of the 256-element array limitation. A second version of the program is shown below. It shows the lengths you have to go to in order to program with the 256-element array limitation. In this program the 1000-element table is divided into four quarters of 250 elements each.

```

5 REM RANDOM VERSION 2A
10 REM ***** B L A N K E T *****
20 REM RANDOM DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM THE
40 REM KEYBOARD
50 REM *****
70 DIM T1(249),T2(249),T3(249),T4(249)
75 T=4 :REM NUMBER OF TABLES
76 N=250 :REM NO OF ELEMENTS
80 GOSUB 200 :REM INITIALIZE TABLES
90 PRINT"HIT A KEY OR <R> TO END":
95 N1=N:N2=N:N3=N:N4=N
100 GET C$:IF C$="" GOTO 100

```

```

105 IF C#=CHR$(13) GOTO 170
110 PRINT"D": REM CLEAR SCREEN
120 X=RND(-TI) REM START NEW SEED
125 C=(ASC(C#)AND128)/2 OR (ASC(C#)AND63)
126 FOR L=1TO1000 REM 1 FOR EACH SPOT
127 TX=T*RND(1)+1 REM PICK A TABLE
128 ON TX GOSUB 300,400,500,600 REM GO PICK AN ELEMENT
130 POKE A,C REM DISPLAY CHAR
140 NEXT L
160 GOTO 95
170 END
199 REM **SUBR TO INITIALIZE TABLES**
200 FOR I=0 TO N-1:T1(I)=I:NEXT
210 FOR I=0 TO N-1:T2(I)=I+250:NEXT
220 FOR I=0 TO N-1:T3(I)=I+500:NEXT
230 FOR I=0 TO N-1:T4(I)=I+750:NEXT
240 RETURN
299 REM **SUBROUTINE FOR TABLE T1**
300 N1=N1-1
305 REM IF EMPTY, GO TO ANOTHER TABLE
310 IF N1<0 THEN ON INT(3*RND(1)+1) GOTO 400,500,600
320 A%=(N1+1)*RND(1) REM PICK AN ELEM
330 A=T1(A%)+32768 REM FORM POKE ADDR
340 TP=T1(A%):T1(A%)=T1(N1):T1(N1)=TP REM SWAP ELEMENTS
350 RETURN
399 REM **SUBROUTINE FOR TABLE T2**
400 N2=N2-1
410 IF N2<0 THEN ON INT(3*RND(1)+1) GOTO 300,500,600
420 A%=(N2+1)*RND(1)
430 A=T2(A%)+32768
440 TP=T2(A%):T2(A%)=T2(N2):T2(N2)=TP
450 RETURN
499 REM **SUBROUTINE FOR TABLE T3**
500 N3=N3-1
510 IF N3<0 THEN ON INT(3*RND(1)+1) GOTO 300,400,600
520 A%=(N3+1)*RND(1)
530 A=T3(A%)+32768
540 TP=T3(A%):T3(A%)=T3(N3):T3(N3)=TP
550 RETURN
599 REM **SUBROUTINE FOR TABLE T4**
600 N4=N4-1
610 IF N4<0 THEN ON INT(3*RND(1)+1) GOTO 300,400,500
620 A%=(N4+1)*RND(1)
630 A=T4(A%)+32768
640 TP=T4(A%):T4(A%)=T4(N4):T4(N4)=TP
650 RETURN

```

Chapter 6: MEMORY MAP

All of the changes in Chapter 6 are based on the fact that the memory map for the old ROMs was reorganized for the new ROMs. The memory map for the old ROMs is shown in Table G-1, at the back of this appendix.

Chapter 6: PET BASIC INTERPRETER

The system locations holding principal pointers in the user program area are different for the old ROMs. Your pointers, in place of Figure 6-2, are as shown in Figure G-1. Figure G-2, replacing Figure 6-4, also reflects these changes.

