

3322

PROGRAMMING WITH
IBM® PC BASIC®
AND THE
PICK® DATABASE
SYSTEM

DAVID L. CLARK

THE
PICK LIBRARY
EDITED BY
JONATHAN E. SISK

**PROGRAMMING WITH
IBM[®] PC BASIC[®]
AND THE
PICK[®] DATABASE
SYSTEM**

The PICK Library

Edited by Jonathan E. Sisk

The following books are in the PICK Library series:

PICK BASIC: A Programmer's Guide

by Jonathan E. Sisk

The PICK Perspective

by Ian Sandler

The PICK Pocket Guide

by Jonathan E. Sisk

PICK for the IBM PC and Compatibles

by John W. Winters, Ph.D. and Dale E. Winters

PICK for Professionals: Advanced Methods and Techniques

by Harvey E. Rodstein

PROGRAMMING WITH
IBM[®] PC BASIC[®]
AND THE
**PICK[®] DATABASE
SYSTEM**
DAVID L. CLARK



TAB Professional and Reference Books

Division of TAB BOOKS Inc.

Blue Ridge Summit, PA

Notices

ACCESS®	
Pick®	
PICK/BASIC®	Pick Systems
BASICA®	
IBM®	International Business Machines Co.
CompuSheet + ®	Interactive Systems
dBASE®	Ashton-Tate
Microsoft®	
MS-DOS®	
QuickBASIC®	Microsoft Corporation
Norton Utilities®	Peter Norton
Radio Shack	
Model 100®	Tandy Corporation
R:BASE®	Microrim, Inc.
Revelation®	Cosmos Inc.
WordStar®	MicroPro International Corp.

DEDICATION

To Suzanne, for insight and understanding.

Published by **TAB BOOKS Inc.**
FIRST EDITION/FIRST PRINTING

© 1990 by **TAB BOOKS Inc.**. Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. The publisher takes no responsibility for the use of any of the materials or methods described in this book, or for the products thereof. Printed in the United States of America.

Library of Congress Cataloging-in-Publication Data

Clark, David L., 1945 –
Programming with IBM PC Basic and the Pick database system / by
David L. Clark.
p. cm.
Includes index.
ISBN 0-8306-3322-7 (H)
1. IBM Personal Computer—Programming. 2. BASIC (Computer program
language) 3. Data base management. 4. Pick (Computer program)
I. Title.
QA76.8.I2594C56 1989
005.265—dc20

89-33691
CIP

TAB BOOKS Inc. offers software for sale. For information and a catalog, please contact
TAB Software Department, Blue Ridge Summit, PA 17294-0850.

Questions regarding the content of this book should be addressed to:

Reader Inquiry Branch
TAB BOOKS Inc.
Blue Ridge Summit, PA 17294-0214

Acquisitions Editor: Larry Hager
Book Editor: Eileen P. Baylus
Series Editor: Jonathan E. Sisk

Contents

Acknowledgments	x
Programs and Routines	xi
Introduction	1
Languages Used	1
Punctuation and Remarks	5
<i>Comments in a Program</i>	
<i>End of Program</i>	
<i>End of Statement</i>	
Special Notations for This Book	7
<i>Endings for Variable Types</i>	
<i>Format of Program</i>	
<i>IBM PC Basic Documentation Different from Code</i>	
Arithmetic	16
Integer Division	16
Remainder from Division (Modulo)	17
Batch Files and PROCs (Stored Procedures)	18
IBM PC DOS Batch files	19
Pick PROCs	19

Boolean Flags	20
In Place of “IF...THEN”	22
To Choose a String or Null	22
To Find the Longest String in an Array	23
To Swap Two Numbers if the First Is Larger	24
Characters	26
Convert	26
<i>ASCII to Character</i>	
<i>Character to ASCII</i>	
Conditions	29
Constants	41
Conversions and Correlatives in Pick	43
Arithmetic	45
Date	52
Group Extract	56
Pattern Matching	57
Text Extract	57
Translate from File	57
User Exits	59
Data Structures	60
Arrays, Fixed	60
<i>Dimension or Declare</i>	
<i>Set Value of Entire Array</i>	
Enumerated and Set	63
Pick	64
<i>Arrays, Dynamic</i>	
<i>Count Elements</i>	
<i>Delete</i>	
<i>Extract</i>	
<i>Insert</i>	
<i>Locate</i>	
<i>Replace</i>	
Editor	77
Error Recovery	79
Files, Items	82
Access System Information	85
Copy, Backup	98
Delete	105
Input	107

Convert File from IBM PC BASIC to Pick

Order of Access

Search with Pick Access Processor

Save.List, Get.List, Copy.List, Edit.List, Delete.List 134

Modulo 144

Names 146

Open, Close 148

Test for Existence

Output 156

Rename 158

Synonyms 160

Keyboard

164

Access System Information 169

BREAK Key 169

Change Keys 172

Control Characters for Editing 242

ESCAPE Key to Interrupt 243

Function Keys 247

Select a Choice from a Menu 249

Numbers

252

Convert 252

Decimal, Hexadecimal, and Other Bases

Negative to Positive

Number to String

String to Number

Operators

257

Printer

261

Access System Information, Change Settings 266

Format Masks 267

Paging, Headings, and Footings 268

Program Running and Compiling

271

Repetition

277

Screen

285

Access System Information 288

Assign Screen Constants 289

Assign Terminal 295

Assign Terminal Parameters 303

Clear Screen 312

Cursor 314

<i>Place the Cursor</i>		
<i>Return Position</i>		
<i>Turn off</i>		
Format Masks	321	
Paging, Headings, and Footings		321
Protect	322	
Redirect Output	324	
Scroll Lock	327	
Video Characteristics	327	
<i>Blinking</i>		
<i>Intensity</i>		
<i>Underline</i>		
<i>Video reverse</i>		
Size, Speed of Program		341
Advice	341	
Test Speed	362	
Strings		366
Assign	366	
Compare	367	
Match Patterns	367	
Concatenate	368	
Extract	369	
Find Length	374	
Find Position of Substring		376
Replace, Insert	377	
Spaces, Repeating Character		378
Subroutines		380
Internal	380	
External	385	
System		388
Access System Information		388
<i>Date</i>		
<i>Time</i>		
Booting, Start-Up	400	
<i>Suppress System Messages</i>		
<i>Terminal Logon</i>		
<i>Terminal Logon with a Stand-Alone Computer</i>		
Call System Commands from a Program		412
Direct Entry of Commands	418	
Partition Change from IBM PC DOS to Pick		419
Partition Change from Pick to IBM PC DOS		423

Transfer

From IBM PC DOS to Pick	425
From Pick to IBM PC DOS	430
Through a Serial Port	431
To and from Tape	434
To Screen or Printer	435

Variables

Initial Values	437
Names, Limits	437
Scope	439

Appendix: Programs and Instructions for Pick Release 2.2 and 3.0

440

Index

452

Acknowledgments

I WOULD LIKE TO THANK THE FOLLOWING INDIVIDUALS AND ORGANIZATIONS FOR THE help that they have provided:

Brian Stone, president of Alcyona Computer Consultants and also president of the Los Angeles Pick Users Group, for the course on PICK/BASIC that he taught in his home in the Hollywood Hills.

Jeffrey Hunt, Vice President of Alcyona Computer Consultants, for unraveling the mysteries of Pick on the IBM PC/XT.

Paul Anderson, Director of Education for Pick Systems, for a course on PICK/BASIC which was cut short by a fire in Malibu Canyon.

Jonathan Sisk, President of the Pick educational firm JES and Associates, for acting as editor of TAB's Pick series.

Richard Pick, for creating a marvelously useful database management system.

Pick Systems and Esprit Systems, for providing software and terminals for the History Computerization Project of the Los Angeles City Historical Society, and to TAB Professional and Reference Books, for publishing the project's course textbook under the upcoming title, *Database Management Concepts for Cataloging and Research*.

Computer Database Management for research, writing, and cataloging.

Programs and Routines

FOR ROUTINES THAT ARE NOT COMPLETE PROGRAMS AND FOR PICK/BASIC PROGRAMS and PROCs, a filename that will meet the IBM PC DOS file naming limitations follows the name of the program or routine in braces.

For IBM PC BASIC or Microsoft BASIC, a distinction is made between a documentation version of a program, presented for ease of understanding, and an executable version.

The following naming conventions indicate the type of program or routine:

Capitalization, spaces

- ☐ All Capitals: Name under Pick operating system
- ☐ Upper- and lowercase, no spaces: Name under IBM PC DOS
- ☐ Upper- and lowercase, spaces in name: A routine, not a complete program

Endings

- ☐ .asm: Assembly language program under IBM PC DOS
- ☐ .ba: Microsoft BASIC (Radio Shack Model 100), executable version
- ☐ .bar: IBM PC BASIC routine, executable version
- ☐ .bas: IBM PC BASIC program, executable version
- ☐ .bat: Batch file under IBM PC DOS
- ☐ .bp: PICK/BASIC program
- ☐ .b03: PICK/BASIC program, version for R82 2.2, 3.0

- ☐ .bpr: PICK/BASIC routine, not a complete program
- ☐ .bps: PICK/BASIC external subroutine, separately compiled
- ☐ .brd: IBM PC BASIC routine, documentation version, not executable
- ☐ .bs3: PICK/BASIC extended subroutine, version for R83 2.2, 3.0
- ☐ .bsd: IBM PC BASIC program, documentation version, not executable
- ☐ .do: Microsoft BASIC (Radio Shack Model 100), documentation version
- ☐ .prc: Pick PROC (stored procedure, like a batch file)

Introduction



INTRODUCTION: LANGUAGES USED

Programming with IBM PC BASIC and the Pick Database System is a comparative language guide for IBM PC BASIC and PICK/BASIC. The book is also a comparative systems guide for IBM PC DOS and the Pick Operating System.

IBM PC BASIC is a variant of Microsoft BASIC, because IBM's language is supplied by the Microsoft Corporation. Close relatives within the Microsoft BASIC family include the simpler version installed on the Radio Shack Model 100 portable computer and other machines less powerful than the IBM PC, and the more advanced QuickBASIC, which is compiled for faster speed of execution. The IBM PC BASIC examples presented in this book will operate under all of these Microsoft BASIC environments. Any exceptions or differences will be shown.

The Radio Shack 100 variant is not included here primarily for that machine, but as an example of the rules that apply to simpler versions of the BASIC language. If the IBM PC BASIC version does not work on your machine, try the Radio Shack 100 version.

The version of IBM PC BASIC employed is interpreted BASICA 3.0. The version of QuickBASIC is 4.0. For IBM PC DOS, the version used was 3.1.

PICK/BASIC is the compiled programming language of the Pick Operating System, which is a system created expressly for multiuser database management. Pick implements at the operating system level all of the facilities required for handling a relational database, and gives PICK/BASIC direct and convenient access to that database. Pick thereby provides the most powerful and flexible database management system available. The Pick System runs on a wide variety of machines, from microcomputers through minis to main-

frames. The version employed here is IBM PC/XT Release 1983 2.0. Differences will be noted with the versions IBM PC/AT Release 1983 2.2 and Open Architecture.

This book is organized alphabetically by topic to provide a quick reference for the programmer who needs to carry out a familiar operation in an unfamiliar language or system. The best procedure for approaching this book is to read it once through lightly for concepts and ideas that would be missed when looking only for individual commands. Then use the book as a reference, looking up general subjects through the Table of Contents and specific keywords and commands through the Index.

Immediately after the Table of Contents appear alphabetical listings of all the programs, routines, and tables. Refer to this Introduction for the conventions used in naming variables and presenting code.

For each operation, this book presents an explanation followed by parallel examples in each of the languages examined. After covering the fundamental points, the book will warn of underlying differences between superficially similar commands, explain the circumstances that could cause error conditions, suggest methods of optimizing performance, cover aspects of each language that have not been documented elsewhere, and provide program examples for duplicating features between different languages and systems.

The book contains extensive testing programs and routines, both to verify that the reader's particular version of each language and system operates in the same manner as the versions illustrated here, and to account for any differences. The emphasis is placed on providing parallels, explanations of undocumented features, complete examples, and facilities useful for those who must move back and forth between different systems, rather than on repeating material available from other sources.

The examples presented include complete programs designed to meet the needs that will arise in a multilingual and multisystem environment. The book includes programs to convert data files between IBM PC DOS and Pick, techniques for resetting the keyboard in order to use the same editing keys under different systems, routines that will allow terminals on a multiuser system to present the sort of attractive interface that single-user systems provide, and methods for employing the same machine as both a stand-alone computer and as a terminal connected to a Pick multiuser database.

The most powerful features available in computing today for capturing and utilizing information are those for entering data in the field with a convenient portable computer; performing storage, editing, maintenance, reporting, and in-office data entry with the Pick database system; and sending the information to an IBM PC DOS word processing or spreadsheet program for final reworking and presentation. During the entire journey, the information should be typed only once, and the operator should have little awareness of the underlying differences in the environments. This book was written in the course of developing the History Database program, which provides these capabilities for historical writing, research, and cataloging.

Information on the languages and systems chosen for this book follows.

IBM PC BASIC

The great advantages of IBM PC interpreted BASIC are:

- ☐ It is easy to learn at a beginning level.

- It is already installed on the IBM PC and, as Microsoft BASIC, on most other desktop and portable computers. When the programmer finds it necessary to do some quick coding on another person's computer, Microsoft BASIC will already be there.
- The same or nearly identical code can be used in compiled form with QuickBASIC or on simpler machines such as the Radio Shack 100. The QuickBASIC language from Microsoft brings to BASIC the compiled speed and structure of Pascal, while retaining the ease of use and string-handling facilities of BASIC. It is at the center of the current revival of the BASIC language. The documentation conventions presented in this guide would help greatly in making any future transition. The Portable 100 is the Swiss army knife of computers. The device weighs only three pounds, costs only \$400, runs on penlight batteries, and is supported by 7,000 Radio Shack outlets. It is ideal for data entry in the field. The currently available version, the Model 102, is identical from a programming standpoint.

Pick

Pick is an operating system built from the ground up, over a period of 20 years, expressly for the purpose of database management. It is accordingly far more powerful for that purpose than other systems constructed originally with a scientific and mathematical orientation. Many tasks that can be accomplished in Pick with little or no programming would involve extensive programming, and limitations in flexibility, with other systems.

The same Pick program will run on many different computers, from an IBM PC/XT up to minicomputers and mainframes. There are some differences among different installations of Pick, but they are almost trifling in comparison with the variations between dialects of Pascal or BASIC. Because the same code will run on very large systems as well as on small and inexpensive devices, the same program can be made available to both large and small organizations, or to a large company with many scattered operations.

Pick is a true multiuser system, designed to allow many people to have access to the same database at the same time. The multiuser aspect is available even on a limited budget. Two extra terminals can be attached to a standard IBM PC/XT, allowing three operators to use the same computer at a cost of about \$400 for each additional terminal. An IBM PC/AT will accommodate 10 operators. Desktop 80386 machines will handle up to 33 operators, still at a cost of \$400 per additional user. It is also possible to use Pick on larger systems with hundreds of terminals. Note that the methods employed do not involve networking, which brings many additional hardware and software complications and expenses.

The information stored by Pick is completely variable in length. IBM PC DOS database programs such as dBASE and R:Base use a fixed-length format requiring that the user stipulate a length for each field, and the program takes up that amount of disk storage space regardless of the amount of data. Pick takes up only the amount of space that the data requires, plus a character to separate one field from another. Some IBM PC DOS programs provide for an extra "memo" field, which can be variable in length but cannot be searched. A database field that cannot be searched is about as useful as a kickstand on a horse.

Remarks cannot interrupt a line; that is, no additional commands can follow the remark on the same line. Note that all remarks (except the first line title, purpose, and date of the program) should be removed from the final code in interpreted BASIC.

PICK/BASIC

*** or ! or REM**

X = X + 1; * This is a remark

The end-of-statement semicolon is required between a statement and a remark on the same line.

Note that the asterisk or star “*” has three different functions:

1) **Remarks in PICK/BASIC:**

 * This is a remark

2) **Multiplication in IBM PC BASIC and PICK/BASIC:**

 Result = 3 * 4

3) **Wildcard, to select all Items:**

 Pick Terminal Control Language:

 COPY Filename *

IBM PC DOS Command Level, in place of a filename or extension:

 COPY Drive: \Directory \ *.*

For maximum uniformity, this book will include the star (asterisk) in all remark designators. The following symbols will be used for remarks:

IBM PC BASIC: ' *

PICK/BASIC: *

INTRODUCTION: PUNCTUATION AND REMARKS: END OF PROGRAM

The following punctuation and keywords are used to end a program:

IBM PC BASIC

END

For example:

END ' * The program ends here

The END command leaves the user in BASIC. On the IBM PC, an IBM PC DOS batch file could be used to invoke the program, and then return the user to a menu display

of user options after the program terminated. (See the section “System: Call System Commands from a Program.”)

STOP

For example:

```
STOP          ' * The program stops abruptly here
```

The STOP command can be used to terminate the program prematurely, primarily for debugging purposes.

SYSTEM

For example:

```
SYSTEM ' * Return to IBM PC DOS command level
```

On the IBM PC, the SYSTEM command in BASIC terminates the BASIC program and returns the user to the IBM PC DOS command level.

Microsoft BASIC (Radio Shack 100)

MENU

For example:

```
MENU          ' * Return the user to Radio Shack's menu
```

Instead of the SYSTEM command, the Radio Shack 100 offers the MENU command. The MENU command will return the user to Radio Shack's menu, which appears when the machine is first turned on.

PICK/BASIC

END

For example:

```
END           * The program ends here
```

The END command returns the operator to Terminal Control Language (TCL) level. The END command is normally used to mark the physical end of a program, which should also be the logical end. If the program was invoked from a PROC (stored procedure, similar to an IBM PC DOS batch file) the PROC will resume when the program terminates. A program also can terminate with a CHAIN, ENTER, or EXECUTE statement that transfers control to another program. (See the section: “System: Call System Commands from a Program.”)

ABORT, STOP

For example:

```
ABORT          ' * The program stops abruptly here  
STOP          ' * ""
```

The STOP or ABORT command can be used to terminate the program prematurely, primarily for debugging purposes.

INTRODUCTION: PUNCTUATION AND REMARKS: END OF STATEMENT

The following punctuation is used to end a program statement:

IBM PC BASIC

: or Carriage Return

For example:

```
FOR I = Start TO Finish : PRINT I : NEXT I
```

A statement can be ended by either a colon or a Carriage Return, but not both. Interpreted BASIC programs should include the maximum number of statements on the same program line, separated by colons, to save time and memory. (See the section: "Size, Speed of Programs.")

This book presents documentation versions of IBM PC BASIC programs with program lines interrupted by Carriage Returns and remarks for clarity. The lines would not be interrupted in executable programs.

PICK/BASIC

; or Carriage Return

For example:

```
FOR I = Start TO Finish; CRT I; NEXT I
```

A statement can be ended by either a semicolon or a Carriage Return, but not both. It is an error to end a line with a semicolon, with no additional statement following on the same line.

Expressions cannot be interrupted by Carriage Returns (as is permitted in Pascal). Thus, in PICK it would be incorrect to write:

```
FOR I  
  = Start To Finish
```

With regard to conditional constructions such as IF...THEN...ELSE or CASE, or repeating conditional structures such as LOOP...WHILE...REPEAT or LOOP...UNTIL...REPEAT, it is not always obvious when end-of-statement punctuation is required, or even allowed. (For examples see the section: "Repetition.")

INTRODUCTION: SPECIAL NOTATIONS FOR THIS BOOK

This book's text differs from sanctioned *Chicago Manual of Style* usage in the following respects:

Quotes: When quotes are used, punctuation is always placed outside of the quotes, rather than within, so that the reader will not become confused as to whether the punctuation is a required convention of the particular computer language. For example, a sentence explaining that “.bas” is the standard ending for the name of an IBM PC Basic program will read:

An IBM PC Basic program filename should end with the extension “.bas”.

If the period ending the sentence had appeared within the quotes, the reader might have thought that the second period was part of the filename.

For Program Code and Text

Capitalization: All keywords for a computer language are capitalized, although capitalization is not required at IBM PC DOS command level. Specific string, variable, and statement names and other parameters supplied by the program or user are presented in upper- and lowercase. Thus the variable Message\$ would be displayed on the screen with the PRINT command.

For Program Code

Indentation:

6 spaces: From the left margin, for lines in a program, commands, or prompts on the screen. When complete programs or subroutines are presented, the 6 spaces are reserved for the program heading and the labels (or line numbers) which identify subroutines.

4 spaces: Within a program, statements nested logically. In IBM PC BASIC, additional statements on the same program line will be shown preceded by the statement-separating colon and indented. Inclusion on the same line is an important control mechanism in interpreted BASIC.

2 spaces: A continuation of a statement or expression, which is broken across lines only because it will not fit across the page. In an executable program, the statement or expression would not be interrupted by a Carriage Return.

{ } Braces: Optional parameter, unless noted as required punctuation.

* Section Name *

***** : A heading for a section or subroutine within a program.

|Section Name| : Section or subroutine names are marked by vertical lines in reference remarks. Because IBM PC BASIC and PICK/BASIC (Release 1983) use numbers, not subroutine names, the subroutine name will always appear in remarks after a call:

GOSUB 9000; *|Error Routines|

* =====* : A

boundary line between major parts of a program, such as between the assign constants, parameters section, and the main program.

----- : A boundary line between units within a program section, such as different subroutines, each of which has a separate RETURN to its caller. If the program is to continue from one routine directly into the next, there will be no boundary line, only a heading or remark.

*** Exit#1 *** : If a subroutine has more than one exit point, each will be marked with a comment as Exit #1, Exit #2, etc. It is preferable that a routine have only one exit.

Begin line: If the opening statement for a multiple-statement line in IBM PC BASIC is a control statement such as IF...THEN, in order to avoid double indentation (4 spaces for the control logic, 4 spaces for inclusion on the same line) the line will begin with the comment "Begin line".

End: All compound statements will terminate with an "End" that will appear in all capitals if it is required and in upper- and lowercase if it is only a comment.

End line: Because inclusion on the same line is a major control mechanism in IBM PC BASIC, the end of a program line that includes more than one statement in IBM PC BASIC will be indicated with the comment "End line".

Because neither IBM PC BASIC nor PICK/BASIC recognize a distinction between local and global variables, the following names will be used to indicate that a variable is strictly local to the section or routine in which it is defined. When a section of code calls a subroutine, the caller will not expect to pass or receive a parameter through any of these variables, nor to depend on finding the variable unchanged. The called subroutine might have used any of these as its own local variables.

- ☐ I, J: Local Counters for Loops
- ☐ Test.Flag Local Boolean
- ☐ U, V: Local Integers
- ☐ X\$, Y\$, Z\$: Local Strings

INTRODUCTION: SPECIAL NOTATIONS FOR THIS BOOK: ENDINGS FOR VARIABLE TYPES

The following naming conventions will be used to indicate the type of the variable and its purpose.

Note that only IBM PC BASIC actually gives a meaning to type endings, using the endings \$, %, ! and #. The other endings are utilized only to clarify the documentation. (See the section: "Variables: Names.")

In some cases a given element, such as an attribute, a value, or an item ID, might be a string or a number. The ending of the variable name would correspond to the string or number type.

- ☐ No ending = Integer, in all languages, with values from -32,767 to 32,767
All IBM PC BASIC programs should begin with the statement "DEFINT A-Z", to make integer the default variable type.
- ☐ \$ = String in IBM PC BASIC or PICK/BASIC
- ☐ ! = IBM PC BASIC: Single precision, 6 digits plus decimal point
- ☐ # = IBM PC BASIC: Double precision, 14 digits plus decimal point

- ☐ **Attrib.** = Pick: An attribute (field) within an item (record)
- ☐ **File.\$** = A file
- ☐ **Item.\$** = Pick: An item (like a record in other databases)
- ☐ **Item.ID** = Pick: The key that identifies an item
- ☐ **SubValue.** = Pick: A subvalue within a value
- ☐ **Value.** = Pick: A value within an attribute
- ☐ **.Bin** = Number in binary
- ☐ **.Flag** = Boolean type, only true or false
- ☐ **.Hex** = Number in hexadecimal
- ☐ **.Name\$** = Name rather than the element, such as File.Name\$, Attrib.Name\$

INTRODUCTION: SPECIAL NOTATIONS FOR THIS BOOK: FORMAT OF PROGRAM

For details on each command, see the Index under the name of the command. To explain the reasons for these conventions, see the section: "Size, Speed of Program." For more information on the required parts of a program, see the section: "Program Running and Compiling."

IBM PC BASIC

Make each line number greater than the one before by 10, and preferably by 20. After thorough debugging, for greater speed and smaller size, eliminate the final zero from all the line numbers.

The filename for a BASIC program on the IBM PC ends with the extension ".bas". A filename on the IBM PC can be eight characters in length. (On the Radio Shack Model 100 a program filename ends with ".ba" and only six characters are allowed.)

Sample Program, Documentation Version

Sample: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

TF = Test.Flag : Temporary Boolean

```

1      '* Sample.bsd: Sample Program.  Date   Author
2      '* One-line explanation of purpose
3      '* IBM PC Basic program. Documentation or executable version.
4      '* IBM PC BASICA 3.0, Radio Shack 100 (Versions tested successfully)
5      '* Copyright notice, if applicable
6      '* Further explanations, if necessary
10     GOTO 9500          '*|Assign Constants, Parameters, Initial Values|
      '*=====*
20     '* Speed-critical subroutines, with the most critical first
      '* Indent any nested subroutines
      '* For each subroutine:
      '******
      '* Subroutine Name *
      '******
      '* Parameters: Used by subroutine
      '* Returns: Values returned by subroutine
      '* Local variables: Not exchanged outside of subroutine
      RETURN
      '*-----*
      '* Single lines separate subroutines that return separately.
      '*   Otherwise the first routine drops through to the second.
      '*=====*
      '******
6000    '* Main Program *
      '******
      '* Set parameters for each subroutine
      GOTO 9990          '*|End Program|
      '*=====*
7000    '* Subroutines that are not speed-critical
      '*=====*
      '******
9000    '* Error Recovery *
      '******
      '*=====*
      '******
9500    '* Assign Constants, Parameters, Initial Values *
      '******
      ON ERROR GOTO 9500      '*|Error Recovery|
      DEFINT A-W              '* Integers without type ending
      DEFSTR X-Z              '* Strings without type ending
      '* Initialize speed-critical constants and variables to null or zero
      '*   with DIM
      DIM Entry$, Key$, Length
      '* Boolean constants
      False = 0
      True  = -1
      '******
      '* Read Parameters File *
      '******
      '* Read parameters from a start-up file
      '* Dimension arrays, perhaps according to parameters from file
      GOTO 6000              '*|Main Program|
      '*=====*
      '******
9900    '* Set Parameters File *
      '******
      '* Put parameters in a file for use at start-up next time
      '* May return, or go straight through to End
      '*-----*

```

```

          '*****
9990      '* End Program *
          '*****
          '* Any last clean-up, reset system
END      '* Radio Shack 100: MENU

```

PICK/BASIC

SAMPLE.PROGRAM Program

```

* SAMPLe.PROGRAM Date Author
* One-line explanation of purpose
* Pick Basic program.
* Pick IBM PC-XT R83 2.0 {Versions tested successfully}
* Filename under IBM PC DOS: Sample.bp
* Copyright notice, if applicable
* Further explanations, if necessary.
*****
* Assign Constants, Parameters, Initial Values *
*****
* Declare common variables shared between program modules
COMMON File.Data$, File.Data.Name$, Print.Parameter(4)
* Dimension arrays not included in COMMON
DIM Fix.Ay$(10)
* Equate constants. These cannot be changed.
EQUATE FALSE TO 0, TRUE TO 1
* Equate simple variables to fixed array elements
EQUATE Left.Margin TO Print.Parameter(1)
* Set prompt character to null
PROMPT ""
*****
* Read Parameters File *
*****
* Read parameters from a start-up file
=====
*****
* Main Program *
*****
* Set parameters for each subroutine before calling
GOTO 9990; *|End Program|
=====
* Subroutines
* Indent any nested subroutines
* For each subroutine:
*****
* Subroutine Name *
*****
* Parameters: Used by subroutine
* Returns: Values returned by subroutine
* Local variables: Not exchanged outside of subroutine
RETURN
*-----*
* Single lines separate each subroutine that returns separately.
* Otherwise the first routine drops through to the second.
=====
*****
9000 * Error Recovery *
*****
=====

```



```

*****
9900 * Set Parameters File *
*****
* Put parameters in a file for use at start-up next time
* May return, or go straight through to End
*-----*
*****
9990 * End Program *
*****
* Any last clean-up, reset system
END

```

INTRODUCTION: SPECIAL NOTATIONS FOR THIS BOOK: IBM PC BASIC DOCUMENTATION DIFFERENT FROM CODE

Various conventions are used in the documentation, for clarity of presentation, which must not appear in the final code submitted to the interpreter or compiler.

IBM PC BASIC

Interpreted BASIC programs longer than a few lines should be written in two versions: one for documentation, and another to execute. Clear documentation requires several features that must be removed from the executable code. Use of the conventions explained here allow you to simulate the clear, structured code of Pascal, while employing the features of BASIC that are necessary to optimize speed and memory usage. The documentation conventions will greatly assist any future transition to QuickBASIC or other more advanced languages.

IBM PC BASIC variable names will be shown here in upper- and lowercase. In the executable program, the names would all be uppercase. IBM PC BASIC will automatically change all lowercase letters that are not between quotes or in DATA statements or comments to uppercase. Thus TotalVersions, after submission to the BASIC interpreter, would become TOTALVERSIONS.

Simpler versions of BASIC, such as that used on the Radio Shack Model 100, do not allow periods in a variable name. In this book, the parts of a variable name will be separated by periods for clarity. Simpler BASIC languages also regard only the first two characters of the name as significant. Thus TOTALVERSIONS would be seen by the interpreter as the same as TOTALELEMENTS, TOTALFIELDS, and as the reserved word TO. (See the section: "Variables: Names.")

In the documentation version of a program, the variable name will appear as Total.Versions, and in the executable program as TV. In moving from documentation to the executable program, the short version of the name will consist of the first letter of each part that is separated by a period in the longer version. A dictionary will be used to tie together the two versions of the name, for example:

Program Name: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

TV = Total.Versions

IBM PC BASIC requires a line number for every line. Short program fragments generally will be shown without line numbers.

Inclusion of statements on the same program line is a major control mechanism in IBM PC BASIC, which provides no other ending for an "IF...THEN...ELSE" construction than ending the line. Compare the following two examples:

```
10 IF Test.Flag THEN PRINT "True" : GOSUB 100
```

```
10 IF Test.Flag THEN PRINT "True"  
20 GOSUB 100
```

In the first example above, the program will go to line 100 only if Test.Flag is true. In the second example, the program will go to line 100 whether or not Test.Flag is True. (For details on conditional branching, see the section: "Conditions.")

As many statements as possible should be placed on a single program line in IBM PC BASIC to save time, memory space, and to avoid extra "GOTO Line.Number" statements. In this guide, statements and clauses meant to continue on the same program line will be shown on separate documentation text lines, indented four spaces and preceded with any required end-of-statement colons. (Note that "ELSE" clauses do not constitute separate statements, and so do not require a preceding colon.) The end of the line will also be marked by an "End line" comment.

If the opening statement for the line is a control statement, such as "IF ..." or "FOR I = 1 TO ...", to avoid double indentation (four spaces for the control logic, four spaces for inclusion on the same line) the line will begin with the comment "Begin line".

Note that in the actual code submitted to the interpreter, comments cannot interrupt a program line. In the final code, all comments would be removed (except for the first line title, author, and date), and the separate statements would appear on the same line.

The following "Show Array" routine illustrates the differences between the documentation and executable code.

Show Array Routine

Show Array Routine: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

D\$()	= Data\$()	: Array holding the data
TE	= Total.Elements	: Total number of elements in Data\$() array

Documentation Version of Routine

```
'*****
'* Show Array *
'*****
'* IBM PC Basic routine. Documentation version.
'* IBM PC BASICA 3.0, Radio Shack 100
'* Stored under filename: ShowArray.brd
'* Begin line
  FOR I = 1 TO Total.Elements
:   PRINT Data$(I)           '* Display each array element
:   NEXT
:   RETURN
'* End line
```

Executable Version

```
10 '*Show Array
20 FOR I=1 TO TE : PRINT D$(I) : NEXT : RETURN
```

Arithmetic



THE MOST COMMON ARITHMETIC OPERATORS, SHOWN HERE, ARE THE SAME IN ALL OF the languages:

+	Add
-	Subtract
*	Multiply
/	Divide

(For the precedence of operators, see the section: “Operator Precedence.”)

ARITHMETIC: INTEGER DIVISION

To return an integer as a result from division, rather than a fraction, use integer division. In each case the number is truncated rather than rounded.

IBM PC BASIC

\

For example:

Result = A \ B

PICK BASIC

**INT and **
For example:

Result = INT(A/B)

ARITHMETIC: REMAINDER FROM DIVISION (MODULO)

To return the sum remaining after division:

The remainder of $7 \setminus 3 = 1$, the remainder of $6 \setminus 3 = 0$.

IBM PC BASIC

MOD
For example:

Remainder = A MOD B

PICK BASIC

MOD or REM
For example:

Remainder = MOD(A, B)
Remainder = REM(A, B)

Batch Files and PROCs (Stored Procedures)



BATCH FILES IN IBM PC DOS AND PROCs IN PICK CAN BE EMPLOYED AS VERY SIMPLE JOB control languages for purposes such as transferring the user to the appropriate DOS directory or Pick account, without the user having to learn the operating system commands required. These languages should not be employed for more than very simple purposes for the following reasons:

- ☐ They are slow. They are interpreted rather than compiled, thus the system must read in and interpret each command at run time.
- ☐ They are not structured. For example, they lack subscripted variables and looping statements other than GOTO. Documentation included in the batch file or PROC will make it run even more slowly, because the system must interpret every line at the time of execution.
- ☐ They are unnecessary. System commands can be called directly from a program. (See the section: “System: Call System Commands from a Program.”)

If it takes more than a few minutes to write the batch file or PROC, the commands should be placed in a program instead.

BATCH FILES AND PROCS: IBM PC DOS BATCH FILES

IBM PC DOS

An IBM PC DOS batch file is called by typing its name at the operating system level. The file consists of operating system commands. The file can transfer control to a program or to another batch file by invoking its name. Control will return to the calling batch file after the called program or batch file has finished execution. The filename of the batch file must end with the extension ".BAT" or ".bat".

The @ ECHO OFF statement will prevent display of commands on the screen. The REM statement can be used to precede remarks, just as in an IBM PC BASIC or PICK/BASIC program.

The following simple batch file, named "Alice.bat", will transfer the user Alice to her own directory on disk "C", invoke the program called "WORD", and then return her to the root directory after the program has finished:

```
@ECHO OFF
REM Switch to Alice Directory, start WORD, return to Root Directory.
C:
CD \Alice
WORD
CD \
```

An IBM PC DOS batch file can be used to simulate some of the features provided under the Pick System of separate accounts and synonym filenames. (See the section: "Files, Items: Synonyms.")

BATCH FILES AND PROCS: PICK PROCS

Pick TCL

Each PROC in the Pick System must begin with the letters "PQ" in the first attribute, meaning "Prestored Query."

A PROC must be called from the master dictionary. It is normal practice to place in the master dictionary only a very short PROC that calls the full PROC. The full PROC is stored in a PROC library. The reasons for not leaving the full PROC in the master dictionary is that the system will run faster if the master dictionary is kept as small as possible.

For the short PROC in the master dictionary, the second attribute will contain the call passing control to the full PROC. The call passing control without return to the caller is placed in parentheses, and contains the name of the file in which the full PROC is stored, and then the name of the PROC. If the PROC name is the same in the master dictionary and the PROC library, it is not necessary to include the name in the call. Because there will be no return to this PROC after the call, it is unnecessary to include anything more:

```
Item ID Proc.Name
01 PQ
02 (Proc.Library)
```

Boolean Flags



BOOLEAN FLAGS OF TRUE AND FALSE WILL OFTEN MAKE CONDITIONAL TESTS CLEAR and also make them faster, particularly when the results of a test must be used several times.

IBM PC BASIC

```
False = 0
True  = -1
Test.Flag = (X$ = Y$)           '*Test.Flag will be either 0 or -1
IF Test.Flag THEN GOSUB 100
```

The assignment to Test.Flag is made clearer with parentheses, but the parentheses are not required. For example:

```
Test.Flag = X$ = Y$           '*Test.Flag will be either 0 or -1
```

Note that the IBM PC BASIC uses -1 for true, rather than 1 , which is the standard value in other languages.

In IBM PC BASIC there is no special Boolean type (as there is in Pascal). Any value except 0 and -1 will pass all positive and negative tests. Thus, if $A = 5$ or -5 , it will pass both “IF A ...” and “IF NOT A ...”. Numeric variables that have not specifically

been assigned as 0 or - 1 can be used, with caution, for positive Boolean tests. Thus to ascertain whether the string Match\$ exists within the string Search\$:

```
Pos.Match = INSTR(Search$, Match$)
IF Pos.Match THEN ...
```

The last statement is equivalent to:

```
IF Pos. Match > 0 THEN ...
```

The Boolean test is approximately twice as fast. You could not use the negative counterpart:

```
IF NOT Pos.Match THEN ...
```

The variable would pass a negative test unless the variable equaled - 1. The reason is that "NOT" acts as a bit-wise operator, changing every bit in the number to its opposite. If you began with a value other than - 1, the result would be a number other than zero; therefore, a IBM PC BASIC would read it as true. Since "AND" and "OR" are also bit-wise operators, do not use Boolean tests on values other than zero and - 1 with "AND" or "OR" either.

Odd or Even Number in IBM PC BASIC

To determine if a number is even or odd, use the IBM PC BASIC Boolean bit-wise operator AND. The operator compares each bit of two numbers. If both bits are 1, the result is 1 (Boolean true). If either or both bits are zero, the result is zero (Boolean false).

The last (least significant) bit of an even number will equal zero, the last bit of an odd number will equal 1. Therefore, to determine if a number is odd or even, AND it with a binary 1, the 8 bits of which equal "00000001." For example:

```
IF (Test.Num AND 1) THEN
    Odd.Num.Flag = True
ELSE
    Odd.Num.Flag = False
' *End
```

or:

```
Odd.Num.Flag = (Test.Num AND 1)
```

PICK/BASIC

```
EQUATE False TO 0, True TO 1
Test.Flag = (X$ = Y$);
IF Test.Flag THEN GOSUB 100
```

*Test.Flag will be either 0 or 1

Parentheses are required in some, but not all, versions of Pick. Because parentheses make the expression clearer, always include them.

Any number other than 0 will pass all positive tests. Thus, if $A = -5$ or 5, it will pass "IF A ...". A number must be zero to pass a negative test. Numeric variables that have not been specifically assigned as 0 or 1 can be used, with more confidence than in IBM PC BASIC, for Boolean tests. Thus to ascertain whether the string Match\$ exists within the string Search\$:

```
Occur.Num = 1
Pos.Match = INDEX(Search$, Match$, Occur.Num)
IF Pos.Match THEN ...
```

The last statement is equivalent to, but faster than:

```
IF Pos.Match > 0 THEN ...
```

A null string in Pick is treated as equivalent to the number zero. However, if a Boolean flag test is made on a string, and the string has characters in it that are not numeric, an error will occur indicating the use of nonnumeric data when numeric was expected. So, do not use a statement such as "IF Data. String\$ THEN ...".

BOOLEAN FLAGS: IN PLACE OF "IF...THEN"

A Boolean flag can be used in place of an "IF...THEN" statement to act on the results of a conditional test. The major difference between the two methods is that an "IF...THEN" test will break the continuous flow of action in a program line, requiring that subsequent statements not in agreement with the final test begin on a new line.

As an alternative, a Boolean flag can be multiplied by a numeric variable to return either the original value or 0. That variable can be used to obtain the same results as would be accomplished with "IF...THEN" clauses. The following sections illustrate the use of such a procedure to choose between a string and a null value, to find the longest string in an array, and to swap two numbers when the first is larger than the second.

The difference between the Boolean flag and "IF...THEN" methods is particularly important to IBM PC BASIC, where inclusion of multiple statements on the same line is a major control mechanism, and also vital for speed and saving memory.

BOOLEAN FLAGS: TO CHOOSE A STRING OR NULL

In the following examples, a numeric variable is multiplied by a Boolean flag. The results will be either zero or the original value of the variable, depending on whether the Boolean flag is true or false. That numeric variable is used to determine the number of characters to extract from a string. If the Num.Chars variable equals zero, the new string New\$ will be null. Such a procedure could be used, for example, to determine whether an error warning should be included in a message, depending on whether the Error.Flag was true or false.

IBM PC BASIC

```
Num.Chars = Num.Chars * - Test.Flag  
New$ = LEFT$(Old$, Num.Chars)
```

Note that because IBM PC BASIC's True = - 1, you must also use a negative operator on Test.Flag. Thus if Test.Flag is true, the expression:

```
Num.Chars * - Test.Flag
```

is equivalent to:

```
Num.Chars * - - 1
```

or:

```
Num.Chars * 1
```

PICK/BASIC

```
Num.Chars = Num.Chars * Test.Flag  
Pos.Start = 1  
New$ = Old$[Pos.Start, Num.Chars]
```

BOOLEAN FLAGS: TO FIND THE LONGEST STRING IN AN ARRAY

The following routine in IBM PC BASIC will find the longest string in an array. (The same procedure would apply to finding the largest number in an array.) The routine would be used in preparation for displaying a list of choices. The longest of the array Choice\$(Choices.Total) must be identified to plan the display.

Find Longest String in Array Routine

```
'*****  
'* Find Longest String in an Array*  
'*****  
'* Find Length of longest Choice$  
'* Parameters: Choice$(Choices.Total)  
'* Returns: Longest = Index to longest element in Choice$( ) array  
'* Local variables:  
'*   New : Length of the newest Choice$  
Longest = 0  
FOR I = 1 to Choices.Total  
:   New = LEN(Choice$(I))
```

The “IF...THEN” method will require breaking the line:

```
: IF Longest < New THEN
    Longest = New
: NEXT I
ELSE
    NEXT I
'* End: If Longest
'* End line
```

The Boolean method will allow the line to continue with other statements:

```
: Longest = Longest - (New - Longest) * (New > Longest)
: NEXT I
```

The Boolean statement operates as follows in each pass through the loop. If the previous longest is longer:

```
Longest = Longest - ((New - Longest) * 0),   therefore:
Longest = Longest - 0,                       therefore:
Longest = Longest
```

If the new value is longer:

```
Longest = Longest - ((New - Longest) * - 1),   therefore:
Longest = Longest - - (New - Longest),          therefore:
Longest = Longest + (New - Longest),           therefore:
Longest = New
```

Recall that the true value in IBM PC BASIC = - 1, that two minuses are equivalent to a plus, that multiplication has precedence over addition and subtraction, and that any value multiplied by zero equals zero.