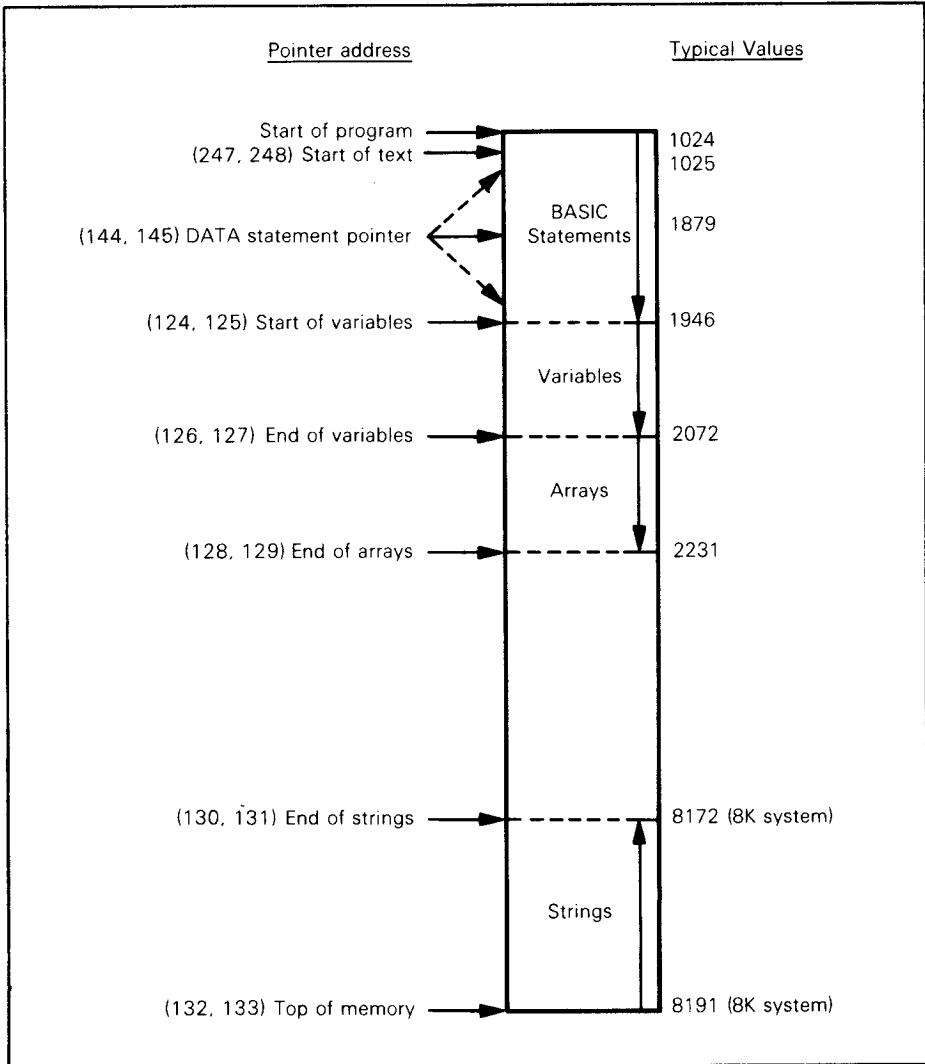


Figure G-1. Principal Pointers in User Program Area

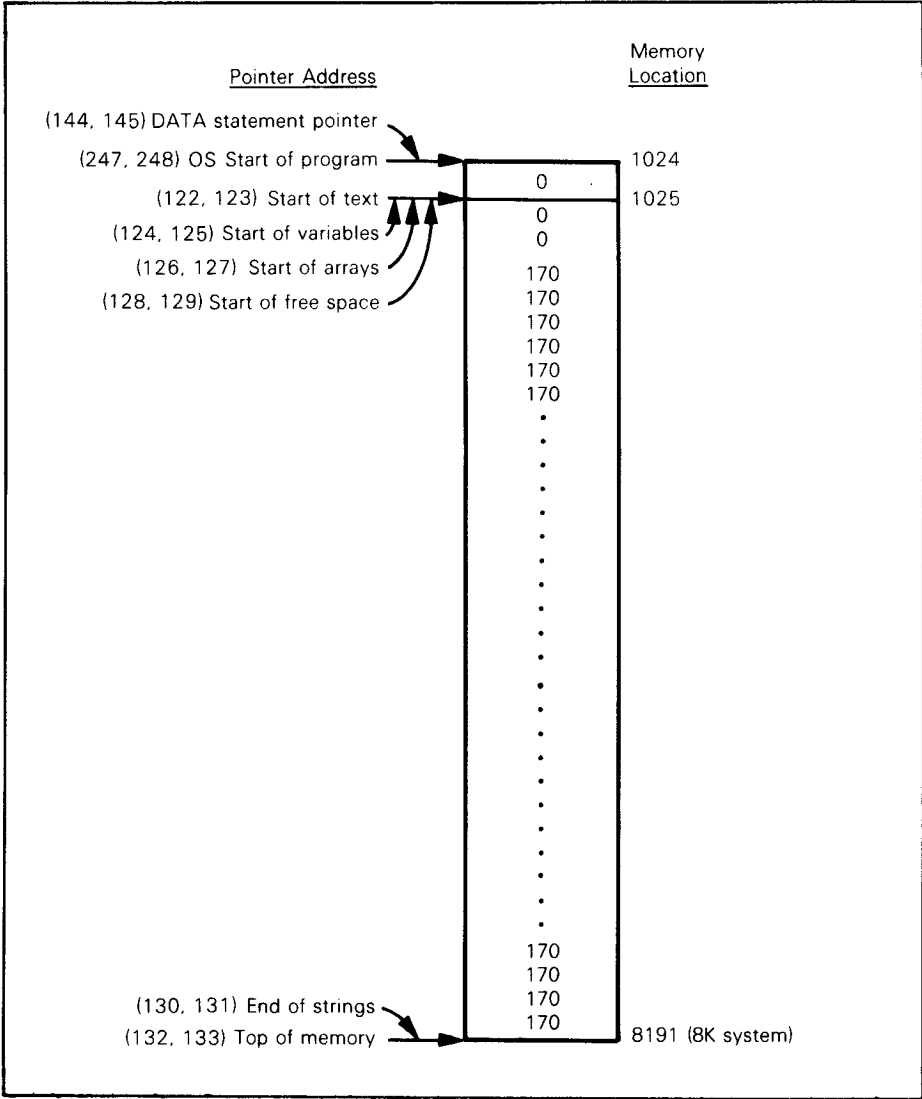


Figure G-2. User Program Area on Power-up

Chapter 6: VARIABLES, Floating Point Variable Format

Use the following program to examine floating point representations:

```
10 INPUT A
20 X=PEEK(125)*256+PEEK(124)+2
30 PRINT A;"=";PEEK(X);PEEK(X+1);PEK(X+2);PEEK(X+3);PEEK(X+4)
40 GOTO 10
```

This is the same one given in Chapter 6 except for the system locations at line 20 being PEEKed.

Chapter 6: CONSTANTS

Instead of pointer (42,43), the pointer in the diagrams is (124,125).

Chapter 6: ARRAY STORAGE FORMAT

Use the following program for viewing sample Array Area entries:

```
10 DIM A(5),B%(2,2),C$(10)      :REM SAMPLE ARRAYS
20 FOR I=0 TO 5:A(I)=I:NEXT
30 FOR I=0 TO 2:FOR J=0 TO 2:B%(J,I)=100+3*I+J:NEXT J,I
40 FOR I=0 TO 10:C$(I)=CHR$(ASC("A")+I):NEXT
50 X=PEEK(127)*256+PEEK(126)    :REM POINT TO ARRAY AREA
60 Y=PEEK(129)*256+PEEK(128)   :REM END OF ARRAYS
70 FOR I=X TO Y
80 PRINT I,PEEK(I)
90 GET D$:IF D$="" THEN GOTO 90:REM HIT KEY FOR NEXT ELEMENT
100 NEXT
```

This is the same as the program in Chapter 6 except for the system locations accessed in lines 50 and 60.

Chapter 6: ASSEMBLY LANGUAGE PROGRAMMING

For the old ROMs, item 2, Top of Core discussion should read as follows:

2. Top of Memory. Memory locations 134 and 135 contain the pointer to the top of memory. On 8K PETs this value is 8192. You can temporarily set the top of memory pointer to a lower address, thereby reserving a number of bytes from the new pointer value to the actual top of memory for storage of an assembly language program. To set the pointer, say, down 1000 bytes, you will need to store the value 7192 (8192-1000) converted into low, high address order, e.g.:

$$7192_{10} = 1C18_{16} \xrightarrow{\text{High}} 1C_{16} = 28_{10} \text{ and } 18_{16} = 24_{10} \xleftarrow{\text{Low}}$$

So 24 is to be stored at location 134 (low byte), and 28 is to be stored at location 135 (high byte). The following instructions can be used:

```

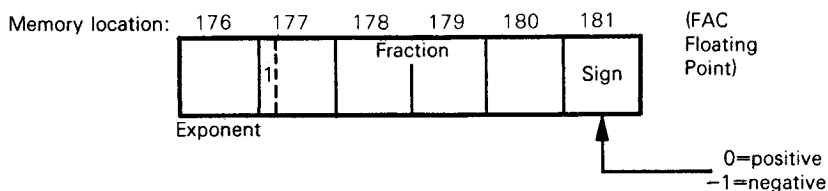
10 AL=PEEK(134):AH=PEEK(135): REM SAVE CURRENT POINTER
20 POKE 132,24:POKE 135,28: REM TOP OF CORE NOW = 7192
.
.
.
100 POKE 134,AL:POKE 135,AH: REM RESTORE POINTER
110 END

```

Chapter 6: USR

Since the accumulator is maintained in different system locations on the old ROMs, the accumulator description will read as follows:

The parameter value is passed to the USR subroutine in system locations that function as a floating point accumulator (FAC) for all functions. The FAC resides in six bytes from memory locations 176 to 181 (B0₁₆–B5₁₆). The FAC has the following format:



Like floating point variables, the exponent is stored in excess 128 format, and the fraction is normalized with the high-order bit of byte 177 (the high-order byte of the fraction) set to 1. The difference between this format and the variable format is that the high-order 1 bit is present in byte 177 of the FAC. An extra byte (181) is used to hold the sign of the fraction. (This is done for ease of manipulation by the functions that use the FAC).

1. PET User Notes, Volume 1, Issue 6, Sept.-Oct. 1978, pg. 14, "Cassette File Usage Summary" by Jim Butterfield.
2. Best of the PET GAZETTE, pg. 38 "On Data Files:", by Michael Richter.

Table G-1. PET Memory Map (Rev 2 ROMs)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
Page 0 (0-255):				
USR Function Locations				
0	0000	76	4C	Constant 6502 JMP instruction
1-2	0001-0002	826	033A	User address jump vector
Terminal I/O Maintenance				
3	0003	0	00	Active input device number (0=keyboard)
4	0004	0	00	No. of nulls to print after CR/LF (0=normal)
5	0005	0	00	Cursor position for POS function (0-255)
6	0006	127	7F	Terminal width (unused)
7	0007	127	7F	Limit for scanning source columns (unused)
8	0008	60	3C	Line number storage preceding buffer
9	0009	3	03	Constant
10-89	000A-0059	48	30	BASIC input line buffer (80 bytes)
90	005A	0	00	General counter for BASIC
91	005B	0	00	Delimiter flag for quote mode scan
92	005C	255	FF	Input buffer pointer, general counter
Evaluation of Variables				
93	005D	0	00	Flag for dimensioned variables
94	005E	0	00	Flag for variable type: 00=numeric FF=string
95	005F	0	00	Flag for numeric variable type: 00=floating point 80=integer
96	0060	0	00	Flag to allow reserved words in strings and remarks
97	0061	0	00	Flag to allow subscripted variable
98	0062	0	00	Flag for input type: 0=INPUT 64=GET 152=READ
99	0063	0	00	Flag sign of TAN function
100	0064	0	00	Flag to suppress output: + normal - suppressed
101	0065	104	68	Index to next available descriptor
102-103	0066-0067	101	0065	Pointer to last string temporary
104-111	0068-006F	2	0002	Table of double-byte descriptors that point to variables (8 bytes)
112-113	0070-0071	14525	38BD	Indirect index #1
114-115	0072-0073	62983	F607	Indirect index #2
116	0074	1	01	Pseudo-register for function operands (6 bytes)
117	0075	234	EA	
118	0076	0	00	
119	0077	0	00	
120	0078	0	00	
121	0079	0	00	

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
Data BASIC Storage Maintenance				
122-123	007A-007B	1025	0401	Pointer to start of text
124-125	007C-007D	1946	079A	Pointer to start of variables
126-127	007E-007F	2072	0818	Pointer to end of variables
128-129	0080-0081	2231	08B7	Pointer to end of arrays
130-131	0082-0083	8192	2000	Pointer to start of strings (moving down)
132-133	0084-0085	8191	1FFF	Pointer to end of strings (top of available RAM)
134-135	0086-0087	8192	2000	Pointer to limit of BASIC memory
136-137	0088-0089	2000	07D0	Line number of current line being executed -1 in 137=direct mode statement
138-139	008A-008B	110	006E	Line number for last line executed before CONT
140-141	008C-008D	1922	0782	Pointer to next line to be executed after CONT
142-143	008E-008F	1150	047E	Line number of current DATA line
144-145	0090-0091	1879	0757	Pointer to current DATA line
146-147	0092-0093	13	000D	Next DATA item within line
148-149	0094-0095	89	0059	Current variable name
150-151	0096-0097	2032	07F0	Pointer to current variable
152-153	0098-0099	2032	07F0	Pointer to next FOR...NEXT variable
154-155	009A-009B	31999	7CFF	Pointer to current operator in ROM table
156	009C	0	00	Mask for current logical operator
157-158	009D-009E	898	0382	Pointer to user function FN definition
159-160	009F-00A0	104	0068	Pointer to a string description
161	00A1	221	DD	Length of string
162	00A2	3	03	Constant used by garbage collection routine
163	00A3	76	4C	Constant 6502 JMP instruction
164-165	00A4-00A5	0	0000	Jump vector for user function FN
166-171	00A6-00AB	129	81	Floating point accumulator #3 (6 bytes)
172-173	00AC-00AD	0	00	Block transfer pointer #1
174-175	00AE-00AF	0	00	Block transfer pointer #2
176-181	00B0-00B5			Floating point accumulator (FAC) #1 (6 bytes)
		0	00	176 00B0 Exponent +128
		0	00	177 00B1 Fraction MSB Floating Point
		0	00	178 00B2 Fraction
		0	00	179 00B3 Fraction MSB Integer
		0	00	180 00B4 Fraction LSB
		0	00	181 00B5 Sign of fraction (0 if zero or positive, -1 if negative)
182	00B6	0	00	Copy of FAC #1 sign of fraction
183	00B7	0	00	Counter for number of bits to shift to normalize FAC #1
184-189	00B8-00BD	0	00	Floating point accumulator #2 (6 bytes)
190	00BE	0	00	Overflow byte for floating argument
191	00BF	0	00	Copy of FAC #2 sign of fraction
192-193	00C0-00C1	258	0102	Conversion pointer