BOOLEAN FLAGS: TO SWAP TWO NUMBERS IF THE FIRST IS LARGER

The “Check Start and End Positions” routine in IBM PC BASIC will swap two values if the first is greater than the second. The routine would be used as part of a “Cut and Paste” operation, to set the starting and ending positions of the string to extract and put into a buffer. The heading of the routine is followed by explanations of its operations.

```
*****
'* Check Start and End Positions *
*****
'* Parameters: Pos.Start.Buffer, Pos.End.Buffer
'* Returns: Pos.Start.Buffer, Pos.End.Buffer
'* Make sure that the End does not come before the Start
```

The "IF...THEN" method requires ending the line:

```
'* Local variables:
'*   Temp : Temporary holder for swap
'* Begin line
    IF Pos.Start.Buffer > Pos.End.Buffer THEN
        Temp           = Pos.Start.Buffer
:       Pos.Start.Buffer = Pos.End.Buffer
:       Pos.End.Buffer  = Temp
    '*End: If Pos.Start.Buffer
'*End line
```

The Boolean method allows the line to continue with other statements:

```
'* Local variables:
'*   Swap.Flag : If True, a swap of values is necessary
'*   Difference : Amount to change between values
Swap.Flag      = Pos.Start.Buffer > Pos.End.Buffer
:   Difference  = Pos.Start.Buffer - Pos.End.Buffer
:   Pos.Start.Buffer = Pos.Start.Buffer - Difference * Swap.Flag
:   Pos.End.Buffer  = Pos.End.Buffer + Difference * Swap.Flag
```

SWAP

For example:

Swap Pos.Start.Buffer, Pos.End.Buffer

The IBM PC BASIC SWAP statement eliminates the use of a temporary holder. (Neither PICK/BASIC nor Microsoft BASIC on the Radio Shack 100 has a SWAP command.)

Characters



CHARACTERS: CONVERT: ASCII TO CHARACTER

The following commands convert an ASCII decimal number into the corresponding character. Note that when displaying a character, control characters such as the Bell (ASCII 7) or Form Feed (ASCII 12) will cause actions to occur, such as sounding the beeper or clearing the screen.

IBM PC BASIC

CHR\$

For example:

```
A.Char$ = CHR$(Ascii.Num)
```

The following AscChar.bas (ASCII to Character) program will display the character corresponding to the ASCII number entered, until ENTER alone (or BREAK) is pressed.

AscChar.bas Program

```
1  '* AscChar.bas: ASCII to Character. David L. Clark 09/22/1988
2  '* Display character for ASCII number, until ENTER alone or BREAK
3  '* IBM PC Basic program. Executable version.
4  '* IBM PC BASICA 3.0, Radio Shack 100
5  '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
```

```

10  DEFINIT A-Z
20  LINE INPUT "Enter ASCII number: "; X$
30  IF X$ = "" THEN GOTO 9999      '*|End Program|
40  A = VAL(X$)
50  PRINT "ASCII = "; A; "  Character = "; CHR$(A) '* If not number, zero
60  GOTO 20                        '* Loop back
9999 END

```

PICK/Basic

CHAR

For example:

A.Char\$ = CHAR(Ascii.Num)

The following ASCII.TO.CHAR program will display the character corresponding to the ASCII number entered, until ENTER alone (or BREAK) is pressed.

ASCII.TO.CHAR Program

```

* ASCII.TO.CHAR David L. Clark 04/27/1987
* Display character for ASCII number entered, until ENTER alone or BREAK.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: AscChar.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  PROMPT ""
  LOOP
    CRT "Enter ASCII number: " :
    INPUT X$
  UNTIL X$ = "" DO
    Ascii = X$;          * If not a number, will be zero
    CRT "ASCII = " : Ascii : "  Character = " : CHAR(Ascii)
  REPEAT
END

```

If a program attempts to set a character to an ASCII value over 255, no error occurs. Pick subtracts 256 from the excess value. Thus CHAR(300) is set to CHAR(44), CHAR(500) is set to CHAR(244).

CHARACTERS: CONVERT: CHARACTER TO ASCII

To derive the ASCII decimal number for a character.

IBM PC BASIC

ASC

For example:

Ascii.Num = ASC(A.Char\$)

Ascii.Num will contain the ASCII decimal number of A.Char\$. If A.Char\$ is a string more than one character long, Ascii.Num will contain the ASCII number of the first character in the string. If A.Char\$ is null, a function call error will result.

The following ChrAsc.bas (Character to ASCII) program will show the ASCII number of a character typed at the keyboard, until BREAK (or CONTROL-C) is pressed.

CharAscii.bas Program:

```

1      '* ChrAsc.bas Copyright 04/27/1987 David L.Clark -- TAB BOOKS Inc.
2      '* Show ASCII number for keyboard character, until BREAK
10     DEFINT A-Z
20     PRINT "Press a key: ";
30     K$ = INKEY$ : IF K$="" GOTO 30
40     A = ASC(K$)
50     PRINT " : ASCII = "; A
60     IF A <> 3 GOTO 20
9999  END

```

PICK/BASIC

SEQ

For example:

Ascii.Num = SEQ(A.Char\$)

Ascii.Num will contain the ASCII decimal number of A.Char\$. If A.Char\$ is a string more than one character long, Ascii.Num will contain the ASCII number of the first character in the string.

The following CHAR.TO.ASCII program will show the ASCII number of a character typed at the keyboard, until BREAK (or CONTROL-C) is pressed.

CHAR.TO.ASCII Program

```

* CHAR.TO.ASCII: Character to ASCII David L. Clark 04/27/1987
* Show ASCII number for keyboard character, until BREAK
* To send output to the printer, run the program with the "P" option:
* "RUN TEST.BP CHAR.TO.ASCII (P)"
      PROMPT " "
      LOOP
          CRT "Press a key: " :
          INPUT Key$, 1 :
          Ascii = SEQ(Key$)
          PRINT "      ASCII = "; Ascii
      UNTIL Ascii = 3 DO
      REPEAT
      END

```


Conditions



THE COMMANDS IN THIS SECTION ARE USED TO TEST A CONDITION, AND THEN TAKE different courses of action depending on the results of the test. (For a testing method that allows the program flow to continue on the same line without a break, see the section: “Boolean Flags: Used in Place of IF...THEN.” For repeating loops, with conditional tests of whether to continue, see the section: “Repetition.”)

In all of the languages, an ELSE clause refers back to the most recent IF without an ELSE. You can therefore climb back up a ladder of ELSE clauses to a previous IF. An example will be shown in each language.

IBM PC BASIC

IF...THEN...ELSE

Only the single-line form is allowed, because the only means that IBM PC BASIC provides for ending the IF structure is to end the line:

```
10 IF Test.Flag THEN X$ = Y$ : GOSUB 100 ELSE X$ = Z$ : GOSUB 200
```

Note the colon to separate multiple statements. No colon is required before the ELSE clause. For documentation purposes, the line can be rendered more clearly as follows. (The executable program should be written as above, not in a multiple-line form. A comment cannot interrupt a line in an executable program.)

```

10  '* Begin line
    IF Test.Flag Then
        PRINT "True"
:    GOSUB 100
    ELSE
        PRINT "False"
:    GOSUB 200
    '* End: If Test.Flag
    '* End line

```

The program will continue with additional statements on the line only after having passed the previous conditional test:

```

10  '* Begin line
    IF Test.Flag.1 THEN
        PRINT "Test 1 = True"
    ELSE
        PRINT "Test 1 = False"
:    IF Test.Flag.2 THEN
        PRINT "Test 1 = False, Test 2 = True"
        '* End: If Test.Flag.2
    '* End: If Test.Flag.1
    '* End line

```

This program will never get to Test.Flag.2 unless Test.Flag.1 is false. Statements that should be acted upon whether or not Test.Flag.1 is true must be placed on a separate line:

```

10  '* Begin line
    IF Test.Flag.1 THEN
        PRINT "Test 1 = True"
    ELSE
        PRINT "Test 1 = False"
    '* End: If Test.Flag.1
    '* End line
20 IF Test.Flag.2 THEN PRINT "Test 2 = True"

```

Remember that statements that follow a conditional test on the same line will not be executed if the condition proves untrue. The execution of a FOR...NEXT loop is faster if the loop is placed on one line. However, the inclusion of a conditional clause in the loop will require ending the line, otherwise the loop will not reach the NEXT statement when the condition is untrue. For example:

```
FOR I = 1 TO 10; PRINT I; IF I < 6 THEN A(I) = 1; NEXT
```

The routine will print the numbers 1 through 6. The routine will not reach the NEXT statement after I becomes greater than 6. The correct method would be to place NEXT on a separate line:

```
FOR I = 1 TO 10; PRINT I; IF I < 6 THEN A(I) = 1
NEXT
```

An alternative to avoid breaking the line is to use a Boolean statement for the assignment:

```
FOR I = 1 TO 10; PRINT I; A(I) = - (I < 6); NEXT
```

Note that a Boolean true in IBM PC BASIC equals -1. (See the section: "Boolean Flags.")

The following short TIfElse program will climb back up a ladder of ELSE clauses to a previous IF.

TIfElse Program, Documentation Version

```
1      '* TIfElse.bas Copyright 1987 David L. Clark -- TAB BOOKS
2      '* Test IF...THEN...ELSE
3      '* IBM PC Basic program. Documentation version.
4      '* IBM PC BASICA 3.0, Radio Shack 100
5      '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10     DEFINT A-Z
20     INPUT "Test #1 (Enter -1 or 0)"; Test.Flag.1
      : IF Test.Flag.1 THEN
      :   INPUT "Test #2 (Enter -1 or 0)"; Test.Flag.2
      :   IF Test.Flag.2 THEN
      :     PRINT "Test #1 = True, Test #2 = True"
      :   ELSE
      :     PRINT "Test #1 = True, Test #2 = False"
      :   '* End: If Test.Flag.2
      : ELSE
      :   PRINT "Test #1 = False"
      :   '* End: If Test.Flag.1
      '* End line
9999  END
```

Note that there is no statement-separating colon before an ELSE clause.

When the phrase "THEN GOTO" or "ELSE GOTO" appears, both words are not required in IBM PC BASIC. Thus the statement:

```
IF Test.Flag THEN GOTO 100 ELSE GOTO 200
```

can be rendered:

```
IF Test.Flag GOTO 100 ELSE 200
```

or:

```
IF Test.Flag THEN 100 ELSE 200
```

The same shortcut does not apply to GOSUB.

For a conditional test that must be repeated, such as reading from a file until reaching the end, an "IF...GOTO Line.Number" construction is commonly used:

```

10 LINE INPUT #1, Line.From.File$
:   PRINT #2, Line.From.File$
:   IF NOT EOF(1) GOTO 10

```

Input would be taken from a file 1 and sent to file 2, until the end of file 1.

However, an "IF...GOTO Line.Number" loop is slower than a FOR...NEXT loop, and requires an extra line number reference. If a great many repetitions are likely, use FOR...NEXT and change the index variable with a Boolean operation:

```

FOR   I = 1 to 1000
:     LINE INPUT #1, Line.From.File$
:     PRINT #2, Line.From.File$
:     I = I + 1000 * - EOF(1)
:     NEXT

```

(See the sections: "Files, Items: Input," "Boolean Flags: Used in Place of IF...THEN," and "Repetition.")

In IBM PC BASIC a series of ELSE IF clauses simulates the CASE statement of other languages:

```

IF Char$ = > "A" AND Char$ <= "Z" THEN PRINT "Upper case letter"
ELSE IF Char$ = > "a" AND Char$ <= "z" THEN PRINT "Lower case letter"
ELSE IF Char$ = > "0" AND Char$ <= "9" THEN PRINT "Digit"
ELSE IF Char$ = > CHR$(0) AND Char$ <= CHR$(31) THEN
  PRINT "Control character"
ELSE PRINT "Punctuation, math or special symbol"
'* End: If Char$

```

ON...GOSUB, ON...GOTO

For example:

```

ON Numeric.Expression GOSUB   Line.Number, Line.Number, ...
ON Numeric.Expression GOTO   Line.Number, Line.Number, ...

```

IBM PC BASIC evaluates the numeric expression and truncates it to an integer. The program branches to the Line.Number whose position in the list corresponds to that integer. The numeric expression is a variable, but each Line.Number must be a literal. For example:

```

ON Number GOSUB 100, 200, 300 : RETURN

```

If Number = 1, the program will branch to line 100; if Number = 2, to line 200; and if 3, then to line 300. After branching, the program continues with the next statement, which can be on the same line.

If Number is negative, an illegal function call error will result. If Number is 0 or is greater than the number of line numbers on the list, the program will continue with the next statement, which can be on the same line.

Unlike the “IF...THEN...ELSE” construction, “ON...GOSUB” does not break the flow of the line. Thus in the previous example, as long as Number is not negative, the RETURN statement will be executed.

The order of the line numbers determines which branch is taken, not their value. Line numbers can also be repeated. Thus the following is legal:

```
ON Number GOSUB 700, 300, 100, 700, 200
```

If Number = 1, the program will branch to line 700, if Number = 2, to line 300, etc.

Any expression that evaluates to a number can be employed as the control variable. Thus INSTR could be used to determine the position of a character or substring within a string, and branch according to that position. In the following example, INSTR and ON...GOSUB are used with a Table of Control Characters. The program branches to an appropriate editing routine according to which key has been pressed:

```
ON INSTR(Table.Ctrl.Chars$, Key$) GOSUB 100, 200, 300 ...
```

(For the remainder of the example, see the section: “Keyboard: Control Characters for Editing.”)

IBM PC BASIC also provides a variety of other conditions under which an ON...GOSUB can be executed. In each case the line number must be a literal, not a variable.

ON COM GOSUB Line.Number

For example:

```
ON COM(Serial.Port.Number) GOSUB 100  '* IBM PC
ON COM GOSUB 100                      '* Radio Shack 100
```

A communication received through the serial port will interrupt the program and cause it to branch to the designated line number. (See the section: “Transfer: Through a Serial Port.”)

ON ERROR GOTO Line.Number, RESUME {Line.Number or NEXT}

For example:

```
ON ERROR GOTO 9000                      '*|Error Routines|
RESUME
RESUME 100
RESUME NEXT
```

The occurrence of an error recognized by the system will interrupt the program and cause it to branch to the designated line number. The RESUME statement resumes the program after the branching. (See the section: “Error Recovery.”)

ON KEY GOSUB Line.Number { , Line.Number, ... }

For example:

```
ON KEY GOSUB 100, 200, 300, , 500 ...
```

Pressing a function key interrupts the program and causes it to branch to the line in a list of line numbers that corresponds to the number of the function key. (See the section: "Keyboard: Function Keys.")

IBM PC BASIC

ON TIMER (Number.of.Seconds) GOSUB Line.Number

For example:

```
Number.of.Seconds = 100
ON TIMER(Number.of.Seconds) GOSUB 100
```

The passage of the designated number of seconds will interrupt the program and cause it to branch to the Line.Number specified. (See the section: "System: Access to System Information: Time.")

Microsoft Basic (Radio Shack 100)

ON TIME\$ = Interrupt\$ GOSUB Line.Number

For example:

```
Interrupt$ = "12:00:00"
ON TIME$ = Interrupt$ GOSUB 100
```

The arrival of the designated time will interrupt the program and cause it to branch to the designated Line.Number. (See: "System: Access to System Information: Time.")

The ON TIMER statement of IBM PC BASIC is not available in Microsoft BASIC on the Radio Shack 100.

ON MDM GOSUB Line.Number

For example:

```
ON MDM GOSUB 100
```

A communication received through the Radio Shack 100's modem will interrupt the program and cause it to branch to the designated line number.

PICK/BASIC

IF...THEN...ELSE

Single line test:

```
IF Test.Flag THEN CRT "True" ELSE CRT "False"
```

A single line test can include multiple statements, separated by the end-of-statement semicolon:

```
IF Test.Flag THEN CRT "True" ; GOSUB 100
```

The program will branch to Label 100 only if Test.Flag is true. If the two statements are placed on separate lines:

```
IF Test.Flag THEN CRT "True"  
GOSUB 100
```

then the program will branch to label 100 whether or not Test.Flag is true.

The following is a single test with both THEN and ELSE clauses:

```
IF Test.Flag THEN CRT "True"; GOSUB 100 ELSE CRT "False"; GOSUB 200
```

Note that there is no end-of-statement marker before the ELSE, because IF ...THEN...ELSE constitutes a single statement. (The same rule applies to IBM PC BASIC.)

Single-line ELSE IF form:

```
IF Test.Flag.1 THEN GOSUB 100 ELSE IF Test.Flag.2 THEN GOSUB 300
```

The program will branch to label 300 if Test.Flag.1 is false and Test.Flag.2 is true.

Multiple-line form:

```
IF Test.Flag THEN  
    CRT "Test = True"  
    GOSUB 100  
END ELSE;           * Note END before ELSE  
    CRT "Test = False"  
    GOSUB 200  
END
```

“END” ends a previous multiline statement. The following END and ELSE are separate. It is only a convention to show them together. If the clause following END ELSE occupies the same line, no final END is necessary:

```
IF Test.Flag THEN  
    A = 1  
    B = 2  
END ELSE C = 3; D = 4  
* End: If Test.Flag
```

or:

```
IF Test.Flag THEN  
    A = 1  
    B = 2  
END ELSE  
    C = 3  
    D = 4  
END
```

Both the THEN and ELSE clauses are optional. You can have an ELSE without a THEN:

```
IF Test.Flag ELSE CRT "Test = False"
```

No THEN clause, multiline ELSE:

```
IF Test.Flag ELSE
    CRT "Test = False"
    GOSUB 200
END
```

Note that the tests lacking a THEN clause probably would be rendered with a negative test:

```
IF NOT(Test.Flag) THEN CRT "Test = False"
```

The lack of a THEN is common for other PICK/BASIC statements that incorporate the "IF...THEN...ELSE" form, such as the OPEN statement, to open a file. For the OPEN statement, an ELSE clause is required. The ELSE clause is activated if the file does not exist:

```
OPEN File.Data.Name$ TO File.Data$ ELSE
    CRT "The file " : File.Data.Name$ : " does not exist"
END
```

You can use a NULL statement to create an empty THEN:

```
OPEN "", File.Data.Name$ TO File.Data$ THEN
    NULL
END ELSE
    CRT "The file " : File.Data.Name$ : " does not exist"
END
```

or in place of the required ELSE:

```
OPEN "", File.Dat.Name$ TO File.Dat$ ELSE NULL
```

Multiline THEN, single-line ELSE:

```
IF Test.Flag THEN
    CRT "Test = True"
    GOSUB 100
END ELSE CRT "Test = False"
```

Statements that follow a conditional test on the same line will not be executed if the condition proves untrue. An entire FOR...NEXT loop can be placed on one line. However, the inclusion of a conditional clause in the loop will require ending the line, other-

wise the loop will not reach the NEXT statement when the condition is untrue. For example, the PICK/BASIC compiler refused to accept the following routine:

```
FOR I = 1 TO 10; CRT I; IF I < 6 THEN A(I) = 1; NEXT I
```

The compiler rejected the looping routine because the routine would not reach the NEXT statement after I became greater than 6. The correct method would be to place NEXT on a separate line:

```
FOR I = 1 TO 10; CRT I; IF I < 6 THEN A(I) = 1  
NEXT I
```

An alternative to avoid breaking the line would be to use a Boolean statement for the assignment:

```
FOR I = 1 TO 10; CRT I; A(I) = (I < 6); NEXT I
```

Note that a Boolean true in PICK/BASIC equals 1. (See the section: "Boolean Flags.")

The following short TEST.IF.THEN.ELSE program will climb back up a ladder of ELSE clauses to a previous IF.

TEST.IF.THEN.ELSE Program

```
* TEST.IF.THEN.ELSE Copyright 11/20/1987 David L. Clark -- TAB BOOKS Inc.  
* Test IF...THEN...ELSE  
* Name under IBM PC DOS: TifElse.bp  
PROMPT ""  
CRT "Test #1 (Enter 1 or 0):" :  
INPUT Test.Flag.1  
IF Test.Flag.1 THEN  
  CRT "Test #2 (Enter 1 or 0):" :  
  INPUT Test.Flag.2  
  IF Test.Flag.2 THEN  
    CRT "Test #1 = True, Test #2 = True"  
  END ELSE  
    CRT "Test #1 = True, Test #2 = False"  
  END; * If Test.Flag.2  
END ELSE  
  CRT "Test #1 = False"  
END; * If Test.Flag.1  
END
```

PICK/BASIC will always evaluate all of the conditions in an IF statement. For example:

```
IF Test.Flag.1 AND Test.Flag.2 AND Test.Flag.3 THEN ...
```

The program will evaluate all three flags, rather than stopping if Test.Flag.1 is false. If Test.Flag.1 were false 90 percent of the time, the program would be doing a great deal of unnecessary work. The statement might be rewritten as:

```
IF Test.Flag.1 THEN IF Test.Flag.2 THEN IF Test.Flag.3 THEN ...
```

Place the test that would most often avoid execution of the remainder of the statement first. If two tests are approximately equal in their likelihood, place the test that takes the least time first. Boolean True and False tests, as those shown in the examples, take the least time. Comparisons of strings generally take more time than comparisons of numbers.

Sequential testing must be used to filter out conditions that could cause an error, such as division by zero:

```
IF U > 0 THEN IF V/U > W THEN ...
```

CASE

To make a series of tests, generally on one variable:

```
EQUATE True TO 1
BEGIN CASE; * Char$
    CASE Char$ = > "A" AND Char$ <= "Z"
        CRT "Upper case letter"
    CASE Char$ = > "a" AND Char$ <= "z"
        CRT "Lower case letter"
    CASE Char$ = > "0" and Char$ <= "9"
        CRT "Digit"
    CASE Char$ = > CHAR(0) AND Char$ <= CHAR(31)
        CRT "Control character"
    CASE True
        CRT "Punctuation, math or special symbol"
END CASE; * Char$
```

The final condition "CASE True" is equivalent to "otherwise."

Although CASE is generally used for testing a single variable, Pick allows any conditional tests in a CASE expression:

```
BEGIN CASE; * Number, Test.Flag
    CASE Number = 1 AND Test.Flag
        PRINT "Number = 1, Test = True"
    CASE Number = 1 AND NOT(Test.Flag)
        PRINT "Number = 1, Test = False"
    CASE Number = 2 AND Test.Flag
        PRINT "Number = 2, Test = True"
    CASE True
        PRINT "No valid choices found"
END CASE; * Number, Test.Flag
```

CASE statements can be nested, like IF clauses:

```
CASE Number.1 = 1
  BEGIN CASE; * Number.2
    CASE Number.2 = 1
      PRINT "Number 1 = 1, Number 2 = 1"
    CASE Number.2 = 2
      PRINT "Number 1 = 1, Number 2 = 2"
  END CASE; * Number.2
```

Semicolons, rather than separate lines can be used for the CASE structure; thus any of the three following configurations will work:

```
BEGIN CASE
  CASE N = 1
    GOSUB 100
  CASE N = 2
    GOSUB 200
END CASE
```

```
BEGIN CASE
  CASE N = 1; GOSUB 100
  CASE N = 2; GOSUB 200
END CASE
```

```
BEGIN CASE; CASE N = 1; GOSUB 100; CASE N = 2; GOSUB 200; END CASE
```

ON...GOSUB, ON...GOTO

For example:

```
ON Numeric.Expression GOSUB Label, Label, ...
ON Numeric.Expression GOTO  Label, Label, ...
```

PICK/BASIC evaluates the numeric expression and truncates to an integer. The program branches to the label whose position in the label list corresponds to that integer. The numeric expression is a variable, but the labels must be literals. For example:

```
ON Number GOSUB 100, 200, 300; RETURN
```

If Number = 1, the program will branch to line 100, if Number = 2, to line 200, and if 3, then to line 300. After branching, the program continues with the next statement, which can be on the same line.

If Number is less than 1, or greater than the number of labels on the list, the program will continue with the next statement, which can be on the same line.

Note that unlike the "IF...THEN...ELSE" construction, the flow of the line is not broken. Thus in the example, no matter what the value of Number, the RETURN statement will be executed.

The order of the labels determines which branch is taken, not their value. Labels can also be repeated. Thus the following is legal:

```
ON Number GOSUB 700, 300, 100, 700, 200
```

If Number = 1, the program will branch to label 700, if Number = 2, to label 300, and so on.

Any expression that evaluates to a number can be employed as the control variable. For example, the INDEX statement could be used to determine the position of a character or substring within a string and branch according to that position. In the following example, INDEX and ON...GOSUB are used with a table of control characters. The program branches to an appropriate editing routine according to which key has been pressed:

```
INDEX(Table.Ctrl.Chars$, Key$, 1)  
ON GOSUB 100, 200, 300 ...
```

(For the remainder of the example, see the section: “Keyboard: Control Characters for Editing.”)

Constants



CONSTANTS ARE USED TO SET CONSTANT VALUES THAT WILL NOT CHANGE DURING THE program.

IBM PC BASIC

IBM PC BASIC makes no distinction between variables and constants.

PICK/BASIC

EQUATE

For example:

```
EQUATE String.Constant$ TO "Literal value"  
EQUATE Number.Constant TO 10  
EQUATE Char.Constant TO CHAR(255)  
EQUATE Array.Equivalent$ TO Array$(1)
```

EQUATE can be abbreviated to EQU, and can be used for a series of assignment statements, each separated by a comma. The CHAR function, converting a number to an ASCII character, is the only function allowed in an EQUATE statement. Place the EQUATE statements together in a constants section at the beginning of the program.

* Assign Constants *

EQUATE Error.Msg\$ TO "Error: File Not Found"

EQU Left.Margin TO 5, Attrib.Mark\$ TO CHAR (254)

EQUATE will not accept a negative number. For example, when converting a file from IBM PC BASIC, you might have to adjust for IBM PC BASIC's use of a negative - 1 as Boolean True, while Pick uses a positive 1. You could not create a negative constant such as:

EQU IBM.PC.Basic.True TO - 1

Instead you would have to use a variable:

IBM.PC.Basic.True = - 1

Constants created with EQUATE cannot always be used where numeric literals are required, such as to dimension an array or to create a label to identify a subroutine. The following sequence is illegal:

EQU Array.Max TO 100

DIM Array\$(Array.Max)

A numeric literal is required:

DIM Array\$(100)

You can also EQUATE a simple variable to an element in a fixed array, to produce more recognizable variable names from an array:

EQU Name\$ TO Array\$(1), Address\$ TO Array\$(2)

This use of EQUATE with an array does not produce constants, but rather variables with different names but holding the same values. Changing the value of one will automatically change the value of the other. (See the section: "Data Structures: Arrays, Fixed: Dimension or Declare.")

Conversions and Correlatives in Pick



PICK'S CONVERSIONS AND CORRELATIVES HAVE NO PARALLEL IN THE OTHER LANGUAGES and systems examined. The Pick operating system, built for the purpose of database management, allows you to define an attribute (field) in terms of other attributes or calculations, which can use values in the same file or in other files. The definitions are contained in the attribute definition item in the dictionary level file, which points to a corresponding data level file. Within the attribute definition item, attribute 7 contains conversion codes, which format the data for output, and attribute 8 contains correlative codes, which put the data into an intermediate format for searching and processing purposes. (For more details on the attribute definition item, see the beginning of the section: "Files, Items.")

Attribute 8, the correlative attribute, creates derived values. This is data that does not exist as a stored value, but is created by applying calculations and concatenations on the data stored in one or more files. Storing only one piece of data, and then deriving different forms for different purposes, not only saves on storage space, it preserves the integrity of the data. Integrity refers to consistency. If the same piece of data is stored multiple times, inconsistencies inevitably will arise where a person's name is spelled in different ways in different places because of mistakes in input or failure to change the data in all its different storage locations when it is learned that the original information was incorrect.

To cite a simple example, conversions and correlatives can be used to allow the user to enter a state designation as California, CALIFORNIA, or CA. The data will be stored in the most economical manner, CA. However, the information will be displayed as California. The user can search for the state as California, CALIFORNIA, or CA. Furthermore, when the date is entered, the system will check a list of state codes contained in

another file, and reject the entry if it does not correspond to an item on the list. All of this can be accomplished without BASIC programming through conversion and correlative codes in the attribute definition item.

The sequence of operation of the conversion and correlatives on an Access statement is shown below. For the attribute Entry.Date the attribute definition item contains the following conversion:

```
Item ID: Entry.Date
07   D/           { External format: MM/DD/YYYY }
```

- 1) The Access statement is entered:

```
SSELECT History.Data WITH Entry.Date < "08/09/1987"
```

- 2) In the case of dates and masked decimal conversions, the conversion code in attribute 7 converts the element in quotes from external to internal format. Thus "08/09/1987" is converted to an integer.
- 3) Any correlative code in attribute 8 is applied.
- 4) The Access processor performs searching and sorting.
- 5) The conversion code is applied to display the data in external format.

If no conversion code was applied, Access would interpret the material between the double quote marks as a literal.

ICONV, OCONV

For example:

```
Data.Internal$   = ICONV(Data.External$, Correlative.Code$)
Data.External$   = OCONV(Data.Internal$, Conversion.Code$)
```

Conversion and correlative codes also can be used in PICK/BASIC. The ICONV function can be used to convert data from external to internal format, and the OCONV function to convert data from internal to external format.

For example, Pick stores the time as the number of seconds past midnight, but can output the time for display in a variety of formats. The code "MT" (Mask Time) can be used as both a conversion and a correlative code, to translate between internal and external format.

For input, an ICONV statement is used to inform the system that a time is to be stored:

```
CRT "Enter the time of the event:"
INPUT Time.External$
Time.Internal = ICONV(Time.External$, "MT")
```

For output, an OCONV statement is used to display the time in a readable format:

```
CRT "The time of the event was: " : OCONV(Time.Internal, "MT")
```


On output, the time will be displayed in 24-hour military format. Other output conversion codes can be employed to use AM and PM, to limit the display to hours, or to add display of the seconds.

Conversion codes can be placed in variables or in constants created with the EQUATE statement. For example, to convert an integer to a dollar amount with two decimal places, a comma for each thousand, and a dollar sign:

```
EQUATE Dollar.Code$ TO "MR2,$"  
Dollar.Amount = OCONV(Amount, Dollar.Code$)
```

or:

```
Dollar.Code$ = "MR2,$"  
Dollar.Amount = OCONV(Amount, Dollar.Code$)
```

Leading and trailing spaces do not affect conversion. For example:

```
ICONV(" 07/04/1776 ", "D");    *Will convert correctly as date
```

Under the new Open Architecture version of Pick, as well as the Pick-like program Revelation, PICK/BASIC subroutines can be called from the attribute definition items to perform conversions, thus eliminating the need for conversion codes for complex operations.

CONVERSIONS AND CORRELATIVES IN PICK: ARITHMETIC

The A codes for correlatives and conversions are the most universal in application. They can be called on to perform calculations and to combine the results of the other types of conversions as the following shows:

```
A{ Precision }; { Attrib.Num } { Operator } { Attrib.Num }
```

The default precision is to handle all numbers as integers. Numbers after the semi-colon reference attributes, thus:

```
A;3*4
```

will multiply attributes 3 and 4.

A literal number should be placed in quotes, thus:

```
A;"3"*4
```

will multiply attribute 4 by the number 3.

Name Operator

An attribute can be referred to by name, by placing the name in parentheses preceded by the letter N. Thus, if attribute 3 is Price and attribute 4 is Quantity, you can multiply them in this manner:

A;N(Price)*N(Quantity)

Any correlative codes contained in the attribute definition items of price and quantity are applied when those attributes are referred to by name, but not when they are referred to by number.

Repetition Operator

A single-valued attribute can be multiplied repeatedly against a multivalued attribute. Thus, if Quantity is multivalued, for quantities of a given piece of merchandise held in different warehouses, a summation of the total cost is achieved by adding an R for repeat:

A;N(Quantity)*N(Price)R

Date and Time Operators

The letter “D” will retrieve the system date, in internal format, and the letter “T” will retrieve the system time, in internal format.

Substring Operator

The A correlative uses brackets for substring extraction in the same way as PICK/BASIC, with the form:

[Pos.Start, Num.Chars]

Derived Attributes

The power of the A correlative lies in its ability to combine the operators mentioned previously with the actions of other correlatives, such as the date correlative, and with “IF..THEN” logic. The following examples consist of attribute definition items for attributes that are computed, rather than used to store data. Attributes that hold no data of their own are referred to as derived or virtual attributes. Derived attributes will return information such as the difference in days between today’s date and the date when a piece of information was entered. The attribute presents information that is derived on the basis of a comparison of today’s date, which is supplied by the date correlative, and the date information stored in another attribute.

When attributes are derived, rather than used to store data, the attribute count (stored in attribute 2 of the attribute definition item) is a mythical number because it does not refer to data. For this number, always use zero rather than a number such as 99. If the number 99 is used, the Access processor will spend time to look through each item for the non-existent attribute number 99.

Example for Date Comparison

As an example of the use of a derived attribute for date comparison, it will be assumed that the database includes an attribute called "Entry-Date" for the date when information is entered. (A PICK/BASIC data entry program fills this attribute automatically, using the PICK/BASIC DATE () function, rather than requiring the user to type the information.) It is also assumed here that the Entry-Date attribute is number 12 in the database.

The following Entry-Date attribute definition item will retrieve the date of entry for printing or display in external format, in the form "MM/DD/YYYY". The Entry-Date-Internal attribute definition item will retrieve the date in Pick internal system format, which is the number of days before or after December 31, 1967. Note the conversion code "D/" in attribute 7 for Entry-Date, and the lack of any conversion or correlative code for Entry-Date-Internal.

Item ID: Entry-Date

01 A
02 12
07 D/
09 R
10 10

Item ID: Entry-Date-Internal

01 A
02 12
09 R
10 10

The attribute Entry-Date-Internal will be used extensively in the derived attributes that follow to retrieve entry date in internal format. It is preferable to use an attribute name in this fashion, rather than the number of the data attribute, because if the data attribute number is changed, then only Entry-Date-Internal need be changed. The derived attributes that use Entry-Date-Internal will continue to function correctly. In addition, those derived attributes can be copied into other data dictionaries for use with other files, or kept in the master dictionary for use with many different files. Only the data attribute number for Entry-Date-Internal must be set individually for each data dictionary.

The first derived attribute returns the number of days that have elapsed since a data item was entered. Note the use of zero as the attribute number for a derived attribute. The computation appears in the correlative attribute, which is number 8. The operator "D" retrieves the current system date, from which is subtracted the date of entry in internal format, producing the numbers of days since the entry was made:

Item ID: Days-Since-Entry

01 A
02 0
08 A;D-N(Entry-Date-Internal)
09 R
10 10

To retrieve all of the items entered within the last seven days, issue the following command at TCL level:

```
LIST History.Data WITH Days.Since.Entry < "7"
```

You can embed “IF...THEN” logic in an A correlative, returning a string or a null depending upon whether a condition has been met. Thus the following derived attributes combine Boolean operators with string extraction to return either “True” or null depending upon whether the data was entered within the last seven days, yesterday, or today:

Item ID: Entered-Last-7-Days

```
01 A
02 0
08 A;"True" [ "1", "4"*(N(Days-Since-Entry)<"7") ]
09 L
10 4
```

Item ID: Entered-Yesterday

```
01 A
02 0
08 A;"True" [ "1", "4"*(N(Days-Since-Entry) = "1") ]
09 L
10 4
```

Item ID: Entered-Today

```
01 A
02 0
08 A;"True" [ "1", "4"*( D = N(Entry-Date-Internal) ) ]
09 L
10 4
```

You can enter search statements in the following form:

```
LIST History.Data WITH Entered-Last-7-Days
LIST History.Data WITH Entered-Yesterday
LIST History.Data WITH Entered-Today
```

To retrieve items older than seven days:

```
LIST History.Data WITHOUT Entered-Last-7-Days
```

A Boolean comparison returns a numerical result of either 1 or 0. Thus the constant “4” was multiplied by the Boolean operation “(D = N(Entry-Date-Internal))” to return either 4 or 0. A string extraction of 0 will yield a null value. The string extraction was necessary to turn a 0 into a null, to allow a simple WITH or WITHOUT search statement.

(TCL search commands, on IBM PC/XT R83 2.0, do not treat a 0 return as null. For more information on Boolean logic see the section: "Boolean Flags.")

Calculating whether an entry was made this week or last is more complex than determining if the entry occurred in the last seven days. The first step is to create attribute definition items to return the day of the week, in numerical form, for the current date and for the date of entry.

The Pick "DW" date conversion code for the day of the week treats Sunday as the end of the week, rather than the beginning, numbering the days from 1 to 7 for Monday through Sunday. The following examples will change Sunday from 7 to 0, treating a week in the normal sense of Sunday through Saturday, numbered 0 through 6.

Item ID: Entry-Weekday

```
01 A
02 0
08 A;N(Entry-Date-Internal)(DW)*(N(Entry-Date-Internal)(DW)#"7")
09 R
10 10
```

Item ID: Today-Weekday

```
01 A
02 0
08 A;D(DW)*(D(DW)#"7")
09 R
10 10
```

The following two attribute definition items will use the "Today-Weekday" attribute to return the internal system date for the beginning day of the current week, and the beginning day of last week:

Item ID: Begin-This-Week

```
01 A
02 0
08 A;D-N(Today-Weekday)
09 R
10 10
```

Item ID: Begin-Last-Week

```
01 A
02 0
08 A;N(Begin-This-Week)-"7"
09 R
10 10
```

Finally, the attributes Entered-This-Week and Entered-Last-Week will be created to return "True" or null:

Item ID: Entered-This-Week

```
01 A
02 0
08 A;"True" ["1","4"*(N(Entry-Date-Internal) >= N(Begin-This-Week))
09 L
10 4
```

Item ID: Entered-Last-Week

```
01 A
02 0
08 A;"True" ["1","4"*(N(Entry-Date-Internal) < N(Begin-This-Week))
    *(N(Entry-Date-Internal) >= N(Begin-Last-Week))
09 L
10 4
```

You can now enter search statements of the form:

LIST History.Data WITH Entered-This-Week

A similar combination of attribute definitions will reveal whether the data was entered in the current or last month or year, using the Pick “DM” date conversion to retrieve the month in numerical form, the “DD” conversion to retrieve the day of the month, and the “DY” conversion to retrieve the year as a four-digit number. The following Begin-Last-Month conversion finds the internal system date for the beginning of the previous month by subtracting the number of days in the previous month from the beginning date for the current month.

Item ID: Begin-This-Month

```
01 A
02 0
08 A;D-D(DD)
09 R
10 10
```

Item ID: Begin-Last-Month

```
01 A
02 0
08 A;N(Begin-This-Month)-(N(Begin.This-Month)-"1") (DD)
09 R
10 10
```

Item ID: Entry-Year

```
01 A
02 0
08 A;N(Entry-Date-Internal) (DY)
09 R
10 10
```

Item ID: Entered-This-Month

```
01 A
02 0
08 A;"True" ["1","4"*(N(Entry-Date-Internal) >= N(Begin-This-Month))]]
09 L
10 4
```

Item ID: Entered-Last-Month

```
01 A
02 0
08 A;"True" ["1","4"*(N(Entry-Date-Internal) < N(Begin-This-Month))
      *(N(Entry-Date-Internal) >= N(Begin-Last-Month))]]
09 L
10 4
```

Item ID: Entered-This-Year

```
01 A
02 0
08 A;"True" ["1","4"*(N(Entry-Year) = D(DY))]
09 L
10 4
```

Item ID: Entered-Last-Year

```
01 A
02 0
08 A;"True" ["1","4"*(D(DY)-N(Entry-Year) = "1")]
09 L
10 4
```

The derived attributes can be used for a variety of purposes beyond those shown here. For example, to return data entered in 1985:

```
LIST History.Data WITH Entry-Year = "1985"
```

Similar derived attributes can be created containing correlative codes such as the following:

To return an amount other than 0 only for invoices more than 90 days old:

```
A;N(Amount)*((D-N(Invoice.Date)) < "91")
```

To show the name of the author only if the category of the work is "Published":

```
A;N(Author.Name)["1","999"*(N(Category) = "Published")]
```

The correlative would perform a string extraction of either 0 or up to 999 characters, depending on whether the entry in the field named Category equaled "Published."

Comparison between A (Arithmetic) and F (Function) Correlatives

A correlatives are converted by the system to F correlatives. However, no significant amount of time is lost in the process because the correlative is precompiled when the Access processor parses the statement.

An A correlative can be used to perform anything that an F can do. The A is much easier to understand than the F. The A allows calculations to be stated in normal order, rather than the reverse Polish notation required with the F. It is also a substantial advantage that, with the A correlative, one can refer to an attribute by name. The ability to use attribute names allows you to combine correlatives without repeating all of the calculations in the single correlative entry. With an F correlative the entire operation must be contained in one attribute.

You would expect to eliminate use of the F correlative entirely. However, there is a catch. The Pick System has a limit of 11 stack return entries. A correlatives use the stack. Therefore if the "Return Stack Full" abort occurs, you must change some of the A correlatives to F.

The undocumented Access "Y" option will assist in the change. If one appends a "Y" option to an Access statement that acts on an attribute with an A correlative, the F correlative into which the A translates will appear at the top of the screen. Because the display will be immediately replaced by the listing display, you should use the "P" option to send output to the printer to keep the "Y" option result on the screen. For example:

```
LIST File.Name Attrib.Name (YP)
```

You then type the F correlative into the attribute definition item in place of the A.

CONVERSIONS AND CORRELATIVES IN PICK: DATE

Pick stores a date internally as a number—the number of days in relation to December 31, 1967. Dates before that date are kept as negative numbers. The numeric form allows arithmetic to be performed on the date in order to respond to search requests, such as the retrieval of newspaper articles appearing within seven days after the date stored in a particular attribute. You also can limit a search request to a range of months, quarters, or days of the year. A search can specify the day of the week, without requiring that you make a separate entry of information in that form.

Always use a four-digit year. If the year is shortened to two digits, different systems will interpret the figure as twentieth or twenty-first century. Pick interprets any year from "29" as twenty-first century, thus 11/21/29 becomes November 21, 2029.

Pick offers a variety of conversion codes to display or interpret the date in different formats, or to extract only a part, such as the year, quarter, month, or day. The conversion code format, beginning with "D" for date, follows:

```
D{ Digits.Year } { Input.Delim and Fields.to.Skip } { Output.Delim or Code }
```

The number of digits shown for the year can be 0 through 9; the default is 4.

An input delimiter and number of fields to skip can be specified for group extraction, when the date (in internal format) is to be pulled out of a string that contains other elements.