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
RAM Subroutines				
194-199	00C2-00C7	230	E6	Routine to fetch next BASIC character
200	00C8	173	AD	Entry to refetch current character
201-202	00C9-00CA	1929	0789	Pointer to source text
203-223	00CB-00DF	201	C9	Work area for RND function
OS Page Zero Storage				
224-225	00E0-00E1	33728	83C0	Pointer to start of line where cursor is flashing
226	00E2	0	00	Column position where cursor is flashing (0-79)
227-228	00E3-00E4	33792	8400	Utility pointer
229-230	00E5-00E6	1929	0789	End of current program
231-233	00E7-00E9	254	FE	Utility
234	00EA	0	00	Flag for quote mode. 0=not quote mode
235-237	00EB-00ED	192	C0	Utility
238	00EE	0	00	No. of characters in current file name
239	00EF	5	05	Current logical file number
240	00F0	255	FF	GPIB primary address
241	00F1	63	3F	GPIB device number
242	00F2	39	27	Max. no. of characters on current line (39,79)
243-244	00F3-00F4	634	027A	Pointer to start of current tape buffer (634 or 826)
245	00F5	23	17	Line number where cursor is flashing (0-24)
246	00F6	10	0A	I/O storage
247-248	00F7-00F8	1024	0400	OS pointer to program
249-250	00F9-00FA	3100	0C1C	Pointer to current file name
251	00FB	0	00	Number of Insert keys pushed to go
252	00FC	9	09	Serial bit shift word
253	00FD	0	00	Number of blocks remaining to read/write
254	00FE	0	09	Serial word buffer
255	00FF	243	F3	Overflow byte for binary to ASCII conversions
Page 1 (256-511)				
256-up	0100-up	32	20	Tape read working storage (up to 511) and conversion stg. 256-318 For error correction in tape reads (62 bytes) 256-266 Binary to ASCII conversion (11 bytes)
511-down	01FF-down	0	00	Stack (down to 256)
Page 2-3 (512-1023)				
OS Working Storage				
512-514	0200-0202	3801352	3A0108	24-hour clock incremented every 1/60 second (jiffy). Resets every 5,184,000 jiffies (24 hours). Stored in low to high order.

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
515	0203	255	FF	Matrix coordinate of key depressed at current jiffy. 1-80=key 255=no key
516	0204	0	00	Status of SHIFT key: 0=unshifted (up) 1=shifted (down)
517-518	0205-0206	37916	941C	Secondary jiffy clock
519	0207	52	34	Interrupt driver flag for cassette #1 ON switch
520	0208	0	00	Interrupt driver flag for cassette #2 ON switch
521	0209	255	FF	Keyswitch PIA
522	020A	0	00	Utility
523	020B	0	00	I/O flag: 0=LOAD 1=VERIFY
524	020C	0	00	I/O status byte
525	020D	0	00	Number of characters in keyboard buffer (0 to 9)
526	020E	0	00	Flag to indicate reverse field on (0=normal)
527-536	020F-0218	85	55	Keyboard buffer (10 bytes)
537-538	0219-021A	34048	8500	Hardware interrupt vector
539-540	021B-021C	0	0000	6502 BRK instruction interrupt vector
541-546	021D-0222			Input routine storage (6 bytes)
		13	0D	542 021E No. of characters on screen line
547	0223	255	FF	Key image
548	0224	1	01	Flag for cursor enable: 0=Enable 1=Disable
549	0225	11	0B	Counter to flip cursor (20 to 1)
550	0226	32	20	Copy of character at current cursor position
551	0227	0	00	Flag for cursor on/off: 0=cursor moved 1=blink started
552	0228	0	00	Flag for tape write
553-577	0229-0241			High byte of screen line addresses 553-559=128 (lines 1-7) 560-565=129 (lines 8-13) 566-572=130 (lines 14-20) 573-577=131 (lines 21-25)
578-587	0242-024B	5	05	Table of logical numbers of open files
588-597	024C-0255	5	05	Table of device numbers of open files
598-607	0256-025F	255	FF	Table of secondary address modes of open files
608	0260	0	00	Flag for input source: 0=keyboard buffer 1=screen memory
609	0261	0	00	I/O utility
610	0262	1	01	Number of open files (index into tables)

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
611	0263	0	00	Default input device number (0=keyboard)
612	0264	3	03	Default output device number (3=screen)
613	0265	0	00	Tape parity byte
614	0266	0	00	I/O utility
615	0267	0	00	I/O utility
616	0268	0	00	Byte pointer in filename transfer
617	0269	0	00	I/O utility
618	026A	255	FF	I/O utility
619	026B	0	00	I/O utility
620	026C	8	08	Serial bit count
621	026D	0	00	Count of redundant tape blocks
622	026E	0	00	Tape utility
623	026F	0	00	Cycle counter flip for each bit read from tape
624	0270	0	00	Countdown synchronization on tape write
625	0271	0	00	Tape buffer 1 index to next character
626	0272	0	00	Tape buffer 2 index to next character
627	0273	0	00	Countdown synchronization on tape read
628	0274	0	00	Flag to indicate bit/byte tape error
629	0275	0	00	Flag to indicate tape error 0=first half-byte marker not written
630	0276	0	00	Flag to indicate tape error 0=2nd half-byte marker not written /Tape dropout counter
631	0277	0	00	Tape dropout counter
632	0278	128	80	Flag for tape read current function
633	0279	9	09	Checksum utility
634-825	027A-0339	1	01	Tape buffer for cassette #1 (192 bytes)
826-1017	033A-03F9	173	AD	Tape buffer for cassette #2 (192 bytes)
1018-1023	03FA-03FF	28	1C	Utility space/unused.
Page 4-32 (1024-8191)				
1024-8191	0400-1FFF	0	00	User program area
Page 33-128 (8192-32767)				
8192-32767	2000-7FFF	0	00	Expansion RAM
Page 129-144 (32768-36863)				
32768-36863	8000-8FFF	12	0C	TV RAM 32768-33767 Display memory (1000 bytes)
Page 145-192 (36864-49151)				
36864-49151	9000-BFFF	0	00	Expansion ROM
Page 193-232 BASIC (49152-59391)				
Pointers to BASIC Routines				
49152-49153	C000-C001	50973	C71D	Pointer -1 to END*
49154-49155	C002-C003	50760	C648	Pointer -1 to FOR
49156-49157	C004-C005	52277	CC35	Pointer -1 to NEXT

* These memory locations contain the address of the byte preceding the specified BASIC routines.

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
49158-49159	C006-C007	51183	C73F	Pointer -1 to DATA
49160-49161	C008-C009	51909	CAC5	Pointer -1 to INPUT#
49162-49163	C00A-C00B	51935	CADF	Pointer -1 to INPUT
49164-49165	C00C-C00D	53104	CF70	Pointer -1 to DIM
49166-49167	C00E-C00F	52003	CB23	Pointer -1 to READ
49168-49169	C010-C011	51356	C89C	Pointer -1 to LET
49170-49171	C012-C013	51100	C79C	Pointer -1 to GOTO
49172-49173	C014-C015	51060	C774	Pointer -1 to RUN
49174-49175	C016-C017	51231	C81F	Pointer -1 to IF
49176-49177	C018-C019	50956	C70C	Pointer -1 to RESTORE
49178-49179	C01A-C01B	51071	C77F	Pointer -1 to GOSUB
49180-49181	C01C-C01D	51145	C7C9	Pointer -1 to RETURN
49182-49183	C01E-C01F	51250	C832	Pointer -1 to REM
49184-49185	C020-C021	50971	C71B	Pointer -1 to STOP
49186-49187	C022-C023	51266	C842	Pointer -1 to ON
49188-49189	C024-C025	55041	D701	Pointer -1 to WAIT
49190-49191	C026-C027	65492	FFD4	Pointer -1 to LOAD
49192-49193	C028-C029	65495	FFD7	Pointer -1 to SAVE
49194-49195	C02A-C02B	65498	FFDA	Pointer -1 to VERIFY
49196-49197	C02C-C02D	53908	D294	Pointer -1 to DEF
49198-49199	C02E-C02F	55032	D6F8	Pointer -1 to POKE
49200-49201	C030-C031	51582	C97E	Pointer -1 to PRINT#
49202-49203	C032-C033	51614	C99E	Pointer -1 to PRINT
49204-49205	C034-C035	51012	C744	Pointer -1 to CONT
49206-49207	C036-C037	50599	C5A7	Pointer -1 to LIST
49208-49209	C038-C039	51055	C76F	Pointer -1 to CLR
49210-49211	C03A-C03B	51588	C984	Pointer -1 to CMD
49212-49213	C03C-C03D	65501	FFDD	Pointer -1 to SYS
49214-49215	C03E-C03F	65471	FFBF	Pointer -1 to OPEN
49216-49217	C040-C041	65474	FFC2	Pointer -1 to CLOSE
49218-49219	C042-C043	51870	CA9E	Pointer -1 to GET
49220-49221	C044-C045	50512	C550	Pointer -1 to NEW
49222-49223	C046-C047	56075	DB0B	Pointer to SGN**
49224-49225	C048-C049	56222	DB9E	Pointer to INT
49226-49227	C04A-C04B	56106	DB2A	Pointer to ABS
49228-49229	C04C-C04D	0	0000	Pointer to USR pointer
49230-49231	C04E-C04F	53860	D264	Pointer to FRE
49232-49233	C050-C051	53893	D285	Pointer to POS
49234-49235	C052-C053	56868	DE24	Pointer to SQR
49236-49237	C054-C055	57157	DF45	Pointer to RND
49238-49239	C056-C057	55487	D8BF	Pointer to LOG
49240-49241	C058-C059	56992	DEA0	Pointer to EXP
49242-49243	C05A-C05B	57246	DF9E	Pointer to COS
49244-49245	C05C-C05D	57253	DFA5	Pointer to SIN
49246-49247	C05E-C05F	57326	DFEE	Pointer to TAN
49248-49249	C060-C061	57416	E048	Pointer to ATN
49250-49251	C062-C063	55014	D6E6	Pointer to PEEK
49252-49253	C064-C065	54868	D654	Pointer to LEN
49254-49255	C066-C067	54089	D349	Pointer to STR\$
49256-49257	C068-C069	54917	D685	Pointer to VAL
49258-49259	C06A-C06B	54883	D663	Pointer to ASC
49260-49261	C06C-C06D	54724	D5C4	Pointer to CHR\$
49262-49263	C06E-C06F	54744	D5D8	Pointer to LEFT\$

** These memory locations contain the address of the first byte of the specified BASIC routines.

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
49264-49265	C070-C071	54788	D604	Pointer to RIGHT\$
49266-49267	C072-C073	54799	D60F	Pointer to MID\$
49268-57343	C074-DFFF			BASIC Routines
				Starting Address Function
				49836 C2AC FOR...NEXT stack check
				49882 C2DA Insert line space marker
				49949 C31D Stack overflow check
				50007 C357 Error message abort
				50057 C389 READY
				50068 C394 Execute line
				50092 C3AC Handle new line
				50224 C430 Rechain lines after insert/delete
				50274 C462 Input line
				50297 C479 Get character from input line
				50317 C48D Keyword encoder
				50466 C522 Line number search
				50513 C551 NEW
				50586 C59A Set pointer to start of program
				50600 C5A8 LIST
				50761 C649 FOR...NEXT
				50869 C6B5 Statement processor
				50930 C6F2 Statement execute
				50957 C70D RESTORE
				50972 C71C STOP
				50974 C71E END
				51013 C745 CONT
				51056 C770 CLR
				51061 C775 RUN
				51072 C780 GOSUB
				51101 C79D GOTO
				51146 C7CA RETURN
				51184 C7F0 DATA
				51198 C7FE Next line scan
				51232 C820 IF
				51251 C833 REM
				51267 C843 ON...GOTO/GOSUB
				51299 C863 Number fetch
				51357 C89D LET=
				51484 C91C Digit check
				51583 C97F PRINT#
				51589 C985 CMD
				51615 C99F PRINT
				51751 CA27 Print string
				51780 CA44 Print character
				51831 CA77 Input data error
				51871 CA9F GET

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
				51910 CAC6 INPUT #
				51936 CAED INPUT
				51991 CB17 Input prompt
				52004 CB24 READ
				52242 CC12 Error messages
				52278 CC36 NEXT
				52370 CC92 Format checker
				52408 CCB8 Expression evaluator
				52538 CD3A Stack argument
				52637 CD9D Symbol evaluator
				52668 CDBC Pi
				53105 CF71 DIM
				53207 CFD7 Variable table look-up
				53415 D0A7 Floating-to-integer
				53860 D264 FRE
				53880 D278 Integer-to-floating
				53893 D285 POS
				53909 D295 DEF
				54089 D349 STR\$
				54724 D5C4 CHR\$
				54744 D5D8 LEFT\$
				54788 D604 RIGHT\$
				54799 D60F MID\$
				54868 D654 LEN
				54883 D663 ASC
				54917 D685 VAL
				55014 D6E6 PEEK
				55033 D6F9 POKE
				55042 D702 WAIT
				55080 D728 Subtraction
				55103 D73F Addition
				55487 D8BF LOG
				55552 D900 Multiplication
				55646 D95E Load number to AFAC
				55650 D962 Load variable to AFAC
				55780 D9E4 Division
				55924 DA74 Load Accumulator (FAC)
				55928 DA78 Load variable to FAC
				55979 DAAB Store variable from FAC
				56075 DB0B SGN
				56106 DB2A ABS
				56222 DB9E INT
				56868 DE24 SQR
				56878 DE2E Raise AFAC to power FAC
				56992 DEA0 EXP
				57157 DF45 RND
				57246 DF9E COS
				57253 DFA5 SIN
				57326 DFEF TAN