The output delimiter is the character that is shown between the numeric forms of month, day, and year. Rather than an output delimiter, an additional subcode can be specified from the following list:

D	Day of month
I	Internal: Show the internal format.
J	Julian day of year: Numeric
M	Month: Numeric
MA	Month: Alphabetic
Q	Quarter: Numeric
W	Weekday: Numeric, begins with Monday as 1
WA	Weekday: Alphabetic
Y	Year

The following codes represent the date July 4, 1776, in the formats shown:

D	04 JUL 1776	
D/	07/04/76	
D2/	07/04/1776	
D-	07-04-1776	
D*	07*04*1776	
DM	7	Month: Numeric
DMA	JULY	Month: Alphabetic
DQ	3	Quarter
DY	1776	Year only
DW	2	Weekday: Numeric
DWA	TUESDAY	Weekday: Alphabetic
I	-69942	Internal format: Number of days before 12/31/1967

Note the use of “D2/” to display a two-digit year, with month, day, and year separated by a slash. The code “D/2” has a completely different effect. Pick expects to find a string already delimited by slashes, with a date, in internal format, to extract from the third element (after the second delimiter). From the string:

Thomas Jefferson/Declaration of Independence/-69942

the code “D/2” returns:

Thomas Jefferson/Declaration of Independence/04 JUL 1776

the code “D2/” returns:

Thomas Jefferson/Declaration of Independence/07/04/1776

Such a date code could be used when a date was assigned as part of an item ID, for quick sorting or retrieval by date of entry. An external version of the date portion of the

item ID makes it easier for the user to understand. The following attribute definition item converts the second element in an item ID from an internal to an external date. The elements in the Item ID are divided by asterisks:

```
Item ID: Item.ID.External
01  A
02  00
07  D*2/
```

The conversion would change this:

```
Test*Code*7321
```

to this:

```
Test*Code*01/16/1988
```

When entering a date, use an input conversion to inform the system that a date has been entered, rather than just a string. The input conversion code for a date is "D," as in this example:

```
CRT "Enter the date of the document:"
INPUT Date.External$
Date.Doc = ICONV(Date.External$, "D")
```

In this example, Date.Doc is a number.

If no year is specified on input, the current year of the system date is assumed. If only two digits are entered for the year, 30–99 will be considered as twentieth century, and 00–29 will be interpreted as twenty-first century.

The following TEST.DATE program will input a date and then display it in the format specified. The internal date is considered a number, and the external date is considered a string. The material shown between brackets is the default value that will be used if ENTER is pressed alone.

Begin by entering the date in the format MM/DD/YYYY, and then try different input formats, to make sure that each will be interpreted in the manner intended. Pick accepted all of the following as valid entries for July 4, 1776:

```
07/04/1776
7/4/1776
JULY 4, 1776
July 4, 1776
july 4, 1776
JULY 4 1776
July 4 1776
july 4 1776
4 JULY 1776
4 July 1776
4 july 1776
4 JUL 1776
4 Jul 1776
4 jul 1776
```

When the year 1776 is abbreviated to 76, Pick interprets the year as 1976.

TEST.DATE Program

```
* TEST.DATE Copyright 07/12/1987 David L. Clark -- TAB BOOKS Inc.
* Demonstrate date conversion codes.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: AscChar.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  PROMPT ""
  EQU True TO 1, False TO 0
  Quit.Flag = False
  * Start with July 4, 1776 and format MM/DD/YYYY as default values
  Date.Entry$ = "07/04/1776"
  Code$ = "D/"
  LOOP UNTIL Quit.Flag DO
    CRT
    CRT "Enter date with:  1.External format  2.Internal format  " :
    CRT "Pick a number [1]:" :
    INPUT Num.Choice
    Ext.Format.Flag = (Num.Choice # 2)
    CRT "Enter code [" : Code$ : "]" :
    INPUT X$
    IF X$ # "" THEN Code$ = X$
    CRT "Enter date [" : Date.Entry$ : "]" :
    INPUT X$
    IF X$ # "" THEN Date.Entry$ = X$
    * Date.Internal is a number
    IF Ext.Format.Flag THEN
      Date.Internal = ICONV(Date.Entry$, "D")
    END ELSE
      Date.Internal = Date.Entry$
    END
    * Date.External$ is a string
    Date.External$ = OCONV(Date.Internal, Code$)
    CRT "Internal date = " : Date.Internal
    CRT "External date = " : Date.External$
    CRT
    CRT "1.Continue  2.Quit  Pick a number [1]:" :
    INPUT Num.Choice
    Quit.Flag = (Num.Choice = 2)
  REPEAT; * Until Quit.Flag
END
```

Pick has no automatic adjustment for the difference between the earlier Julian calendar and the present Gregorian calendar. Pick assumes the modern Gregorian usage. The adjustment has to be made by a PICK/BASIC program or by a conversion or correlative in an attribute definition item. The adjustment has to vary for the purpose intended, because the revised calendar was adopted in different places at different times.

The Gregorian calendar suppressed 10 days where it was adopted by 1700, or 11 days thereafter, and specified that no year divisible by 100 could be a leap year unless the year was also divisible by 400.

Pick does not have a facility for displaying years as B.C. Pick accepts "JANUARY 1, 0000" as a valid entry for the first day of the present era. But Pick ignores attempts to use a minus sign to indicate a B.C. date. Thus, for the Ides of March of which Julius Caesar was warned, the entries "MARCH 15, -0044" and "03/15/-0044" are interpreted as March 15, 44 A.D. However, when given a correct internal format date, Pick's output conversion works properly. Thus, when using the test date program, the date in internal format of "-734808" produces the output "03/15/-044." To retain the display of the year in four digits, allowing for the minus sign, the conversion code "D5/" produces "03/15/-0044." But the "D5/" specification also leaves a leading zero before A.D. dates. The accuracy of ancient dates might be impaired, however, by differences between the Julian and Gregorian calendars. Programming is necessary to interpret the date on entry to store the correct internal number.

Pick also does not accept on entry a year beyond "9999." The year is truncated on input. Apparently, in Pick cosmology the universe is to end in another 8,000 years. However, as with B.C. dates, if the correct internal format date is stored, the output conversion works properly. Thus, the internal date "2933689" produces the output "01/01/10000," for January 1 of the year 10,000. The internal date "9999999" displays as "07/03/29346," and "-9999999" displays as "06/29/-25411." Note that the default specification of four digits for the year, with the conversion code "D/," did not truncate the output, which included five digits and the minus sign. Leading zeros for the year have been cut.

The examples in this book assume the use of the American style of displaying a date numerically in the format "MM/DD/YYYY." The European style places the day before the month, in the format "DD/MM/YYYY." The command SET-DATE-EUR sets the system to the European style, while the command SET-DATE-STD designates the American style as standard usage. (See the section "System: Access System Information: Date.")

The date conversion also can be used as a print format mask. For example, to display today's date in external format MM/DD/YYYY:

```
CRT DATE() "D/"
```

Note that no operator appears between the value to be formatted and the format mask.

CONVERSIONS AND CORRELATIVES IN PICK: GROUP EXTRACT

The group extract correlative or conversion extracts elements between delimiters. The code has the following format (the spaces shown here are added, and should not appear in the actual code):

```
G { Elements.to.Skip } Delimiter Elements.to.Take
```

For example, an item ID might be composed of several elements, such as a Collection#, Part#, Sub-Part#, Entry#, each separated by an asterisk:

```
1*13*2*1234
```

To reference the item ID, place a zero in the attribute for attribute mark count of the attribute definition item. To extract the individual elements of the item ID, place the following codes in the correlative attribute:

For Collection#:	G*1
For Part#:	G1*1
For Sub-Part#:	G2*1
For Entry#:	G3*1
For Part# and Sub-Part#:	G1*2

CONVERSIONS AND CORRELATIVES IN PICK: PATTERN MATCHING

The following is the format of the pattern matching correlative:

P(Num.Chars Match.Type)

A blank is returned if there is no match. There is no space between the parameters.
Match types:

- A Alphabetic
- N Numeric
- X Alphanumeric

To require a match of nine numbers:

P(9N)

Multiple match choices can each be enclosed in parentheses and separated by semicolons. To allow either nine numbers or one alphabetic character followed by eight numbers:

P(9N);(1A8N)

CONVERSIONS AND CORRELATIVES IN PICK: TEXT EXTRACT

The following is the format of the text extract correlative:

T Start.Pos,Num.Chars

For example, to take the second through tenth characters:

T2,10

CONVERSIONS AND CORRELATIVES IN PICK: TRANSLATE FROM FILE

The following is the format of the translate from file correlative:

TFile.Name;Code;Sort.Attrib;Display.Attrib

The following are choices for code:

```
C  If data not found, use original data
X  " " " " " , result null
```

The translate from file correlative is generally employed with a piece of data serving as an item ID for an item in another file, in order to pull information from that other file. For example, to allow the user to type in either "CA," "California," or "Calif.," to store "CA," to sort on "California," and to display "Calif.," you create a file named "State .File" with the following item:

```
Item ID: CA
01 California
02 Calif.
```

The attribute definition item:

```
Item Id: State
08 TState.File;X;1;2
```

If translating a multivalued attribute, Pick will turn value marks into spaces. Alternately, a number can be placed after the X or C code to extract only a single value, for example:

```
08 TState.File;X1;;3
```

The first value will be used.

Although the T correlative generally is employed to translate from a different file, it also can be used to translate an item back to itself in order to extract a specific value from an attribute. For example, to extract the value for July from the second attribute in an item in File.Data, when that attribute holds one value for each month in sequence, you would create an attribute definition item such as:

```
Item ID: July
01 A
02 0
08 TFile.Data;X7;2;2
```

The result is the retrieval of the seventh value (because July is the seventh month) from the second attribute.

To allow the exclusion of stop words, such as "The" from a book title:

```
08 F;3(G 1);(TExclude.File;C;;1)
```

Note the space after G in the group extract correlative. The C code in the translate file correlative means that if "The" is not found, use the original. Thus you could ask for "House of Usher" or "The House of Usher."

The translate file correlative is very slow, because it must reopen the file for each translation performed. Thus to list 100 translated attributes, the translate correlative must

reopen the file 100 times. Within a BASIC program, rather than using the OCONV function to translate, open the target file once and access it directly.

CONVERSIONS AND CORRELATIVES IN PICK: USER EXITS

User exits consists of the hexadecimal numbers for the entry points of assembly language routines. Because they are generally undocumented and often unsupported, they should be avoided. The following "Sort with User Exit" routine illustrates the use of User Exit U1072 to sort the attributes in an item in ascending alphabetical order. Sorting also can be accomplished with the LOCATE command, operating on a dynamic array. (See the section: "Data Structures: Pick: Arrays, Dynamic, Locate.")

Sort with User Exit Routine

```
*****
* Sort with User Exit *
*****
* David L. Clark 09/22/1988
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: SortUser.bpr
EQUATE Attrb.Mark$ TO CHAR(254)
Item$ = "Zebra" : Attrb.Mark$ : "Tiger" : Attrb.Mark$ : "Ape"
Item$ = OCONV(Item$, "U1072")
FOR I = 1 TO 3
    CRT Item$<1> : ", " :
NEXT I
```

The routine will display:

Ape, Tiger, Zebra

Data Structures



DATA STRUCTURES: ARRAYS, FIXED: DIMENSION OR DECLARE

The size of a fixed array must be declared before the array can be used. This requirement is in contrast to the use of a dynamic array, for which new elements can be created as needed. A fixed array is therefore less flexible to use. In addition, a fixed array will take up some amount of space for every element, even when all of the elements are not used.

A fixed array, however, is faster to use than a dynamic array. The optimum combination of the two types is to store a large body of data in dynamic array form, but to transfer any information that a program accesses repeatedly into a fixed array.

IBM PC BASIC

DIM

For example:

```
DIM Array.1.Dimen$ (Num.Elements), Array.2.Dimen$ (Num.Rows, Num.Cols)  
Array.1.Dimen$ (Element) = X$
```

Num.Elements, Num.Rows, and Num.Cols can be variables rather than literal numbers. The program can determine the size needed for an array while the program is running by taking information from a parameters file or from the user. The number of elements is limited only by memory. Any number of arrays, separated by commas, can be dimensioned by a single DIM statement.

An array will always begin with element number zero; thus, the statement “DIM X\$(10)” will create an array of 11 elements, X\$(0) through X\$(10). If an array variable is used without a preceding DIM statement, IBM PC BASIC automatically creates an array of 11 elements—0 through 10. Thus, if there were no preceding DIM statement for the array A\$, the statement:

```
A$(10) = X$
```

would not cause an error, but:

```
A$(11) = X$
```

would cause a “subscript out of range” error.

All of the elements in an array must be of the same type—e.g., an integer array can contain only integers. When an array is dimensioned, either automatically or through the DIM statement, the system will allocate two bytes for each integer element, four bytes for each single-precision element, eight bytes for each double-precision element, and three bytes for the pointer to each string element. To dimension an array to a size larger than necessary wastes memory.

The dimensions of an array cannot be changed, except after a CLEAR, RUN, MERGE, NEW, LOAD, CHAIN (IBM PC only), or MAXFILES (Radio Shack 100 only) statement has erased all of the variables, or the ERASE command has been used on the IBM PC (see below).

IBM PC BASIC

ERASE

For example:

```
DIM      Array.1$(10), Array.2$(10,10)
ERASE    Array.1$,   Array.2$
DIM      Array.1$(20), Array.2$(20,20)
```

The ERASE command will erase specific arrays from memory to release memory space or to change the dimensions of an array.

OPTION BASE 1

For example:

```
OPTION BASE 1           '* Used before any arrays have been dimensioned
```

Although IBM PC BASIC normally starts arrays at element zero, you can change the default procedure by giving the command OPTION BASE 1 before any arrays have been dimensioned. After this command, you can only switch back to the default procedure by erasing all of the variables with a CLEAR or equivalent statement. OPTION BASE 1 will save the memory space that would otherwise be allocated to element zero.

Neither ERASE nor OPTION BASE 1 are available in Microsoft Basic on the Radio Shack 100.

PICK/BASIC

DIM

For example:

```
DIM Array.1.Dimen$(Num.Elements)
DIM Array.2.Dimen$(Num.Rows, Num.Cols)
Array.1.Dimen$(Element) = X$
```

Num.Elements for a one-dimensional array, and Num.Rows, Num.Cols for a two-dimensional array, must be literal numbers, they cannot be variables or EQUATED constants. The size of the array is fixed when the program is compiled, not when it runs.

Several arrays can be dimensioned with a single DIM statement:

For example:

```
DIM A(10), B(10), C(10)
```

An array always will begin with element number 1; thus the statement "DIM X\$(10)" will create an array of 10 elements, X\$(1) through X\$(10).

Because types are not defined in Pick, it is possible for some elements of an array to hold numbers, while others hold strings. An array might be used in this manner with the MATREAD statement to quickly take in all the attributes of an item from a file, placing each attribute in an element of the array. However, such mixing of strings and numbers generally should be avoided. (For input from files, see the section: "Files, Items: Input.")

COMMON

For example:

```
COMMON Array.1.Dimen$(Num.Elements)
COMMON Array.2.Dimen$(Num.Rows, Num.Cols)
```

If an array is declared with the COMMON statement, so that it can be shared by both a program and an external subroutine, the COMMON statement takes the place of a DIM statement. It is an error to include both statements. (For details of the COMMON statement, see the section: "Subroutines.")

EQUATE

For example:

```
EQUATE File.Name$ TO Parameter$(1), Project$ TO Parameter$(2)
```

A simple variable is faster than an array variable, and can have a more recognizable name. Use the EQUATE statement to give both variables the same value. When two variables are EQUATED, changing the value of one will automatically change the value of the other. A literal number must be used when equating a simple variable to an array element. In the example here, the numbers "1" and "2" could not have been replaced with variables. (For details on the EQUATE statement, see the section: "Constants.")

DATA STRUCTURES: ARRAYS, FIXED: SET VALUE OF ENTIRE ARRAY

IBM PC BASIC

Array elements must be assigned individually in IBM PC BASIC:

```
'* Begin line
  For I = 1 TO Total
:     Array.1$(I) = Array.2$(I)
:     Array.3$(I) = X$
:   NEXT I
'* End line
```

PICK/BASIC

MAT

To assign the value of X\$ to every element in the array Array.1\$:

```
MAT Array.1$ = X$
```

To copy every element of Array.2\$ to Array.1\$:

```
MAT Array.1$ = MAT Array.2$
```

To copy one array to another both arrays must have the same total number of elements. One array can be single-dimensioned and the other double-dimensioned, as long as their totals match. The double-dimensioned array will be assigned by row, then column.

DATA STRUCTURES: ENUMERATED AND SET

PICK/BASIC

A partial equivalent to the Pascal ENUMERATED type and SET type can be created with PICK/BASIC (or IBM PC BASIC) by concatenating the elements into a string, each separated by a delimiter. For example, the SHOW.DAY.IN.FRENCH program will obtain the name for the day of the week in French, rather than the English name that would be derived with the "DWA" conversion.

SHOW.DAY.IN.FRENCH Program

```
* SHOW.DAY.IN.FRENCH David L. Clark 09/04/1988
* Test Enumerated type: Show Name of day of the week in French
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ShwDayFr.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  * Assign Table
  Days$ = "/Dimanche/Lundi/Mardi/Mercredi/Jeudi/Vendredi/Samedi/"
```

```

EQU False TO 0, True TO 1
* Return day of the week number
Date.Internal = DATE()
Day.Number = MOD(Date.Internal, 7)
* Return day name in French
* Add 2 to number: 1 to avoid zero, 1 for FIELD
French.Name.of.Day$ = FIELD(Days$, "/", Day.Number + 2)
CRT "The day of the week is: " : French.Name.of.Day$
* Go in the opposite direction, to confirm a value in a Table
Occur.Num = 1
IF INDEX(Days$, French.Name.of.Day$, Occur.Num) THEN
    Value.Found.Flag = True
END ELSE
    Value.Found.Flag = False
END
END; * End program

```

The purpose of the “/” delimiter is to separate the elements and prevent a false selection, which might occur if one element ran into the next. For example, if a string contained “aabbcc” a positive match would result not only on “aa”, “bb”, and “cc” but also on “ab” and “bc”. If it is known that the element to be compared to the table contains only a single character, such as in the case of a control character entered at the keyboard, then the delimiters are not necessary. (For an example of how to create a table of control characters, see the section: “Keyboard: Control Characters for Editing.”)

In PICK/BASIC, a dynamic array also could be used for an equivalent operation, with searching through the LOCATE statement.

DATA STRUCTURES: PICK: ARRAYS, DYNAMIC

PICK/BASIC

Dynamic arrays differ from fixed arrays in that when using a dynamic array, new elements can be created as needed. In contrast, the total size of a fixed array must be declared before the array can be used. The dynamic array’s ability to expand makes it the logical structure to hold a database.

Pick stores the data in a database in items, which are in turn stored in files. Pick items are dynamic arrays. The normal data processing procedure in Pick is to read an item in from a file, make changes to the data, and then write the changed item back to the file. (The methods used to handle or access the item while it is still in a file are treated in the section: “Files, Items.”)

In Pick reference manuals, the same data is usually referred to as an item while it is in a file, and as a dynamic array while the individual pieces of information are being manipulated by a program. A file item contains additional information related to file storage, such as the item’s length and its item ID.

In Pick there is no requirement that the elements of either a fixed or a dynamic array be of a uniform type. Some elements can contain strings, and others numbers.

An entire dynamic array, however, is really one large string, within which elements are separated by delimiters. The section on “Strings” demonstrates how one might first find the positions of delimiters in a string, and then extract the characters that lay before,

after, or between the specified delimiters. All of those procedures are relevant for, and can be used with, dynamic arrays.

When a single element is extracted from a dynamic array, Pick will consider that element to be a string. A program can convert the element to numeric storage by assigning it to a single variable and adding zero. (See the section: “Numbers: Convert: String to Number.”)

A Pick item or dynamic array has the added feature that certain characters have been established as delimiters with specific meanings, and that special operations have been provided to manipulate dynamic arrays, based upon those delimiting characters. A Pick item or dynamic array can be divided into attributes, each of which can in turn be split into values, each of which can be further divided into subvalues. The delimiting character that marks the end of an attribute, generally referred to as the attribute mark, is ASCII 254. The value mark is ASCII 253. The subvalue mark is ASCII 252.

The delimiter appears only at the end of the element that it separates. In a file, the entire item is set off by a segment mark, ASCII 255. The program does not need to provide the segment mark, and indeed the program should not write a segment mark to a file. None of the characters ASCII 252–255 should be allowed as data entries, because Pick will read them as delimiters.

To facilitate the handling of dynamic arrays, a Pick program generally will contain in its assign constants section the following statements:

```
EQUATE Attrib.Mark$ TO CHAR(254)
EQUATE Value.Mark$ TO CHAR(253)
EQUATE SubValue.Mark$ TO CHAR(252)
```

(For the use of EQUATE, see the section: “Constants.”)

You can display an item in the form in which the item exists in a file with the following commands at Pick Terminal Control Language (TCL) level. Note that the parameters in the commands are literals, not variables. The first command obtains the Frame.Number, which is used in the second command to DUMP the item:

```
ITEM Filename Item.ID
DUMP Frame.Number
```

When the item is displayed, the attribute mark will appear as a caret: “^”, the value mark will appear as a right bracket: “]”, and the subvalue mark will appear as a backslash: “\”. If the item is brought into the Pick Editor with the “EDIT Filename Item.ID” command, each attribute mark will be treated as a Carriage Return—Line Feed, causing each attribute to begin a new line. (To enter the system delimiters from the Editor, see the section: “Editor.”)

The following commands in PICK/BASIC will manipulate parts of a dynamic array. Some commands include both an old and a new version, the new version employing the angle brackets “<” and “>” in a manner similar to the use of parentheses with a fixed array, to indicate the elements of the array. Thus “Dyn.Ay\$<4,3,2>” indicates the fourth attribute, third value, second subvalue of the dynamic array Dyn.Ay.

DATA STRUCTURES: PICK: ARRAYS, DYNAMIC: COUNT ELEMENTS

DCOUNT

Use DCOUNT to count the number of elements in a dynamic array. For example:

```
Total.Elements = DCOUNT(Search$, Delimiter$)
```

The string function DCOUNT is often used with dynamic arrays. DCOUNT will determine the total number of elements in a string that lie between the specified delimiters. If the string to search is null, DCOUNT will return zero. If the delimiter is not found, but the string is not null, DCOUNT will return 1, considering the entire string as one element. DCOUNT will count as separate elements the material before the first delimiter and after the last delimiter. If the string begins with a delimiter, DCOUNT will count a null element before the delimiter. Similarly, if the string ends with a delimiter, DCOUNT will count a null element after that delimiter. (See the section: "Strings: Extract.")

When used with dynamic arrays, an attribute mark, value mark, or subvalue mark would be indicated as the delimiter. (These delimiters would only begin or end a dynamic array if a null element were indicated.) Thus to find the number of attributes in the dynamic array Dyn.Ay\$:

```
EQUATE Attrib.Mark$ TO CHAR(254)
Total.Attribs = DCOUNT(Dyn.Ay$, Attrib.Mark$)
```

To find the number of values within an attribute or subvalues within a value, the EXTRACT function using angle brackets is employed to extract the subelement for searching. Thus to find the number of values in the attribute specified by Attrib.Num:

```
EQUATE Value.Mark$ TO CHAR(253)
Num.of.Values = DCOUNT(Dyn.Ay$ < Attrib.Num > , Value.Mark$)
```

or:

```
Search$ = Dyn.Ay$ < Attrib.Num >
Delimiter$ = Value.Mark$
Num.of.Values = DCOUNT(Search$, Delimiter$)
```

(For dynamic array extraction, see the section: "Data Structures: Pick: Arrays, Dynamic: Extract.")

DATA STRUCTURES: PICK: ARRAYS, DYNAMIC: DELETE

DELETE

To delete an element of a Pick dynamic array:

```
Dyn.Ay$ = DELETE(Dyn.Ay$, Attrib.Num { , Value.Num, SubValue.Num })
```

The specified attribute, value, or subvalue will be deleted from the dynamic array Dyn.Ay\$. Each of the parameters can be a variable or literal. If there are no Value.Num and SubValue.Num parameters, or those parameters are zero, then the entire attribute is deleted. If there is no SubValue.Num parameter, or that parameter is zero, the entire value is deleted. If the Attrib.Num, Value.Num, or SubValue.Num parameter is -1, then the last attribute, value, or subvalue is specified.

When an attribute, value, or subvalue is deleted, the associated attribute mark, value mark, or subvalue mark is also removed. Thus the command:

```
Dyn.Ay$ = DELETE(Dyn.Ay$, 4)
```

would remove the fourth attribute from the dynamic array Dyn.Ay\$. If there had been more than 4 attributes in Dyn.Ay\$, then attribute number 5 would now be number 4, attribute number 6 would now be number 5, and so on. To set an attribute to a null value, use the REPLACE command rather than DELETE.

The result of the DELETE command also can be assigned to a new dynamic array:

```
New.Dyn.Ay$ = DELETE(Dyn.Ay$, Attrib.Num, Value.Num, SubValue.Num)
```

The original Dyn.Ay\$ is not affected.

The DELETE command can be used without an assignment to display the result of a DELETE operation. Thus the command:

```
CRT DELETE(Dyn.Ay$, 4, 3, 2)
```

displays the dynamic array Dyn.Ay\$, without subvalue 2 of value 3 of attribute 4.

The following operation will delete a single attribute from an item in a file. Note that the DELETE commands for file operations issued at Pick TCL level and in PICK/BASIC each remove an entire item, and cannot be used to delete only one element of an item. (See the section: "Files, Items: Delete.")

```
OPEN File.Data.Name$ TO File.Data$ ELSE STOP
READ Dyn.Ay$ FROM File.Data$, Item.ID ELSE STOP
WRITE DELETE (Dyn.Ay$, Attrib.Num) ON File.Data$, Item.ID
```

DATA STRUCTURES: PICK: ARRAYS, DYNAMIC: EXTRACT

EXTRACT

Use EXTRACT to extract an element from a Pick dynamic array. For example:

```
X$ = EXTRACT(Dyn.Ay$, Attrib.Num { , Value.Num, SubValue.Num})
X$ = Dyn.Ay$<Attrib.Num { , Value.Num, SubValue.Num } >
```

The two versions of the command have the same result. The variable X\$ will contain the specified attribute, value, or subvalue from the dynamic array Dyn.Ay\$. Each of the

parameters can be a variable or literal. If there are no Value.Num and SubValue.Num parameters, or those parameters are zero, then the entire attribute is extracted. If there is no SubValue.Num parameter, or that parameter is zero, the entire value is extracted. If the Attrib.Num, Value.Num, or SubValue.Num parameter is -1, then the last attribute, value, or subvalue is specified.

The following operation will search for a subvalue equal to the string Match\$. Note that Access statements search for attributes and values, but not for subvalues.

```
OPEN File.Data.Name$ TO File.Data$ ELSE STOP
READ Dyn.Ay$ FROM File.Data$, Item.ID ELSE STOP
IF Dyn.Ay$ < Attrib.Num, Value.Num, SubValue.Num > = Match$ THEN
  Test.Flag = True
END
```

DATA STRUCTURES: PICK: ARRAYS, DYNAMIC: INSERT

INSERT

Use INSERT to insert an element into a Pick dynamic array. For example:

```
Dyn.Ay$ = INSERT(Dyn.Ay$, Att.Num{ , Val.Num, SubVal.Num, } {;} New$)
```

The variable New\$ will be inserted into the specified attribute, value, or subvalue of the string dynamic array Dyn.Ay\$. Each of the parameters can be a variable or literal. If there are no Val.Num and SubVal.Num parameters, or those parameters are zero, then an entire attribute is inserted. If there is no SubVal.Num parameter, or that parameter is zero, the entire value is inserted. If the Att.Num, Val.Num, or SubVal.Num parameter is -1, then insertion after the last attribute, value, or subvalue is specified.

If either the Val.Num or SubVal.Num parameters are omitted, then a semicolon must be placed before the value to be inserted. To insert the variable New\$ as the fifth attribute of the dynamic array Dyn.Ay\$:

```
Dyn.Ay$ = INSERT(Dyn.Ay$, 5; New$)
```

When an attribute, value, or subvalue is inserted, an associated attribute mark, value mark, or subvalue mark is also inserted. Thus the command:

```
Dyn.Ay$ = INSERT(Dyn.Ay$, 4; "This is a new value")
```

would insert the string as the fourth attribute of the dynamic array Dyn.Ay\$. If there had been more than 4 attributes in Dyn.Ay\$, then the previous attribute number 4 would now be number 5, attribute number 5 would now be number 6, and so on.

The result of the INSERT command also can be assigned to a new dynamic array:

```
New.Dyn.Ay$ = INSERT(Dyn.Ay$, Att.Num, Val.Num, SubVal.Num, New$)
```

The original Dyn.Ay\$ is not affected.

The following operation will insert a single value into an attribute in an item in a file. Note that the PICK/BASIC WRITEV command for writing to a file will add a single attribute, but not just a single value or subvalue. (For an explanation of the WRITEV command, see the section: "Files, Items: Output.")

```
OPEN File.Data.Name$ TO File.Data$ ELSE STOP
READ Dyn.Ay$ FROM File.Data$, Item.ID ELSE STOP
Dyn.Ay$ = INSERT(Dyn.Ay$, Attrib.Num, - 1; New.Value$)
WRITE Dyn.Ay$ ON File.Data$, Item.ID
```

The INSERT function might not yield the expected results if the new attribute to be inserted into a dynamic array is null. If the program later expects to find a certain number of attributes, whether null or not, the expected number will not be found.

An alternative method is to concatenate the attribute to the dynamic array, including an attribute mark. However, you must avoid placing an attribute mark before the first attribute in the dynamic array or after the last.

In the following routine "Values Separated by Semicolons," elements delimited by a semicolon and space in the string Entry\$ are loaded as attributes into the dynamic array Dyn.Ay\$. Because the FIELD function, used to extract the element, accepts only a single character as delimiter, an additional TRIM function is required to remove the space. (For details on the string operations, see the section: "Strings: Extract.")

Values Separated by Semicolons Routine

```
*****
* Values Separated by Semicolons *
*****
* David L. Clark 09/22/1988
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ValueSep.bpr
EQUATE Attrib.Mark$ TO CHAR(254)
Delim$ = ";"; * Single character delimiter
Total.Elements = DCOUNT(Entry$, Delim$)
Element$ = FIELD(Entry$, Delim$, 1); * No leading space on 1st
Dyn.Ay$ = Element$
FOR I = 2 TO Total.Elements
    Element$ = TRIM(FIELD(Entry$, Delim$, I)); * Trim leading space
    Dyn.Ay$ = Dyn.Ay$ : Attrib.Mark$ : Element$
NEXT I
```

Another method, when the attributes are not to be loaded in sequential order and the total number of elements is known, is first to fill the dynamic array with attribute marks, and then to insert the attributes at specific locations:

```
* Elements from a fixed array are loaded into a dynamic array at
* specific locations determined by Attrib.Num()
EQUATE Attrib.Mark$ TO CHAR(254)
Dyn.Ay$ = STR(Attrib.Mark$, Total.Attrb - 1)
FOR I = 1 TO Total.Attrb
    Dyn.Ay$ < Attrib.Num(I) > = Fixed.Ay$(I)
NEXT I
```

The following TEST.INSERT.NULL program demonstrates the problems encountered when inserting a new attribute that might be null. A null attribute cannot be inserted at the start or end of a dynamic array.

At each test, the attempt was made to insert five attributes.

When the insertion was made in the middle of an item that already had two attributes, it did not matter if the attributes to be added were null.

When both the item and attributes were nonnull, all five attributes could be inserted at the start or end of the item, with the result of Total Attributes = 6.

When the item was null but the attributes were not null, all five attributes could be inserted at the start or end of the item, Total Attributes = 5.

When the item was not null but the attributes were null, all five attributes could be inserted at the start of the item, with the result that Total Attributes = 6. But at the end of the item, only one null attribute was inserted. No additional null attributes could be inserted after the first, with the result that Total Attributes = 2.

When both the item and the attributes were null, no insertion took place either at the start or the end of the item, and Total Attributes = 0.

Conclusion: The safe way to add an attribute to the beginning or end of an item, when the item, the attribute, or a previous attribute might be null, is through concatenation rather than insertion.

TEST.INSERT.NULL Program

```
* TEST.INSERT.NULL David Clark 09/09/1989
* Test dynamic array null insertion
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TInsNull.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  EQU AM$ TO CHAR(254);          * Attribute Mark
  EQU True TO 1, False TO 0
  PROMPT ""
  Clear.Screen$ = @(-1)
  Continue.Flag = True
  LOOP WHILE Continue.Flag DO
    *****
    * Set Initial Values *
    *****
    CRT Clear.Screen$ :
    CRT \Starting value for Item for each test \ :
    CRT \1. "Start" 2. Null\
    CRT \ Pick a Number [1]:\ :
    INPUT Num.Choice$
    IF Num.Choice$ = "2" THEN Start$ = "" ELSE Start$ = "Start"
    CRT
    CRT \ Value for Attribute to add 1. "Attrib" 2. Null\
    CRT \ Pick a Number [1]:\ :
    INPUT Num.Choice$
    IF Num.Choice$ = "2" THEN Attrib$ = "" ELSE Attrib$ = "Attrib"
    CRT
    CRT \ 1. Display Results 2. Print Results\
    CRT \ Pick a Number [1]:\ :
    INPUT Num.Choice$
    Print.Flag = (Num.Choice$ = "2")
```

```

IF Print.Flag THEN PRINTER ON
*****
* Show Initial Values *
*****
PRINT
PRINT "For each test, the attempt is made to add 5 Attributes"
PRINT "Item to start each test = " : Start$
PRINT "Attribute to add = " : Attrib$
Item$ = "Start" : AM$ : "End"
PRINT
PRINT "For initial comparison only, starting value of Item = " :
PRINT Item$
*****
* Show Insertion in the Middle for Comparison *
*****
Start.Item$ = Item$
FOR I = 1 TO 5
    Item$ = INSERT(Item$, 2; Attrib$)
NEXT I
Total.Attribs = DCOUNT(Item$, AM$)
PRINT "Total Attributes after Insert in middle = " :
PRINT Total.Attribs
PRINT "Item = " : Item$
*****
* Add Attribute to End *
*****
* Use Extract function to Insert
Item$ = Start.Item$
FOR I = 1 TO 5
    Item$<-1> = Attrib$
NEXT I
Total.Attribs = DCOUNT(Item$, AM$)
PRINT
PRINT "Total Attributes after Extract-Insert at end = " :
PRINT Total.Attribs
PRINT "Item = " : Item$
* Use Insert function
Item$ = Start.Item$
FOR I = 1 TO 5
    Item$ = INSERT(Item$, -1; Attrib$)
NEXT I
Total.Attribs = DCOUNT(Item$, AM$)
PRINT
PRINT "Total Attributes after Insert at end = " : Total.Attribs
PRINT "Item = " : Item$
* Use Concatenation with Attribute Mark
Item$ = Start.Item$
FOR I = 1 TO 5
    Item$ = Item$ : AM$ : Attrib$
NEXT I
Total.Attribs = DCOUNT(Item$, AM$)
PRINT
PRINT "Total Attributes after Concatenation at end = " :
PRINT Total.Attribs
PRINT "Item = " : Item$
*****
* Insert Attribute at Start *
*****
* Use Insert function

```

```

Item$ = Start.Item$
FOR I = 1 TO 5
    Item$ = INSERT(Item$, 1; Attrib$)
NEXT I
Total.Attribs = DCOUNT(Item$, AM$)
PRINT
PRINT "Total Attributes after Insert at start = " : Total.Attribs
PRINT "Item = " : Item$
* Use Concatenation with Attribute Mark
Item$ = Start.Item$
FOR I = 1 TO 5
    Item$ = Attrib$ : AM$ : Item$
NEXT I
Total.Attribs = DCOUNT(Item$, AM$)
PRINT
PRINT "Total Attributes after Concatenation at start = " :
PRINT Total.Attribs
PRINT "Item = " : Item$
* Turn off printer
IF Print.Flag THEN PRINTER CLOSE; PRINTER OFF
*****
* Continue or Quit *
*****
CRT
CRT "          1. Continue          2. Quit          Pick a Number [1]:" :
INPUT Num.Choice$
Continue.Flag = (Num.Choice$ # "2")
REPEAT; * While Continue.Flag
END

```

DATA STRUCTURES: PICK: ARRAYS, DYNAMIC: LOCATE

LOCATE

LOCATE is used to locate a matching value in an element of a Pick dynamic array, returning the index number of the attribute, value, or subvalue containing the matching value. To find Match\$ within Dyn.Ay\$:

```

LOCATE(Match$, Dyn.Ay${ , Att.Num, Value.Num}; Index{; Order$}) THEN
    Found.Flag = True
END ELSE
    Found.Flag = False
END

```

The dynamic array Dyn.Ay\$ is searched for an element matching the string Match\$. Each of the parameters can be a variable or literal. If there are no Att.Num or Value.Num parameters, or those parameters are zero, then the entire dynamic array is searched for an attribute equal to Match\$. If there is no Value.Num parameter, or that parameter is zero, then the specified attribute is searched for a value equal to Match\$. If there is a Value.Num parameter, then that value is searched for a subvalue equal to Match\$.

An ELSE clause is mandatory, but the NULL statement can be used. (For Pick's IF...THEN...ELSE construct, see the section: "Conditions.")

The optional parameter Order\$ indicates the order of searching, and will speed up the searching process if the elements to be searched have been maintained in a specified order,

such as an alphabetical list. Order\$ is a string, and must either be placed in a variable or surrounded by right single quote marks or double quotes. Note that when sorting or searching, numbers are right justified, strings are left justified. The following specifications can be used:

'AL' =	Ascending,	Left-justified
'AR' =	Ascending,	Right-justified
'DL' =	Descending,	Left-justified
'DR' =	Descending,	Right-justified

The parameter Index will return the index number of the attribute, value, or subvalue that contains Match\$. An added feature of the LOCATE function is very useful for maintaining lists in order. If the Order\$ parameter is included, and the matching value is not found, then Index will return the position within the dynamic array, attribute, or value in which Match\$ should be inserted, in order to maintain the specified order. This information can be used with the INSERT function. If no Order\$ parameter is supplied, the default position for insertion is at the end of the dynamic array. For example:

```
Order$ = "AL";                                * For a list in alphabetical order
* Look through the entire dynamic array for a matching Attribute
LOCATE(Match$, Dyn.Ay$, Index; Order$) ELSE
    * Match$ not found, so place it where it should be
    Attrib.Num = Index
    Dyn.Ay$ = INSERT(Dyn.Ay$, Attrib.Num; Match$)
END
```

Select Lists

The LOCATE command could be used to keep select lists of item IDs in order as new item IDs are added. For example, item IDs are stored in attribute 1 of each item in the file "Subjects.Index," which is an index file to subject terms used in the database "History-Data." A new item ID is obtained from a new data entry or from merger with another select list.

The LOCATE statement will be used to determine whether the item ID is already present in the attribute containing the list of item IDs. If the new item ID is already present, no action is needed. If the item ID is not already stored, it will be added to the list at a place that maintains the order of the list, which in this case is numerical (right justified). In the case shown, the subject term "Harbors" is used as the item ID of the Item in the Subjects.Index file.

```
Item.ID$ = "Harbors"
Attrib.Num = 1
OPEN "Subjects.Index" TO File.Index$ ELSE STOP
* Lock the Item while updating
READVU Select.List$ FROM File.Index$, Item.ID$, Attrib.Num
Order$ = "AR"
```

```

LOCATE(New.Item.ID, Select.List$, Index; Order$) THEN
    RELEASE File.Index$, Item.ID$
END ELSE
    Select.List$ = INSERT(Select.List$, Index, New.Item.ID)
    WRITEV Select.List$ ON File.Index$, Item.ID$, Attrib.Num
END

```

(For locking, see the section: "Files, Items: Input.")

In the following example, the LOCATE statement is placed in a loop to sort a list of values held within an attribute:

```

* Use LOCATE to sort a list of Values
New.Dyn.Ay$ = ""
Order$ = "AL";           * Alphabetical
I = 1
LOOP
    Value$ = Old.Dyn.Ay$ < Attrib.Num, I >
UNTIL Value$ = "" DO
    LOCATE(Value$, New.Dyn.Ay$, Attrib.Num; Index; Order$) ELSE NULL
    New.Dyn.Ay$ = INSERT(New.Dyn.Ay$, Attrib.Num, Index, 0, Value$)
    I = I + 1
REPEAT
Old.Dyn.Ay$ = New.Dyn.Ay$

```

The following program, TEST.LOCATE.SORT, will demonstrate the use of LOCATE to sort a list of attributes within a dynamic array. The program will prompt the user to enter a list of names, each separated by a semicolon and space. The program will then sort the list, and display the original and sorted lists side by side.

TEST.LOCATE.SORT Program

```

* TEST.LOCATE.SORT David L. Clark 03/25/1988
* Use LOCATE to sort a list.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TLocSort.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
    *****
    * Assign Constants *
    *****
    PROMPT ""
    Clear.Screen$ = @(-1)
    Order$ = "AL";           * Left-justified for strings
    *****
    * Get list *
    *****
    CRT
    CRT "Enter a list of names, each separated by a semicolon and space:"
    INPUT Names.Entry$

```

```

*****
* Put Names in a Dynamic Array *
*****
Total.Names = DCOUNT(Names.Entry$, ";")
Names.List$ = ""
* Normally the names would be inserted in order, but this routine
* will sort later, as would be done for an already existing list.
FOR I = 1 TO Total.Names
    Names.List$<I> = TRIM(FIELD(Names.Entry$, ";", I))
NEXT I
*****
* Sort Names *
*****
Sorted.List$ = ""
FOR I = 1 TO Total.Names
    Name$ = Names.List$<I>
    LOCATE(Name$, Sorted.List$; Index; Order$) ELSE NULL
    Sorted.List$ = INSERT(Sorted.List$, Index; Name$)
NEXT I
*****
* Display Original and Sorted Lists *
*****
CRT
CRT "ORIGINAL LIST" : @(39) : "SORTED LIST"
CRT
FOR I = 1 TO Total.Names
    CRT Names.List$<I> : @(39) : Sorted.List$<I>
NEXT I
END

```

DATA STRUCTURES: PICK: ARRAYS, DYNAMIC: REPLACE

REPLACE

Use REPLACE to replace an element in a Pick dynamic array. For example:

```

Dyn.Ay$ = REPLACE(Dyn.Ay$, Att.Num{, Val.Num, SubVal.Num,} {;} New$)
Dyn.Ay$<Att.Num {, Val.Num, SubVal.Num}> = New$

```

The two versions of the command have the same result. the variable New\$ will replace the specified attribute, value, or subvalue of the string dynamic array Dyn.Ay\$. Each of the parameters can be a variable or literal. If there are no Val.Num and SubVal.Num parameters, or those parameters are zero, then an entire attribute is replaced. If there is no SubVal.Num parameter, or that parameter is zero, the entire value is replaced.

If the indicated Att.Num, Val.Num, or SubVal.Num does not already exist, then the new element is inserted with the required number of leading attribute, value, or subvalue marks. If the Att.Num, Val.Num, or SubVal.Num parameter is - 1, then insertion after the last attribute, value, or subvalue is specified.

If either the Val.Num or SubVal.Num parameters are omitted from the command version containing the word "REPLACE", then a semicolon must be placed before the replacement value. Thus to replace the fifth attribute of the dynamic array Dyn.Ay\$ with the variable New\$:

```
Dyn.Ay$ = REPLACE(Dyn.Ay$, 5; New$)
```

or:

```
Dyn.Ay$<5> = New$
```

The result of the **REPLACE** command also can be assigned to a new dynamic array:

```
New.Dyn.Ay$ = REPLACE(Dyn.Ay$, Att.Num, Val.Num, SubVal.Num, New$)
```

The original **Dyn.Ay\$** is not affected. The version with the word “**REPLACE**” was necessary to make this assignment in one step.

The following operation will replace a single value in an attribute in an item in a file. Note that the **PICK/BASIC WRITEV** command for writing to a file will replace a single attribute, but not just a single value or subvalue. (For an explanation of the **WRITEV** command, see the section: “Files, Items: Output.”)

```
OPEN File.Data.Name$ TO File.Data$ ELSE STOP
READ Dyn.Ay$ FROM File.Data$, Item.ID ELSE STOP
Dyn.Ay$ < Att.Num, Val.Num > = New.Value$
WRITE Dyn.Ay$ ON File.Data$, Item.ID
```


Editor



IBM PC DOS

Do not even consider using the IBM PC DOS system editor EDLIN. Instead buy a good word processor. Microsoft Word, with which this book was written, is strongly recommended.

Pick

The Pick system editor is only a simple line editor, not a full screen editor and word processor. The best description of the Pick editor appears in *Exploring the Pick Operating System* by Jonathan Sisk and Steve Van Arsdale. Only a few commands, which are useful for a search and replace operation and for viewing a data item, will be covered here. All of the Pick programs appearing in this book were written in Microsoft Word and transferred over to Pick with the COPYDOS command.

To Replace with a Prestored Command

To replace a string throughout a document, making a choice each time whether the string is to be replaced or not, create the following prestored command, designated as P1. In the example, “Old” is to be replaced with “New,” but each case will be verified to avoid changing embedded strings, and thus making “Oldsmobile” into “Newsmobile.” “RU” for replace universal will perform the replacement, “L” for locate will find the next example. The left bracket indicates the ESCAPE key, not the left bracket on the keyboard.

A series of commands can be placed in a prestore sequence by separating the commands with the ESCAPE character.

First create the prestore:

```
P1 RU/Old/New/[L /Old
```

Now search for the first case of “Old”:

```
L/Old
```

When the case appears, type:

```
P1
```

to replace it and continue the search, or type:

```
A
```

for “Again” to leave the text unchanged and perform the search again.

System Delimiters

The system delimiters of attribute mark (ASCII 254), value mark (ASCII 253), and subvalue mark (ASCII 252) can be viewed and manipulated directly in the Editor. This method is recommended for experienced users only, who understand the database file structure.

The attribute mark will be treated by the Editor as a Carriage Return—Line Feed, causing each attribute to appear on a separate line. You can create a new line, and hence new attribute, by going into insert mode with the “I” command, and then proceeding to type the new attribute. However, if you try to press ENTER only, to insert a null or empty attribute, the Editor will take this to mean exiting from the insert mode. You must put in a dummy value, such as “XXX”, and then replace that value with a null. Rather than using the dummy and replace method, you can insert a null attribute by typing CTRL - ^ . (Hold down the CTRL key and type “^”.) The “^” is the caret mark that on most keyboards is typed with SHIFT-6, so to add the CTRL key, you press CTRL-SHIFT-6.

The value mark will appear on the screen as a right bracket: “]”. To add a value mark, type CTRL-] . (Hold the CTRL key and type “]”.) The subvalue mark will appear as a backslash: “\”. To add a subvalue mark, type CTRL-\ . (Hold down the CTRL key and type “\”.)

Error Recovery



ERROR RECOVERY IS USED TO OBTAIN THE CODE IDENTIFYING THE TYPE OF ERROR, and then branch to an error-handling routine to take appropriate action.

IBM PC BASIC

ON ERROR GOTO Line.Number, ERR, ERL, RESUME {Line.Number or NEXT}

For example:

```
10   ON ERROR GOTO 9000           '*|Error Routines|
20   Num.Chars = -10             '* Negative parameter not allowed
30   X$ = LEFT$(Y$, Num.Chars)   '* Function call error
40   '* Program will resume at this point
    '*-----*
    '******
9000 '* Error Routines *
    '******
    PRINT "Error Number ="; ERR; " Error Line ="; ERL
9010 RESUME NEXT
```

The occurrence of an error recognized by the system will interrupt the program and cause it to branch to the designated Line.Number. The Line.Number must be a literal number, not a variable. The system variable ERR will contain the number of the error and Error Line (ERL) will contain the line number where the error occurred.

The RESUME statement resumes the program after the branching. After a RESUME statement, ERR will be cleared to 0, but ERL will still contain the line number of the last error.

RESUME with no parameters will return to the line that caused the error. (The error handling routine should have corrected the problem.) RESUME NEXT will resume on the line after the line that caused the error. RESUME Line.Number will resume at the designated line.

STOP, CONT

STOP will stop the program. After a STOP statement, the user is dumped into BASIC. The program could then be continued by entering the command "CONT".

An error-handling routine could determine, according to the error number, which of the following to carry out:

IF ERR = Fatal.Error	THEN STOP	'*Stop the program
IF ERR = Not.Fatal	THEN RESUME NEXT	'*At the next line
IF ERR = Corrected	THEN RESUME	'*At the same line
IF ERR = Special	THEN RESUME 1000	'*At a designated line

ERROR

For example:

ERROR Error.Number

You also can simulate an error, supplying your own error number. In that way you can invoke the error-handling routine for a failure that would not be caught as an error by the system. Such a case might be the lack of a needed parameter for a program. After branching to the error-handling routine, ERR and ERL will contain values just as they would if a system error had occurred.

PICK/BASIC

SYSTEM(0), ERRMSG

For example:

Error.Code = SYSTEM (0); * Number from ERRMSG file

The ERRMSG file contains all the Pick error messages, as well as general system messages and additional messages that can be added by the programmer. Each message is identified by a code, generally a number, but it can contain letters. The existence of an active "error code" does not indicate that an error has occurred, because the same process is used for other system messages, such as displaying the time and date when one logs onto the system. (The item "LOGON" in the ERRMSG file will automatically be displayed to all users when they log onto the system.)

To test if an error has occurred, use the ELSE clause available with PICK/BASIC commands:

```
File.Data.Name$ = "History.Data"  
OPEN File.Data.Name$ TO File.Data$ ELSE STOP 201, File.Data.Name$
```

If the file does not exist, and therefore an error occurs, Error Message 201 will be displayed, with the filename:

```
[201] 'History.Data' IS NOT A FILE NAME
```

PICK TCL

PRINT-ERR

For example:

```
PRINT-ERR ERRMSG Error.Code$
```

The Pick TCL PRINT-ERR verb will display the message associated with an error code from the ERRMSG file.

Files, Items



Pick

The Pick file structure is much more complex than that of IBM PC DOS, because the Pick operating system is set up specifically for database usage. A Pick file is composed of items (records in other databases). Each item in turn is made up of attributes (fields).

Each file belongs to an account, in a manner similar to the IBM PC DOS grouping of files into directories. The Pick structure, however, is much more complete and more useful. For example, different accounts can have different security levels, and can be granted or denied access to files in other accounts. The use of any system command also can be granted or denied to an account, including access to the Pick Editor. Each user easily can be given his or her own account. When the person enters the account, a routine unique to that account automatically can be invoked to show the user a menu of choices or carry out any other operation.

The Pick system is dictionary driven. Each level of the system is defined in the dictionary of the next higher level. Thus, the SYSTEM file, in the system programmer (SYSPROG) account, contains the definition of the master dictionary (MD) of each account, thereby defining each account. Each account master dictionary contains the definition of each file residing in that account.

A Pick file has two levels: a dictionary level and a data level. The dictionary level contains the definitions for one or more data level files, as well as the definitions for the attributes in those files. When one file dictionary defines multiple data files, the same data

structure can be used for different files. The data from different projects could be kept separate, while using the same format.

All of the definitions, as with all Pick information, are stored in items in files. Thus an attribute is defined in an attribute definition item in a file dictionary. The dictionary is employed principally by the Access processor when searching and displaying data from a file. (For details on the Access processor, see the section: "Files, Items: Input: Search with Pick Access Processor.")

(Items are presented here with the number of each attribute displayed on the left, followed by a descriptive name or typical entry for the attribute, and additional comments in braces. When numbers in sequence are skipped, the intervening attributes should be null.)

An attribute definition item is composed of attributes, and has the following format:

Item ID:	Attribute.Name	{ Name by which Attribute is accessed }
01	A	{ X if not to be displayed }
02	Attribute Mark Count	{ Position within the Item }
03	Column Heading	{ To display rather than the Item ID }
04	Controlling or Dependent	
05		
06		
07	Conversion	
08	Correlative	
09	Justification	
10	Column Width	

A data attribute, for the date on which a data item was entered, might have the following definition:

Item ID:	Entry.Date
01	A
02	3
03	
04	
05	
06	
07	D/
08	
09	R
10	10

The following attribute definition item will create an attribute that can be used to display remarks contained in the second line of a program or PROC. If the programmer followed the convention consistently of placing a brief explanation of the program or PROC on the second line, this attribute would prove very useful. The attribute definition item should be placed in the MD so that the attribute can be used with any file of PROCS and programs. The attribute will be called Remarks.

The correlative in attribute 8 of the definition item will perform a text extract, taking only characters 1 through 65 of the line.

The attribute definition item, in the master dictionary:

Item ID: Remarks

01	A
02	2
08	T2,65
09	T
10	65

To display the names of all the programs in the file Test.Programs, in alphabetical order, with their explanatory remarks, the following command would be issued at Pick TCL level:

`SORT Test.Programs Remarks`

A file is defined in a file definition item, with the following format:

Item ID: File.Name	{ Name by which the file is accessed }
01 D	{ Other choices: Q, DC, DX, DY }
02 Base Frame	{ Do not change! }
03 Modulo	{ Number of Groups allocated initially }
04 Separation	{ Number of Frames in a Group. Usually 1. }
05 Retrieval code	{ Code match necessary to access the file }
06 Update code	{ Code match necessary to change the file }
07	
08	
09 Justification	{ For the Item IDs of Items in the file }
10 Column Width	{ For the Item IDs of Items in the file }
11	
12	
13 Re-allocation	{ New Modulo and Separation after restore }

The first attribute of the file definition item will begin with “D” if the file is actually stored in that account. If the file definition item is a synonym pointing to a different account, or simply to a different filename, then the first attribute will contain “Q.”

If the file contains binary code in the form of BASIC programs, the first attribute will contain “DC.” A file of saved select lists also will be designated as “DC.”

A code of “DX” in the first attribute instructs the SAVE processor not to save the file. A code of “DY” will save the file definition and space allocation, but not the contents of the file.

In attribute 9, indicate “R” for right justification if the item IDs of the items in the file will be numeric. Indicate “L” for left justification if the item IDs will be alphabetic.

An account is defined in an account definition item. The item is created in the SYSTEM dictionary in the SYSPROG account by the action of the CREATE-ACCOUNT command at PICK TCL level. The item has basically the same format as a file definition item because it is actually the definition of the master dictionary file of the account. Thus the modulo and separation apply to the master dictionary file, they do not affect the files defined in the master dictionary and thus held in that account.

Item ID: Account.Name	{ Name of the individual user or department }
01 D	{ Q for a synonym Account }
02 Base Frame	{ Do not change! }
03 Modulo	{ Number of Groups allocated initially }
04 Separation	{ Number of Frames in a Group. Usually 1. }
05 Retrieval code	{ Code provided to access files }
06 Update code	{ Code provided to change files }
07 Password	{ Used to access the Account }
08 Privilege Level	{ Low to high: SYS0, SYS1, SYS2 }
09 Justification	{ For display of File names in the Account }
10 Column Width	{ For display of File names in the Account }

In attribute 9 of the account definition item, an “R” signifies that the user will not be given access to TCL. If the **BREAK** key is pressed, and then “END” typed, rather than giving access to TCL the Logon PROC will be restarted. The “U” signifies updating the ACC file for accounting, which will keep a record of usage by that user. Both “R” and “U” can be used.

A synonym account definition item can be provided for each user:

Item ID: Name.of.User	
01 Q	
02 Account.to.Access	
09 RU	
10 20	

FILES, ITEMS: ACCESS SYSTEM INFORMATION

IBM PC DOS

DIR

For example:

DIR { Drive: } { \ Paths \ } { Filename.Extension }

The current drive and directory are the default choices.

In response to the DIR command, IBM PC DOS will show a display such as the following. The first line of the heading displays the drive and the volume name, if a volume name has been given to the disk. The second line of the heading shows the directory name. After a blank line, the next two lines represent directory levels. Two dots represent the current directory, one dot represents the directory above it. The time and date show when the current directory was created.

The detail lines that follow represent each file in the directory, showing the filename, extension, size in bytes, and date and time when the file was last changed. The bottom line reports the number of files in the directory and the total number of bytes available on the disk.

IBM PC DOS Directory Display

Volume in drive C is HISTORY-DATABASE

Directory of C:\PICK

			< DIR >	5-05-86	8:32a
..			< DIR >	5-05-86	8:32a
COPYPICK	EXE	71458	6-19-87	9:21a	
DICT	TST	160	4-02-88	10:09p	
DUMP	TST	2279	4-02-88	10:08p	
HISTORY	BP	26730	3-26-88	8:31p	
SHOWSIZE	BP	5130	4-03-88	7:27p	
TRANDOS	BP	58128	3-27-88	3:13p	
TRANTEST	BP	495	1-08-88	4:57p	
7 File(s)		1175552 bytes free			

The listing can be narrowed down with specifications for filename and extension. Wild card characters can be used as part of the specification. A question mark substitutes for a single character, and an asterisk substitutes for an entire filename or extension. To see a listing of all the IBM PC BASIC programs files, in a directory named "Basic," on the "C" drive:

```
DIR C:\Basic\*.bas
```

IBM PC BASIC

FILES

For example:

```
FILES { Drive: } { \ Paths \ }Filename.Extension
```

The BASIC FILES command displays a list of filenames in the same way as the IBM PC DOS "DIR" command. Use the LOF command to get the length of a file:

```
LOF
File.Length = LOF(File.Number)
```

File.Number must be a literal number, the number under which the file was opened. (See the section: "Files: Open.")

Microsoft BASIC (Radio Shack 100)

FILES

The FILES command is the same on the Radio Shack 100 as on the IBM PC, except that on the 100 the command has no parameters or wild cards, and it displays only the filenames, without information on file size, date, or time. If the command is called from within a BASIC program, an asterisk will follow the name of the BASIC program currently in use.

There is no LOF command to find the length of a file on the Radio Shack 100. Instead, the PEEK command would be used to take information from the Radio Shack 100's file directory. The directory for document, BASIC, and machine language files begins at address 63930. Each file information block in the directory is 11 bytes long. The last entry begins at 64138.

The first byte in the file information block contains the file type number. These file type numbers have the following meanings:

- 0 = A file that has been killed
- 128 = Basic program
- 192 = Document file, closed
- 194 = Document file, open
- 240 = Machine Language program

Bytes two and three of the file information block contain the address of the file, with the least significant byte first. The last eight bytes of the file information block hold the filename, with the main portion of the name in the first six bytes. If the name is less than six characters long, the remainder of the six bytes contain spaces. The file extension is held in the last two bytes.

The size of a file can be obtained by subtracting the difference between the file's address and the next highest address. If the program is adding data to a file, the file must be closed and reopened each time that the amount of memory remaining and the file size is accessed, in order to read those figures accurately. Otherwise each will be changed only in jumps of 256 bytes, with the result that the figure for file size could be too low or for memory too high by as much as 255.

If there is no other file at a higher (larger) location in memory, the file address should be compared to its own ending address. The ending address of the file at the highest memory location is stored at decimal 64434 and 64435.

PICK TCL and PICK/BASIC

Note that in all the examples at PICK TCL there are no variables, as there would be in a PICK/BASIC program. Thus references to "Filename" and "Item.ID" require a literal filename or item ID and not a variable. The following PICK/BASIC statements are identified expressly as such.

The LISTFILES command will display a list of filenames at terminal command level. The TEST.EXECUTE.FILE program shown next will retrieve a list of filenames for display with a PICK/BASIC program.

Both LISTFILES and the BASIC program retrieve the filenames in the form of item IDs of file definition items held in a dictionary. A file in the Pick System is defined at the next higher dictionary level, thus the data level of a file is defined at the dictionary level, and that dictionary in turn is defined in the master dictionary of the account. The master dictionary of each account is defined in the SYSTEM file, which is the master dictionary of the SYSPROG Account.

LISTFILES

```
LISTFILES {Dict.Name} {(P)}  
LISTFILES {Dict.Name} {(LPTR)}
```

to display a list of the files defined in a dictionary. If no dictionary name is included, the MD of the account is used as the default dictionary, thus displaying the names of all the files in the account. Note that the parentheses are required to send the output to the printer, even with the "LPTR" modifier. Without the parentheses, the computer looks for a dictionary file named "LPTR."

The display includes a code to indicate the type of file: D = Data File, DC = Program File, Q = Synonym. (For an explanation of synonyms, see the section: "Files: Synonym Names.") The display then shows the frame where the file begins, the modulo (number of groups included in the file) and the separation (number of frames per group, usually 1).

To allow more than the default 10 spaces on the display for the name of each file, change attribute 10, the column width attribute in the item that defines the dictionary, to the appropriate number of spaces. This can be changed with the Pick Editor. To edit the master dictionary item in the master dictionary file for the account, at TCL type:

EDIT MD MD

Change line 10, which will be attribute 10, to the number of spaces desired for the filename display. If the MD item presently has fewer than 10 lines, add extra blank lines to it. Then place an "L" in line 9 (Attribute 9) for left justification, and the number of spaces for the filenames in line 10.

The following program demonstrates different methods of retrieving a list of filenames for display within a PICK/BASIC program. The program also demonstrates methods of using the EXECUTE statement to retrieve a list of item IDs.

A filename in the Pick System is the item ID of the file definition item of the dictionary in which the file is defined. The first attribute of the file definition item begins with "D." (A synonym contains "Q" instead.) The attribute name "*A1" is a default system name for retrieving the first attribute of any item.

The user is prompted for the name of the dictionary file, with "MD" offered as the default choice. Then the user is prompted for a file prefix, for a string with which the filename should begin. This feature allows the program to display only those files designated for a particular project or user.

With regard to the EXECUTE statement, note that search terms must be surrounded with quotes. The convention used by this book is to enclose double quotes in backslashes.

The first method uses the SORT command and captures the output of SORT into a dynamic array. The individual filenames are then extracted from the array. Note that the (C) option is used with SORT to suppress the heading and column names and the end of list message. The output from SORT on the IBM PC/XT Version 2.0 begins with a Form Feed and a Carriage Return. The program therefore deletes the first two attributes from the dynamic array. Should SORT produce a different output on a different system now or

in the future, the SORT method might not work.

The second method uses the SSELECT command and captures the filenames in the form of a select list. The READNEXT statement is used to read the individual names from the list. This method is faster, and avoids possible problems with changes in the output of the SORT command.

Using SSELECT, you could try to extract the number of items selected from the system message, and use that number in a FOR...NEXT loop. However, that method creates the same dangers as using SORT, if the exact message output should change.

TEST.EXECUTE.FILE Program

```
* TEST.EXECUTE.FILE David L. Clark 08/14/1987
* Use EXECUTE to get filename.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TExecute.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants, Parameters, Initial Values *
*****
PROMPT ""
EQUATE Attrib.Mark$ TO CHAR(254)
EQUATE True TO 1, False TO 0
Clear.Screen$ = @(-1)
*=====
*****
* Main Program *
*****
CRT Clear.Screen$ :
CRT "Name of Dictionary [None required]: " :
INPUT Dict.Name$
IF Dict.Name$ = "" THEN Dict.Name$ = "MD"
CRT "Prefix for Filename [None required]: " :
INPUT Prefix$
Search.Term$ = ' ' : Prefix$ : ']' ; * Must surround with quotes
*****
* Get Filenames with SORT *
*****
Command$ = \SORT ONLY DICT \ : Dict.Name$ : \ = \ : Search.Term$
Command$ = Command$ : \ WITH *A1 = "D]" (C) \
EXECUTE Command$ CAPTURING List.Files$
* Delete the first 2 elements
List.Files$ = DELETE(List.Files$,1)
List.Files$ = DELETE(List.Files$,1)
GOSUB 100; *|Display Filenames|
*****
* Get Filenames with SSELECT *
*****
Command$ = \SSELECT DICT \ : Dict.Name$ : \ = \ : Search.Term$
Command$ = Command$ : \ WITH *A1 = "D]" \
EXECUTE Command$; Capture.Message$=""; Return.Code$=""
* EXECUTE Command$ RETURNING Return.Code$ CAPTURING Capture.Message$
* Use Capture Message?
CRT; CRT; CRT
CRT "Return Code = " : Return.Code$
CRT "Capture Message = " : Capture.Message$
```

```

* If any Item IDs have been selected, Capture.Message$ will begin
*   with the Error Message number in brackets, a space, and then the
*   number of Item IDs, followed by a space.
Total.Files$ = FIELD(Capture.Message$, " ", 2)
IF NUM(Total.Files$) THEN
    Total.Files = Total.Files$ + 0;    * Store as a number
END ELSE
    Total.Files = 0
END
CRT "Total Files = " : Total.Files
* With Total.Files taken from List.Message$, a faster FOR...NEXT loop
*   could be used below, rather than LOOP...UNTIL. This advantage
*   would matter only with a very long Select List.
List.Files$ = ""
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
    READNEXT Filename$ THEN
        List.Files$<-1> = Filename$
    END ELSE
        Quit.Flag = True
    END
REPEAT
GOSUB 100;                                *|Display Filenames|
* End of routine
GOTO 9990;                                *|End Program|
=====
100 *****
* Display Filenames *
*****
CRT
CRT
Display$ = "Files:"
* To put each filename on a separate line, add CHAR(13) : CHAR(10)
Total.Files = DCOUNT(List.Files$, Attrib.Mark$)
FOR I = 1 TO Total.Files
    Display$ = Display$ : " " : List.Files$<I>
NEXT I
CRT Display$ :
RETURN
=====
9990 *****
* End Program *
*****
END

```

*** A9999**

For example:

```
LIST Filename 'Item.ID' *A9999
```

At Pick TCL, to find the size of an item, specify attribute “*A9999” as an output parameter. “*A9999” is a system attribute that Pick provides to access an item’s size. Alternatively, you could create an attribute with a name such as Item.Size for easy refer-

ence by a user to the size of an item. Following is an attribute definition item for an Item .Size attribute:

```
Item ID: Item.Size
01  A
02  9999
07  MR0
09  R
10  10
```

Within PICK/BASIC you cannot ask for an attribute by its dictionary name, thus you cannot access item size by asking “*A9999” or for attribute number 9999. Under the new open architecture release of Pick, PICK/BASIC has the added feature that it can make use of dictionary definitions. Under prior releases, you would have to READ the item and then measure its length in bytes with the string function LEN. The size of the item includes the system delimiters of attribute, value, and subvalue marks:

```
READ Item$ FROM File$, Item.ID$ ELSE STOP
Item.Size = LEN (Item$)
```

The SHOW.SIZE program presented here will give the Pick user some of the features that an IBM PC DOS user expects for obtaining the size of a file.

For a data file, the SHOW.SIZE program displays the item ID and size, creating for that purpose an attribute definition item named ITEM.SIZE, with an attribute count of 9999, in the account’s MD, as shown in the ITEM.SIZE example. Pick does not automatically keep track of the time that a file was last written to, as does IBM PC DOS. A Pick database should contain an attribute for the date of entry or editing if that information is important. For data files, the program also gives the option of entering an asterisk in place of an item ID, to show the sizes of all the items in the file. In that case, the program also displays a file statistics report with total size, item count, and average item size.

For a compiled program, SHOW.SIZE will display lines such as the following:

```
Object code: Bytes = 194  Frames = 1  Date = 02 APR 88  Time = 13:20
Source code: Bytes = 356
```

The number of frames used can be up to one-third greater than you would expect from dividing the size of the object code by 500 bytes per frame. The difference is accounted for by the symbols table, which holds symbolic references to all of the variables for use by the debugger. The symbols table can be eliminated by using the “(S)” option when compiling. (For compiler options, see the section: “Program Running and Compiling.”)

The number of frames used and the date and time of compilation are stored in the dictionary level of the program file, with the program name as item ID. In the following example, attribute 5 holds the time and date of compilation, attribute 3 holds the number of frames used, and attribute 2 holds the frame number where the object code begins.

```
001 CC
002 12945
003 4
```

004
005 13:20:45 02 APR 1988

To obtain the size of the object code in bytes, you must perform a DUMP of the first frame holding the code. The syntax of the DUMP command at Pick TCL is:

DUMP Frame.Number{,Option}

To obtain the first frame of a program whose code begins at frame number 12945, and to retrieve the information in hexadecimal form, the command is:

DUMP 12945,X

The SHOW.SIZE program uses the EXECUTE statement to perform the DUMP command, capturing the output into the variable Capture.Output\$. The first few lines of the output of the DUMP command are shown here:

FID:	12945 :	0	0	0	0	(3291 : 0 0 0 0)
0001	00C2006E	49046C74	01010101	0109FF27		1 :.B.nl.1t.....:"
0011	01074E09	2D31FF79	18001E01	05000000		17 :.N. - 1_Y.....:
0021	004E2088	18002801	05000000	004E204E		33 :.N ...(.....N N:
0031	092D31FF	7918032	01050000	00007530		49 :. - 1_y..2.....u0:

The only part of interest to the SHOW.SIZE program is the first two bytes of the hexadecimal output: "00C2". The first two bytes hold the size of the object code, in hexadecimal.

Note carefully that when using the CAPTURING clause of the EXECUTE statement to capture output for parsing and dissection, any change in the display format between different Pick versions will require a change in the routine. The SHOW.SIZE program took the object code size from the output of the DUMP command by extracting four characters, starting at character position 7, from attribute 5:

Object.Size.Hex = Capture.Output\$<5> [7, 4]

The hexadecimal number was converted to decimal form with the statement:

Object.Size.Bytes = OCONV(Object.Size.Hex, "MCXD")

SHOW.SIZE Program

User Exits: U201E, UA070

```
* SHOW.SIZE David L. Clark 04/24/1988
* Show the size of a program or data Item
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ShowSize.bp
```



```

* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants, Initial Values *
*****
PROMPT ""
EQU Attrib.Mark$ TO CHAR(254)
EQU False TO 0
File.Name.Default$ = "BP"
Item.Size.ID$ = "ITEM.SIZE";      * Attribute Definition Item in MD
EQU Peek.User.Exit$ TO "U01FF";  * Read virtual memory byte
EQU True TO 1
*****
* Create ITEM.SIZE Attribute Definition in Master Dictionary *
*****
OPEN "MD" TO File.Master.Dict$ ELSE
    CRT "Cannot find Master Dictionary"
    GOTO 9999;                      *|End Program|
END
* Check if Item already exists
READV Attrib.Item.Size$ FROM File.Master.Dict$, Item.Size.ID$, 1 ELSE
    * Create Item
    Item.Size$ = "A";              * Attribute definition
    Item.Size$<2> = 9999;           * Returns size of the Item
    Item.Size$<9> = R;             * Right-justified
    Item.Size$<10> = 10;           * Display width
    WRITE Item.Size$ ON File.Master.Dict$, Item.Size.ID$
END
*****
* Get User Request Loop *
*****
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
    CRT
    CRT "1. Compiled Program   2. Data Item   3. Quit       " :
    CRT "Pick a number [1]: " :
    INPUT Response$
    Quit.Flag = (Response$ = "3")
    IF NOT(Quit.Flag) THEN
        Code.Flag = (Response$ # "2")
        CRT
        CRT "Filename [" : File.Name.Default$ : "]: " :
        INPUT File.Name$
        IF File.Name$ = "" THEN
            File.Name$ = File.Name.Default$
        END ELSE
            File.Name.Default$ = File.Name$
        END
        IF Code.Flag THEN
            CRT "Program name: " :
            INPUT Item.ID$
            GOSUB 100;                *|Show Size of Program|
        END ELSE
            CRT "Item ID [* for entire file]: " :
            INPUT Item.ID$
            GOSUB 200;                *|Show Size of Data Item|
        END
    END
END
REPEAT; * Until Quit.Flag
GOTO 9999;                          *|End Program|

```

```

100  *=====*
*****
* Show Size of Program *
*****
* Parameters: File.Name$, Item.ID$
*****
* Get Frames, Time, Date, Base *
*****
* Open dictionary level of file
OPEN "DICT", File.Name$ TO File.Dict$ ELSE
    CRT "Cannot find file: " : File.Name$
    RETURN
END
* Read dictionary Item for the program
READ Item.Dict$ FROM File.Dict$, Item.ID$ ELSE
    CRT "Cannot find program: " : Item.ID$
    RETURN
END
Object.Size.Frames = Item.Dict$<3>
Object.Time$ = Item.Dict$<5>[1, 5]
Object.Date$ = TRIM(Item.Dict$<5>[9, 99])
* Get the frame where object code stored
Frame.Num = Item.Dict$<2>
*****
* Get Object Code Size by Dump *
*****
* Dump the first Frame of code. The display actually starts with
*   byte 12 of the Frame, not the true 1st byte.
EXECUTE \DUMP \ : Frame.Num : \,X\ CAPTURING Capture.Output$
* The Item size in hexadecimal is stored in the first four chars
*   displayed for the frame.
* If the DUMP display changes for different implementations,
*   the extraction specification below must be changed.
Object.Size.Hex = Capture.Output$<5>[7, 4]
Object.Size.Bytes.Dump = OCONV(Object.Size.Hex, "MCXD")
*****
* Get Object Code Size by User Exit *
*****
* The Item size in hexadecimal is stored in bytes 12 and 13 of the
*   Frame.
Offset = 12
Address$ = Frame.Num : Attrib.Mark$ : Offset
Size.1.Hex.Byt = OCONV(Address$, Peek.User.Exit$)
Offset = 13
Address$ = Frame.Num : Attrib.Mark$ : Offset
Size.2.Hex.Byt = OCONV(Address$, Peek.User.Exit$)
Object.Size.Hex = Size.1.Hex.Byt : Size.2.Hex.Byt
Object.Size.Bytes.User.Exit = OCONV(Object.Size.Hex, "MCXD")
*****
* Get Source Code Size *
*****
OPEN File.Name$ TO File.Source$ THEN
    READ Item.Source$ FROM File.Source$, Item.ID$ THEN
        Source.Size = LEN(Item.Source$);    * Includes delimiters
    END ELSE
        Source.Size = 0
    END
END ELSE

```

```

        Source.Size = 0
END
*****
* Show Size *
*****
CRT
CRT "Object code: " :
CRT "Object Date   = " : Object.Date$ :
CRT "      Time    = " : Object.Time$
CRT "      Frames  = " : Object.Size.Frames :
CRT "      Bytes   = " : Object.Size.Bytes.Dump :
CRT "      Bytes   = " : Object.Size.Bytes.User.Exit :
CRT "Source Bytes  =   : Source.Size
RETURN
*-----*
*****
200  * Show Size of Data Item *
*****
* Parameters: File.Name$, Item.ID$
IF Item.ID$ # "" THEN
    EXECUTE \LIST \ : File.Name$ : \ '\ : Item.ID$ : \ ' ITEM.SIZE\
END ELSE
    EXECUTE \SORT \ : File.Name$ : \ TOTAL ITEM.SIZE\
    EXECUTE \STAT \ : File.Name$ : \ ITEM.SIZE\
RETURN
*=====*
*****
9999 * End Program *
*****
END

```

In PICK/BASIC

For example:

```

Attrib.Count = OCONV(Item$, "U201E)
Value.Count = OCONV(Attrib$, "UA070")

```

User Exit U201E returns the number of attributes in an item in a file, or in a dynamic array held as a variable by a PICK/BASIC program. User exit UA070 will return the number of values within an attribute. The DCOUNT statement could be used for the same purpose as the user exits within a PICK/BASIC program:

```

EQUATE Attrib.Mark TO CHAR(254), Value.Mark$ TO CHAR(253)
Attrib.Count = DCOUNT(Item$, Attrib.Mark$)
Value.Count = DCOUNT(Attrib$, Value.Mark$)

```

(For DCOUNT, see the section: "Strings: Extract.")

The user exits, unlike DCOUNT, can be used directly in Access statements, if attribute definition items employing the user exits are created. The user exit would be placed in attribute 8, the correlative attribute, of the attribute definition item:

```

Item ID: Attrib.Count
01  A

```

```

07 MR0           { Display as a number }
08 U201E         { User Exit to count Attributes }
09 R
10 10

```

Item ID: Value.Count

```

01 A
07 MR0           { Display as a number }
08 UA070         { User Exit to count Values }
09 R
10 10

```

When conducting a search, you could access the number of attributes in a particular item. For example, if an item in an index file stores each item ID on a select list in a separate attribute, to find the total number of item IDs stored for the subject of harbors:

```
LIST Subjects.Index 'Harbors' Attrib.Count
```

CAT

For example, type

```
CAT Filename
```

to display information about the programs in a program file. The display will show something similar to:

Filename	BASE	LEN	DATE COMPILED
Program.1	4310	1	21:24:14 06 MAY 1986
Program.2	4318	6	16:16:06 03 APR 1986

The display shows the base frame where the program is stored, the length of the program in frames, and the date when it was last compiled. CAT is a PROC in the PROCLIB file. It is not explained in the Pick manual. The command used by the CAT procedure is:

```
SORT DICT Filename WITH *C1 = "CC" *C2 *C3 *C5
```

To add to this PROC the option of sending the listing to the printer rather than the terminal, include the following just before the final line:

```
IF A3 = LPTR H LPTR
```

Then invoke the command with the line-printer parameter:

```
CAT Filename LPTR
```

Do not try to use the alternate "(P)" parameter for the printer.

To display a list of cataloged BASIC programs:

```
SORT MD WITH *A6 = "Filename" {(Options)}
```

The command will display a sorted list of those programs from the designated program file that have been cataloged. "*A6" is a default designation for the sixth attribute in an item.

COMPARE

For example:

```
COMPARE Filename Item.IDs {(Options)}
```

The system will prompt:

WITH:

to which you reply:

```
WITH: {(New.Filename)} Item.IDs
```

The system will show the elements that are different, placing a "D" in the left margin for deleted lines and an "I" for inserted lines. The following options are available:

- A All elements of each attribute, rather than only first 3.
- N NOPAGE on output to terminal
- P Output to printer
- S Suppress display of identical items
- Z Show errors only, in Assembly format

COMPARE is a verb. It is not mentioned in the Pick manual.

CT

For example:

```
CT Filename Item.ID {Item.ID Item.ID}
```

CT stands for copy-to-terminal, and will cause the item to be displayed on the terminal screen. CT is the same as the COPY command, except that it is automatically directed to the terminal, and the system will not prompt for an item ID. At least one item ID must be included in the command, or the process will abort.

CT is a PROC in the PROCLIB file. It does not appear in the index of the Pick manual.

FILES, ITEMS: COPY, BACKUP

IBM PC DOS Command Level

COPY

For example:

```
COPY Filename.Ext NewFile.Ext
```

The IBM PC DOS COPY command creates a new file, to which the old file is copied. If a file already exists with the intended new filename, the old file is destroyed without warning. Therefore always check that no file with that name already exists with the DIR command:

```
DIR NewFile.Ext
```

If no such file exists, the reply will be:

```
File not found
```

BACKUP, RESTORE, VERIFY

For example:

```
VERIFY ON  
BACKUP {Original.Drive} {Path} {Filename.ext} Storage.Drive  
RESTORE Storage.Drive Original.Drive {Path} {Filename.ext}  
VERIFY OFF
```

If a file on a hard disk is too big to store on one floppy diskette, the BACKUP command must be used. IBM PC DOS will prompt for the next diskette as each is filled. Be sure to have enough diskettes on hand. Any other data on the diskettes will be erased. As always, the current drive and directory are the default choices. If no filename is specified, all the files in the directory will be backed up.

The great disadvantage of the BACKUP command is that a large database file might stretch out over 20 diskettes and if one diskette goes bad, the entire backup might be worthless. The IBM PC DOS BACKUP and RESTORE procedure does not provide a selective method, such as the SEL-RESTORE command that Pick offers, for selecting and restoring individual records from a database backup. (Conclusion: Use each system for what it does best. Use Pick for database management, and IBM PC DOS for word processing, spreadsheets, and graphics.)

To add a little bit of security to the IBM PC DOS method, the VERIFY ON command should be issued before a backup. When DOS writes the data to the backup diskette, the system will check to make sure that it can read the data back. Since the VERIFY option slows down disk operations, give the VERIFY OFF command after the backup is completed. To find out if VERIFY is currently on or off, issue the command VERIFY by itself.

The RESTORE command will restore files from a series of backup diskettes. The drive and directory for restoration must be the same that were used for the original backup. You cannot BACKUP from one directory and then RESTORE to another.

To back up the file History.dat in the history directory onto a series of floppy diskettes:

```
VERIFY ON
BACKUP C:\History\History.dat A:
VERIFY OFF
```

To restore the same file from diskettes:

```
RESTORE A: C:\History\History.dat
```

Pick TCL

COPY

To copy items in a file:

```
COPY Filename Item.ID {Item.ID ...} {(Options)}
```

The system will prompt:

TO:

Reply with:

```
TO: { (New.Filename) {New.Item.ID ...}
```

Note the left parenthesis before the new filename. If the parenthesis was not included, Pick would think that “New.Filename” was an item ID.

The file to which the items are to be sent must already exist. (To create a file, see the section: “Files, Items: Open.”)

Each item ID must be unique. Pick does not permit different items in the same file to have the same item ID. For the COPY command, you must supply either a new filename, or new item IDs, or both. After issuing the command:

```
COPY Filename Item.1 Item.2 Item.3
```

The following options are available for the second part of the command. To copy items with new IDs into a new file:

```
TO: (New.Filename New.Item.1 New.Item.2 New.Item.3
```

To copy items with the same IDs into a new file:

```
TO: (New.Filename
```

To copy items with new IDs into the same file:

```
TO: New.Item.1 New.Item.2 New.Item.3
```

To copy an entire file, copy all the items (with the same IDs or new IDs) for both the dictionary and data levels of the file. The asterisk is a wildcard character standing for "All Items":

```
COPY DICT Old.Filename*  
TO: (DICT New.Filename { New.Item.IDs}  
COPY Old.Filename*  
TO: (New.Filename { New.Item.IDs}
```

SELECT, SSELECT

Items can be chosen with the SELECT or SSELECT command, and the select list of item IDs then passed to the COPY command:

```
SELECT Filename { Conditions for selection }  
COPY Filename  
TO: (New.Filename
```

For example:

```
SELECT History.Data WITH Project = "Chamber of Commerce"  
COPY History.Data  
TO: Chamber.History.Data
```

All the data items for the Chamber of Commerce project will be copied. (For more details on the SELECT command and the use of select lists, see the section: "Files, Items: Input: Search with Pick Access Processor.")

Sampling

When carrying out a new procedure, it is wise to test the procedure before using it to modify a data file. For that purpose you might create a sample file with a small sample from the data file. The number of items option with the COPY verb will copy only the number specified. For example, to copy 100 items from File.Data to File.Sample:

```
COPY File.Data * (100)  
TO: (File.Sample
```

Note the asterisk to indicate all items, followed by the option in parentheses that limits the number.

Under some Pick implementations, but not under IBM PC/XT R83, 2.0, a number of items option or a SAMPLING clause can be used with the SELECT command.

SET-FILE

To copy a file from a different account, a Q-pointer must be used. The Q-pointer will be a file synonym, a file definition item that points to a file in a different account. The

command SET-FILE will create a temporary file synonym with the name "QFILE". Every time the SET-FILE command is used, the identity of QFILE will be changed, to point to a different file. (For file synonyms, see the section: "Files, Items: Synonyms.")

The SET-FILE command is given from the account into which the copy is to be made. When the command SET-FILE is used, the system will prompt:

ACCOUNT NAME?

Respond with the name of another account, the account from which the copy is to be made. The system will then prompt: "FILE NAME?" Give the name of the file that is to be copied. The system will give the final message:

'QFILE' UPDATED

For example, to copy the SET-DATE verb from the master dictionary file in the SYSPROG account for use in the current account:

```
SET-FILE
ACCOUNT NAME? SYSPROG
FILE NAME? MD
'QFILE' UPDATED
COPY QFILE SET-DATE
TO: (MD
```

The Pick commands to back up a file onto floppy diskettes or tape allow you to back up a single file, an account, or all the accounts on the system.

All of the Pick commands for diskettes are the same as the commands for tape.

Begin each session using diskettes or tape with the commands to attach and rewind a tape:

```
T-ATT
T-REW
```

End the session with the command to detach the tape:

```
T-DET
```

ACCOUNT-SAVE

The ACCOUNT-SAVE verb will back up an entire account, including the account's master dictionary and the dictionary and data levels of all files. The command must be issued from the SYSPROG account, where the ACCOUNT-SAVE PROC is stored. The following sequence will back up the history account:

```
LOGTO SYSPROG
T-ATT
T-REW
ACCOUNT-SAVE
```

Pick will prompt:

TAPE LABEL IF DESIRED:

The user can add information to the label that Pick provides, which contains the name of the account and the date and time of the backup.

Pick will then prompt:

ACCOUNT NAME?

Respond by entering the actual name of the account:

ACCOUNT NAME? History

ACCOUNT-RESTORE, DELETE-ACCOUNT

The ACCOUNT-RESTORE verb restores an account from tape or from a series of diskettes. The account can be new, perhaps supplied by a software vendor, rather than an account previously used on the present system. In either case, no account with the same name can exist on the current system. If any such account exists, it must be deleted. To delete the history account:

LOGTO SYSPROG
DELETE-ACCOUNT

Pick prompts:

ACCOUNT NAME?

Respond with the name:

ACCOUNT NAME? History

Pick will display a listing of all the files in the account and ask if it is still to be deleted. Respond by answering "YES":

DO YOU STILL WANT TO DELETE THE ACCOUNT? YES

To restore the history account:

LOGTO SYSPROG
T-ATT
T-REW
ACCOUNT-RESTORE History

Pick will prompt for the name under which the account was originally saved. The name on the diskettes can be different than the name to which the account will be restored. In the case shown here, the account was saved with the name "History":

ACCOUNT NAME ON TAPE? History

SEL-RESTORE

For example:

SEL-RESTORE Filename {Item.IDs} {(Options)}

The **SEL-RESTORE** command allows you to restore an individual file or item from a backup. The backup might contain an entire account or all of the accounts on the system. Pick will search through the backup for the file and items requested. The “(O)” for “Over-Write” option will cause the restore to over-write any existing items with the same item IDs.

The restore must take place from the account to which the restore will be made, and the file must already exist in the account.

To restore items with the item IDs of “ID.1” and “ID.2” from the History.Data file in the history account, and over-write those items if they already exist:

```
LOGTO History
T-ATT
T-REW
SEL-RESTORE History.Data ID.1 ID.2 (O)
```

Pick then prompts for the name of the account and the file on the tape (or diskette). In this case, the names are the same as on the current system:

```
ACCOUNT NAME?      History
FILE NAME?         History.Data
```

SAVE

For example:

SAVE (DT)

Pick’s file-save procedure, activated by the **SAVE** command, saves all of the accounts onto tape or diskettes. The operating system code from Pick Systems is not backed up. The operating system only can be restored from the original system diskettes or tape.