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
57344-59391	E000-E7FF			<p align="center">Screen Editor</p> <p align="center">Starting Address Function</p> <p>57416 E048 ATN</p> <p>57525 E0B5 Initialize BASIC system</p> <p>57910 E236 Clear screen</p> <p>57981 E27D Character fetch</p> <p>Video driver</p> <p>58282 E3AA Scroll processor</p> <p>58346 E3EA Video display routine</p> <p>58185 E349 Quote mode (\$EA) switcher</p> <p>58346 E3EA Print character</p> <p>58713 E559 Scroll 1 line</p> <p>58758 E586 Interrupt Request (IRQ)</p> <p>Interrupt handler</p> <p>Clock update</p> <p>Keyboard scan</p> <p>Keyboard encoding table</p>
58004-58986	E294-E66A			
58987-59012	E66B-E684			
59013-59198	E685-E73E			
59199-59227	E73F-E75B			
59228-59348	E75C-E7D4			
Page 233-240 I/O Ports and Expansion I/O (PIA's and VIA) (59392-61439)				
Keyboard PIA (59408-59411)				
59408	E810	233	E9	I/O Port A and Data Direction register
59409	E811	60	3C	Control Register A — screen blanking 52=Screen off (blanked) 60=Screen on
59410	E812	255	FF	I/O Port B and Data Direction register 255=all keys except: 254=RVS key 253=key 251=SPACE key 247= < key
59411	E813	61	3D	Control Register B — #1 cassette motor 53=motor on 61=motor off
IEEE Port PIA (59424-59427)				
59424	E820	255	FF	I/O Port A and Data Direction register PEEK (59424) reads input data.
59425	E821	188	BC	Control Register A — set output line CA2 POKE 59425,52=low POKE 59425,60=high
59426	E822	255	FF	I/O Port B and Data Direction register POKE 59426,data writes output data POKE 59426,255 before a read to Port A
59427	E823	60	3C	Control Register B — set output line CB2 POKE 59427,52=low POKE 59427,60=high

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
59456	E840	254	FE	Parallel User Port VIA (59456-59471) I/O Port B 207=#2 cassette motor on 223=#2 cassette motor off WAIT 59456,23,23 waits for vertical retrace of display Bit 1=PB1 (NFRD on IEEE connector) output line Bit 3=PB3 (ATN on IEEE connector) output line I/O Port A with handshaking Data Direction register for I/O Port B Data Direction register for I/O Port A For each bit 1=output, 0=input =0 all input =255 all output (Low, high order) Read Timer 1 Counter; write to Timer 1 Latch and (high byte) initiate count (Low, high order) Read Timer 1 Latch Read Timer 2 Counter low byte and reset interrupt; write to Timer 2 low byte PEEK (59464) Clock decrements every microsecond POKE 59464,n sets SR rate of shift from high (n=0) to low (n=255) for music from User Port. Read Timer 2 Counter high byte; write to Timer 2 high byte and reset interrupt. PEEK (59465) Clock decrements every millisecond Serial I/O Shift register (SR) POKE 59466,15 or 51 or 85 to generate square wave output at CB2 for playing music from User Port. Auxiliary Control register. =16 Sets SR to free-running mode for music from User Port. =0 for proper operation of tape drive Peripheral Control register =12 for graphics on shifted characters =14 for lower-case letters on shifted characters Interrupt Flag register Interrupt Enable register I/O Port A without handshaking
59457	E841	255	FF	
59458	E842	30	1E	
59459	E843	0	00	
59460-59461	E844-E845	25248	62A0	
59462-59463	E846-E847	65381	FF65	
59464	E848	113	71	
59465	E849	200	C8	
59466	E84A	1	01	
59467	E84B	0	00	
59468	E84C	14	0E	
59469	E84D	0	00	
59470	E84E	128	80	
59471	E84F	255	FF	
Page 241-256 Operating System (61440-65535)				
61622-61904	F0B6-F1D0			File Control
				Starting Address Function
61905-63532	F1D1-F82C			61905 F1D1 Get a character (without cursor)
				61921 F1E1 Input a character (with cursor)

Table G-1. PET Memory Map (Rev 2 ROMs) (Continued)

Memory Address		Sample Value		Description
Decimal	Hexadecimal	Decimal	Hexadecimal	
				62002 F232 Display a character
				62026 F24A Close all files
				62121 F2A9 CLOSE
				62250 F32A STOP search
				62278 F346 Tape playback
				62402 F3C2 LOAD
				62481 F411 Display filename
				62515 F433 Fetch file number
				62556 F45C Number fetch
				62647 F4B7 VERIFY
				62724 F504 Fetch filename
				62741 F515 Fetch tape character
				62753 F521 OPEN
				62824 F568 Record SAVE routine
				62894 F5AE Tape header search
				62947 F5E3 Clear current tape buffer
				62957 F5ED Write tape end block
				63101 F67D Set up tape end pointer
				63108 F684 SYS
				63134 F69E SAVE
				63153 F6B1 SAVE memory block on cassette
				63273 F729 Update secondary jiffy clock
				Tape Control
				63582 F85E Check for cassette on
				63615 F87F Tape read to buffer
				63684 F8C4 Write block to tape
				63765 F915 Interrupt wait
				Power-On Diagnostics
63533-64789	F82D-FD15			64824 FD38 System reset SYS (64824) simu- lates power-on reset
				64909 FD8D Reset BASIC (does not affect User Pro- gram)
				64912 FD90 EOT-buffer compare
				Jump Vectors
65472-65516	FFC0-FFEC			
65472-65474	FFC0-FFC2	76 62753	4C F521	JMP OPEN
65475-65477	FFC3-FFC5	76 62121	4C F2A9	JMP CLOSE
65487-65489	FFCF-FFD1	76 61921	4C F1E1	JMP RDT
65490-65492	FFD2-FFD4	76 62002	4C F232	JMP WRT
65493-65495	FFD5-FFD7	76 62402	4C F3C2	JMP LOAD
65496-65498	FFD8-FFDA	76 63134	4C F69E	JMP SAVE
65499-65501	FFDB-FFDD	76 62647	4C F4B7	JMP VERIFY
65502-65504	FFDE-FFED	76 63108	4C F684	JMP SYS
65508-65510	FFE4-FFE6	76 61905	4C F1D1	JMP GETC
65514-65516	FFEA-FFEC	76 63273	4C F729	JMP Clock Update
65530-65535	FFFA-FFFF			6502 Interrupt Vectors
65530-65531	FFFA-FFFB	51808	CA60	Non-maskable interrupt (NMI)
65532-65533	FFFC-FFFD	64824	FD38	System reset (RESET)
65534-65535	FFFE-FFFF	58987	E66B	Interrupt request, break (IRQ+BRK)

PET Features

Features of PET BASIC

Calculator/program modes
8 commands
26 statements
29 functions
Extended BASIC features
Strings
ASCII character representation
Number representation: Floating point (5 bytes)
Integer (2 bytes)
To 9 or 10 significant digits
Variables up to 255 characters, 2 characters significant
Mix of 40 and 80 character source lines
Expression evaluation for statement and function parameters
Fully programmable graphic keys
Fully programmable cursor control keys
Arrays of any dimensions
Line editor
 Add line
 Delete line
 Change line
Programmed breaks
Nested subroutines up to 26
Tape read/write under program control
Program overlay capability
PEEK/POKE to load/store memory bytes
Programmable system clock for time of day
Machine language interface
Fully programmable external I/O capability

Operational Features	
Chassis	Portable, self-contained Steel cabinet 44 pounds 16½" wide, 18½" deep, 14" high I/O connectors for: 8-bit parallel I/O port IEEE 488 interface Tape cassette unit
CPU	MCS 6502 microprocessor
Memory	4K, 8K, 16K, 32K bytes RAM user memory (4K no longer available) Programmable 1K screen memory 14K bytes ROM memory Resident BASIC interpreter and operating system 2K Input/Output Expansion ROM 12K additional, total ROM 26K Expansion RAM (up to 28K) additional, total RAM 32K
Keyboard	Compact keyboard has 73 keys, 2/3 conventional size Full-size keyboard has 74 keys including Shift Lock 140 characters/functions 2 shift keys 26 alphabetic keys Numeric pad 28 special symbols ASCII characters and symbols all unshifted 62 graphic keys Graphic keys all shifted Reverse-field video characters Alternate programmable character set 6 cursor control functions Key editing Replace characters Insert characters Delete characters Program load and go/stop key
Video Display Screen	9" (22.86 cm) high-resolution screen 1000 character display, 25 lines by 40 characters 8x8 dot matrix Full cursor control
Tape Cassette Unit	Built into PET console or externally connected Uses standard audio magnetic tape cassettes Four tape control keys: Play, Fast Forward, Rewind, Record Programmable Tape Read/Write 1000 baud optimal

Index

- ABS function, 150
- Add a line, 100
- Addend, 190
- Addition, 63, 190
- Alphabet, BASIC, 53
- Alternate character set, 12
- AND, 66-68
- Animation, 232
- Array Header, 322
- Arrays
 - dimensions, 69
 - size, 70
- Array storage format, 322
- ASC function, 156
- ASCII, 188, 326
- Assembly Language Programming, 329
- Assignment statement, 52
- ATN function, 150
- Augend, 190
- BASIC, 3
- BASIC commands
 - CLR, 83, 112
 - CONT, 83, 112
 - LIST, 83, 98, 101, 113, 114
 - LOAD, 74, 83, 100, 114
 - NEW, 83, 117
 - RUN, 83, 117
 - SAVE, 83, 98, 119
 - VERIFY, 84, 120
- Byte, 5
- Calculator mode, 47, 48, 84
- Cassette tape controls, 35
 - Fast Forward, 35
 - Play, 36
 - Record, 35
 - Rewind, 35
 - Stop, 36
- Cassette tape drive, 32, 33
- Cassette tape unit, hookup, 8
- Cassette tapes
 - care of, 39
 - pre-programmed, 3, 74
- CHR\$ function, 105, 156, 188
- CLEAR SCREEN/HOME key, 25, 26, 96
- CLOSE statement, 82, 123
- CLR command, 83, 112
- CMD statement, 82, 124
- Concatenating strings, 90, 180
- Console input/output
 - GET, 81
 - INPUT, 81
 - PRINT, 80
- Constants, 320
- CONT command, 112
- COS function, 150

Cursor blink enable, 106
 CURSOR LEFT/RIGHT key, 27, 28
 CURSOR UP/DOWN key, 26, 27, 51

Data block, 251
 Data conversions, 143
 Data file, 74
 DATA statement, 80, 125
 Decision making
 IF . . . THEN, 80
 ON . . . GOSUB, 80
 ON . . . GOTO, 80
 DEF FN, 164
 Defining data
 DATA, 80, 125
 DIM, 80, 126
 LET . . . =, 80
 READ, 80
 RESTORE, 80
 Delete a line, 100
 DELETE key, 107
 Degmagnetizer, 38, 39
 Dimension statement, 71, 80, 126
 Direct mode, 47
 Division, 64

Editing, screen, 49
 Elements, BASIC, 53
 END statement, 127
 End-of-file mark, 123, 242
 End-of-tape mark, 123, 241
 EXP function, 151
 Exponentiation, 64

Field, 232
 Field separator, 96
 File input/output
 CLOSE, 82
 CMD, 82
 GET #, 82
 INPUT #, 82
 OPEN, 82
 PRINT #, 82
 File names, 141
 length, 74
 matching, 74
 Files
 data, 232
 program, 232
 Floating Point Array Element Format, 323
 Floating point numbers, 53
 FOR . . . NEXT statement, 79, 127
 FRE function, 161
 Functions, arithmetic group
 ABS, 71
 ATN, 72
 COS, 72
 EXP, 71
 INT, 71
 LOG, 72
 RND, 72
 SIN, 72
 SGN, 71
 SOR, 71
 TAN, 72
 Functions, BASIC, 71
 arguments, 71
 Functions, format group

 POS, 73
 SPC, 73
 TAB, 73
 Functions, string group
 ASC, 72
 CHR\$, 72
 LEFT\$, 72
 LEN, 72
 MID\$, 72
 RIGHT\$, 72
 STR\$, 72
 VAL, 72
 Functions, system group
 FRE, 73
 PEEK, 73
 SYS, 73
 TI\$, TI, 73
 USR, 73
 Functions, user-defined, 73

GET statement, 81, 103, 129
 GET # statement, 82, 130
 GOSUB/RETURN statement, 79
 GOSUB statement, 131
 GOTO statement, 79, 131
 Graphic character set, 255
 Graphic keys, 13
 Graphic mode, 255
 Graphics, 255