When a file-restore is performed from the file-save diskettes or tape, any new file modulus, placed in attribute 13 of a file definition item, will be applied.

An empty file-save will generate system statistics. Otherwise, you must use the “D” option to save data and the “T” option to save the material to tape or diskette.

To save all of the accounts, including the data in them:

```
LOGTO SYSPROG
T-ATT
R-REW
SAVE (DT)
```

To restore all of the accounts (for IBM PC/XT Release 1983 Version 2.0), first place the original "Pick PC System Diskette #1" in the A: Drive. Now reboot the system by pressing CTRL-ALT-DEL. Pick will display the following prompt:

OPTIONS (A,F,K)

Option "A" will restore the operating system only, without affecting the data files. Option "K" will kill or delete the Pick partition.

To perform a file-restore, to restore the data files, enter option "F" for "Files." Pick will first restore the operating system, prompting for each additional diskette to be inserted. After Pick has finished reloading the operating system diskettes, the following prompt will appear:

Load PICK PC DATA FILES #1 then type 'C' to continue

Now insert each of the diskettes onto which the data has been saved.

T-DUMP, T-LOAD

For example:

T-DUMP	Filename	{Item.ID}	{Conditions}	{Modifiers}	{(Options)}
T-LOAD	Filename	{Item.ID}	{Conditions}	{Modifiers}	{(Options)}

Individual files and items can be dumped to or loaded from tape or diskette with the T-DUMP and T-LOAD commands. These commands and the SAVE commands are mutually exclusive. You cannot T-LOAD an item saved to diskette with the ACCOUNT-SAVE command. To retrieve the individual item use SEL-RESTORE.

T-DUMP and T-LOAD are Access verbs, with the same selection options available as other Access commands such as LIST and SORT. You also can precede the command with a SELECT statement to apply selection criteria. (For details on selection, see the section: Files, Items: Input: Search with Pick Access Processor.)

To dump the entire History.Data file to tape or diskette:

T-DUMP History.Data

To dump only those items in which the subjects field (defined through an attribute definition item in the file dictionary) contains the word "Harbors" as an individual value:

T-DUMP History.Data WITH Subjects = "Harbors"

The T-LOAD command will load files and items dumped originally with the T-DUMP command. The file named must already exist on the system. If the "(O)" for "Over-Write" option is included, T-DUMP will over-write items with identical item IDs.

To load the History.Data file from tape or diskette, without over-writing any items:

T-LOAD History.Data

To load only those items in which the subjects field includes “Harbors”, over-writing any items on the system with identical Item IDs:

T-LOAD History.Data WITH Subjects = “Harbors” (O)

FILES, ITEMS: DELETE

DELETE is used to delete an entire file or item from the system.

IBM PC BASIC

KILL

For example:

```
KILL File$  
KILL "Filename.Extension"
```

The filename can be a variable or a literal enclosed in quotes. The file can be open or closed.

The extension to the filename must be included in the NAME and KILL commands, although it is not required in the OPEN, input, or output commands.

PICK/BASIC

DELETE

For example:

```
DELETE File.Data$, Item.ID$
```

In this case, an item rather than an entire file is deleted. The item specified by its item ID is deleted in the file previously opened and assigned to the file variable File.Data\$. Either a variable or literal enclosed in quotes can be used for Item.ID\$. If the item does not exist, no action is taken, and there is no error.

(To delete an individual attribute, value, or subvalue within an item, see the section: “Data Structures: Pick: Dynamic Arrays: Delete.” To search for particular items to delete, according to the values that they hold, see the section: “Files, Items: Input: Search with Pick Access Processor.”)

CLEARFILE

For example:

```
CLEARFILE File.Data$
```

The data level of the file opened and assigned to the specified file variable will be cleared of all its data. The file itself will not be deleted, it will simply be empty of data. The dictionary level of the file cannot be cleared in this way.

Pick TCL

DELETE-FILE

For example:

```
DELETE-FILE { DICT } Filename
```

Use this command to delete an entire file. The dictionary file cannot be deleted before the data files that the dictionary points to.

DELETE

For example:

```
DELETE { DICT } Filename {Item.ID Item.ID . . .}
```

Use this command to delete a single item rather than an entire file. If no item ID is entered, the system prompts with:

```
NEXT ITEM:
```

You can continue to enter item IDs, receiving a prompt after each one. To stop the process, press ENTER without giving an item ID.

DELETE is not explained in the Pick manual. It is a PROC, stored in PROCLIB.

Items can be chosen for deletion with the SELECT command, and the select list then passed to the DELETE command:

```
SELECT Filename {Conditions}  
DELETE Filename
```

For example:

```
SELECT History.Data WITH Project = "Chamber of Commerce"  
DELETE History.Data
```

All the data items for the Chamber of Commerce project will be deleted. Note however, that this procedure, and many other utility commands and procedures provided by Pick, will not function correctly if an item ID contains a space. Therefore although it is not illegal to include a space in a Pick item ID, there are great disadvantages in doing so.

(For more detail on SELECT and Select Lists, see the section: "Input: From a File: Search with Pick Access Processor.")

PICK/BASIC programs are stored with the source code in an item in the data level of a file, and the object code in an item in the dictionary level. To delete entirely a program that is no longer needed:

```
DECATALOG Program.Name  
DELETE Program.File Program.Name  
DELETE DICT Program.File Program.Name
```

CLEAR-FILE

For example:

```
CLEAR-FILE DATA Filename  
CLEAR-FILE DICT Filename
```

The Pick TCL CLEAR-FILE command operates in a manner similar to the PICK/BASIC CLEARFILE command, with the difference that the data or dictionary level of the file must be specified. The specified section of the file will be left empty. Clearing the dictionary level will not delete file definition items.

FILES, ITEMS: INPUT

IBM PC BASIC

LINE INPUT #

For example:

```
LINE INPUT #File.Number, Line.From.File$
```

LINE INPUT # will take all of the text, including commas, blanks, and quote marks, up to the next Carriage Return or 255 characters, whichever occurs first. The file must be identified by its assigned file number, which must be a literal number. LINE INPUT # is not mentioned in the Radio Shack 100 manual.

LINE INPUT # does not place the Carriage Return—Line Feed characters (ASCII 13 and 10) into the assigned input variable. Therefore to input lines (of 255 characters or less) from file #1 and send them to file #2 would require that the output statement include a Carriage Return:

```
LINE INPUT #1, Line$  
PRINT #2, Line$
```

A semicolon at the end of the PRINT statement suppresses the Carriage Return.

The simplest method for creating and later reading a file is to place each separate element, such as a field for a database or a parameter used by a program at start-up, on a separate line, and then read each element with a LINE INPUT # statement. Placing each element on a separate line is the best technique from the standpoint of compatibility with other languages. PICK BASIC will be able to read one line at a time with the READV statement.

The comma, used as a delimiter between elements by the INPUT # statement, shown later, is not recognized as a delimiter by the other languages. Furthermore, some elements can contain embedded commas within quoted strings. (The complicated string extraction methods necessary to read such files are shown in the section: "Files, Items: Input: Convert From IBM PC BASIC to Pick.")

LINE INPUT # has two disadvantages in comparison with INPUT #. The first is that placing each element on a separate line requires two characters for separation: Carriage Return and Line Feed (ASCII 13 and 10), while a comma separator only requires one

character. Thus separate lines require more memory. This factor is important only when dealing with a file with many elements on a machine with a small memory. If the comma-delimited elements require surrounding quotes, then three separating characters must be used.

The second disadvantage is that the LINE INPUT # will only assign one variable, whereas the INPUT # command can assign many. This disadvantage can be avoided by placing the elements in an array, and reading them with a FOR...NEXT loop. The FOR...NEXT method with LINE INPUT # is naturally appropriate for database fields, where the number of fields in the record will be known.

EOF

For example:

```
IF NOT EOF(File.Number) THEN LINE INPUT #1, Line$
```

A file input or file transfer routine must check for the end of the file. The boolean EOF function will return true (which in IBM PC BASIC is -1) when the end of the file has been reached. The File.Number parameter must be a literal—it cannot be a variable. It is the number under which the file was opened.

The following “Read Database Fields” routine will read database fields of less than 255 characters each from one file and transfer them to another.

Read Database Fields Routine

```
100  '* Read Database Fields. David L. Clark 09/22/1988
103  '* IBM PC Basic routine. Documentation version.
104  '* IBM PC BASICA 3.0, Radio Shack 100
105  '* Stored under filename: ReadData.brd
150  OPEN File.Input$ FOR INPUT AS #1
160  OPEN File.Output$ FOR OUTPUT AS #2
200  '* Begin line
      IF NOT EOF(1) THEN
        FOR I = 1 TO Total.Fields
          LINE INPUT #1, Field$(I)
          PRINT      #2, Field$(I)
        NEXT
        GOTO 200                '* Loop back
      ELSE
        CLOSE
      '* End: If Not End of File
    '* End line
```

Additional considerations apply if the element to be read is longer than 255 characters. If LINE INPUT # has taken 255 characters before reaching a Carriage Return, the action of the command will stop at that point. The next LINE INPUT # statement will start with character 256. A routine transferring text from one file to another must check the length of the input string. If the input is less than 255 characters, then a Carriage Return—Line Feed should be added to the output, because a complete line or paragraph has been taken. If the input equals 255 characters, no Carriage Return—Line Feed should be added, because there is more to come in the same line or paragraph.

If the transfer is meant for a database file, in which each Carriage Return signals the end of a field or attribute, care must be taken for the possibility that a line could total exactly 255 characters. In that case, the first LINE INPUT # would take all the characters, leaving the Carriage Return. The next LINE INPUT # would take only the Carriage Return, returning null to the program. If the program was following a FOR...NEXT loop to take in a specified number of fields, it would think that it had two fields, when it only had one, upsetting the expected database structure. Either the database file creation routine should limit the line length to less than 255 characters, or the database transfer routine must check if any line equals exactly 255 characters and, if it does, make a second LINE INPUT # to take the dangling Carriage Return.

Alternatively, a plain text transfer method could be used instead. Although the text transfer routine would make two LINE INPUT # statements for one data field of 255 characters, the routine would only send one Carriage Return—Line Feed to the output device or file, thus preserving the database structure. The text transfer method is slower, because it must check for line length and the end of the file after each input, rather than following a FOR...NEXT loop through all the fields. However, the program processor is probably faster than a transfer device such as a serial port or modem in any case.

The following “Read Plain Text” routine will transfer plain text from one file to another.

Read Plain Text Routine

```

100  '* Read Plain Text. David L. Clark 09/22/1988
103  '* IBM PC Basic routine. Documentation version.
104  '* IBM PC BASICA 3.0, Radio Shack 100
105  '* Stored under filename: ReadText.brd
150  OPEN File.Input$ FOR INPUT AS #1
160  OPEN File.Output$ FOR OUTPUT AS #2
200  '* Begin line
      IF NOT EOF(1) THEN
        LINE INPUT #1, Line$
      :    IF LEN(Line$) < 255 THEN
          PRINT #2, Line$
          GOTO 200
      :    ELSE
          PRINT #2, Line$;
          GOTO 200
      :    '* End: If Len(Line$)
      ELSE
        CLOSE
      '* End: If Not End of File
  '* End line

```

INPUT\$

For example:

Chars.From.File\$ = INPUT\$(Num.Chars, File.Number)

The INPUT\$ statement will take the designated number of characters from a file. The

file must be identified by its assigned file number, which must be a literal number. To input 1 character from file number 2:

```
INPUT$(1, 2)
```

INPUT\$ returns any character found, including commas, quote marks, Carriage Returns, and Line Feeds. Because INPUT\$ gives no warning of an end of file condition, and then aborts when the end is reached, it only can be used safely to input one character at a time.

INPUT #

For example:

```
INPUT #File.Number, Variable { , Variable ... }
```

The variables to be input from the file are separated by commas in both the program and the data file. Spaces placed before or after a number or string will have no effect. Spaces placed within a string will be included in the string.

The following is an example of a program line:

```
INPUT #1, N1, N2, N3, S1$, S2$, S3$
```

The following is an example of a line in file to be input:

```
1, 2, 3, First String, Second String, Third String
```

The TstInput.bas (Test Input) program demonstrates the use of INPUT # with files in different formats. In each case, Test Input will input three numbers followed by three strings. Ten different examples will be given of how an IBM PC BASIC file might store the data. The program allows you to select which example will be used, and then presents the results of using INPUT # with that example.

Another version of Test Input is presented for input of the data from the same ten examples into a PICK/BASIC program. Thus the same IBM PC BASIC file, perhaps containing a list of parameters for the operation of a program, can be used in both languages.

Both versions of the Test Input program use the same input files as examples. The examples are presented here. The Test Input program will give the user a choice of examples with the following menu:

CHOOSE FROM EXAMPLES

Without Quotes

1. 1 line, no nulls
2. 6 lines, no nulls
3. 2 lines, null in middle
4. 2 lines, null start, end
5. 3 lines, middle blank

With Quotes

6. 1 line, no nulls
7. 6 lines, no nulls
8. 2 lines, null in middle
9. 2 lines, null start, end
10. 3 lines, middle blank

11. Other

Pick a Number [1]:

The files holding the data are named "EXAM1.DOC" through "EXAM10.DOC" for examples 1 through 10. Number 11—"Other"—would be supplied by the person using the program. The program will ask for the name of the file and the number of separate elements that it contains.

IBM PC BASIC can use three types of delimiters between separate elements:

- 1) A Carriage Return, marking the end of a line
- 2) A comma between elements in a single line
- 3) Quote marks to surround a string, allowing the string to contain embedded commas

The examples are divided into two major categories of five each, according to whether or not quotes are used. Examples 1 through 5 use no quotes, while Examples 6 through 10 do use quotes. Thus Example 1 presents the 6 elements on one line with no null values and no quotes.

Example 1:

1,2,3,First,Second,Third

Quotes must be used to surround strings that contain embedded commas. A string to be input can have embedded spaces, but if the commas are not to be interpreted as delimiters, the entire string must be surrounded by quotation marks.

Example 6:

1,2,3,"String, First","String, Second","String, Third"

The input of one element with INPUT # will continue up to the end of the line, if no comma delimiter is encountered. The next variable will be input beginning with the start of the next line. A separate LINE INPUT # is not necessary to take the end of the line. Thus the same result will be obtained if the data from Example 1 were placed on six separate lines, as in Example 2. Note that the end-of-line substitutes for a comma delimiter, and that therefore both are not used.

Example 2:

1
2
3
First
Second
Third

The same applies when quoted elements are placed on separate lines.

Example 7:

1
2
3

"String, First"
"String, Second"
"String, Third"

A comma delimiter standing alone will cause the input of a null element. Thus in Examples 3 and 8, the elements are divided into two lines. The middle number and the middle string are missing from each line. Instead, the comma delimiting the first element in the line is followed immediately by the comma meant to delimit the second element. As a result, the program reads the second element as null. In the case of numbers, a null is interpreted as zero.

Example 3:

1,,3
First,,Third

Example 8:

1,,3
"String, First",,"String, Third"

If a comma begins a line, the first element will be interpreted as null. If a comma ends a line, the program will read a null final element there. Hence in the example with the elements on six separate lines, if each element had been followed by a comma, the program would have seen not six elements, but 12. Examples 4 and 9 contain null elements at the start and end of each line.

Example 4:

,2,
,Second,

Example 9:

,2,
, "String, Second",

If a line is to contain only a single null, the line should be blank, ending with a Carriage Return. In Examples 5 and 10, the second number will be read as zero.

Example 5:

1
{ Blank line }

3,First,Second,Third

Example 10:

1
{ Blank line }
3,"String, First", "String, Second", "String, Third"

The Test Input program will input three numbers and then three string from each example chosen. In some cases, some of the elements will be null. The program will first display the entire input file (using the LINE INPUT # statement) and then show the results of the INPUT # operation. For Example #1, the program will display the following:

```
Line #1: 1,2,3,First,Second,Third
Numbers:
1 = 1 | 2 = 2 | 3 = 3
Strings:
FIRST = First | Second = Second | Third = Third
```

You also can test your own input file by designating "Other" as the choice for input, and then supplying the name of the file and the number of separate elements that it contains.

TInput Program, Documentation Version Test Input Program: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

```
E$ = Element$()      : Array of separate elements from user-supplied file
EX$ = Example$       : Number of example file, EXAM1 - EXAM10.
F$ = File$
L$ = Line$
TE = Total.Elements  : Total separate elements in file
UF = User.File.Flag  : Input file created by user
```

```
1  '* TInput.bsd: Test Input David L. Clark 06/30/1987
2  '* Input 3 numbers and 3 strings from files "EXAM1" - "EXAM10".
3  '* IBM PC Basic program. Documentation version.
4  '* IBM PC BASICA 3.0, Radio Shack 100
5  '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10 '* *****
   '* Assign Constants, Parameters, Initial Values *
   '* *****
   DEFINT A-Z
   :   DEFSTR X-Z
   '* End line
   '* =====*
20 '* *****
   '* Main Program *
   '* *****
   '* Choose File *
   '* *****
   CLS
22 PRINT"                CHOOSE FROM EXAMPLES"
   :   PRINT
   '* End line
24 PRINT "Without Quotes          With Quotes"
26 PRINT "1. 1 line,  no nulls    6. 1 line,  no nulls"
28 PRINT "2. 6 lines, no nulls    7. 6 lines, no nulls"
```

```

30 PRINT "3. 2 lines, null in middle      8. 2 lines, null in middle"
32 PRINT "4. 2 lines, null start, end    9. 2 lines, null start, end"
34 PRINT "5. 3 lines, middle blank      10. 3 lines, middle blank"
: PRINT
  '* End line
36 PRINT "                                11. Other"
: PRINT
: PRINT "                                Pick a Number [1]:";
  '* End line
40 LINE INPUT Example$
: User.File.Flag = Example$ = "11"      '* User-supplied file
  IF Example$ = "" THEN Example$ = "1"  '* Default choice
  '* End line
42 '* Begin line
: IF User.File.Flag THEN
  PRINT
: PRINT "Enter Filename: ";
: LINE INPUT File$
: PRINT "Enter Number of Elements: ";
: LINE INPUT X$
: Total.Elements = VAL(X$)
: DIM Element$(Total.Elements)
  ELSE
: Total.Elements = 6
: File$ = "EXAM" + Example$ + ".DOC"    '* IBM PC only
: File$ = "EXAM" + Example$ + ".DO"     '* Radio Shack 100 only
  '* End: If user-supplied File
  '* End line
  '* Open file the 1st time
46 OPEN File$ FOR INPUT AS #1
  '*****
50 '* Show File *
  '*****
  '* If the File is empty, will do nothing
  CLS
: I = 0
52 '* Begin line - Top of Loop
  IF NOT EOF(1) THEN
    I = I + 1
: LINE INPUT #1, Line$
: PRINT "Line #"; MID$(STR$(I), 2); ": "; Line$
: GOTO 52
  '* End: If EOF ...
  '* End line
  '*****
60 '* Input Elements *
  '*****
  '* Close and re-open the file to begin input again from the start
  CLOSE
: OPEN File$ FOR INPUT AS #1
: IF User.File.Flag THEN
  FOR I = 1 TO Total.Elements
: INPUT #1, Element$(I)
: NEXT
  ELSE
  '* Input 3 numbers followed by 3 strings
: INPUT #1, N1, N2, N3, S1$, S2$, S3$
  '* End: If User-supplied file
  '* End line

```

```

70  '*****
    '* Show Elements *
    '*****
PRINT
:   IF User.File.Flag THEN
        FOR I = 1 TO Total.Elements
:       PRINT "Element #"; MID$(STR$(I), 2); " = "; Element$(I)
:       NEXT
    ELSE
:       PRINT "Numbers:"
:       PRINT "1 ="; N1; " | 2 ="; N2; " | 3 ="; N3
:       PRINT "Strings:"
:       PRINT "First = "; S1$; " | Second = "; S2$; " | Third = "; S3$
80  '* End line
    '*****
    '* Repeat or Quit *
    '*****
PRINT
:   PRINT "1. Repeat      2. Quit"
:   PRINT "Pick a Number [1]: " :
:   LINE INPUT X$
:   IF X$ <> "2" THEN
        CLOSE
:       GOTO 20          '*|Main Program| - Loop back
    '* End: If Quit
    '* End line
    '*****
9999 '* End Program *
    '*****
END

```

(The executable version of the program for the IBM PC eliminates all of the comments and all of the Carriage Returns within a single program line, as explained in the section: "Introduction: Special Notations for Guide: IBM PC BASIC Documentation Different from Code.")

Pick

The procedures used to search a file in Pick and the usage of the Pick Access processor will be examined in the section on "Searching with Pick Access." This section explains file input from PICK/BASIC in parallel with the methods used by IBM PC BASIC.

PICK/BASIC

READ, READV, MATREAD

An ELSE clause is required with each command that reads from a file. A THEN clause is optional. The syntax is the same as in the IF . . . THEN . . . ELSE conditional test. (See the section: "Conditions.")

To read one item from a file, and assign the entire item to a dynamic array:

```
READ Dyn.Ay FROM File.Data$, Item.ID$ {THEN . . .} ELSE . . .
```

To read one attribute from an item in a file, and assign that attribute to a variable, the **READV** statement includes a reference to the number of the attribute within the item:

```
READV Attrib$ FROM File.Data$, Item.ID$, Att.Num {Then ...} ELSE ...
```

To read one item from a file, and assign each attribute in the item to an element in a fixed array:

```
MATREAD Fixed.Ay FROM File.Data$, Item.ID$ {THEN ...} ELSE ...
```

Attribute References for READV

The **READV** statement can refer to a specific attribute to read either as a literal number or as a variable. Thus, to retrieve the first attribute with a literal number reference:

```
READV Attrib$ FROM File$, Item.ID$, 1 ELSE STOP
```

or with a variable reference:

```
Attrib.Num = 1  
READV Attrib$ FROM File$, Item.ID$, Attrib.Num ELSE STOP
```

READV cannot use a dictionary definition. If the first attribute were defined as “Author” in the dictionary of the data file, it would be illegal to try to reference the attribute by its dictionary name:

```
Dict.Name$ = "Author"  
READV Attrib$ FROM File$, Item.ID$, Dict.Name$ ELSE STOP; * Illegal
```

The program could first make a search through the dictionary of the file for an attribute definition item with an item ID of “Author,” and then extract attribute 2 from the item, to get the attribute mark count, and place the attribute number in a variable called **Attrib.Num**. However, the conversions and correlatives used by the dictionary would not be applied to the attribute. The program would instead retrieve the attribute as it actually existed in the file. Alternatively, the program could also read the conversions and correlatives from the attribute definition item and convert them into BASIC code or apply them through the **ICONV** and **OCONV** statements. (**ICONV** and **OCONV** can be used only with a limited set of correlatives. See the section: “Conversions and Correlatives in Pick.”)

Under the new open architecture version of Pick, direct use of file dictionaries is provided. The program can refer to an attribute by dictionary name, and also employ the conversions and correlatives stored in the dictionary.

Filling a Fixed Array with MATREAD

If the item read in with **MATREAD** contains more attributes than the dimensioned array has elements, the remaining attributes are placed in the last element as a dynamic array. For example:

```
DIM Fixed.Ay$(5)  
MATREAD Fixed.Ay$ FROM File$, Item.ID$
```


If the item had 10 attributes, the first 4 elements of Fixed.Ay\$ would hold 1 attribute each, the last element would hold attributes 5 through 10, each separated by an attribute mark (ASCII 254).

The following READ operation reads an item into a dynamic array. If the item exists, elements of the array are extracted and assigned to simple variables. (For dynamic array extraction, see the section: "Data Structures: Pick: Arrays, Dynamic: Extract.")

If the item does not exist, the program branches to an error handling routine:

```
READ Dyn.Ay$ FROM File.Data$, Item.ID$ THEN
    Name$ = Dyn.Ay$<1>
    Address$ = Dyn.Ay$<2>
END ELSE
    Error.Message$ = "Item: " : Item.ID$ : " not found"
    GOTO 9000;      *|Error Routines|
END
```

If only a single attribute is needed from the item, use READV. Otherwise a single READ or MATREAD to take the whole item is faster than two or more READV statements.

SELECT, READNEXT

For example:

```
SELECT File$ {TO Select.List$}
READNEXT Item.ID$ {FROM Select.List$} {THEN ...} ELSE ...
```

The ELSE clause in the READNEXT, READ, or READV statements is the equivalent of the EOF (End Of File) test in IBM PC BASIC, and is used to ensure that the program will not try to continue reading past the end of the file or item.

To read through an entire file, perhaps a free-form text file rather than a structured database file, you can use SELECT to create a list of all the item IDs in the file. (To use search criteria, rather than selecting all of the items, see the section on "Searching with Pick Access.")

You would then use READNEXT to read the item IDs from the select list one at a time. When the ELSE clause of READNEXT is true, there are no more item IDs on the select list:

```
OPEN File.Text.Name$ TO File.Text$ ELSE STOP
SELECT File.Text$
READNEXT Item.ID$ ELSE STOP
```

The following short program, CREATE.TEXT.FILE, will create a test file for experimentation out of an existing database file. The test file will be limited to 100 items. It will share the dictionary of the database file.

The name of the test file will be the same as the database filename, with "Test." placed at the front of the name.

CREATE.TEST.FILE Program

```
* CREATE.TEST.FILE David L. Clark. 07/11/1987
* Create a file for testing.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: CreatTFi.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* For the modulo, 31 is used. The file is meant to be very temporary.
* It is assumed that the Dictionary level and Data level of the database
* file have the same name. If not, the user should enter both names in
* response to the prompt, separated by a comma.
  PROMPT ""
  EQU True TO 1, False TO 0
  CRT "Enter name of database file:"
  INPUT File.Data.Name$
  OPEN File.Data.Name$ TO File.Data$ ELSE CRT " No such file"; STOP
  File.Test.Name$ = "Test." : File.Data.Name$
  EXECUTE "CREATE-FILE DATA File.Data.Name$, File.Test.Name$ 31"
  OPEN File.Data.Name$, File.Test.Name$ TO File.Test$ ELSE STOP
  SELECT File.Data$
  End.of.File.Flag = False
  FOR I = 1 TO 100 UNTIL End.of.File.Flag DO
    READNEXT Item.ID$ THEN
      READ Item$ FROM File.Data$, Item.ID$ THEN
        WRITE Item$ ON File.Test$, Item.ID$
      END; * Item$ exists
    END ELSE
      End.of.File.Flag = True
    END; * If not yet End of File
  NEXT I
  * Pick For...Next counter has a value at end of loop
  CRT "Total Items in Test file " : File.Test.Name$ : " = " : I
  END
```

You also can create a select list of item IDs and store it for later use, which opens the possibility that the list might be out of date. Some of the items might no longer exist. In that case, as the program reads each item with the READ statement, the ELSE clause will indicate if there are no more items in the file.

When reading a list of item IDs with the READNEXT statement placed in a loop, the individual item IDs can be placed directly into simple variables or elements of a fixed array, but they cannot be placed directly into elements of a dynamic array. Dynamic array placement requires two steps: first to retrieve the item ID, and then to place it. The following examples illustrate placement of a list of item IDs into fixed and dynamic arrays. Note that for placement into a fixed array, the loop must be limited to the dimension of the array.

Fixed Array

```
DIM Fixed.Array$(100)
EQUATE Array.Max TO 100, False TO 0, True TO 1
Quit.Flag = False
FOR I = 1 TO Array.Max UNTIL Quit.Flag
  READNEXT Fixed.Array$(I) ELSE
    Quit.Flag = True
  END
NEXT I
```

Dynamic Array

```
EQUATE False TO 0, True TO 1
Dyn.Ay$ = ""
Quit.Flag = False
I = 0
LOOP UNTIL Quit.Flag DO
    I = I + 1
    READNEXT Item.ID$ THEN
        Dyn.Array$<I> = Item.ID$
    ELSE
        Quit.Flag = True
    END
REPEAT
```

Note that READNEXT does not like to be interrupted. You cannot cut short the READNEXT process by skimming off the first item ID returned and then ignoring the rest.

For example, consider the following routine to take the most recently created select list, stored as an item under a sequential item ID or an item ID set by date and time. The routine uses EXECUTE to perform a descending sort, employing an attribute defined as "Item.ID" for this purpose. The routine retrieves the first item ID presented. Once that item ID is retrieved, the routine cancels the READNEXT process:

```
Command$ = \SSELECT \ : File.Select.Lists.Name$ : \ BY-DSND Item.ID \
EXECUTE Command$ CAPTURING Capture.Output$
READNEXT List.ID$ THEN
    READ Select.List$ FROM File.Select.Lists$, List.ID$
    END ELSE
        NULL
    END; * Read
END ELSE
    NULL
END; * ReadNext
```

This method will cause unexpected results when a second SSELECT and READNEXT is performed on the file. The second operation will retrieve the item IDs remaining from the first. Instead, use the following method to cycle through the unwanted item IDs, while only reading the item from the first:

```
Command$ = \SSELECT \ : File.Select.Lists.Name$ : \ BY-DSND Item.ID \
EXECUTE Command$ CAPTURING Capture.Output$
No.More.Items.Flag = False
Most.Recent.List.Flag = True
LOOP UNTIL No.More.Items.Flag DO
    READNEXT List.ID$ THEN
        IF Most.Recent.List.Flag THEN
            Most.Recent.List.Flag = False;
            * Clear flag
```

```

        READ Select.List$ FROM File.Select.Lists$, List.ID$
        END ELSE
        NULL
    END; * Read
    END; * If 1st Item ID retrieved
END ELSE
    No.More.Items.Flag = True
    END; * ReadNext
REPEAT; * Until no more Item IDs left

```

(For more detail on READNEXT, see the section: “Files, Items: Input: Search with Pick Access Processor.”)

EQUATE

The fastest access to three or more attributes is to use MATREAD to place all the attributes in a fixed (dimensioned) array, with the desired array elements designated as simple variables through an EQUATE statement at the beginning of the program:

```

* Assign Constants
DIM Data$(10)
EQUATE Name$ TO Data$(1), Address$ TO Data$(2)
* Open file, assign Item ID
OPEN File.Data.Name$ TO File.Data$ ELSE STOP
* Input from file
MATREAD Data$ FROM File.Data$, Item.ID$ THEN
    CRT "Name: " : Name$
    CRT "Address: " : Address$
END ELSE
    GOTO 9000;          *|Error Routines|
END

```

In a dimensioned array, the time needed to access any element is the same, regardless of its position in the array. Simple variables provide even faster access, and array elements can become simple variables through the EQUATE statement. Elements in a dimensioned array, such as an item, in which the separate elements are designated by delimiters, take time to access according to how far down the elements are within the dimensioned array. The farther down, the more time it takes to find and count the delimiters before reaching the designated element.

To MATREAD a large item, assigning all of the attributes to array elements, is less efficient than to READ the item, when only a few attributes at the beginning of the item are needed.

READU, READVU, MATREADU

The “U” version of each read statement locks the item (or the group to which the item belongs) until the lock is released. While the lock is in place, no one else can access the item (or group) with the “U” version of the read statements. Note that “Non-U” access is

still legal. It is assumed that access with intent to change will always take place in the “U” mode.

A lock is released automatically when the item, or an attribute of the item, is written back to the file (unless the “U” version of the write statement is used). If no change is to be made to the item, then the RELEASE statement is issued:

```
* Replace Old.Name$ with New.Name$ in a data file
READVU Name$ FROM File.Data$, Item.ID$, Attrib.Num ELSE STOP
IF Name$ = Old.Name$ THEN
    WRITEV New.Name$ ON File.Data$, Item.ID$, Attrib.Num
END ELSE
    RELEASE File.Data$, Item.ID$
END
```

LOCKED

Recent releases of the Pick system include a LOCKED clause in the statements that read file input to give the user trying to access an item an alternative to waiting for the item (or group to which the item belongs) to become unlocked. If no LOCKED clause is included, the user’s terminal will beep until the locked item becomes free.

The LOCKED clause functions as part of the IF . . . THEN . . . ELSE structure of the READ, READV, and MATREAD statements. If LOCKED is true, the commands in the LOCKED clause will be executed. If LOCKED is false, the program will pass to the next statements, which will be the THEN and ELSE clauses. (For IF . . . THEN . . . ELSE structure and syntax, see the section: “Conditions.”)

Examples of the LOCKED Clause

```
READ ITEM$ FROM File$, Item.ID$
LOCKED
    GOSUB 100;    *|Recover from Lock|
END THEN
    GOSUB 200;    *|Edit Item|
END ELSE
    GOSUB 9000;   *|Error Routines|
END
READV Attrib$ FROM File$, Item.ID$, Attrib.Num
LOCKED
    GOSUB 100;    *|Recover from Lock|
END THEN
    GOSUB 300;    *|Edit Attribute|
END ELSE
    GOSUB 9000;   *|Error Routines|
END
MATREAD Array$ FROM File$, Item.ID$
LOCKED
    GOSUB 100;    *|Recover from Lock|
END THEN
```

```

        GOSUB 400;    *|Edit Array|
    END ELSE
        GOSUB 9000;  *|Error Routines|
    END

```

A routine to recover gracefully from a LOCKED condition might resemble the following "Recover from Lock." (For the use of screen constants such as Erase.Screen\$ and Row.Message, see the section: "Screen." For a routine to choose from a list, see the section "Keyboard: Select a Choice from a Menu.")

Recover from Lock Routine

```

*****
* Recover from Lock *
*****
* David L. Clark 09/22/1988
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: RcvrLock.bpr
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Parameters: File.Name$, Item.ID$
* Returns: Locked.Items.List$
* Local variables: Display$, Response$
Erase.Screen.from.Pos$ = @(-3)
Row.Bottom = SYSTEM(3)
Row.Message = Row.Bottom
Display$ = @(0, Row.Message - 5) : Erase.Screen.from.Pos$
Display$ = Display$ : "Item# " : Item.ID$ : " in file " : File.Name$
Display$ = Display$ : " is presently in use. Do you wish to:"
Display$ = Display$ : @(0, Row.Message - 3)
Display$ = Display$ : "1.Continue, placing this Item ID on the"
Display$ = Display$ : " Locked Items List"
Display$ = Display$ : @(0, Row.Message - 2)
Display$ = Display$ : "2.Wait until this Item becomes free"
Display$ = Display$ : @(30, Row.Message) : "Pick a Number [1]: "
CRT Display$ :
INPUT Response$ :
CRT @(0, Row,Message - 5) : Erase.Screen.from.Pos$
IF Response$ # "2" THEN
    INSERT(Locked.Items.List$, -1; Item.ID$)
END ELSE
    Display$ = @(0, Row.Message - 1)
    Display$ = Display$ : "Press ESCAPE if you decide not to"
    Display$ = Display$ : " continue waiting."
    Display$ = Display$ : @(0, Row.Message)
    Display$ = Display$ : "Otherwise the program will wait until the"
    Display$ = Display$ : " Item becomes free"
    CRT Display$ :
    GOSUB 110;                                *|Wait Until Free or Escape|
END
RETURN
*-----*
*****
110 * Wait Until Free or Escape *
*****

```

```

* Parameters: File$, Item.ID$
* Returns: Item$ or Locked.Item.List$
* Local variables:
  * Quit.Flag
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
  READ Item$ FROM File$, Item.ID$
  LOCKED
    IF SYSTEM(14) > 0 THEN;    * A key has been pressed, so quit
      Quit.Flag = True
      INSERT(Locked.Items.List$, -1; Item.ID$)
    END
  END THEN
  * If reach this point, Item has become free
  Quit.Flag = True
  GOSUB 200;                  *|Edit Item|
END ELSE
  Quit.Flag = True
  GOSUB 9000;                 *|Error Routines|
END
REPEAT;                      * Until Quit.Flag
RETURN

```

FILES, ITEMS: INPUT: CONVERT FILE FROM IBM PC BASIC TO PICK

The TEST INPUT program is the PICK/BASIC equivalent of the TstInput.bas program shown in the IBM PC BASIC portion of the section “Files, Items: Input.” The program demonstrates the input of data into a PICK/BASIC program from a file created originally under IBM PC BASIC. Thus a file of data previously created with IBM PC BASIC could be loaded into a Pick database. Very similar methods could be employed to convert database Records from dBASE, R:Base, and other IBM PC DOS database programs into Pick. (For a simpler method of transferring Records from R:Base to Pick, see the section: “Transfer: From IBM PC DOS to Pick.”)

Ten different examples will be given of how an IBM PC BASIC file might store the data. In each case, Test Input will input three numbers followed by three strings. The examples reside in the file “FILE.INPUT,” in 10 different items, named “EXAM-1.DOC” through “EXAM10.DOC.” The program allows you to select which example will be used, and then presents the results of using that example.

Both versions of the Test Input program use the same examples. In Pick, the examples are separate items. In IBM PC BASIC, the examples are separate files. The examples and the operation of the program are shown with the original version of the program in the IBM PC BASIC portion of the section “Files, Items: Input.”

When a file from another system is brought into Pick, Pick treats each line ending with a Carriage Return—Line Feed as a separate attribute, which can be read individually with the READV statement or extracted from the item with dynamic array functions. Thus access to one line at one time is very easy. Pick will not take any data element less than an attribute (line) from a file, unlike IBM PC BASIC, which will take individual numbers or strings within a line.

After reading the item and dividing it into separate lines, Pick’s string functions will be used to extract the separate elements within the line.

(For detailed explanations of the string functions see the section: "Strings: Extract.")

IBM PC BASIC can divide elements into separate lines, or within a single line separate elements with commas. In addition, quote marks can surround a string, allowing the string to contain embedded commas. The Test Input program in PICK/BASIC must allow for all three kinds of delimiters: end-of-line, commas, and quotes. The PICK/BASIC Test Input program contains two separate routines to extract elements from within a line, according to whether or not the routine must adjust for the possibility of embedded commas hidden within quoted strings.

The routine for extracting elements without quote marks is much shorter and simpler. The string function DCOUNT first determines the number of elements to extract from the line, according to the presence of comma delimiters. Then a FOR...NEXT loop using the FIELD function pulls out the elements between the commas.

The DCOUNT function will count as an element, and the FIELD function will extract, an element before the first delimiter on a line and an element after the last delimiter. To IBM PC BASIC, a delimiter at the start of a line signifies a preceding null element, and a delimiter at the end of a line signifies that a null element follows. PICK/BASIC's DCOUNT and FIELD functions make the same interpretation. Thus for a line of:

```
,2,
```

DCOUNT counts three elements. If DCOUNT is used as the basis for the occurrence number in a looping FIELD statement, an initial null element and a final null element will be assigned:

```
Elements.in.Line = DCOUNT(Line$, Delim$)
FOR I = 1 TO Elements.in.Line
    Element$(I) = FIELD(Line$, Delim$, I)
NEXT I
```

If FIELD is given an occurrence number beyond the number of delimiters, a null value is returned, and there is no error. When FIELD is given an occurrence number of three for the line shown above, a null element is returned.

If there is only a single element on a line, IBM PC BASIC would not use a comma delimiter. If the line is not null, DCOUNT will count one element. However, if the line is null, DCOUNT will return zero. To count a null element for the line, an adjustment must be made setting the count to 1:

```
Elements.in.Line = DCOUNT(Line$, Delim$)
* If line null, count it as a null element rather than skipping it.
IF Elements.in.Line = 0 THEN Elements.in.Line = 1
FOR I = 1 TO Elements.in.Line
    ...
NEXT I
```

Note here an important difference between a FOR...NEXT loop in Microsoft BASIC on the Radio Shack 100, and IBM PC BASIC as well as PICK/BASIC. On the Radio

Shack 100, a FOR...NEXT loop would execute once even if the starting value was greater than the ending value, thus a loop beginning with the statement "FOR I = 1 TO Elements.in.Line" would execute even if Elements.in.Line were zero. For the other forms of FOR...NEXT such a loop would not execute.

If any of the elements are strings surrounded by quote marks, the quote marks must be deleted, and a check must be made whether such strings contain embedded commas. In the routine that extracted elements without quote marks, within each line the index value for the Element\$() array matched the occurrence number parameter for the FIELD statement, with the addition of the count of elements previously extracted from the item. If embedded commas lie within quoted strings, the two numbers will not match so neatly, and a more elaborate extraction process must be performed. Note that quote marks will appear correctly only in pairs and that a single quoted string can contain several commas.

When extracting quoted strings, one must allow for the possibility that some of the elements are null, and not end the operation if the extraction returns a null value. Instead, the operation will end after it has reached the end of the line, and null values will be assigned to the corresponding delimited elements.

IBM PC BASIC does not require that a space separate a comma and the element that follows, as would be the case in normal English punctuation. Such a space might have been inserted, however, if a user created the file directly. Furthermore, when IBM PC BASIC prints an integer, whether to the screen, to a printer, or to a file, an extra space follows the number, and an extra space precedes a positive number. Any such extra spaces would be removed by PICK/BASIC's TRIM function. TRIM also removes any repeating spaces, reducing any sequence of two or more spaces to a single space. Such extra spaces might have been included in a quoted title or page heading for formatting purposes.

No unwanted spaces are to be expected within a quoted string, assuming that the IBM PC BASIC program that created the file correctly removed the extra spaces around numbers when converting them to string format. The quoted string form would therefore allow extra spaces for formatting purposes. The routine will still exclude any extra spaces occurring before the first quote mark from elements placed in the same line, in case a user has created the file directly. (For conversion of numbers to strings, see the section: "Numbers: Convert: Number to String.")

The program will read both numbers and strings from the item into a string array, which is allowed by Pick's lack of typing for variables. In the routine "Show Elements," each of the elements that is supposed to be a number is tested with the NUM function, and then forced to display as a number by adding zero. Null is thus interpreted as zero. Prior initialization will not force null to show as zero.

The routine "Get Elements" calls one of the two subroutines that actually extract the elements from each line. All the parameters that the subroutine needs it gets from the caller. The only values created by the subroutine that are retained after the subroutine ends are those that the subroutine returns to the caller, designated as "Receives from Subroutine" in the documentation under the caller's heading. This method of handling parameters corresponds to the structure of Pascal.

The caller specifies what delimiter is used between the elements. Thus the same extraction routines could be used with different delimiters.

TEST.INPUT Program

```

* TEST.INPUT David L. Clark 11/20/1987
* Input from IBM PC Basic File to Pick.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TInput.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Input 3 numbers and 3 strings from file "FILE.INPUT", with a choice
* of 10 example Items numbered "EXAM1.DOC" through "EXAM10.DOC",
* or input an Item created by the user.
*****
* Assign Constants, Parameters, Initial Values *
*****
PROMPT ""
EQU Attrib.Mark$ TO CHAR(254)
EQU True TO 1, False TO 0
EQU Quote$ TO CHAR(34)
* Assume maximum of 20 Elements
DIM Element$(20)
* Pick's lack of typing for variables allows numbers to be taken
* into a string array.
EQU N1 TO Element$(1), N2 TO Element$(2), N3 TO Element$(3)
EQU S1$ TO Element$(4), S2$ TO Element$(5), S3$ TO Element$(6)
Clear.Screen$ = @(-1)
File.Name$ = "FILE.INPUT"
Quit.Flag = False
=====
*****
* Main Program *
*****
OPEN File.Name$ TO File$ ELSE STOP
LOOP UNTIL Quit.Flag DO
    *****
    * Choose Item *
    *****
    CRT Clear.Screen$
    CRT "                CHOOSE FROM EXAMPLES"
    CRT
    CRT "Without Quotes                With Quotes"
    CRT "1. 1 line, no nulls            6. 1 line, no nulls"
    CRT "2. 6 lines, no nulls            7. 6 lines, no nulls"
    CRT "3. 2 lines, null in middle        8. 2 lines, null in middle"
    CRT "4. 2 lines, null start, end        9. 2 lines, null start, end"
    CRT "5. 3 lines, middle blank          10. 3 lines, middle blank"
    CRT
    CRT "                11. Other"
    CRT
    CRT "                Pick a Number [1]: " :
    INPUT Example$
    User.Item.Flag = (Example$ = "11");
    IF Example$ = "" THEN Example$ = "1";
    IF User.Item.Flag THEN
        CRT
        CRT "Enter Item ID: " :
        INPUT Item.ID$
        CRT "Enter Number of Elements: " :
        INPUT Total.Elements

```

```

END ELSE
    Item.ID$ = "EXAM" : Example$ : ".DOC"
    Total.Elements = 6
END; * If user-supplied Item
READ Item$ FROM File$, Item.ID$ ELSE
    CRT "No such Item: " : Item.ID$
    STOP
END; * Read Item$
Total.Lines = DCOUNT(Item$, Attrib.Mark$)
*****
* Show Item *
*****
CRT
* If the Item is empty, will do nothing
FOR Current.Line = 1 TO Total.Lines
    Line$ = Item$<Current.Line>
    CRT "Line #" : Current.Line : " : " : Line$
NEXT Current.Line
*****
* Get Elements *
*****
* Parameters: Item$, Total.Lines
* Returns: Element$(), Elements.Counted
* Sends to subroutine: Delim$, Elements.Counted, Line$
* Receives from subroutine: Element$(), Elements.in.Line
* Local variables:
*   Current.Line : Line counter
*   Quotes.Flag : Specify whether method handles quotes
*   Num.Choice$ : Input from user
Delim$ = ","
CRT
CRT "Use method that will adjust for Quotes?"
CRT "      1.No      2.Yes"
CRT "      Pick a Number [1]:" :
INPUT Num.Choice$
Quotes.Flag = (Num.Choice$ = "2")
Elements.Counted = 0
* If the Item is empty, will do nothing
FOR Current.Line = 1 TO Total.Lines
    Line$ = Item$<Current.Line>
    IF Quotes.Flag THEN
        GOSUB 200;          *|Extract Elements with Quotes|
    END ELSE
        GOSUB 100;          *|Extract Elements|
    END
    Elements.Counted = Elements.Counted + Elements.in.Line
NEXT Current.Line
*****
* Show Elements *
*****
CRT
IF User.Item.Flag THEN
    FOR I = 1 TO Total.Elements
        CRT "Element #" : I : " = " : Element$(I)
    NEXT I
END ELSE
    CRT "Numbers:"
    * Force numbers to display as numbers, including Null as Zero

```

```

FOR I = 1 TO 3
  IF NUM(Element$(I)) THEN;      * Test for number. Null = 0.
    Element$(I) = Element$(I) + 0
  END ELSE
    CRT "Element #" : I : "is not numeric: " : Element$(I)
    STOP
  END; * If a number
NEXT I
CRT "1 = " : N1 : " | 2 = " : N2 : " | 3 = " : N3
CRT "Strings:"
CRT "First = " : S1$ : " | Second = " : S2$ : " | Third = " : S3$
END; * If a user-supplied Item
CRT "Elements counted = " : Elements.Counted
*****
* Repeat or Quit *
*****
* Local variables:
*   Num.Choice$ : Input from user
CRT
CRT "1. Repeat      2. Quit"
CRT "Pick a Number [1]:" :
INPUT Num.Choice$
Quit.Flag = (Num.Choice$ = "2")
REPEAT; * Until Quit.Flag
GOTO 9990;

```

100

```

*****
* Extract Elements *
*****
* Parameters: Delim$, Elements.Counted, Line$
* Returns: Element$(), Elements.in.Line
* Local variables:
*   Finish      : Last element to assign
*   I            : Counter for current Element$() within array
*   J            : Counter for current Element$() within Line$
*   Start        : First element to assign
* Assume no quoted strings with embedded delimiters.
* If no embedded delimiters within quotes, then the number of
* delimiters matches the number of elements. DCOUNT will give the
* number of elements, which can be taken with FIELD in a FOR...NEXT
* loop, rather than using slower string extraction.
Elements.in.Line = DCOUNT(Line$, Delim$)
* If line null, count it as a null element rather than skipping it.
IF Elements.in.Line = 0 THEN Elements.in.Line = 1
Start = Elements.Counted + 1;      * Current element within array
J = 1;                             * Current element within line
Finish = Elements.Counted + Elements.in.Line
FOR I = Start TO Finish
  Element$(I) = TRIM(FIELD(Line$, Delim$, J))
  J = J + 1
NEXT I
RETURN

```

200

```

*****
* Extract Elements with Quotes *
*****
* Parameters: Delim$, Line$, Elements.Counted
* Returns: Element$(), Elements.in.Line
* Local variables:

```

```

*   I                      : Counter for current Element$() within array
*   Length                 : Length of Line$
*   Num.Chars              :
*   Occur.Num              : Count of elements in Line$
*   Pos.After.Element      : Pos.Start for next Element$() or end of line
*   Pos.Q.in.Element       : Position of 1st quote within Element$()
*   Pos.Q1                 : Position of the element's begin quote
*                           within Line$
*   Pos.Q2                 : Position of the element's end quote
*                           within Line$
*   Pos.Start              : Start position for element
* Allow for quoted strings with embedded delimiters
* Note that position within Element$() and position within Line$
* are not the same. The Element$() might start at position 21
* within the Line$. Thus position 1 within the Element$() would
* correspond to position 21 within the Line$. When quote marks are
* present, the Element$() must be extracted from the Line$ by the
* Line$ position.
I = Elements.Counted + 1
Elements.in.Line = 0
Length = LEN(Line$)
IF INDEX(Line$, Delim$, 1) = 0 THEN;          * No delimiter
    * Take all of the line, even if null
    Elements.in.Line = 1
    * If element in quotes, trim off quotes, but not extra spaces
    IF Element$(I)[1, 1] = Quote$ THEN
        Element$(I) = Element$(I)[2, Length - 2]
    END ELSE
        * Trim off beginning, trailing or extra space
        Element$(I) = TRIM(Line$)
    END
END ELSE;                                     * Delimiter found
    Occur.Num = 0
    Pos.After.Element = 0
    LOOP UNTIL Pos.After.Element > Length DO
        Occur.Num = Occur.Num + 1
        * Don't trim spaces until get positions
        Element$(I) = FIELD(Line$, Delim$, Occur.Num)
        Pos.Q.in.Element = INDEX(Element$(I), Quote$, 1)
        IF Pos.Q.in.Element = 0 THEN;         * Not a quoted string
            * Trim off beginning, trailing or extra space
            Element$(I) = TRIM(Element$(I))
            Pos.After.Element = COL2()
        END ELSE;                             * Quote found
            * Get the positions within Line$, not just in the
            * Element$() first extracted, which may have been
            * truncated by an embedded delimiter within quotes
            Pos.Q1 = COL1() + Pos.Q.in.Element
            Pos.Q2 = Pos.Q1 + INDEX(Line$[Pos.Q1+1, Length], Quote$, 1)
            IF Pos.Q2 = 0 THEN;                * Missing End Quote
                CRT "Error in File " : File.Name$
                CRT "Begin Quote without an end: " : Line$
                STOP
            END ELSE
                * Take the chars in Quotes, excluding Quote Marks.
                * The element may be a null between Quotes.
                * Don't trim any extra spaces in the string.
                Pos.Start = Pos.Q1 + 1
                Num.Chars = Pos.Q2 - Pos.Q1 - 1
            END
        END
    END

```

```

        Element$(I) = Line$(Pos.Start, Num.Chars]
        * Account for commas embedded in the quoted string
        Occur.Num = Occur.Num + COUNT(Element$(I), Delim$)
        * The comma delimiter comes after the 2nd Quote.
        * Put next Pos.Start after the delimiter.
        * Assume no extra spaces after 2nd Quote.
        Pos.After.Element = Pos.Q2 + 1
    END; * If missing 2nd quote
    END; * If Pos.Q.in.Element = 0
    Elements.in.Line = Elements.in.Line + 1
    I = I + 1
    Pos.Start = Pos.After.Element
    REPEAT; * Until Pos.After.Element > Length
END; * If a delimiter in the line
RETURN
=====
*****
9990 * End Program *
*****
END

```

FILES, ITEMS: INPUT: ORDER OF ACCESS

Because Pick's variable-length item structure is far superior for database storage to the random access files of IBM PC BASIC, which are composed of fixed-length records, the discussion of random access will be restricted to Pick. Simpler versions of BASIC, such as that on the Radio Shack Model 100, generally provide sequential access only.

Pick

Pick has only one type of file structure. Items in a Pick file are accessed in a random manner, according to the groups into which their Item.IDs hash. When an item is changed, it is written back to the end of the group. To access the items in a particular manner from PICK/BASIC, a sorted list of item IDs should be created, and the items accessed in the order of their item IDs on the list, using the READNEXT statement with a select list. (For Select Lists and READNEXT, see the section: "Files, Items: Input.")

For such a sorted list to be useful, the item IDs must be assigned and sorted in such a manner to reflect the desired sequence. A file of Library of Congress subject terms could simply use the terms as the item IDs, and then perform an alphabetical (left to right) sort. A data file, on the other hand, might use a series of numbers for Bibliography#, Part#, Subpart#, Entry#.

Numbers require a right to left sort. The type of sort performed on the item IDs in a file is determined by attribute 9 of the data file definition item, which is an item in the dictionary level of the file, with an item ID made up of the filename. Attribute 9 determines the justification used for displaying the item IDs. For a right to left sort, attribute 9 should contain "R" for right justification rather than "L" for left. The following sequence of item IDs was obtained with the SORT verb and right justification:

```

1-1-1-1
1-1-1-2
1-1-1-1000
1-2-1-1
1-5x7-N-1
1-5x7-N-2
1-5x7-N-1000
1-8x10-N-1
1-8x10-N-2
1-8x10-N-1000
1-35mm-1-1
1-35mm-1-2
1-35mm-1-1000

```

with left justification:

```

1-1-1-1
1-1-1-1000
1-1-1-2

```

FILES, ITEMS: INPUT: SEARCH WITH PICK ACCESS PROCESSOR

To search a sequential file in IBM PC BASIC, you must input each data item and examine its contents. Random access file procedures under IBM PC DOS have not been examined in this book because the Pick operating system is far superior for the purpose of storing and maintaining a database. You could not duplicate the database features of Pick in IBM PC DOS even if this book were the length of an encyclopedia.

Pick TCL

LIST, SORT, SELECT, SSELECT

These commands are used to find the items whose data meets specific conditions. Commands entered at Pick TCL level that might have conditions attached are called Access commands. Access commands, like all TCL commands, can be used from a PICK/BASIC program with the EXECUTE statement, as will be explained shortly. The Pick Access processor, including display options, will be described in this section.

All Access commands have the following format:

```

Verb Filename      { 'Item.ID' { 'Item.ID' ... } } { BY Sort instructions }
                  { WITH Attrib.Name Operator "Value" } { Attrib.Names to output }
                  { Modifiers } { (Options) }

```

LIST and SORT will print or display the items, SORT doing so in sorted order. SELECT and SSELECT will create a select list of the item IDs of the items that met the conditions, SSELECT doing so in sorted order, according to the sorting instructions. (For sorting, see the section: "Files, Items: Input: Order of Access.")

The following is a simplified review of Access terminology.

Search Command Format

Command	File	WITH	Field-name	Operator	"Object"
SELECT	PHOTOS	WITH	PHOTOGRAPHER	=	"Adams, Ansel"

To sort by item ID use the command SSELECT (for Sort-Select) rather than SELECT. To sort by fields include BY Field, such as: BY PHOTOGRAPHER BY PHOTO-DATE. For several conditions use AND or OR. AND links conditions, OR separates them. Repeat WITH field name for different fields and for AND = tests on the same field.

Examples:

For Field ENTRY# between 2 and 5:

SELECT PHOTOS WITH ENTRY# > "2" AND < "5"

For Field ENTRY# 2 or 5:

SELECT PHOTOS WITH ENTRY# = "2" OR = "5"

For harbor and ships in same photo:

SELECT PHOTOS WITH SUBJECTS = "Harbors" AND WITH SUBJECTS = "Ships"

For harbor photos, but only in Long Beach*:

SELECT PHOTOS WITH SUBJECTS = "Harbors" AND WITH CITY = "Long Beach"

For harbor photos, but only outside of Long Beach*:

SELECT PHOTOS WITH SUBJECTS = "Harbors" AND WITH CITY # "Long Beach"

For photos of architecture, sorted by city and then by date:

SSELECT PHOTOS WITH SUBJECTS = "Architecture" BY CITY BY PHOTO-DATE

To find by ITEM-ID use right single quotes, do not use WITH, AND, OR.

For ITEM-IDs 1, 3, and after 10:

SELECT PHOTOS '1' '3' > '10'*

Operators to test individual fields in an item:

Equals:	WITH Field	=	"Object"
Not Equal:	WITH Field #		"Object"
Empty:	WITHOUT Field		

Not Empty:	WITH FIELD	
Contains:	WITH FIELD =	" [Object] "
Not Contains:	WITH Field #	" [Object] "
Begins with:	WITH Field =	"Object" "
Ends with:	WITH Field =	" [Object]"
Comes Before:	WITH Field <	"Object"
Before or Equal:	WITH Field < =	"Object"
Comes After:	WITH Field >	"Object"
After or Equal:	WITH Field > =	"Object"

To use a NOT operative in a Pick Access statement, the "NOT" should precede the attribute name:

```
SELECT History.Data WITH Subjects "Automobiles"
AND WITH NOT Name "Ford Motor Co." "General Motors, Inc."
```

Those items with automobiles as a subject, but not including cars by Ford or General Motors, will be selected. Note that the "=" operator and the "OR" connective are assumed, and do not have to be expressly included.

Attribute Definition Item

The Access commands are dictionary driven in the sense that the words that make up the commands (other than the data values that you are looking for) must be contained in dictionaries. Thus when you enter the command:

```
LIST Bibliog.File WITH Author = "McWilliams, Carey"
```

The word "Author" has been defined in the attribute definition item of the dictionary of the file Bibliog.File as the name of an attribute. All of the words in the command are contained in dictionaries except for the author's name, Carey McWilliams.

Controlling and Dependent Relationships

Attribute 4 of the attribute definition item indicates controlling and dependent relationships between attributes. If data attribute 7 controls data attributes 8 and 9, then dictionary attribute 4 in the attribute definition item for data attribute 7 will contain the following:

```
04 C;8,9
```

Dictionary attribute 4 for data attributes 8 and 9 will contain the following:

```
04 D;7
```

The controlling and dependent relationships would be employed in a print limiting clause in an exploding sort. For example, the Lockheed Aircraft Company has its headquarters in Burbank, but factories at many locations. If different photographs have been cataloged into a single item, using location as a multivalued attribute, and you wish to dis-

play information only for the photos of Lockheed's Burbank facility, the search statement is:

```
SORT Photos BY-EXP Location
  WITH Subjects = "Aeronautics"
  AND Subjects = "Factories"
  AND Location = "Burbank"
  Name Date Location = "Burbank"
```

Conversions and Correlatives

Conversions and correlatives in the attribute definition items of the dictionary level of the data file can play a role in searching. Attribute 7 of the attribute definition item stores conversion codes, attribute 8 stores correlative codes. Conversions will convert the values within quotes in an access statement to internal format, and later convert after processing from internal to external format, to produce a more readable output for the user. Correlatives convert before processing, to make some change for use by the search statement.

For example, a translate from file correlative can be used to access related information in a different file or to extract a specific value within an attribute. The correlative value "MCU" (Mask Character Uppercase) would convert the data to uppercase for searching purposes.

(For more information on conversions and correlatives, see the section: "Conversions and Correlatives in Pick.")

SAVE-LIST, GET-LIST, COPY-LIST, EDIT-LIST, DELETE-LIST

The default file for storing select lists is the POINTER-FILE of the current account. However, another file can be specified in any of the commands handling select lists. The file must have "DC" in the first attribute of its file definition item in the master dictionary. (For file definition items, see the beginning of the section: "Files, Items.")

Immediately after a SELECT or SSELECT statement, to create a select list for later use under a chosen list name:

```
SAVE-LIST {Filename} List.Name
```

To retrieve the list for use:

```
GET-LIST {Filename} List.Name
```

To take the list into the Pick Editor:

```
EDIT-LIST {Filename} List.Name
```

To make a copy of the list:

```
COPY-LIST {Filename} List.Name {(Options)}
```

The same options are available as for the COPY verb:

D = Delete the original Select List after the copy
P = Copy to the Printer
T = Copy to the Terminal

If copy to the terminal is not specified, the system will prompt:

TO:

Respond by providing either a new filename or a list name for the second list:

TO: {(New.Filename) { New.List.Name}

To delete the list:

DELETE-LIST {Filename} List.Name

QSELECT

For example:

QSELECT Filename 'Item.ID' { 'Item.ID' } {(Attrib.Num)}

Use QSELECT to create a select list from the attributes in one or more items or, if an attribute number is specified, from the values in a multivalued attribute. An asterisk in the place of the item IDs will select all of the items.

Index Files, QSELECT

For example:

QSELECT File.Index 'Item.ID' { 'Item.ID' } {(Attrib.Num)}
LIST File.Data

The use of index files can greatly speed searching. QSELECT is used to create a select list from a list of item IDs held in an index file. The item IDs can be stored one per attribute, or all in the same attribute, as separate values. Items are then retrieved from a data file according to the item IDs on the select list.

For example, the index file "Subjects.Index" contains a separate item for each subject term used in the "History.Data" file. Attribute 1 of each item in the Subjects.Index file contains a list of the item IDs of the items in the History.Data file in which the term appears in the attribute that has been defined as "Subjects." Thus if the item ID in the Subject.Index file is "Harbors," attribute 1 will contain a list of the History.Data items having to do with "Harbors."

To retrieve all of the History.Data items that contain the subject term "Harbors":

```
QSELECT Subjects.Index 'Harbors' (1)
LIST History.Data
```

The indexed search will be much faster than asking:

```
SORT History.Data WITH Subjects = "Harbors"
```

To retrieve both "Harbors" and "Ships":

```
QSELECT Subjects.Index 'Harbors' 'Ships' (1)
LIST History.Data
```

In the last example, some items will be retrieved twice, because they contain both the subject terms "Harbors" and "Ships." A PICK/BASIC routine should be used to delete duplicate item IDs when merging select lists, which will be easier if the lists are kept in order with the LOCATE statement. Keeping the select list in order will also cause the items to be retrieved in item ID order without having to use the SORT command. (For LOCATE, see the section: "Data Structures: Pick: Dynamic Arrays: Locate.")

The select list also can be used as the basis for a further search. For example, if it is known that in the file History.Data all items that contained "Todd Shipyards" in the names attribute also would contain "Harbors" in the subjects attribute, the index to harbors could be used to shorten the search:

```
QSELECT Subjects.Index 'Harbors' (1)
LIST History.Data WITH Names = "Todd Shipyards"
```

Mixing Implicit and Explicit Item ID Lists

An implicit item ID list is created whenever a selection verb is used that activates a select list rather than causing an immediate action. These verbs are: SELECT, SSELECT, QSELECT, and GET-LIST. For example, if a SELECT statement is followed by a LIST statement, LIST uses the implicit item ID list formed by the SELECT.

An explicit item ID list is formed by specifying particular item IDs. For example:

```
LIST File.Data '1' '2' '3'.
```

The items with IDs 1, 2, and 3 will be listed.

If a SELECT statement is issued, followed by a LIST statement that includes selection criteria, those criteria will only be applied to the items chosen by the previous SELECT. The LIST only operates on the implicit item ID list, not on the entire file. For example:

```
SELECT   File.Data WITH Names   = "Hornblower, Horatio"
LIST     File.Data WITH Subjects = "Ships"
```

The only ship information shown will concern Captain Horatio Hornblower.

But what if the LIST statement included an explicit item ID list rather than selection criteria? The Pick IBM PC/XT R83 2.0 manual contradicts itself on this point. On page

187 the manual states that the implicit list will over-ride the explicit, but on page 191 the manual claims the opposite, stating that the previous selection will be ignored. The latter version is correct. Thus if the following two statements were issued:

```
SELECT File.Data WITH Subjects = "Ships"  
LIST      File.Data '1' '2' '3'
```

items with IDs "1," "2," and "3" would be listed regardless of any connection with ships. The preceding selection would be ignored.

To combine explicit item ID lists with selection criteria, the two should be combined into one statement, or the explicit list must come first. For example:

```
SELECT File.Data WITH Subjects = "Ships"  
LIST      File.Data '1' '2' '3'
```

or:

```
SELECT      File.Data '1' '2' '3'  
LIST        File.Data WITH Subjects = "Ships"
```

Only the items with IDs of "1," "2," or "3" will be examined for any connection with ships.

Although the QSELECT verb creates an implicit item ID list, the verb operates in the same manner as explicit item ID list formation. QSELECT creates a list of item IDs directly from existing attributes or values, it does not make a search of a file to apply selection criteria. Thus, if the item named "List.123" in the file "File.Lists" contained the values "1," "2," and "3" in the first three attributes, the effect of the command:

```
QSELECT File.Lists List.123
```

would be to select the items with IDs of "1," "2," and "3," regardless of any prior selection. QSELECT must be treated in the same manner as an explicit item ID statement. QSELECT must precede a selection statement, rather than follow it, for example:

```
QSELECT File.Lists List.123  
LIST      File.Data WITH Subjects = "Ships"
```

If, from within a BASIC program, you wished to employ stored select lists without caring about distinctions between implicit and explicit item ID lists, then you could READ the stored list from a file, place single quote marks around each item ID, and add the list to the selection statement.

If the selection command had already been issued, then you would have to take the less efficient method of treating the list of item IDs as a selection criterion. The statement:

```
SELECT File.Data '1' '2' '3'
```

uses an explicit item ID list, but with the insertion of the equals sign:

```
SELECT File.Data = '1' '2' '3'
```

A search of the file has become necessary. This method can be several times slower than the use of an explicit list, but it avoids the problem of an explicit list canceling out a previous implicit list.

Before employing the less efficient search method, you might use the SYSTEM(11) function to test whether an implied list was actually in effect:

```
List.Active.Flag = SYSTEM(11)
```

The following TEST.SELECT.ID program mixes implicit and explicit item ID lists.

TEST.SELECT.ID Program

```
* TEST.SELECT.ID David L. Clark 02/01/1988
* Mix implicit and explicit Item Lists
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TSelectID.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* The first method, using the explicit list first, took 2 seconds.
* The second method, using the selection first, took 6 seconds.
*****
* Assign Constants *
*****
PROMPT ""
EQUATE Attrib.Mark$ TO CHAR(254)
*****
* Get Parameters from User *
*****
* User provides data filename, lists filename, and list ID,
* and the condition for a search.
CRT "File data name: " :
INPUT File.Data.Name$
CRT "File lists name: " :
INPUT File.Lists.Name$
OPEN File.Lists.Name$ TO File.Lists$ ELSE STOP
CRT "List ID: " :
INPUT List.ID$
CRT "Condition for search: " :
INPUT Condition$
*****
* Explicit List First, Then Selection *
*****
Time.Begin = TIME()
* Retrieve stored Select List
EXECUTE \QSELECT \ : File.Lists.Name$ : \ \ : List.ID$
* Selection command
EXECUTE \SELECT \ : File.Data.Name$ : Condition$
Time.End = TIME()
CRT
CRT "Time passed = " : Time.End - Time.Begin : " seconds"
CRT "Press ENTER: " :
INPUT Response$
```

```

CRT
* Show Items
EXECUTE \LIST \ : File.Data.Name$
*****
* Selection First, Then Explicit List *
*****
Time.Begin = TIME()
* Selection command
EXECUTE \SELECT \ : File.Data.Name$ : Condition$
* Retrieve Stored Select List
READ Item.List$ FROM File.Lists$, List.ID$ ELSE STOP
* Insert single quotes between Item IDs
Total.Elements = DCOUNT(Item.List$, Attrib.Mark$)
List$ = ""
FOR I = 1 TO Total.Elements
    List$ = List$ : '\' : Item.List$<I> : '\' \
NEXT I
* Execute command with equals sign to make list a selection
Search.Command$ = \SELECT \ : File.Data.Name$ : \ = \ : List$
EXECUTE Search.Command$
CRT
Time.End = TIME()
CRT
CRT "Time passed = " : Time.End - Time.Begin : " seconds"
CRT "Press ENTER: " :
INPUT Response$
CRT
* Show Items
EXECUTE \LIST \ : File.Data.Name$
END

```

Default Attribute List

Rather than specifying which attributes are to be displayed each time at TCL level, it is convenient to keep a default attribute list for display. When accessing a file with LIST or SORT, Pick will show a default attribute list of those attributes numbered continuously from 1. If there are attributes with item IDs of 1, 2, 3, 4, 5, and 7, the first 5 attributes will be shown automatically. Such a list is created by building synonym attribute definition items in the dictionary level of the file with item IDs beginning with the number "1" and continuing without a break in sequence until the end of the list. The attribute described under item ID "1" will be displayed as the first in the number, number "2" will be second, and so on.

The number of an attribute in terms of its storage in a data file item does not have to correspond to that attribute's item ID in the dictionary level of the file. Thus, if the attribute for storing names is number 12 in the data item, the attribute can be made to display as number 3 in the default attribute list. A synonym attribute definition item would be placed in the dictionary of the file under the item ID "3." The item would contain the following attributes:

```

01 A
02 12      { Attribute Mark Count in the data Item }

```

03 Names

01

To hold a place in the continuous default attribute list as some attributes are removed, create dummy attribute definition items with an "X" rather than an "A" in attribute 1, which will cause the attribute defined not to display. If attribute 5 is removed from the data structure, a dummy "5" is created:

Item ID: 5

01 X

(For more details on attribute definition items, see the beginning of the section: "Files, Items.")

The remainder of the attribute definition item is not required.

Access Display Options

To place the attribute name or column heading on two separate lines for display or printing, insert a value mark (ASCII 253) in attribute 3 of the attribute definition item. The value mark is inserted with the Editor by pressing CTRL-]. On the screen the value mark will appear as a right bracket. For example:

03 Date of]Entry

will cause a column display of:

Date of
Entry

08/06/87

If the column heading (attribute 3 in the attribute definition item) is not as wide as the display width (attribute 10 in the attribute definition item) then Access will fill out the space with dots. To avoid the dots, add spaces to the column heading. You cannot see the spaces when inserting them with the Editor.

Justification

Attribute 9 of the attribute definition item in the file dictionary controls justification and the sorting order of the attribute when it is displayed by access. The following choices are available:

- ☐ R Right justify, for numbers
- ☐ L Left justify, without word wrapping

- T Text, to left justify with word wrapping on a space
- U Left justify, with display continuing across the screen rather than wrapping.

BREAK-ON

To break on an attribute without actually displaying that attribute, create a special break attribute. In its attribute definition item, in attribute 3, for display name, put just a value mark. In attribute 10, for column width, put a zero.

Differences in Search Results among TCL, Editor, and PICK/BASIC

The various processes available in Pick—the Terminal Control Language (TCL), the Editor, and the BASIC programming language—all must perform searches in order to retrieve items from files. Occasionally, search results vary according to the process used, when different processes use different methods.

The TCL Access statements LIST, SELECT, and T-DUMP perform a sequential search. The command EDIT, used to take an item into the Editor, and the PICK/BASIC READ statement, use Pick's hashing algorithm to find the item according to its item ID. If for some reason an item is in the wrong group, EDIT and READ will look in the correct group and fail to find the item. However, LIST, SELECT, and T-DUMP will find it. To correct the problem, T-DUMP the entire file and then T-LOAD it again. (For T-DUMP and T-LOAD, see the section: "Files, Items: Copy, Backup.")

***A9998**

For example:

```
SORT History.Data WITH Names = "Ford Motor Co." Description *A9998
```

A running count of items accessed can be obtained from attribute 9998 in the same manner that the size of each item can be retrieved from attribute 9999. Within an Access statement such as LIST or SORT the running count can be displayed by placing "*A9998" on the output list. Pick automatically places an attribute definition item in each file with an item ID of "*A9998" to access the system running count attribute. Alternatively, an attribute with a name such as "Running.Count" could be created to access the same information:

```
Item ID: Running.Count
01 A
02 9998
03 Running.Count
```

PICK/BASIC

ICONV

For example:

```
ICONV(Search.Value$, "MCU")
```

Values specified for searching can be converted to Pick's internal format with the ICONV statement. In the previous example, the value entered by the user is converted to all uppercase characters with the "MCU" (Mask Character Uppercase) code. For dates and times, ICONV will change the values entered into Pick's internal numerical format. The data values in the file can be subject to conversion and correlative codes contained in their attribute definition items.

EXECUTE, READNEXT

Use the EXECUTE statement to employ Access commands such as SSELECT from a PICK/BASIC program. The following "Search File, Input Items" routine will search a file for items that meet certain conditions, then input only those items in sorted order according to the item IDs. The list of items will be saved for future use. The READ statement should not encounter any nonexistent items from a current search, but might if a select list was used that had not been updated recently. The Object\$ for searching is surrounded with backslashes to allow the use of double and single quotes within the Object\$. The backslashes in turn are surrounded with double quotes. The use of the CAPTURING clause suppresses display of the system message for the number of items selected.

Search File, Input Items Routine

```
*****
* Search File, Input Items *
*****
* David L. Clark 09/22/1988
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: SrchFile.bpr
EQUATE True TO 1, False TO 0
OPEN Dict$, File.Data.Name$ TO File.Data$ ELSE STOP
Command$ = \SSELECT\ : File.Data.Name$ : \ WITH \
Command$ = Command$ : Attrib.Name$ : Operator$ : "\" : Object$ : "\""
EXECUTE Command$ CAPTURING Output$
EXECUTE \SAVE-LIST Select.List\; * May use list again later
EXECUTE \GET-LIST Select.List\
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
    READNEXT Item.ID$ THEN
        READ Item$ FROM File.Data$, Item.ID$ THEN
            GOSUB 100; *|Data Editing Routines|
        END ELSE
            CRT "No Item for: " : Item.ID$; * Item does not exist
        END; * Read
    END ELSE
        Quit.Flag = True; * Item list ended
    END; * Readnext
REPEAT; * Until Quit.Flag
```

The TEST.SELECT.LIST program will create a select list with the SSELECT command, and then store the list as an item.

The select list will hold the item IDs of all the attribute definition items in the MD. The list will exist in the POINTER-FILE under the name "MD.ATTRIBS.PTR." At this point the list is in pointer format, not item format.

The COPY-LIST verb is then used to copy the list to a regular file (not a pointer file) named "SELECT.LISTS." Because a filename was specified, the COPY-LIST process will save the list as a regular item. The item will be named "MD.ATTRIBS.ITEM."

When the COPY-LIST verb is used to copy a pointer list to another pointer list, a list with the same name will be over-written, but when COPY-LIST file is used to copy a pointer list to an item, an item of the same name will not be over-written. Therefore the program first carries out a DELETE for the target item ID. There is no ill effect if no such item exists.

The item holding the list will be READ by the program as a dynamic array. A program could use such a list in item format without having to loop through a READNEXT process. A list in item format will allow an editing or display routine to move backward and forward within the list, so that the user may return to a previous item displayed or edited, without having to repeat the search. One could also jump directly to any item ID on the list by index number, or locate any item ID with the LOCATE statement.

To conclude the example, QSELECT is used to create a pointer list from the item.

Variables, rather than literals, have been used for the file and list names so that the same routine can be used for different files and lists.

TEST.SELECT.LIST Program

```
* TEST.SELECT.LIST David L. Clark 01/27/1988
* Store a Select List as an Item
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TselList.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  * Assign Constants
  PROMPT ""
  Attrib.Mark$ = CHAR(254)
  File.Select.Lists.Name$ = "SELECT.LISTS"
  Item.Select.List.ID$    = "MD.ATTRIBS.ITEM"
  Pointer.List.Name$      = "MD.ATTRIBS.PTR"
  * Open file to hold the Select List
  OPEN File.Select.Lists.Name$ TO File.Select.Lists$ ELSE
    CRT \File "\ : File.Select.Lists.Name$ : \" not opened\
    STOP
  END; * Open file
  * Delete any existing Item with the same name.
  DELETE File.Select.Lists$, Item.Select.List.ID$
  * Select the Attribute Definition Items in the Master Dictionary
  * The CAPTURING clause suppresses system messages
  EXECUTE \SSELECT MD WITH *A1 = "A" CAPTURING Message$
  * Save to POINTER-FILE first
  EXECUTE \SAVE-LIST \ : Pointer.List.Name$ CAPTURING Message$
  * Load buffer for COPY-LIST, to copy to the Select Lists file
  * Note left paren before filename, space between file and Item
  DATA \(\ : File.Select.Lists.Name$ : \ \ : Item.Select.List.ID$
  * Copy, deleting the Pointer List in POINTER-FILE
  EXECUTE \COPY-LIST \ : Pointer.List.Name$ : \ (D) CAPTURING Message$
  * Read in the list in Item format
  READ List$ FROM File.Select.Lists$, Item.Select.List.ID$ ELSE
    CRT \No Item named "\ : Item.Select.List.ID$ : \" was found\
    STOP
```

```

END; * Read
* Count elements in the list
Total.Elements = DCOUNT(List$, Attrib.Mark$)
* Display the list
CRT
CRT \The following Attributes are defined in the Master Dictionary:\
CRT
FOR I = 1 TO Total.Elements
    CRT List$<I> : \ \ :
NEXT I
CRT
* Pause
CRT
CRT "Press ENTER to continue: " :
INPUT Response$
CRT
* To complete the example, create a Pointer List from the Item
Command$ = \QSELECT \ : File.Select.Lists.Name$ : \ \
Command$ = Command$ : Item.Select.List.ID$
EXECUTE Command$ CAPTURING Message$
* Use the Pointer List to show the Attribute Definition Items
EXECUTE \LIST-ITEM MD\ CAPTURING Message$
END

```

(For more details and examples for the EXECUTE statement, see the section: "System: Call System Commands from a Program." For the LOOP structure, see the section: "Repetition." For the "IF...THEN...ELSE" structure, which is also used by READ-NEXT and READ, see the section: "Conditions.")

FILES, ITEMS: MODULO

Pick

Under the Pick Operating System each file has a modulo, which is the number of groups that the items are distributed to. The distribution of items among groups is determined by a mathematical hashing formula, applied to the item ID. Sequential numbers for item IDs will provide the most even distribution. Item IDs with similar or identical endings will cause the worst distribution. (For the selection of item IDs, see the section: "Files, Items: Names.")

The modulo is set originally with the CREATE-FILE command, and should be reset when the file changes significantly in size. If the modulo is too small, the system will keep running, because Pick will attach additional frames to any group that overflows, whose items fill up the original frame assigned. Items not in the original frame are said to be in overflow.

If many items are in overflow, the system will slow down, because every time it looks for an item, Pick will first run through the original frame assigned to the group, and will then load and examine each extra frame attached. The term frame-fault refers to the action of loading and examining an extra frame. If many frame-faults occur, the system will slow down to the speed of disk access, rather than running at the speed of Random Access Memory (RAM).

The goal of modulo selection is to allow Pick to place items only in the original frames assigned, without going into overflow and thereby causing frame-faults. Therefore the modulo should be large enough to give the system a sufficient number of groups to accommodate the items in one frame per group. At the same time, the modulo should not be so large as to leave groups more than half empty. Empty groups will waste disk space, and will also cause system slow-down, if Pick has to extract a given number of items out of a larger number of groups.

To change the modulo of a file, use the following procedure. This procedure assumes a file separation of 1, which is the only separation parameter that can be employed on most of the newer Pick implementations.

- 1) Look up the frame size of the system in use. The frame size will be 500, 1000, or 2000 bytes. For the IBM PC/XT R83 2.0, the frame size is 500.
- 2) Estimate the total number of items to be stored in the file within the next year.
- 3) Estimate the average size per item.
- 4) Calculate: $\text{total size} = \text{average size of item} \times \text{total number of items}$.
- 5) Calculate: $\text{modulo} = \text{total size} / \text{frame size}$
- 6) If the total number of items is smaller than the modulo (because the average size per item is greater than the frame size), use the total number of items as the modulo instead.
- 7) If the modulo is divisible by 2 or 5, use as the modulo the next highest number that is not divisible by 2 or 5.
- 8) Make sure that the sum of all the file modulos will not be greater than 95 % of the Maximum Frame Identification (Max FID) number on the system. If the storage space will not accommodate all the files at their optimum modulo size, then the modulos must be reduced until greater storage capacity is obtained.
- 9) Place the new modulo to be used in attribute 13 of the file definition item. For a dictionary level file, the file definition item is held in the master dictionary of the account. For a data level file, the file definition item is held in the dictionary level of that file. The new modulo will take effect when the next file restore is performed.

Newer Pick implementations do not hold very large items in the primary file space allocated when the modulo is chosen. These systems instead store only a pointer, which points to the actual storage location of the item elsewhere. If the system stores very large items elsewhere, adjust the average size by item estimate accordingly.

Before performing file size reallocation during the file restore process, look at each of the files that are bigger than memory. Decide whether or not it is necessary to reallocate those files yet. To restore with reallocation will take several times longer than to restore without reallocation. The number of times longer than the restore with reallocation will take is equal to the average number of items per group.

FILES, ITEMS: NAMES

IBM PC DOS

Filenames must begin with a letter. The name must be no longer than eight characters, and can include numbers and hyphens as well as letters. (Other symbols should be avoided.) A period follows and then an extension of up to three characters. The following extensions have the designations shown:

- ☐ .BAS = Interpreted BASIC program
- ☐ .COM = Command file
- ☐ .EXE = Executable command file
- ☐ .LIB = Library file, of compiled routines to link into a command file
- ☐ .OBJ = Compiled object code, before linking into a command file

The extension for document files is not predetermined, but can be chosen by the user. However, many programs expect a particular extension. For example, document files in Microsoft Word normally have the extension .DOC, although other extensions are allowed.

In IBM PC BASIC, if the user has entered an incorrect filename, error number 64, "Bad File Name," will result.

Upper- and lowercase letters are treated the same, and all are displayed as uppercase.

Two files in the same directory cannot have the same name.

DOS directory names can be as long as you wish, but the total length of the path from the root directory to the filename, including a backslash between each directory level, is 63 characters. You can use a period in a directory name. Whereas only three characters are allowed after a period in a filename, more than three characters can be included after a period in a directory name.

For best results and easiest file organization, do not assign a directory name of more than eight characters, do not use any characters other than letters and numbers, and do not store files more than one directory level below the root directory. A noticeable decrease in speed occurs if command files must be called up from several directory levels deep. It is a recommended practice to store command files in the root directory, with data and word processing documents in subdirectories set aside for individual users and types of information.

Microsoft BASIC (Radio Shack 100)

Filenames must begin with a letter. The name must be no longer than six characters, and can include numbers as well as letters. (Other symbols should be avoided.) A period follows and then an extension. The extension is two letters long and is predetermined as one of the following:

- ☐ .DO = Document file
- ☐ .BA = Basic program
- ☐ .CO = Machine language program

In Microsoft BASIC on the Radio Shack 100, if the user has entered an incorrect filename, error number 55 "Bad File Name" will result.

Upper- and lowercase letters are treated the same, and all are displayed as uppercase.

Two files on the Radio Shack 100 cannot have the same name. There are no separate directories or accounts for files.

Pick

Filenames must start with a letter. Filenames and item IDs can be up to 50 characters in length and can include any character other than a comma or semicolon. In practice, filenames generally contain only letters, numbers, periods, and dashes; item IDs include only letters, numbers, periods, and asterisks; and neither filenames nor item IDs include spaces. Pick variable names allow only the period and "\$" as special characters, therefore, for consistency, it would be best not to use dashes in filenames. Several punctuation marks can occur in the same filename or item ID, thus one could name a file "Gold.Rush.History" and name items "Panning.for.Gold," "Overland.Trail," and so on.

Upper- and lowercase letters are treated differently. If a file is created with the name "History" it must be accessed as "History" and not "HISTORY" or "history."

Two files in the same account and two items in the same file cannot have the same name.

Item ID

Every item, whether it contains data or the definition of an account, a file or an attribute, is identified by an item ID. When planning a database, consider that the simplest type of item ID for sorting purposes on a data file is numeric.

An item ID should not include embedded spaces or parentheses. If it contains spaces or parentheses, then the entire item ID must be surrounded with quotes when entering a statement to access the item. For example, you could invoke the Editor to edit the following items without the need to include quotes:

```
EDIT Address.File Smith,*John Jones,*Tom
```

An asterisk has been substituted for a space in the names that serve as the item IDs. If the item IDs included a space or parentheses, then quotes would be necessary:

```
EDIT Address.File "Smith, John" "Jones(Tom)"
```

It is much less convenient to include quotes.

When asking for a list of items, the asterisk symbol (*) serves to indicate that you wish to retrieve all of the items in the file. For example:

```
EDIT Address.File *
```

Obviously, do not use an asterisk alone as an item ID.

In regard to choosing item IDs, sequential numbers will provide the best distribution of items into groups, thereby gaining advantages with regard to both speed and storage

space. Sequential numbers also provide the easiest and fastest mechanism for ensuring that new item IDs will be unique. The program need only store and update the highest number used so far, rather than searching to check if an item ID has already been used.

The opposing consideration is that sequential numbers will have no meaning for the user, and so they do not lend themselves to direct query. The user would find it easier to ask for "General*Motors" than to ask for Item "4397." If numerical item IDs are used, virtually all queries to the file will require looking at the contents of the fields, a much slower process than taking only the item ID.

The speed of searching on fields can be improved by the use of index files. An index file contains the item IDs of all the items in which a particular field holds a specific value. Indexing compels the use of short item IDs. Long item IDs slow down searches and updates, undermining the purpose of the index. Many indexing systems handle with difficulty, or not at all, a list of item IDs that exceeds the length of a frame.

FILES, ITEMS: OPEN, CLOSE

IBM PC BASIC

OPEN

For example:

```
OPEN Filename FOR Mode AS #File.Number
```

```
File.Input$ = "Input.do"
```

```
OPEN File.Input$ FOR INPUT AS #1
```

```
File.Output$ = "Output.do"
```

```
OPEN File.Output$ FOR OUTPUT AS #2
```

```
File.Add.More.Data$ = "Data.do"
```

```
OPEN File.Add.More.Data$ FOR APPEND AS #3
```

Mode and File.Number must be literals, not variables. The files in the previous examples will be accessed in the program as #1, #2, #3 for input and output.

The OUTPUT and APPEND statements create a file if one does not already exist with the designated filename. The OUTPUT statement destroys any previous contents of a file by that name. APPEND adds the new data to the end of a file, but there is no error if such a file does not already exist.