HOME key, 51

IF . . . THEN statement, 80, 132
 Immediate mode, 47, 78, 84
 INPUT statement, 81, 133, 202
 INPUT # statement, 82, 134
 INSERT/DELETE key, 29, 30, 107
 INT function, 151
 Integer Array Element Format, 323
 Integer Variable Format, 319
 Integers, 57
 Interpreter, PET BASIC, 1, 77, 107, 310
 Item separators, 182, 247

K, 5
 Key Groups
 Alphabetic, 15
 Cursor control, 25-30
 Function, 21-25
 Graphic, 13, 19, 20
 Numeric, 15
 Serial symbols, 16-18
 Keyboard
 Compact, 11, 12
 Full size, 11, 13
 Keyboard rollover, 167-173
 Keywords, 312

LEFT\$ functions, 156, 184
 LEN function, 157, 200
 LET . . . = statement, 80, 135
 Line editor, 310
 Line numbers, 76, 84, 85
 LIST command, 83, 98, 101, 113, 114
 LOAD command, 73, 74, 83, 100, 114
 LOG function, 152
 Logical file number, 138, 139
 Memory Map, 308

MID\$ functions. 157, 184, 200
 Minuend, 202
 Multiplication, 64, 215

 NEW command, 117, 315
 NOT, 66-68
 Numbers
 floating point, 53
 integer, 57
 magnitude, 54, 56
 roundoff, 54
 scientific notation, 54, 143

 ON. . . GOSUB statement, 80, 136
 ON. . . GOTO statement, 80, 136
 OPEN statement, 82, 137
 Operators
 arithmetic, 62
 Boolean, 66
 hierarchy, 65
 relational, 65
 OR, 66-68

 PEEK function, 162
 PET Models, 3, 4, 5
 CBM, 7
 Model 2001-8, 7
 Model 2001-16, 7
 Model 2001-32, 7
 Physical device number, 140
 PLAY key, 116
 Pointer, 308
 POKE statement, 82, 143
 POS function, 159
 PRINT formats, 86
 continuous, 86, 98, 144
 line, 86, 144
 tabbed, 93, 144
 PRINT statement, 58, 80, 86, 143, 185
 PRINT, syntax, 47
 PRINT# statement, 82, 145
 Printer/screen concatenation, 182
 Program mode, 76, 84
 Program storage, 74
 Program termination
 END, 83
 STOP, 83
 WAIT, 83
 Programmed cursor control, 96, 186

 RAM, 2, 5
 expansion, 306
 video, 307
 RAM devices, 42
 removal from PET, 44
 Random numbers, 153
 READ statement, 80, 146
 Read-Only Memory (ROM), 2
 Read/Write Memory (RAM), 2, 74
 READY message, 105
 REM statement, 82, 147
 Replace a line, 100
 Reserved words, 62
 RESTORE statement, 80, 148
 RETURN key, 21, 22
 RETURN statement, 148
 REVERSE ON/OFF key, 22
 RIGHT\$ function, 158, 184, 194
 RND function, 153

 Rollover, 167-173
 ROM, 2, 5
 expansion, 307
 ROM devices, 43
 new, 5
 old, 5
 Roundoff, 54
 RUN command, 83, 98, 101, 117
 RUN/STOP key, 23-25

 SAVE command, 83, 98, 119
 Scientific notation, 54, 143
 Screen display, 31
 Screen memory, 327
 Secondary address code, 140
 SGN function, 154
 SHIFT key, 21
 SHIFT LOCK key, 21
 SIN function, 154
 6502 microprocessor, 303
 SQR function, 155
 ST function, 162
 Standard character set, 12
 Statement separators, 84, 98
 Statements, BASIC, 78, 123
 STATUS WORD command (ST), 242
 STOP key, 103
 STOP statement, 148
 STR\$ function, 158
 String Array Element Format, 323
 Strings, 57, 144, 180
 alphabetic, 182
 concatenating, 180
 graphic, 182
 numeric, 182, 183
 String Variable Format, 320
 Subtraction, 63, 202
 Subtrahend, 202
 SYS function, 163, 331

 TAB function, 160
 TAN function, 155
 Tape head demagnetizer, 38, 39
 Terminal Interface Monitor, 331
 TI function, 163
 TI\$ function, 163
 Time Delay, 232
 Typeover, 107

 Unconditional branches
 FOR. . . NEXT, 79
 GOTO, 79
 GOSUB/RETURN, 79
 User-defined function, 73, 164
 User Program Area, 315
 USR function, 163, 331, 332

 VAL function, 158, 193, 199
 Variables, 14, 58, 317
 display, 49
 floating point, 60
 integer, 60
 name, 14, 59, 61
 string, 61
 value, 14, 59
 VERIFY command, 84, 120

 WAIT statement, 149
 Write protect, 40

OSBORNE/McGraw-Hill GENERAL BOOKS

An Introduction to Microcomputers series

by Adam Osborne

Volume 0 — The Beginner's Book

Volume 1 — Basic Concepts

Volume 2 — Some Real Microprocessors (1978 ed.)

Volume 3 — Some Real Support Devices (1978 ed.)

Volume 2 1978-1979 Update Series

Volume 3 1978-1979 Update Series

The 8089 I/O Processor Handbook

by Adam Osborne

The 8086 Book

by G. Alexy and R. Rector

8080 Programming for Logic Design

by Adam Osborne

6800 Programming for Logic Design

by Adam Osborne

Z80 Programming for Logic Design

by Adam Osborne

8080A/8085 Assembly Language Programming

by L. Leventhal

6800 Assembly Language Programming

by L. Leventhal

Z80 Assembly Language Programming

by L. Leventhal

6502 Assembly Language Programming

by L. Leventhal

Z8000 Assembly Language Programming

by L. Leventhal et al.

Running Wild: The Next Industrial Revolution

by Adam Osborne

PET and the IEEE 488 Bus (GPIB)

by E. Fisher and C. W. Jensen

OSBORNE/McGraw-Hill SOFTWARE

Practical BASIC Programs

by L. Poole et al.

Some Common BASIC Programs

by L. Poole and M. Borchers

Some Common BASIC Programs PET Cassette

Some Common BASIC Programs PET Disk

Some Common BASIC Programs TRS-80 Cassette

Payroll with Cost Accounting - CBASIC

by Lon Poole et al.

Accounts Payable and Accounts Receivable - CBASIC

by Lon Poole et al.

General Ledger - CBASIC

by Lon Poole et al.