If the INPUT file cannot be found, an error occurs, and the system returns error code number 53 in the system variable ERR on the IBM PC, and 52 on the Radio Shack 100. If the program branches to an error-handling routine when an error occurs, that routine could check the value of ERR and display a message such as "File Not Found."

CLOSE

For example:

```
CLOSE #File.Number      '*Close 1 file
```

```
CLOSE                  '*Close any and all open files
```


The File.Number must be a literal, not a variable. If no File.Number is designated, then all open files are closed.

END

For example:

END

All files are automatically closed when a program ends. On the Radio Shack 100, the command "MENU" also will end the program, and return the user to the machine's start-up menu.

BASICA /F

For example:

BASICA /F = Max.Number.of.Files

The maximum number of files that can be open at the same time for input and output is declared with the /F switch when BASIC is started. The number for the maximum number of files must be a literal number.

The /F setting cannot be higher than the number of open files allowed by the IBM PC DOS Operating System. The operating system creates predefined file numbers. The number of files available for a program to use will come above five IBM PC DOS predefined files, and a BASIC predefined file for SAVE, CHAIN, NAME, and MERGE operations. Therefore the program must set a number above six to have files available for its use.

IBM PC DOS

FILES =, CONFIG.SYS

For example:

FILES = Max.Number.of.Files

The maximum number of files that can be open at the same time for input and output is declared with the "FILES = " statement in the root directory's CONFIG.SYS file. The number for the maximum number of files must be a literal. IBM PC DOS looks in the CONFIG.SYS file when the system is booted. No other commands or batch files can precede it.

The default value is 8, which includes the 5 predefined files of standard input (keyboard), standard output (terminal screen), error, auxiliary, and standard printer (line printer, parallel port).

The **BASICA** command of IBM PC BASIC does not apply to Microsoft BASIC on the Radio Shack 100.

On the Radio Shack 100, the maximum number of files that can be open at the same time for input and output is declared with the **MAXFILES** statement. The number for the maximum number of files can be a variable.

Each file buffer requires an extra 256 bytes of memory, therefore keep the number as low as possible. The default setting is 1, but the setting made by the last program run will stay in effect when another program starts. Any substantial program should begin with the setting that the program requires.

The **MAXFILES** command causes a **CLEAR**—all the variables are set to null, the variable definition statements **DEFINT**, **DEFSTR**, and so forth are obliterated, and there is no **RETURN** from a subroutine whose **GOSUB** preceded the **MAXFILES**. Therefore the **MAXFILES** command should not be placed in a subroutine, and the command must precede the assigning of variables that will be needed later in the program.

MAXFILES can be accessed as a variable. Thus:

```
PRINT MAXFILES
```

will display the current **MAXFILES** setting, and:

```
Num.Files = MAXFILES
```

will assign to the variable **Num.Files** the current **MAXFILES** setting.

PICK/BASIC

OPEN

For example:

```
OPEN "DICT", "HISTORY" TO File.History.Dict$ ELSE STOP
OPEN  "", "HISTORY" TO File.History.Data$ ELSE STOP
OPEN  "HISTORY" TO File.History.Data$ ELSE STOP
```

The file is opened for both input and nondestructive output. To open the dictionary rather than data level of the file, put **"DICT"** at the beginning of the statement. To open the data level, you can either indicate a null in place of **"DICT"** or eliminate the dictionary portion of the statement, thus the second and third examples are equivalent. **"HISTORY"** is the actual name of the file on the system. Both **"DICT"** and the filename can be either string literals or variables.

To use a variable for a data filename:

```
OPEN File.Data.Name$ TO File.Data$ ELSE STOP
```

To open a data file whose dictionary points to several data files:

```
OPEN File.Dict.Name$, File.Data.Name$ TO File.Data$ ELSE STOP
```

The OPEN statement assigns a file variable, by which the file can be accessed in the program, distinguishing between the data and the dictionary levels of the file. If the file is not found, the ELSE statement will take affect, allowing the program to stop or branch to an error handling routine.

Any number of Pick files can be open at the same time. There is no command for closing a file. (Under Open Architecture 2.1, there is a limit of 44 files open. This limitation will probably be removed in the next release of the program.)

Because the opening of a file is a time-consuming operation, a program should open as many as possible of the files that it needs only once. All of the files from which a program takes default parameters should be opened as part of the beginning assign constants, parameters, initial values section, by a routine such as "Name, Open Parameter Files," shown here.

Name, Open Parameter Files Routine

```

*****
8500  * Name, Open Parameter Files *
*****
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: NamOpnFi.bpr
File.Input.Name$ = "PARAMETERS"
OPEN File.Input.Name$ TO File.Param$ ELSE GOTO 9000;  *|File Error|
File.Input.Name$ = "ATTRIB.DEFINITIONS"
OPEN File.Input.Name$ TO File.Attrib$ ELSE GOTO 9000;  *|File Error|
File.Input.Name$ = "OLD.SEARCHES"
OPEN File.Input.Name$ TO File.Searches$ ELSE GOTO 9000; *|File Error|
RETURN
*-----*
*****
9000  * File Error *
*****
CRT "File: " : File.Input.Name$ : " has not been found
CRT "Press ENTER:" :
INPUT Response$
GOTO 9999;                               *|End Program|
=====
*****
9999  * End Program *
*****
* Place ending cleanup routines here
END

```

If the same program can be used with a number of different databases, between which the user might switch back and forth, the following "Open Data File" routine will place the file variables of opened files in a dimensioned array, so that the same file need not be reopened until the limit of the array has been reached. Note that the filenames, which are ordinary strings, can be kept in a dynamic array, but file variables, which act as pointers to files, can be kept in a fixed array but not in a dynamic array.

Open Data File Routine

```
*****
* Open Data File *
*****
* David L. Clark 09/22/1988
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: OpenFile.bpr
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Parameters: File.Data.Name$, Files.Data.Name.Open$,
*   Files.Data.Open$(), Files.Data.Max
* Returns: Files.Data.Name.Open$, Files.Data.Open$(),
*   opens File.Data$ for input or output.
*   For error message: File.Input.Name$
* Local variables:
*   Index : Position of File.Data.Name$ in the Files.Data.Name.Open$
*           dynamic array, matching the position of the corresponding
*           file in the dimensioned array Files.Data.Open$()
File.Input.Name$ = File.Data.Name$
LOCATE(File.Data.Name$, Files.Data.Name.Open$, Index) THEN
    * Already open. Assign File.Data$ to file variable from array,
    * according to the matching Index value.
    File.Data$ = Files.Data.Open$(Index)
END ELSE
    * Not already open. Open now.
    OPEN File.Data.Name$ TO File.Data$ THEN
        * Index will equal the total number of files already open.
        * Don't let Index exceed Files.Data.Max
        IF Index < Files.Data.Max THEN Index = Index + 1
        * Place the file name in the dynamic array and the
        * file variable in the dimensioned array at the
        * corresponding Index position.
        Files.Data.Name.Open$(Index) = File.Data.Name$
        Files.Data.Open$(Index)      = File.Data$
    END ELSE
        GOTO 9000;                *|Error Routine|
    END; * Open File.Data.Name$
END; * If File.Data.Name$ already open
```

Pick TCL

CREATE-FILE

For example:

CREATE-FILE Filename Dict.Modulo Data.Modulo

A file is not created with the PICK/BASIC OPEN statement. The file must first be created with the operating system command CREATE-FILE. The parameters Dict.-Modulo and Data.Modulo are the modulos of the dictionary file and data file respectively. The modulo is the number of groups that the items in the file will be spread between.

If the same dictionary will point to multiple data files:

CREATE-FILE Dict.Filename, Data.Filename Dict.Modulo Data.Modulo

The dictionary level file must be created before or concurrently with the data level file that the dictionary points to. In the last two examples, both levels were created at the same time.

To create a dictionary level file only:

```
CREATE-FILE DICT Filename Dict.Modulo
```

To create a data level file when the dictionary file already exists:

```
CREATE-FILE DATA Dict.Filename { , Data.Filename } Data.Modulo
```

The filename of the data file would be written separately if the dictionary file pointed to several data files.

FILES, ITEMS: OPEN, CLOSE: TEST FOR EXISTENCE

The routines shown in each language will open and test a file for input. If there is an error, the routine will branch to an error handling routine.

The failure to open a file has different consequences, according to the circumstances:

- 1) The input file consists of default data values: If the file does not exist, the program begins with the values set to null. No special action is necessary.
- 2) The input file is a file to edit or print: An error message is displayed, then the program returns to the main menu.
- 3) The input file contains parameters necessary to the operation of the program: An error message is displayed, and the program terminates.

IBM PC BASIC

In IBM PC BASIC, if an error occurs the only alternative to crashing is to jump immediately to the error handling routine, rather than processing the error in the routine that caused it. This difficulty is avoided by having the error handling routine set the Boolean variable `Error.Flag` to true and then returning to the original routine in which the error occurred. The original routine will test the `Error.Flag` to make sure that the operation was completed successfully. (For Boolean variables, see the section: "Boolean Flags.")

IBM PC BASIC allows the program to simulate an error with a program-defined error number:

```
ERROR Error.Num
```

(For the `ERROR` statement, see the section: "Error Recovery.")

In the following example, the `Edit Old Data` routine calls the `Test, Open File.Input$` routine to test if the file exists, opening the file if it exists or setting the `Error.Flag` to true if the file does not exist. The `Edit Old Data` routine sets the program-defined `Error.Num` to 90 to indicate that failure to open the file would be a nonfatal error. (An `Error.Num` of 80 would indicate a fatal error.) After returning from the `Test, Open File.Input$` routine,

the Edit Old Data routine tests the Error.Flag to see if the file has been opened, or if a nonfatal error has occurred. (A fatal error would have terminated the program.) If the Error.Flag is true, it is cleared to false, and the program returns to the main menu.

In the Test, Open File.Input\$ routine, if an error occurs, the program goes automatically to the group of routines named Error Routines. If the Error Recovery routine detects an ERR of 53 on the IBM PC or 52 on the Radio Shack 100 for "File Not Found," or an ERR of 64 on the IBM PC or 55 on the Radio Shack 100 for "Bad File Name," the Error Recovery routine sets the Error.Flag to true and with the command "RESUME NEXT" returns the program to the next statement after the statement that tried to open the file.

The Test, Open File.Input\$ routine tests the Error.Flag immediately after trying to open the file. If the Error.Flag is true, the routine activates an error with an Error.Num that will cause the Error Routine to take appropriate action, displaying a message to explain the error, and terminating the program if the error was fatal.

If no action was necessary on failure to open a file, not even the display of an error message, the Test, Open File Input\$ routine would not be called.

Test, Open File.Input\$ Routine

```

*****
'* Test, Open File.Input$ *
*****
'* David L. Clark 09/22/1988
'* Demonstrate procedure for opening and testing for existence a file
'* IBM PC Basic routine. Documentation version.
'* IBM PC BASICA 3.0, Radio Shack 100
'* Stored under filename: TOpenFil.brd
'* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
2400 '* Edit Old Data *
*****
File.Input$ = File.Name$ + File.Ext$
:   Error.Num = 90           '* Non-Fatal "File Missing"
:   GOSUB 8800              '* |Test, Open File.Input$|
:   IF Error.Flag THEN
:       Error.Flag = False   '* Clear Error.Flag
:       RETURN              '* No file to edit
:   ELSE
:       ...
'* End line
'*-----*
8800 '* Test and Open File.Input$ *
*****
'* Parameters: File.Input$, Error.Num
'* Returns: ERROR, or opens File.Input$ for use
OPEN File.Input$ FOR INPUT AS #1
:   IF NOT Error.Flag THEN
:       RETURN
:   ELSE
:       ERROR Error.Num      '* Go to |Error Routines| for action
:       RETURN              '* Error.Flag carried back to caller
:   '* End: If Not Error.Flag
'* End line

```

```

'-----*
*****
9000 '* Error Routines *
*****
*****
'* Error Recovery *
*****
'* Parameters: File.Input$, ERR (System Error#), ERL (Line#)
'* Returns: Error.Flag, Error.Num, Message$
Error.Flag = True
:   CLOSE                                     '* Close any open file
:   IF ERR = 53 OR ERR = 64 THEN               '* IBM PC only
:   IF ERR = 52 OR ERR = 55 THEN               '* Radio Shack 100 only
      RESUME NEXT
    ELSE
      *****
      '* Error Message *
      *****
      IF ERR = 80 OR ERR = 90 THEN
        Message$ = File.Input$ + "File missing"
        '* Test ERR for other errors, supply messages...
        '* End: If ERR = 80 or 90
        '* End: If ERR = 52 or 55
      '* End line
      *****
9200 '* Error Exit *
*****
CLOSE
:   CLS
:   BEEP
:   '* IBM PC version for display statements
:   LOCATE 2,37
:   COLOR 7,0
:   PRINT "ERROR "
:   COLOR 0,7
:   LOCATE 3, 20
:   PRINT Message$
:   '* Radio Shack 100 version for display statements
:   PRINT @57, Video.Reverse$; " ERROR "; Undo.Video.Reverse$
:   PRINT @130, Message$
:   '* For both machines
:   IF ERR > 89 THEN
      RESUME NEXT                                     '* Carries Error.Flag back to caller
    ELSE
      GOTO 9999                                     '*|End Program|
      '* End: If ERR > 89
    '* End line
    '-----*
    *****
9999 '* End Program *
*****
END

```

PICK/BASIC

Testing for the existence of a file is much easier in PICK/BASIC. The mandatory ELSE clause in the OPEN file statement will take effect if the file does not exist.

Test, Open File.Input\$ Routine

```
*****
* Test, Open File Input$ *
*****
* Demonstrate procedure for opening and testing for existence a file
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TOpenFil.bpr
* Parameters: File.Input.Dict$, File.Input$, Error.Num.
* Returns: Branches to |Error Routines| or opens File.Input$
OPEN File.Input.Dict$, File.Input$ TO File.Input$ ELSE GOSUB 9000
                                           *|Error Routines|
```

The procedures for reading an individual item from a file also include an ELSE clause that will take effect if the item cannot be read. (See the section: “Files, Items: Input.”)

FILES, ITEMS: OUTPUT

IBM PC BASIC

PRINT

For example:

```
PRINT #File.Number, U; V; X$; Y$; ...
```

File.Number must be a literal number, not a variable. If commas are used as separators, the elements will be printed 14 spaces apart. Positive integers print with a blank in front, and negative integers print with a minus. (To eliminate the spacing around numbers, see the section: “Numbers: Convert: Number to String.”)

An ending semicolon suppresses the Carriage Return. To send an extra Carriage Return, PRINT a null value:

```
PRINT #1, X$
PRINT #1, ""
PRINT #1, Y$
```

Just “PRINT #1” by itself, equivalent to the empty PRINT statement sent to the screen, will cause a syntax error. You also could put a Carriage Return—Line Feed in a variable, rather than starting a new PRINT statement.

In IBM PC BASIC, format masks can be applied when “printing” to a file in the same manner as output to a screen or printer.

PICK/BASIC

WRITE, WRITEV, MATWRITE

To write an entire dynamic array to an item in a file, assigning each element from the array to an attribute in the item:

```
WRITE Dyn.Ay ON File.Data$, Item.ID$
```


To write a variable or literal to an attribute in an item in a file:

```
WRITEV New.Attrib$ ON File.Data$, Item.ID$, Attrib.Num
```

To write an entire fixed (dimensioned) array to an item in a file, assigning each element from the array to an attribute in the item:

```
MATWRITE Fixed.Ay ON File.Data$, Item.ID$
```

In each case, if the item specified does not exist, it will be created.

With regard to WRITEV, if the item exists but the attribute specified does not exist, then the attribute will be created. If the Attrib.Num parameter is zero, the new element will be inserted at the front of the item as the first attribute, and all the existing attributes will be pushed down by one. If the Attrib.Num parameter is - 1, the new element will be appended to the item as the last attribute.

WRITEU, WRITEVU, MATWRITEU

The “U” version of each write statement will not unlock the group into which the item is hashed. The version of these statements without an ending “U” will unlock the group. While the lock is in place, no one else can access the group with the “U” version of the read statements. The normal practice is to unlock the group after the item has been written. Unlocking also can be performed with the RELEASE statement. (For locks, see the section: “Files, Items: Input.”)

PICK TCL

REFORMAT, SREFORMAT

The PICK TCL verbs REFORMAT and SREFORMAT use the same syntax as the PICK TCL LIST and SORT verbs:

```
SORT      Photos BY-EXP Subjects Subjects Item.ID  
SREFORMAT Photos BY-EXP Subjects Subjects Item.ID
```

The difference is that REFORMAT and SREFORMAT can direct their output to another file or to a tape. These commands are generally used to restructure a file, by outputting selected attributes to a second file, perhaps in a different order, or to create a cross-reference file.

After the command, the prompt appears:

```
FILE:
```

For example:

```
FILE: Subjects.Index
```

Supply either a filename or “TAPE” to direct output to the tape drive or floppy disk.

Among the modifiers and options normally employed with LIST and SORT, control breaks and totals cannot be used with REFORMAT and SREFORMAT. Also, do not use ID-SUPP or the (I) option, to suppress item IDs.

When output is redirected to another file, the first output specification will be used as an item ID to create items in the destination file. Thus in the previous example, an index file or cross-reference file for subjects would be filled with items. The item ID for each item would be a subjects term from the Photos file, the first attribute would be the item ID of the item in the Photos file in which that subjects term appeared. Note however, that SREFORMAT will not create a multivalued attribute. If the same subjects term appeared in several items in Photos, SREFORMAT would only retain the item ID of the last item encountered.

FILES, ITEMS: RENAME

Use the command RENAME to change the name of an existing file.

IBM PC BASIC

NAME...AS...

For example:

```
NAME   File$ AS New.Name$
NAME   "Data.do" AS "Report.do"
```

The new and old names can be literals or variables. The file can be open or closed. The extension to the filename must be included in the NAME and KILL commands, although it is not required in the OPEN, input, or output commands.

The following TstFile.bas (Test File) program will create a file, rename it, then kill it, and show the results of each operation.

TstFile.bas Program

```
1  '* TestFile.bas: Test File  David L. Clark  05/04/1987
2  '* Open, Rename and Kill a File
3  '* IBM PC Basic program. Executable version.
4  '* IBM PC BASICA 3.0, Radio Shack 100
5  '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10 F$ = "Before.do"
20 OPEN F$ FOR APPEND AS #1
30 CLOSE #2          '* No difference if open or closed for Rename, Kill
40 FILES             '* Show files
50 N$ = "After.do"
60 NAME F$ AS N$
70 F$ = N$           '* Changing filename will not change the variable
80 FILES
90 KILL F$
100 FILES
9999 END
```

IBM PC DOS Command Level

RENAME

For example:

```
RENAME File1.Ext File2.Ext
```

PICK TCL

COPY

For example:

```
COPY Filename { * or Item.IDs } {(D)}  
TO: {(Filename) { New.Item.IDs }
```

Files and items are renamed by copying them to a new name. The “D” option will also delete the old item or items.

To rename an item, COPY the item, designating a new item ID. The new item will be created by the COPY operation, and the old one deleted by the “D” option:

```
COPY Filename Old.Item.ID { Old.Item.ID... } (D)  
TO: New.Item.ID { New.Item.ID }
```

To rename a file, you first must create a file with the new name:

```
CREATE-FILE New.Filename Dict.Modulo Data.Modulo
```

(For details on creating a file, see the section: “Files, Items: Open, Close.”)

Copy all the items (with the same names or new names) for both the dictionary and data levels:

```
COPY DICT Old.Filename*  
TO: (DICT New.Filename { New.Item.IDs }  
COPY Old.Filename*  
TO: (New.Filename { New.Item.IDs }
```

Note the left parenthesis before the New.Filename. Finally, delete the old file:

```
DELETE-FILE Old.Filename
```

(For more details on the COPY verb, see the section: “Files, Items: Copy, Backup.”)

PICK TCL

EXCHANGE

For example:

```
EXCHANGE Filename Item.1 Item.2
```

This command is used to exchange the names of two items. After the command, the system will prompt with a rather confusing message. If the item names were "Item.1" and "Item.2" the message would read:

```
COPY Filename Item.1 Item.2 %TEMP% (O,D)
%TEMP% Item.1 Item.2__
?
```

Press ENTER to continue with the process, anything else to abort. The following messages will then appear:

```
1 Item.1 to %TEMP%
2 Item.2 TO Item.1
3 %TEMP% TO Item.2
3 ITEMS COPIED
```

EXCHANGE is not explained in the Pick Manual. It is a PROC in the PROCLIB file.

FILES, ITEMS: SYNONYMS

IBM PC DOS

The IBM PC DOS operating system does not provide for file synonyms, but a program file can be called up by different names through the use of batch files. For example, several individuals, all using the same word processing program, could store their documents in separate directories. The AutoExec.bat file, which the system will automatically execute when the computer is turned on, would include the PATH command:

```
PATH C: \
```

This command allows a program stored in the root directory of the C drive to be invoked from other directories or drives.

Each individual might then have a batch file named with his or her initials, so that the person could call the command file simply by typing in the initials. The person's subdirectory for documents could optionally have the same name:

Initials.bat Batch File

```
@ECHO OFF
REM Initials.bat: David L. Clark 09/09/1989
REM Name of the file for each person's initials
REM Switch to Initials Directory, start Microsoft Word program,
REM then return to main Menu.
REM IBM PC DOS batch file.
C:
REM Go to the directory named for person's initials
CD \ %0
```

```
WORD %1
REM The Menu command is meant to transfer control to a 2nd Batch file
REM with the filename Menu.bat
Menu
→
```

The Menu command invokes a second batch file, which simply presents a menu of the user's options by using the TYPE command to display the contents of a plain text file: "TYPE Menu.txt".

The Microsoft Word program allows you to name a document for editing when calling the program, with the command:

```
WORD Filename
```

The % 0 parameter returns the command the user typed. If the user's initials are "DC," the batch file will take him to the "DC" directory. If the %1 parameter contains something, then WORD is invoked with that parameter.

IBM PC DOS does not care if commands are entered in upper- or lowercase, unlike PICK, which insists on uppercase.

Pick

A file synonym in Pick allows an account to access a file in a different account. The file definition item in the MD of the account from which the file will be accessed (not the original account that holds the file) has the following form:

```
01 Q
02 Original.Account.Name
03 Original.Filename
```

Synonyms also are called Q-pointers, because attribute 1 holds a "Q" rather than a "D" for a file, and because they point to the original file. It is not necessary to fill in the remaining attributes 4 – 10, unless characteristics such as the column width and justification of item ID display or the codes to restrict retrieval and updating of items in the file are to be different.

If the file to be accessed in the other account is the MD of that account, it is not necessary to include attribute 3. If the synonym is a different filename in the same account, attribute 2 would be left blank.

Reflexive Q-pointers are used for files where only a dictionary level is needed, such as the MD of an account or a library of PROCs. The dictionary level of a file always points to the data level. In the case of a reflexive Q-pointer, the data level of the file is a synonym that points back to the dictionary level. The advantage is that to access the file, it will not be necessary to type "DICT Filename," instead "Filename" alone will suffice.

The file will be created with the CREATE-FILE statement, but with a DICT specification, so that only a dictionary level will be created:

```
CREATE-FILE DICT Filename Modulo, Separation
```

The command will place a data-level file definition item in the dictionary level of the file. This data-level definition item will simply be a Q-pointer back to the dictionary level. Only attribute 1 will contain a value, which will be "Q." To allow more room for display of the Item.IDs, rather than the default 10 spaces, create an attribute 9 with "L" for left justification and an attribute 10 with "20" for 20 spaces.

For a PROC library History.Proc.Lib the file definition item in the MD of the account will resemble the following:

```
01 D
02 12860
03 11
04 1
09 L
10 20
```

The data-level definition item within the dictionary level of the file has the same name (item ID) as the file itself. To edit this item, the command is:

```
EDIT DICT History.Proc.Lib History.Proc.Lib
```

The item can consist of only a single attribute:

```
01 Q
```

To provide more than the default 10 spaces for the display of item.IDs, change it to:

```
01 Q
09 L
10 20
```

SET-FILE

The SET-FILE command will create a temporary Q-pointer or file synonym named "QFILE." Every time the SET-FILE command is used, the identity of QFILE will be changed, to point to a different file. The SET-FILE command is convenient for quickly making a copy from a file in a different account. (For copying files, see the section: "Files, Items: Copy, Backup.")

When the command SET-FILE is used, the system will prompt:

```
ACCOUNT NAME?
```

Respond with the name of the account in which the file resides, which can be different from the current account. The system will then prompt:

```
FILE NAME?
```

Give the name of the file. The system will give the final message:

```
'QFILE' UPDATED
```

Q-pointers also can be used to give each user a different password, a different level of security, and a different logon PROC to display a different initial menu of choices. Each user might logon on to the system by typing his or her initials. The Q-pointer in the SYSTEM dictionary file would resemble the following:

Item ID: DLC	{ Initials of user }
01 Q	{ Name of Account used }
02 History	
03	
04	{ Retrieval key }
05 ABCD	{ Update key }
06 MNOP	{ Password, held in encrypted form }
07 R1H8GP	{ System privilege level: SYS0, SYS1 or SYS2 }
08 SYS2	{ If press BREAK, return to Logon, do not go to the Debugger }
09 R	
10	
20	

Keyboard



IBM PC BASIC

LINE INPUT

For example:

```
LINE INPUT {;} { "Prompt string"; } Response$      '* IBM PC
LINE INPUT   { "Prompt string"; } Response$      '* Radio Shack 100
```

The **LINE INPUT** statement takes a string typed at the keyboard and followed by pressing the **ENTER** key and places the string in the variable indicated. The computer waits for a response, and the response is echoed to the screen. The input can have the maximum string length allowed in IBM PC BASIC of 255 characters.

The Carriage Return—Line Feed characters entered with the **ENTER** key are not included in the string variable returned. If **ENTER** is pressed without typing any characters, the string variable is assigned a null value.

LINE INPUT is the most useful form of keyboard input. It is a more reliable method to take the user's response as a string and convert it to a number if necessary, and to allow a null response as a default choice, than to require that the user enter a number correctly.

A semicolon immediately following **LINE INPUT**, and before any prompt string, will prevent an automatic Carriage Return—Line Feed from occurring after the input is taken. If **LINE INPUT** takes place on the bottom row of the screen, and the automatic Carriage Return—Line Feed is not suppressed, then the screen will scroll. The feature of preventing

an automatic Carriage Return—Line Feed is not available for simpler forms of Microsoft Basic, such as that installed on the Radio Shack 100. However, the Radio Shack 100 has a Scroll Lock function to prevent scrolling. (See the section: “Screen: Scroll Lock.”)

An optional prompt string can be included in the LINE INPUT statement. The prompt string must be a constant enclosed in quotes, it cannot be a variable. The prompt string, if present, is separated from the response string variable by a semicolon. The LINE INPUT prompt appears wherever the cursor happens to be. The LINE INPUT statement will not place the prompt at a specific position. (For placement of the cursor on the screen see the section: “Screen: Cursor: Place the Cursor.”)

Rather than including the prompt in the LINE INPUT statement, the prompt can be placed in a separate PRINT statement, in which case the prompt can be either a literal or a variable. A semicolon should follow the PRINT statement in order to keep the cursor on the same line:

```
LINE INPUT "Type a response, then press ENTER: "; Response$
```

or

```
Prompt$ = "Type a response, then press ENTER: #"
PRINT Prompt$;
LINE INPUT Response$
```

INPUT

For example:

```
INPUT { "Prompt string"; } Variable { , Variable ... }
```

The INPUT statement will take a list of variables separated by commas all at the same time from the keyboard, followed by pressing the ENTER key. Alternatively, you can enter values one at a time, following each value with the ENTER key. The computer will wait for a response.

String values can be enclosed in quotes, although quotes are not required unless the string includes leading blanks or commas or colons. If you attempt to enter a string for a numeric variable, the message “?Redo from Start” is displayed and the operation begins over again. INPUT displays a question mark prompt that cannot be suppressed. LINE INPUT is much easier on the user.

PICK/BASIC

INPUT

For example:

```
INPUT Response$ { , Num.Chars } { : }
```

The INPUT statement takes input typed at the keyboard and followed by pressing the ENTER key and places it in the variable indicated. The computer waits for a response, and the response is echoed to the screen. The maximum length allowed for a single input made in this fashion is 140 characters, the length of Pick's primary input buffer. The Carriage Return—Line Feed characters entered with the ENTER key are not included in the variable returned. If ENTER is pressed without typing any characters, the input variable is assigned a null value.

The optional colon will suppress an automatic Carriage Return—Line Feed at the end of the input, leaving the cursor after the last character typed.

The optional parameter Num.Chars, which can be a literal or a variable, causes an automatic line feed and ends the input when the specified number of characters have been entered. The Num.Chars parameter is most useful for taking a single keystroke without pressing ENTER for data entry and editing. (See: "Keyboard: One Keystroke.")

If a prompt character is active, it will be shown to request input.

PROMPT

For example:

```
PROMPT Prompt.Char$
```

The Prompt.Char\$ parameter can be a 1-character variable or literal. Any characters after the first will be ignored. The PROMPT command stays in effect until a new PROMPT command is issued.

If no PROMPT command is given, the question mark "?" will appear as the default prompt character with every INPUT statement. Most Pick programs begin by assigning a null prompt character, and place prompts to the user in separate CRT statements:

```
* Assign Constants, Parameters
PROMPT ""
* Body of the program
CRT @(Column, Row) : "Enter a response:" :
INPUT Response$ :
```

Note the four different uses of the colon. The first colon combines the column and row specification with the quoted prompt statement. A second colon appears within the quoted statement prompting the user, so that the screen will show:

Enter a response:

A third colon appears on the end of the line after the final quote, to suppress the Carriage Return—Line Feed and keep the cursor immediately after the prompt. The last colon appears after the INPUT statement, again to suppress a Carriage Return—Line Feed.

ECHO OFF, ECHO ON, ECHO Test.Flag

For example:

```
ECHO OFF
ECHO ON
ECHO Test.Flag
```

The ECHO OFF command suppresses the echo of keyboard characters onto the screen. ECHO ON reenables the echo function. The ECHO Test.Flag command will suppress the echo if the expression Test.Flag evaluates to false (zero), and will reenable the echo if the expression is true (nonzero).

In the IBM PC/XT implementation of Pick, the control characters will be echoed despite the ECHO OFF command. (To change the control characters to other values that will not be echoed, see the section “Keyboard Input: Change Keys.”)

INPUT @

For example:

```
INPUT @(Column, Row) : Response$ {Num.Chars} {Mask$}
```

The “INPUT @” form of the INPUT statement prompts at the row and column coordinates specified. Row and column can be literals or variables. The Num.Chars parameter functions the same as in the shorter INPUT form.

The Mask\$ expression can combine features of pattern matching, conversion, and justification. For example, you could check for a valid date format, for the number of decimal places, or for the length of the response. The default mask is “L(#50)”. If the response does not match the mask, the “INPUT @” statement is repeated.

The masking function will perform both input and output conversions. With a date specification, for example, the date will be displayed and accepted for input in external format but stored in internal format. The internal format should be placed in the Mask\$ variable.

If the variable Response\$ already has a value, it will be displayed at the specified position, using Mask\$ as an output mask. Whether or not a current value is displayed, the cursor will be placed one position back of the column specification. If a prompt character is active, it is then printed. (It is common practice to use a space as the prompt character when using the “INPUT @” form, in order to move the cursor forward into the correct position.) If the user presses ENTER without typing anything, the current value of Response\$ will be retained as the default value.

INPUTERR, INPUTNULL, INPUTTRAP

The INPUTERR, INPUTNULL, and INPUTTRAP statements provide additional support for the “INPUT @” command.

For example:

```
INPUTERR "Error Message"
```

INPUTERR will print an error message at the bottom of the screen and set a system flag that this has been done. When the next valid entry is made, the system will automatically clear the message.

For example:

```
INPUTNULL Null.Char$
```

INPUTNULL will allow the designation of a character to signify that a null value should be entered for Response\$, rather than continuing the current value. The underscore

character “_” is the default Null.Char\$. Thus if the current value of Response\$ were “William Mulholland” and ENTER alone was pressed, that value would remain. If one typed an underscore (or another specified Null.Char\$) Response\$ would become null.

```
INPUTTRAP Table.of.Chars$ GOSUB 10, 20, 30, 40 ...
INPUTTRAP Table.of.Chars$ GOTO 10, 20, 30, 40 ...
```

The INPUTTRAP statement will look for a character or characters that appear in a Table.of.Chars\$ for use in a GOTO or GOSUB clause. Branching will occur according to the position of Response\$ within the Table.of.Chars\$, in the same manner as the “ON ... GOSUB ...” or “ON ... GOTO ...” format. With the GOSUB form, after branching to the subroutine the program will return to the point after the INPUTTRAP statement, not after the “INPUT @” statement.

The “INPUT @” function can grow to become quite complex, although in some circumstances it will shorten the code by many lines. The procedure recommended here is to use the simpler form of the INPUT statement, with a null prompt character and screen cursor control through the “CRT @(Column, Row)” command, and to place these in a single subroutine that is called whenever keyboard response is required. That method tends to provide the most visually consistent form of interface for the user.

ICONV, OCONV

The ICONV (for input conversion) function will convert the user’s entry to internal Pick format. All input originally comes as a string. If the entry cannot be converted, a null will be returned. A conversion error would occur, for example, if the entry were meant to be a date and the user typed something that could not be converted into a date. The input routine should check for a null, to indicate an invalid entry. Input conversion should also be employed to “normalize” the data, which means to convert the data into a standardized form. If possible, the conversion should strip out unwanted elements without requiring manual correction by the user.

The OCONV (for output conversion) function will display a value in external format. For example, it will convert a date from internal to external format for display.

The following “Input a Date” routine will input the date of a document. If the user enters a null, today’s date will be used as the default choice. If the date is not valid, the user will be warned and prompted again.

Input a Date Routine

```
*****
* Input a Date *
*****
* David L. Clark 09/22/1988
* Check for a valid date on input.
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: InptDate.bpr
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants *
*****
PROMPT ""
EQUATE True TO 1, False TO 0
```

```

Bell$ = CHAR(7)
Clear.Line$ = @(-4)
Date.Today = DATE()
Row.Bottom = SYSTEM(3)
EQUATE Col.Message TO 30
Pos.Message$ = @(Col.Message, Row.Bottom)
Valid.Date.Flag = True
*****
* Loop to Get Input *
*****
LOOP UNTIL Valid.Date.Flag
  * Display today's date as the default in format MM/DD/YYYY.
  Display$ = @(0, 10) : Clear.Line$ : "Date of Document ["
  Display$ = Display$ : OCONV(Date.Today, "D/") : "] : "
  CRT Display$ :
  INPUT Entry$ ;; * Input as a string
  IF Entry$ = "" THEN Doc.Date = DATE.Today; * Default choice
  Entry.Date = ICONV(Entry$, "D")
  * Check for valid date. If not valid, result of ICONV will be
  * null. Also check that date entered not greater than today.
  IF Entry.Date = "" OR Entry.Date > Date.Today THEN
    Valid.Date.Flag = False
    CRT @Pos.Message$ : Entry$ : " Is not a valid date entry" :
  END; * If Entry.Date
REPEAT

```

In some circumstances, when numeric input is required, you could simply strip out nonnumeric characters with an OCONV and mask character numeric:

```
Number = OCONV(Entry$, "MCN") + 0
```

It is usually preferable to check if the input was a valid number, using the NUM function. (To check for a valid number, see the section: "Numbers: Convert: String to Number." For OCONV, see the section "Conversions and Correlatives in Pick.")

PICK/BASIC will not accept more than 140 characters at one time from an INPUT statement. The following TEST.INPUT.LONG program will verify INPUT's 140 character limit on a particular system. Because the INPUT limitation makes the transfer of a file through the serial port from another computer more difficult, it is to be hoped that the restriction will be removed.

TEST.INPUT.LONG Program

```

* TEST.INPUT.LONG David L. Clark 04/04/1988
* Test length of input accepted.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TInpLong.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Length of input on IBM PC-XT R83 2.0 limited to 140 chars. Pick performs
* an automatic Carriage Return -- Line Feed and terminates input.
  * Assign Constants *
  PROMPT ""
  * Get Input *
  CRT
  CRT "Hold down a letter or number key: " :

```

```

INPUT Input$
CRT
CRT "Input = " : Input$
CRT "Length accepted = " : LEN(Input$)
CRT "Release key"
* End Program *
END

```

KEYBOARD: ACCESS SYSTEM INFORMATION

To discover which keystrokes return which characters, see the PICK/BASIC program SHOW.ASCII or the IBM PC BASIC program ShowAsci.bas in the section: "Keyboard: Change Keys."

KEYBOARD: BREAK KEY

In some cases, the BREAK key can be disabled by changing the ASCII value returned by the key. (For details on changing keys see the section: "Keyboard: Change Keys.")

IBM PC BASIC

The BREAK key cannot be disabled directly from IBM PC BASIC.

Microsoft BASIC (Radio Shack 100)

POKE

For example:

```

POKE 63056, 64    '* Disable BREAK, PAUSE
POKE 63056, 0     '* Reenable BREAK, PAUSE

```

The POKE of 64 shown here will disable the BREAK and PAUSE functions for any INKEY\$ input from the keyboard. Pressing BREAK (or CTRL-C) will return ASCII number 3. Pressing PAUSE (or CTRL-S) will return ASCII 19. PAUSE also will be disabled for any keyboard input with the INPUT\$ statement. However, BREAK will not be disabled for input with the INPUT\$ or LINE INPUT statements. Thus you can use CTRL-C within a data entry and editing routine that inputs one keystroke at a time with INKEY\$, but still allow the user the BREAK out of a problem during other processes that employ LINE INPUT.

POKE 64 also causes the function keys F1-F8 to return ASCII 248-255. POKE 64 does not affect the PASTE key, which continues to return ASCII 0 (assuming that a BASIC CLEAR command has cleared the paste buffer). Thus the PASTE key, function keys, CTRL-C, and CTRL-S all can be used as separate control keys for editing. (For more details, see the section: "Keyboard: Change Keys.")

If a BREAK should occur with the LINE INPUT or INPUT\$ statement after a POKE 64, the function keys will not work when the user is dumped into BASIC. The user must type MENU to get back to the Radio Shack Menu. When BASIC is reentered from the Radio Shack Menu, the function keys will work as normal.

The second POKE example, “POKE 63056, 0”, will restore normal conditions. Ending the program with the command “MENU”, rather than with “END”, will have the same effect, thus this POKE is not necessary unless it precedes a direct exit into BASIC.

PICK/BASIC

BREAK-OFF, BREAK-ON, BREAK Test.Flag

For example:

```
BREAK-OFF
BREAK-ON
BREAK Test.Flag
```

The BREAK-OFF command disables the BREAK key. The BREAK-ON command reenables the BREAK key. The BREAK command followed by an expression (shown here as Test.Flag) will disable the BREAK key if the expression evaluates to false (zero), and will reenable the BREAK key if the expression is true (nonzero).

The break on and off commands are cumulative. Thus if the command BREAK-OFF has been given twice, to reenable the BREAK key the BREAK-ON command must also be given twice.

If the BREAK key is pressed while BREAK is disabled, and then the program reenables BREAK, the BREAK will occur as soon as BREAK is reenabled.

It is not necessary to reenable BREAK at the end of a program. It will be reenabled automatically after the program ends.

PICK TCL

User Exits: U218D, U318D

The following PROCs will disable or reenable the use of the BREAK key at TCL level.

DISABLE.BREAK PROC

```
PQ
C DISABLE.BREAK to disable BREAK key.
C Pick Proc.
C Pick IBM PC-XT R83 2.0
C Filename under IBM PC DOS: DisBreak.prc
U218D
P
```

ENABLE.BREAK PROC

```
PQ
C ENABLE.BREAK to re-enable BREAK key after disabled.
C Pick Proc.
C Pick IBM PC-XT R83 2.0
C Filename under IBM PC DOS: EnBreak.prc
U318D
P
```

KEYBOARD: CHANGE KEYS

It is not easy to use a Pick system to deliver editing facilities that a person accustomed to an IBM PC DOS word processor such as Microsoft Word will consider acceptable. To make the editor perform consistently across different machines and keyboards is even more difficult. Nevertheless, the goal to achieve for optimum database usage is the combination of Pick's database facilities and portability with DOS-like editing for data entry and update. The following section is long and complicated, but demonstrates that a Pick database can have a decent editor.

The example will employ the IBM PC/XT running both Pick and IBM PC BASIC, in order to demonstrate keyboard compatibility across systems. The popular Wyse 60 will be employed as a terminal with Pick. The Wyse 60 terminal can be purchased with four alternative keyboards. The AT keyboard should be employed for maximum compatibility with the IBM PC/XT and PC/AT. In addition, as an example of a stand alone computer that also can be made into a terminal, the portable Radio Shack Model 100 will be employed, in both its stand alone and terminal modes.

One of the primary differences between the standard Pick editors and DOS word processors is that in Pick the user must toggle between an entry mode, to type characters for input, and a command mode, to move the cursor, delete a character, or perform any function other than BACKSPACE or ENTER. Pick people have found this arrangement to be acceptable for databases consisting of fields that are short in length, and in the data-entry shop environment of keyboard operators whose typing speeds would overwhelm complex editing algorithms. But the reaction of a DOS user will be, "You must be kidding. Don't tell me how sophisticated and powerful your system is when I can't move the cursor immediately to wherever I want it to be, like I can with DOS." Even the portable Radio Shack 100 includes a built-in word processor that allows cursor movement and character deletion without the need to press a toggle key.

The object of this section will be to find a method whereby the pressing of a normal letter, number, or punctuation key will accomplish data entry, while pressing a control character will cause an editing action to occur, without the need for a toggle to distinguish between entry and command mode. To accomplish this purpose, pressing a control character such as DELETE or CTRL-K must deliver to the editing routine an ASCII number, without causing any independent action. The Editor will supply the action by branching to a subroutine according to the ASCII number of the key pressed.

(For the entry of a single keystroke, see the section: "Keyboard: One Keystroke." The editor can branch to a subroutine according to the position of the key in a table of control characters. For the creation of a table of control characters, see the section: "Keyboard: Control Characters for Editing.")

The most widely used control characters run from ASCII 1 through 31 and include 127. CTRL-A through CTRL-Z occupy ASCII 1 through 26, the ESCAPE key takes ASCII 27. The cursor arrow keys RIGHT, LEFT, UP, and DOWN often occupy ASCII 28 through 31. DELETE returns ASCII 127.

But the normal control keys often are not usable by the editor without resetting. For example, CTRL-C returns ASCII number 3, but also can cause a BREAK, and CTRL-S returns ASCII number 19, but also causes a PAUSE action, freezing the keyboard until CTRL-Q is pressed to restart. The action is not prevented by a command such as ECHO

OFF or other normal means of suppressing the display of a typed character. Some of the keys must therefore be reset to other numbers in order to be used by the Editor.

Those keys that cannot be reset must be avoided. If a user is to employ two different machines, such as an IBM PC/XT as well as a terminal attached to an XT, it would be dangerous to employ CTRL-S for cursor left on the XT keyboard when the same key would cause PAUSE on a remote terminal. That situation is indeed the case with CTRL-S. The key can be reset on the PC/XT keyboard, but not on the Wyse 60 terminal used in this demonstration. The MODEM OFF command, given from the terminal, will disable the PAUSE function, but at the cost of preventing the terminal from warning the central processing unit when the terminal's buffer is full. Garbled characters might result. The situation is particularly vexing because DOS word processors such as WordStar use CTRL-S for cursor left, and standard Pick does not implement the cursor arrow keys on the PC/XT keyboard. The solution has been to turn to a third party vendor for implementation of the cursor keys.

Pick Release 1983 2.2 and 3.0 allow use of the cursor keys. The new version runs on 80286 and 80386 machines, but not on the PC/XT.

If the system allows a choice of numbers for resetting a key, the following considerations apply:

- 1) The normal keyboard characters of ASCII 32 through 126 should not be disturbed.
- 2) In the IBM extended set of characters above ASCII 127, the foreign language characters run from ASCII 128 through 168 and 173 through 175, the graphics characters run from 176 through 223, and the Greek and mathematical symbols run from 169 through 172 and 224 through 254. The best characters for replacement by control codes would be the graphic characters 176 through 223, which you would not want to allow the user to enter accidentally as data. (See the IBM PC BASIC manual, Appendix D: "ASCII Character Codes.")
- 3) The Pick operating system interprets ASCII 251 as buffer start, 252 as subvalue mark, 253 as value mark, 254 as attribute mark, and 255 as segment mark. An editing routine should never allow the entry of characters ASCII 253 through 255 as data. To use foreign language characters, the Pick System provides for as many as four "lead-in" keys that modify the result of the following keystroke. The lead-in keys must be numbered ASCII 251 through 254. (In Release 1983 2.2 and 3.0, ASCII 250 through 254).
- 4) Documentation of the program and the creation of a table of control characters will be easier if consecutive numbers are used whenever possible.

The keys are identified as they appear on the keyboard, repeating the definition when more than one label appears on a single key cap. Thus the Radio Shack 100 does not have a separate DELETE key to return ASCII 127 for the delete function, but uses SHIFT-BACKSPACE instead. The key bears the label "DEL" above the label "BKSP." The key function returning ASCII 127 appears in the following listings as both "DEL" and "SH-BKSP." The same usage applies to the BREAK key, which is used in combination with

SHIFT on the Radio Shack 100 and with CTRL on the IBM PC. The control key combinations not shown simply return the same ASCII number that the key returns without CTRL.

Symbols Used by the Tables

- Not used

The key does not appear on the keyboard, it gives no response, or it returns a regular alphabetic, numeric, or punctuation character rather than a control character. The key can be left in this condition and not used, because pressing it accidentally will cause no harm. For example, pressing CTRL in combination with a key can return the same letter, number, or punctuation mark as pressing the key without CTRL. On Pick Release 1983 for the IBM PC/XT pressing SHIFT-PRINT SCREEN produces no result nor does pressing the cursor arrow keys. The cursor keys are treated instead as a numeric keypad and return the same ASCII number as the regular numeric keys. With CTRL pressed, the cursor keys return nothing. There is no F9, F10, or INSERT key on the Radio Shack 100 keyboard, and there is no PAUSE or PASTE key on the IBM PC keyboard.

Avoid

The key causes an action to occur, even when any available ECHO OFF command has been given. The key should be reset, if possible, unless you wish to retain the action that the key causes. Examples of actions caused are BREAK and PAUSE. When the Radio Shack 100 is used as a terminal, the function keys are used to upload or download files, or to exit terminal mode, and thus are not available for use by an editing routine.

* Value changed

Under the reset column, which provides an example of a keyboard layout for editing after various keys have been reset, an asterisk indicates that the particular key has been changed to the value shown. The descriptions for each device and system indicate which keys can and cannot be reset.

The following numbers indicate the ASCII decimal number actually returned by the keystroke, which under Pick can differ from the ASCII number that Pick Systems list in the KEYBOARDS file. See the Pick keyboard description below for details.

ASCII Value Returned by Each Key Table

Key	Pick			Wyse			Radio Shack		
	Pick R83 IBMXT INPUT Patch	Pick R83 IBMXT PCKEYS User Exit	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT PCKEYS User Exit	Wyse 60 AT PCKEYS User Exit Reset	Radio Shack 100 Alone Reset	Radio Shack 100 Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPC BASIC
CT-A	1	1	1	1	1	1	1	1	1
CT-B	2	2	2	2	2	2	2	2	2
CT-C	3	3	3	3	3	3 #	3	3	3
CT-D	4	4	4	4	4	4	4	4	4

		Pick R83	Pick R83	Wyse 60 AT	Wyse 60 AT			Radio Shack 100	
	Pick R83	R83 IBMXT	IBMXT PCKEYS	Wyse 60 AT	Wyse 60 AT	Radio Shack 100	Radio Shack 100	TELCOM PCKEYS	
Key	INPUT Patch	User Exit	User Exit	User Exit	User Exit	Shack 100 Alone	Shack 100 Alone	User Exit	IBMPC BASIC
CT-E	5	5	5	5	5	5	5	5	5
CT-F	6	6	6	6	6	6	6	6	6
CT-G	7	7	7	7	7	7	7	7	7
CT-H	8	8	8	8	8	8	8	8	8
CT-I	9	9	9	9	9	9	9	9	9
CT-J	10	10	10	10	10	10	10	10	10
CT-K	11	11	11	11	11	11	11	11	11
CT-L	12	12	12	12	12	12	12	12	12
CT-M	255	13	13	13	13	13	13	13	13
CT-N	14	14	14	14	14	14	14	14	14
CT-O	15	15	15	15	15	15	15	15	15
CT-P	16	16	16	16	16	16	16	16	16
CT-Q	17	17	17	17	17	17	17	17	17
CT-R	18	18	18	18	18	18	18	18	18
CT-S	19#	19#	-*	19#	19#	19#	19	19#	19
CT-T	20	20	20	20	20	20	20	20	20
CT-U	21	21	21	21	21	21	21	21	21
CT-V	22	22	22	22	22	22	22	22	22
CT-W	23	23	23	23	23	23	23	23	23
CT-X	24	24	24	24	24	24	24	24	24
CT-Y	25	25	25	25	25	25	25	25	25
CT-Z	26	26	26	26	26	26	26	26	26
ALT-A	-	-	-	-	-	-	-	-	0+30
ALT-B	-	-	-	-	-	-	-	-	0+38
ALT-C	-	-	-	-	-	-	-	-	0+46
ALT-D	-	-	-	-	-	-	-	-	0+32
ALT-E	-	-	-	-	-	-	-	-	0+18
ALT-F	-	-	-	-	-	-	-	-	0+33
ALT-G	-	-	-	-	-	-	-	-	0+34
ALT-H	-	-	-	-	-	-	-	-	0+35
ALT-I	-	-	-	-	-	-	-	-	0+23
ALT-J	-	-	-	-	-	-	-	-	0+36
ALT-K	-	-	-	-	-	-	-	-	0+37
ALT-L	-	-	-	-	-	-	-	-	0+38
ALT-M	-	-	-	-	-	-	-	-	0+50
ALT-N	-	-	-	-	-	-	-	-	0+49
ALT-O	-	-	-	-	-	-	-	-	0+24
ALT-P	-	-	-	-	-	-	-	-	0+25
ALT-Q	-	-	-	-	-	-	-	-	0+16
ALT-R	-	-	-	-	-	-	-	-	0+19
ALT-S	-	-	-	-	-	-	-	-	0+31
ALT-T	-	-	-	-	-	-	-	-	0+20
ALT-U	-	-	-	-	-	-	-	-	0+22
ALT-V	-	-	-	-	-	-	-	-	0+47
ALT-W	-	-	-	-	-	-	-	-	0+17
ALT-X	-	-	-	-	-	-	-	-	0+45
ALT-Y	-	-	-	-	-	-	-	-	0+21
ALT-Z	-	-	-	-	-	-	-	-	0+44

			Pick R83		Wyse		Radio Shack 100		
	Pick R83	Pick R83	IBMXT PCKEYS	Wyse 60 AT	Wyse 60 AT	Radio Shack 100	Radio Shack 100	TELCOM PCKEYS	
Key	INPUT	User Exit	User Exit Reset	User Exit	User Exit Reset	Alone	Alone Reset	User Exit Reset	IBMPC BASIC
ALT-0	-	-	-	-	-	-	-	-	0+129
ALT-1	-	-	-	-	-	-	-	-	0+120
ALT-2	-	-	-	-	-	-	-	-	0+121
ALT-3	-	-	-	-	-	-	-	-	0+122
ALT-4	-	-	-	-	-	-	-	-	0+123
ALT-5	-	-	-	-	-	-	-	-	0+124
ALT-6	-	-	-	-	-	-	-	-	0+125
ALT-7	-	-	-	-	-	-	-	-	0+126
ALT-8	-	-	-	-	-	-	-	-	0+127
ALT-9	-	-	-	-	-	-	-	-	0+128
GPH-f	-	-	-	-	-	130	130	130	-
GPH-SH-F	-	-	-	-	-	238	238	238	-
GPH-i	-	-	-	-	-	142	142	142	-
CT-@	253	253	0*	0	0	0	0	-	0+3
CT-^	30	30	30	30	30	30	30	30	30
CT--	31	31	31	-	-	-	-	-	31
CT-_	31	31	31	31	31	31	31	31	31
CT-[	27	27	27	27	27	27	27	27	27
CT-{	27	27	27	27	27	-	-	-	27
CT-]	29	29	29	29	29	29	29	29	29
CT-}	29	29	29	29	29	-	-	-	29
CT-;	-	-	254*	-	-	-	-	-	-
CT-:	-	-	254*	-	-	-	-	-	-
CT-'	-	-	251*	-	-	-	-	-	-
CT-"	-	-	251*	-	-	-	-	-	-
CT-`	-	-	252*	0	0	-	-	-	-
CT-~	-	-	252*	30	30	-	-	-	-
CT-\	28	28	28	28	28	-	-	-	28
CT-	28	28	28	28	28	-	-	-	28
CT-.	-	-	253	-	-	-	-	-	-
CT->	-	-	253	-	-	-	-	-	-
ALT--	-	-	-	-	-	-	-	-	0+130
ALT-=	-	-	-	-	-	-	-	-	0+131
BKSP	-	8	8	8	8	8	8	8	8
BREAK	3#	3#	3#	3#	3#	3#	3	-	-
DEL	-	193	127*	127	127	127	127	127	0+83
END	-	194	18*	-	18*	-	-	-	0+79
ENTER	255	13	13	13	13	13	13	13	13
ESC	27	27	27	27	27	27	27	27	27
HOME	-	1	17*	30	17*	-	-	-	0+71
INS	-	192	16*	27+114	16*	-	-	-	0+82
LABEL	-	-	-	-	-	#	0	#	-
PG DN	-	196	26*	27+75	26*	-	-	-	0+81
PG UP	-	195	23*	27+74	23*	-	-	-	0+73
PASTE	-	-	-	-	-	0	0	142*?	-

Key	Pick R83		Pick R83		Pick R83		Radio Shack 100		Radio Shack 100	
	IBMXT		IBMXT		IBMXT		TELCOM		TELCOM	
	PCKEYS		PCKEYS		PCKEYS		PCKEYS		PCKEYS	
	INPUT	User	User	Exit	User	Exit	Shack 100	Shack 100	User	IBMPC BASIC
Patch	Exit	Reset	Exit	Reset	Exit	Reset	Alone	Reset	Reset	
PAUSE	-	-	-	-	-	-	19#	19	19#	-
PRINT	-	-	-	-	-	-	#	0	#	-
TAB	9	9	9	9	9	9	9	9	9	9
SH-BKSP	-	8	8	8	8	8	127	127	127	8
SH-DEL	-	-	-	127	127	127	127	127	127	-
SH-END	-	-	-	-	18*	-	-	-	-	-
SH-ENTER	255	13	13	#	13*	13	13	13	13	13
SH-ESC	27	27	27	27	27	27	27	27	27	27
SH-HOME	-	-	-	27+123	17*	-	-	-	-	-
SH-INS	-	-	-	27+113	16*	-	-	-	-	-
SH-PG DN	-	-	-	27+75	5*	-	-	-	-	-
SH-PG UP	-	-	-	27+74	4*	-	-	-	-	-
SH-PASTE	-	-	-	-	-	0	0	142*?	-	-
SH-PAUSE	-	-	-	-	-	3#	3	-	-	-
SH-PRINT	-	-	-	-	-	#	0	-	-	-
SH-PRTSC	-	250	250	27+80	27+122*	-	-	-	-	#
SH-TAB	-	-	-	27+73	9*	9	9	9	9	0+15
CT-BKSP	-	127	127	8	8	8	8	8	8	127
CT-BREAK	3#	3#	3#	3#	3#	3#	3	-	-	3#
CT-DEL	-	-	-	-	-	-	127	127	127	-
CT-END	-	-	-	-	-	-	-	-	-	0+117
CT-ENTER	10	10	13*	13	13	13	13	13	13	13
CT-ESC	27	27	27	27	27	27	27	27	27	27
CT-HOME	-	-	-	-	-	-	-	-	-	0+119
CT-INS	-	-	-	-	-	-	-	-	-	-
CT-PG DN	-	-	-	-	-	-	-	-	-	0+118
CT-PG UP	-	-	-	-	-	-	-	-	-	0+132
CT-PASTE	-	-	-	-	-	0	0	142*?	-	-
CT-PAUSE	-	-	-	-	-	19#	19	19#	-	-
CT-PRTSC	-	-	-	-	-	-	-	-	-	#
CT-TAB	-	-	9*	9	9	9	9	9	9	-
DOWN	-	10	31*	2	31*	31	31	31	31	0+80
LEFT	-	21	29*	6	29*	29	29	29	29	0+75
RIGHT	-	6	28*	4	28*	28	28	28	28	0+77
UP	-	26	30*	1	30*	30	30	30	30	0+72
5	-	250	250	-	-	-	-	-	-	-
SH-DOWN	-	-	-	10	2*	2	2	2	2	-
SH-LEFT	-	-	-	8	1*	1	1	1	1	-
SH-RIGHT	-	-	-	12	6*	6	6	6	6	-
SH-UP	-	-	-	11	20*	20	20	20	20	-
SH-5	-	-	-	-	-	-	-	-	-	-
CT-DOWN	-	-	-	-	-	26	26	26	26	-
CT-LEFT	-	-	-	-	-	17	17	17	17	0+115
CT-RIGHT	-	-	-	-	-	18	18	18	18	0+116

Key	Pick R83	Pick R83	Pick R83	Wyse 60 AT	Wyse 60 AT	Radio Shack	Radio Shack	Radio Shack 100	Radio Shack 100
	IBMXT	IBMXT	IBMXT	PCKEYS	PCKEYS	Shack	Shack	TELCOM	TELCOM
	INPUT	PCKEYS	PCKEYS	User	User	100	100	PCKEYS	PCKEYS
	Patch	User	User	Exit	Exit	Alone	Alone	User	IBMPC
		Exit	Reset	Exit	Reset	Reset	Reset	Reset	BASIC
CT-UP	-	-	-	-	-	23	23	23	-
CT-5	-	-	-	-	-	-	-	-	-
F1	-	-	176*	1+64+13	27+48*	#	248*	#	0+59
F2	-	-	177*	1+65+13	27+49*	#	249*	#	0+60
F3	-	-	178*	1+66+13	27+50*	#	250*	#	0+61
F4	-	-	179*	1+67+13	27+51*	#	251*	#	0+62
F5	-	-	180*	1+68+13	27+52*	#	252*	#	0+63
F6	-	-	181*	1+69+13	27+53*	#	253*	#	0+64
F7	-	-	182*	1+70+13	27+54*	#	254*	#	0+65
F8	-	-	183*	1+71+13	27+55*	#	255*	#	0+66
F9	-	-	184*	1+72+13	27+56*	-	-	-	0+67
F10	-	-	185*	1+73+13	27+57*	-	-	-	0+68
SH-F1	-	-	186*	1+96+13	27+58*	#	-	#	0+84
SH-F2	-	-	187*	1+97+13	27+59*	#	-	#	0+85
SH-F3	-	-	188*	1+98+13	27+60*	#	-	#	0+86
SH-F4	-	-	189*	1+99+13	27+61*	#	-	#	0+87
SH-F5	-	-	190*	1+100+13	27+62*	#	-	#	0+88
SH-F6	-	-	191*	1+101+13	27+63*	#	-	#	0+89
SH-F7	-	-	192*	1+102+13	27+64*	#	-	#	0+90
SH-F8	-	-	193*	1+103+13	27+65*	#	-	#	0+91
SH-F9	-	-	194*	1+104+13	27+66*	-	-	-	0+92
SH-F10	-	-	195*	1+105+13	27+67*	-	-	-	0+93
CT-F1	-	-	-	128	128	#	-	#	0+94
CT-F2	-	-	-	129	129	#	-	#	0+95
CT-F3	-	-	-	130	130	#	-	#	0+96
CT-F4	-	-	-	131	131	#	-	#	0+97
CT-F5	-	-	-	132	132	#	-	#	0+98
CT-F6	-	-	-	133	133	#	-	#	0+99
CT-F7	-	-	-	134	134	#	-	#	0+100
CT-F8	-	-	-	135	135	#	-	#	0+101
CT-F9	-	-	-	136	136	-	-	-	0+102
CT-F10	-	-	-	137	137	-	-	-	0+103
ALT-F1	-	-	-	-	-	-	-	-	0+104
ALT-F2	-	-	-	-	-	-	-	-	0+105
ALT-F3	-	-	-	-	-	-	-	-	0+106
ALT-F4	-	-	-	-	-	-	-	-	0+107
ALT-F5	-	-	-	-	-	-	-	-	0+108
ALT-F6	-	-	-	-	-	-	-	-	0+109
ALT-F7	-	-	-	-	-	-	-	-	0+110
ALT-F8	-	-	-	-	-	-	-	-	0+111
ALT-F9	-	-	-	-	-	-	-	-	0+112
ALT-F10	-	-	-	-	-	-	-	-	0+113
ALT+Num?	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes

After picking a candidate ASCII value to reset a key to, check the “Keys that Return Each ASCII Number” table to see if two different keystrokes will return the same value, or if the same value will be returned by a different keystroke on an different machine. For example, you cannot easily use CTRL-H as a help key, because the ASCII value 8 that it returns is also returned by the TAB key. In the same manner, the ASCII 9 returned by CTRL-I is also returned by the BACKSPACE key, and the ASCII 13 produced by CTRL-M also results from pressing ENTER.

Pressing SHIFT or CTRL in combination with a key sometimes produces a different result than pressing the key alone, and sometimes it does not. Thus one machine might distinguish CTRL-ENTER from ENTER or CTRL-TAB from TAB, while another machine might not.

A difference between this table, “ASCII Value Returned by Each Key,” and the following table, “Keys That Return Each ASCII Number,” is that the latter table shows a final result, in the form of a single ASCII number, for those keys that return more than one character on the Wyse 60 terminal. In the SHOW.ASCII program, ASCII 27, the escape character, has been designated the lead key for character combinations. If the escape character is detected, then the keyboard input routine will look for a second character. If there is a second character, the routine will add 128 to its ASCII number.

Keys That Return Each ASCII Number Table

ASCII	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT Keybd PCKEYS User Exit Reset	Radio Shack 100 Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPC BASIC
0	CT-@ - - - - - - - -	CT-@ CT-` - - - - - - -	CT-@ - LABEL PASTE PRINT SH-PASTE SH-PRINT CT-PASTE CT-PRINT	- - - - - - - - -	- - - - - - - - -
1	CT-A -	CT-A SH-LEFT*	CT-A SH-LEFT	CT-A SH-LEFT	CT-A -
2	CT-B -	CT-B SH-DOWN*	CT-B SH-DOWN	CT-B SH-DOWN	CT-B -
3	CT-C BREAK# CT-BREAK# -	CT-C BREAK# CT-BREAK# -	CT-C BREAK CT-BREAK SH-PAUSE	CT-C - - -	CT-C - CT-BREAK# -
4	CT-D -	CT-D SH-PG UP*	CT-D -	CT-D -	CT-D -
5	CT-E	CT-E	CT-E	CT-E	CT-E
6	CT-F - -	CT-F SH-PG DN* SH-RIGHT*	CT-F - SH-RIGHT	CT-F - SH-RIGHT	CT-F - -

ASCII	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT Keybd PCKEYS User Exit Reset	Radio Shack 100 Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPc BASIC
7	CT-G	CT-G	CT-G	CT-G	CT-G
8	CT-H	CT-H	CT-H	CT-H	CT-H
	BKSP	BKSP	BKSP	BKSP	BKSP
	SH-BKSP	SH-BKSP	-	-	SH-BKSP
	-	CT-BKSP	CT-BKSP	CT-BKSP	-
9	CT-I	CT-I	CT-I	CT-I	CT-I
	TAB	TAB	TAB	TAB	TAB
	-	SH-TAB*	SH-TAB	SH-TAB	-
	CT-TAB*	CT-TAB	CT-TAB	CT-TAB	-
10	CT-J	CT-J	CT-J	CT-J	CT-J
11	CT-K	CT-K	CT-K	CT-K	CT-K
12	CT-L	CT-L	CT-L	CT-L	CT-L
13	CT-M	CT-M	CT-M	CT-M	CT-M
	ENTER	ENTER	ENTER	ENTER	ENTER
	SH-ENTER*	SH-ENTER*	SH-ENTER	SH-ENTER	SH-ENTER
	CT-ENTER*	CT-ENTER	CT-ENTER	CT-ENTER	CT-ENTER
14	CT-N	CT-N	CT-N	CT-N	CT-N
15	CT-O	CT-O	CT-O	CT-O	CT-O
16	CT-P	CT-P	CT-P	CT-P	CT-P
	INSERT*	INSERT*	-	-	-
17	CT-Q	CT-Q	CT-Q	CT-Q	CT-Q
	-	-	CT-LEFT	CT-LEFT	-
	HOME*	HOME*	-	-	-
	-	SH-HOME*	-	-	-
18	CT-R	CT-R	CT-R	CT-R	CT-R
	-	-	CT-RIGHT	CT-RIGHT	-
	END*	END*	-	-	-
	-	SH-END*	-	-	-
19	-	CT-S#	CT-S	CT-S#	CT-S
	-	-	PAUSE	PAUSE#	-
	-	-	CT-PAUSE	CT-PAUSE#	-
20	CT-T	CT-T	CT-T	CT-T	CT-T
	-	SH-UP*	SH-UP	SH-UP	-
21	CT-U	CT-U	CT-U	CT-U	CT-U
22	CT-V	CT-V	CT-V	CT-V	CT-V
23	CT-W	CT-W	CT-W	CT-W	CT-W
	PG UP*	PG UP*	-	-	-
	-	-	CT-UP	CT-UP	-
24	CT-X	CT-X	CT-X	CT-X	CT-X
25	CT-Y	CT-Y	CT-Y	CT-Y	CT-Y
26	CT-Z	CT-Z	CT-Z	CT-Z	CT-Z
	PG DN*	PG DN*	-	-	-
	-	-	CT-DOWN	CT-DOWN	-
27	ESC	ESC	ESC	ESC	ESC
	SH-ESC	SH-ESC	SH-ESC	SH-ESC	SH-ESC
	CT-ESC	CT-ESC	CT-ESC	CT-ESC	CT-ESC
	CT-[	CT-[	CT-[	CT-[	CT-[
	CT-{	CT-{	-	-	CT-{
28	RIGHT*	RIGHT*	RIGHT	RIGHT	-
	CT-\	CT-\	-	-	CT-\

ASCII	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT Keybd PCKEYS User Exit Reset	Radio Shack 100 Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPC BASIC
29	CT- LEFT*	CT- LEFT*	- LEFT	- LEFT	CT- -
	CT-] CT-}	CT-] CT-}	CT-] -	CT-] -	CT-] CT-}
30	UP* CT-^ -	UP* CT-^ CT-^	UP CT-^ -	UP CT-^ -	- CT-^ -
31	DOWN* CT-_ CT--	DOWN* CT-_ -	DOWN CT-_ -	DOWN CT-_ -	- CT-_ CT--
127	- - - CT-BKSP	DEL SH-DEL - -	DEL SH-DEL SH-BKSP -	DEL SH-DEL SH-BKSP -	- - - CT-BKSP
128	-	CT-F1	-	-	-
129	-	CT-F2	-	-	-
130	-	CT-F3 -	- GPH-f	- GPH-f	- -
131	-	CT-F4	-	-	-
132	-	CT-F5	-	-	-
133	-	CT-F6	-	-	-
134	-	CT-F7	-	-	-
135	-	CT-F8	-	-	-
136	-	CT-F9	-	-	-
137	-	CT-F10	-	-	-
176	F1*	F1*	-	-	-
177	F2*	F2*	-	-	-
178	F3*	F3*	-	-	-
179	F4*	F4*	-	-	-
180	F5*	F5*	-	-	-
181	F6*	F6*	-	-	-
182	F7*	F7*	-	-	-
183	F8*	F8*	-	-	-
184	F9*	F9*	-	-	-
185	F10*	F10*	-	-	-
186	SH-F1*	SH-F1*	-	-	-
187	SH-F2*	SH-F2*	-	-	-
188	SH-F3*	SH-F3*	-	-	-
189	SH-F4*	SH-F4*	-	-	-
190	SH-F5*	SH-F5*	-	-	-
191	SH-F6*	SH-F6*	-	-	-
192	SH-F7*	SH-F7*	-	-	-
193	SH-F8*	SH-F8*	-	-	-
194	SH-F9*	SH-F9*	-	-	-
195	SH-F10*	SH-F10*	-	-	-
238	-	-	GPH-SH-F	GPH-SH-F	-
248	-	-	F1*	-	-
249	-	-	F2*	-	-
250	-	-	F3*	-	-
	SH-PRTSC 5 KEYPAD	SH-PRTSC* -	- -	- -	- -

ASCII	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT Keybd PCKEYS User Exit Reset	Radio Shack 100 PCKEYS Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPC BASIC
251	- CT-'*	- -	F4*	- -	- -
252	- CT-"*	- -	F5*	- -	- -
253	- CT-`*	- -	F6*	- -	- -
254	- CT-~*	- -	F7*	- -	- -
255	- CT-.* CT->*	- -	F8*	- -	- -

In the table "Keys and ASCII Number for Each Editing Function," quotes mark the place in each column for those keys and ASCII values that remain the same for all the devices and systems shown on the same row. The ASCII value for each key appears below the key name. For the WYSE 60 AT keyboard, ASCII values greater than 127 have been produced by using escape (ASCII 27) as a lead character, followed by a second character for which the keyboard input routine adds 128 to the character's ASCII value.

When the Radio Shack 100 is employed as a terminal the function keys are reserved for terminal functions, such as uploading or downloading a file, and are not usable for editing purposes. A two-key combination, similar to the use of lead and follow keys for foreign language characters, will be used to replicate the functions keys. Pressing GRPH-f, followed by a number 1 through 0, will serve the same purpose as pressing F1 through F10. Pressing GRPH-SHIFT-F, followed by a number 1 through 0, will serve the same purpose as pressing SHIFT-F1 through SHIFT-F10.

Key and ASCII Number for Each Editing Function Table

Function	ALL	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT PCKEYS User Exit Reset	Radio Shack 100 PCKEYS Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPC Basic
Next Field	ENTER 13	"" ""	"" ""	"" ""	"" ""	"" ""

		Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT PCKEYS User Exit Reset	Radio Shack 100 Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPC BASIC
Function	ALL					
Previous Field	CT-M	""	""	""	""	""
	13	""	""	""	""	""
	TAB	""	""	""	""	""
	9	""	""	""	""	""
	CT-I	""	""	""	""	""
Quit	9	""	""	""	""	""
	-	-	SH-TAB	SH-TAB	SH-TAB	SH-TAB
	-	-	9*	9	9	0+15
	ESC	""	""	""	""	""
	27	""	""	""	""	""
Backspace	CT-[	""	""	""	""	""
	27	""	""	""	""	""
	BKSP	""	""	""	""	""
Delete	8	""	""	""	""	""
	CT-H	""	""	""	""	""
	8	""	""	""	""	""
	DEL	""	""	""	""	""
	-	127*	127	127	127	0+83
Left 1 character	LEFT	""	""	""	""	""
Right 1 character	-	29*	29*	29	29	0+75
	RIGHT	""	""	""	""	""
Up 1 line	-	28*	28*	28	28	0+77
	UP	""	""	""	""	""
Down 1 line	-	30*	30*	30	30	0+72
	DOWN	""	""	""	""	""
Left 1 word	-	31*	31*	31	31	0+80
	CT-A	""	""	""	""	""
	1	""	""	""	""	""
	-	-	SH-LEFT	SH-LEFT	SH-LEFT	-
	-	-	1*	1	1	-
Right 1 word	CT-F	""	""	""	""	""
	6	""	""	""	""	""
	-	-	SH-RIGHT	SH-RIGHT	SH-RIGHT	-
Start of Field	-	-	6*	6	6	-
	CT-Q	""	""	""	""	""
	17	""	""	""	""	""
	-	-	-	CT-LEFT	CT-LEFT	CT-LEFT
	-	-	-	17	17	0+115
End of Field	-	HOME	HOME	-	-	HOME
	-	17*	17*	-	-	0+71
	CT-R	""	""	""	""	""
	18	""	""	""	""	""
	-	-	-	CT-RIGHT	CT-RIGHT	CT-RIGHT
	-	-	-	18	18	0+116
	-	END	END	-	-	END

Function	ALL	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT PCKEYS User Exit Reset	Radio Shack 100 User Alone Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMPC BASIC
Top of Page	- CT-T 20 - -	18* "" "" - -	18* "" "" SH-UP 20*	- "" "" SH-UP 20	- "" "" SH-UP 20	0+79 "" "" - -
Bottom of Page	CT-B 2 - -	"" "" - -	"" "" SH-DOWN 2*	"" "" SH-DOWN 2	"" "" SH-DOWN 2	"" "" - -
Previous Page	CT-W 23 - - -	"" "" - - PG UP	"" "" - - PG UP	"" "" CT-UP 23	"" "" CT-UP 23	"" "" - - PG-UP
Next Page	CT-Z 26 - - -	"" "" - - PG DN	"" "" - - PG DN	"" "" CT-DOWN 26	"" "" CT-DOWN 26	"" "" - - PG DN
Start of Form	CT-D 4 - - -	"" "" - - -	"" "" SH-PG UP 4*	"" "" - - -	"" "" - - -	"" "" - - CT-PG UP
End of Form	CT-E 5 - - -	"" "" - - -	"" "" SH-PG DN 5*	"" "" - - -	"" "" - - -	"" "" - - CT-PG DN
Jump to Field	CTRL-J 10	"" ""	"" ""	"" ""	"" ""	"" ""
Page of Form	CT-X,Num 24	"" ""	"" ""	"" ""	"" ""	"" ""
Over-Type, Insert	CT-O 15	"" ""	"" ""	"" ""	"" ""	"" ""
Clear Field	CT-C 3	"" ""	"" ""	"" ""	"" ""	"" ""
Get Previous Field	CT-G 7	"" ""	"" ""	"" ""	"" ""	"" ""
Select text	CT-L 12	"" ""	"" ""	"" ""	"" ""	"" ""
Copy Selected text	CT-Y 25	"" ""	"" ""	"" ""	"" ""	"" ""

Function	ALL	Pick R83 IBMXT PCKEYS User Exit Reset	Wyse 60 AT PCKEYS User Exit Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	Radio Shack 100 TELCOM PCKEYS User Exit Reset	IBMP C BASIC
		Reset	Reset	Reset	Reset	BASIC
Cut Selected text	CT-U 21	""	""	""	""	""
Paste Selected text	CT-P 16	""	""	""	""	""
	-	INS	INS	-	-	INS
Erase Value	CT-V 22	16*	16*	-	-	0+82
New Item ID Number	CT-N 14	""	""	""	""	""
Next Item Short Cut	CT-K 11	""	""	""	""	""
Help	-	F1	F1	-	-	F1
	-	176*	176*	-	-	0+59
	-	-	-	-	GPH-f, 1 130, "1"	-
Prev Item Short Cut	-	F2	F2	-	-	F2
	-	177*	177*	-	-	0+60
	-	-	-	-	GPH-f, 2 130, "2"	-
Start of List	-	F3	F3	-	-	F3
	-	178*	178*	-	-	0+61
	-	-	-	-	GPH-f, 3 130, "3"	-
Resume List	-	F4	F4	-	-	F4
	-	179*	179*	-	-	0+62
	-	-	-	-	GPH-f, 4 130, "4"	-
	-	-	-	-	-	-

IBM PC BASIC

The ShowAscii.bas (for show ASCII) program was used to obtain the values indicated in the tables of key values for the IBM PC running IBM PC BASIC. The only method employed to reset the key values was to disable the function keys so that they would return an ASCII value only. No other resetting was necessary.

ShowAscii Program, Documentation Version

```
'* ShowAscii.bsd David L. Clark 04/23/1988
'* Show ASCII number for keystroke.
'* IBM PC Basic program. Documentation version.
'* IBM PC BASICA 3.0
'* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
'* Extended keys return CHR$(0) + 2nd character.
'*****
'* Assign Constants, Initial Values *
'*****
```

```

DEFINT A-Z
LOCATE ,,1                                '* Turn on cursor
Key$ = ""                                  '* Will contain keystroke
Quote$ = CHR$(34)                          '* Quote mark for display
'*****
'* Reset Keys *
'*****
'* Set function keys to return ASCII code only
KEY OFF : FOR I = 1 TO 10 : KEY I, "" : NEXT I
'*****
'* Display Exit Method *
'*****
PRINT "Enter "; Q$; "@"; Q$; " to end"
'*****
'* Get Keystroke Loop *
'*****
WHILE Key$ <> "@"
    PRINT "Press a Key: ";
    Key$ = ""
    WHILE Key$ = ""
        Key$ = INKEY$
    WEND
    Ascii$ = STR$(ASC(LEFT$(Key$,1)))      '* 1st or only char
    '* Check for more than 1 ASCII value from 1 keystroke
    Length = LEN(Key$)
    '* Any extra characters are already in Key$.
    '* No separate input required, as in Pick Basic.
    FOR I = 2 TO Length                    '* Will not execute if Length = 1
        Ascii$ = Ascii$ + " " + STR$(ASC(MID$(Key$,I,1)))
    NEXT I
    '*****
    '* Version to Show ASCII Only *
    '*****
    PRINT "    ASCII = "; Ascii$
    '*****
    '* Version to Show ASCII and Character *
    '*****
    '* This version has been commented out
    PRINT Key$; " = ASCII "; Ascii$
' *
WEND
'*****
'* End Program *
'*****
LOCATE ,,0                                '* Turn off cursor
SYSTEM                                    '* Return to operating system

```

Microsoft BASIC (Radio Shack 100)

POKE

The following POKE commands can be used to reset the Radio Shack 100 keys to return the ASCII values shown in the tables of key values in this section.

To disable the PAUSE and BREAK, and reset the function keys:

```

POKE 63056, 64 '* Reset keys
POKE 63056, 0  '* Restore keys

```

The POKE to memory address 63056 only affects the Radio Shack 100 when used on a stand-alone basis. The POKE has no effect when the Radio Shack 100 is used through TELCOM as a terminal connected to a Pick System.

The command POKE 63056, followed by a comma and any number from 64 to 127, disables the BREAK and PAUSE keys for INKEY\$ input from the keyboard, causing them to simply return their ASCII values of 3 and 19. CTRL-C and CTRL-S will also return ASCII values of 3 and 19, rather than causing a BREAK or PAUSE, therefore these keys can be used for editing functions. However, BREAK will not be disabled for input with the INPUT\$ or LINE INPUT statements. Thus you can allow the user to break out of the program at specified points. (For details on the BREAK key, see the section: "Keyboard: BREAK Key.")

The command "POKE 63056, 64" also sets the function keys F1-F8 to ASCII 248-255 and retains the PASTE key at ASCII 0 (assuming that a BASIC CLEAR command has cleared the paste buffer).

With regard to the function keys, they are most useful for editing when assigned the ASCII values 248-255. It is doubtful that the user would wish to use the graphic characters otherwise returned by those values. Any POKE from 64 through 127 or from 192 through 255 will set the function keys F1-F8 to 248-255, and also disable the "ON KEY GOSUB ..." function key interrupt. Any POKE from 0 through 63 will restore the function keys to normal and reenables the function key interrupt. (For ON KEY, see the section: "Keyboard: Function Keys.")

Any POKE from 128 through 191 will neither restore the function keys nor set them to 248-255. Key F1 will return the letter "s" plus a Carriage Return; F2 and F3 will each return a space followed by a double quote mark; F4, F6, and F7 will return ASCII 0; F5 and F8 will return ASCII 13 for Carriage Return.

With regard to the PASTE key, with no POKE, or with any POKE at the address 63056 from 0 through 128, PASTE returns ASCII 0. From POKE 129 through 255 the ASCII value returned by PASTE increases by 2, from 2 to 254.

If there are characters in the paste buffer, pressing PASTE will return the ASCII value of each of those characters. The CLEAR command in BASIC, which must precede any program needing more than 256 bytes of string memory space, clears the paste buffer, restoring PASTE to ASCII 0.

After POKE 64 the LABEL key will continue to return ASCII 0, and cause the BASIC function key command labels "Files, Load, Save, Run, List, Menu" to show above the function keys on the bottom of the screen. The computer will hang if the PRINT key is pressed with no printer attached.

If a BREAK should occur with the LINE INPUT or INPUT\$ statement after a POKE 64, the function keys will not work when the user is dumped into BASIC. The user must type "MENU" to get back to the Radio Shack Menu. When BASIC is reentered from the Radio Shack Menu, however, the function keys will work as normal.

The second POKE example, "POKE 63056, 0", will restore normal conditions. Ending the program with the command "MENU" rather than with "END" will have the same effect, thus this POKE is not necessary unless it precedes a direct exit into BASIC.

The following is a summary of the effects of different POKE values at address 63056:

- 0 – 63: restores **BREAK** and **Pause**
restores function keys
restores **PASTE** to ASCII 0
- 64 – 127: disables **BREAK** and **PAUSE**
sets function keys F1-F8 to 248-255
restores **PASTE** to ASCII 0
- 128 – 191: disables **BREAK** and **PAUSE**
does not restore function keys or set them to 248-255
sets **PASTE** to ASCII value increasing by 2 from 0 at POKE 128
- 192 – 255: disables **BREAK** and **PAUSE**
sets function keys F1-F8 to 248-255
sets **PASTE** to ASCII value increasing by 2 from 0 at POKE 128

To disable the **PRINT** key and printer

```
'* Disable the PRINT key and printer
POKE 64228, 136
POKE 64229, 20
'* Restore the PRINT key and printer
POKE 64228, 243
POKE 64229, 127
```

Pressing the **PRINT** key or giving the **LPRINT** command in **BASIC** will have no effect. The principal reason for employing this **POKE** is that pressing the **PRINT** key when no printer is attached will cause the Radio Shack 100 to hang.

To disable the **LABEL** key:

```
POKE 74173, 0      '* Disable the LABEL key
POKE 64173, 1      '* Restore the LABEL key
```

Pressing the **LABEL** key normally causes the function key labels to appear at the bottom of the screen. The same applies for all of the Radio Shack 100 built-in programs: **BASIC**, **TEXT**, and **TELCOM**. The **LABEL** key is a toggle, pressing it will alternately cause the labels to appear and then to disappear. The principal reason for employing this **POKE** is to prevent the label line from appearing when it is not desired within a **BASIC** program.

Ending a **BASIC** program with the **MENU** command, to return to the Radio Shack Menu rather than remaining in **BASIC**, will also restore the **LABEL** key.

Action should be taken to clear the paste buffer, otherwise the contents of the buffer will spill out if the user accidentally presses the **PASTE** key. Thus every **BASIC** program should begin with a **CLEAR** statement, to clean up any debris from the paste buffer:

```
CLEAR                                     '* Set the PASTE key to ASCII 0
```


The CLEAR command also can be employed in the form:

CLEAR String.Space, High.Memory

to clear a defined amount of string space in memory, and can include a high memory parameter to protect space used for machine language programs. If no amount of string space is specified, the default is 256 bytes.

The CLEAR command will clear the paste buffer, thereby setting the PASTE key to return ASCII zero for input with the INKEY\$ statement when the Radio Shack 100 is used on a stand-alone basis. (Note that when ENTER alone is pressed for INKEY\$, the value returned is ASCII 13 rather than null.)

When the Radio Shack 100 is to be employed as a terminal, pressing the PASTE key after the paste buffer has been cleared will have no effect. No ASCII zero will be returned that could be utilized by an editing routine. Therefore another method of accomplishing a paste action must be employed, such as using CTRL-P. If any characters have been left in the paste buffer before exiting to terminal mode, those characters will spill out when the PASTE key is pressed. (For more information on exiting to terminal mode, see the section: "System: Booting, Start-Up: Terminals.")

When the Radio Shack 100 is used as a terminal through TELCOM, the function keys are reserved for TELCOM functions and are not reachable for editing purposes. The following commands will add abilities to the function keys in TELCOM only:

```
'* Set F6 to display directory of files on the Radio Shack 100
POKE 64268, 58
POKE 64269, 31
'* Restore F6 to no effect
POKE 64268, 247
POKE 64269, 127
'* Set F7 to show remaining memory available on the Radio Shack 100
POKE 64270, 172
POKE 64271, 126
'* Restore F7 to no effect
POKE 64270, 247
POKE 64271, 127
```

The ShwAsc (for show ASCII) program was used to obtain the values indicated in the tables of key values in this section, when the Radio Shack 100 was employed on a stand-alone basis. (For the values returned when the 100 was employed as a terminal, see the PICK/BASIC program "SHOW.ASCII.") The program contains the various methods explained for resetting key values. Comment out these sections selectively to observe their effect.

ShwAsc Program, Documentation Version

```
'* ShwAsc.do David L. Clark 04/25/1988
'* Show ASCII for keystroke
'* Microsoft Basic (Radio Shack 100) program. Documentation version.
'* Radio Shack 100
'* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
'* A keystroke will return 1 char only
'*****
```

```

'* Reset Keys *
*****
CLEAR          '* Clear Paste buffer, set PASTE key to 0
POKE 63056, 64 '* Disable BREAK, PAUSE, set function keys to 248-255
POKE 64173, 0   '* Disable LABEL key
POKE 64228, 136 : POKE 64229, 20 '* Disable PRINT key, printer
*****
'* Assign Constants, Initial Values *
*****
DEFINT A-Z
PRINT CHR$(27)+"P"          '* Turn on cursor
Key$ = ""                   '* Will contain keystroke
Quote$ = CHR$(34)           '* Quote mark for display
*****
'* Display Exit Method *
*****
PRINT "Enter "; Quote$; "@"; Quote$; " to end"
*****
'* Get Keystroke Loop *
*****
FOR I = 1 TO 2              '* I will be reset
    PRINT "Press a key: ";
    FOR J = 1 TO 2          '* J will be reset
        Key$ = INKEY$
        IF Key$ = "" THEN J = 1 ELSE J = 2
    NEXT J
    Ascii = ASC(Key$)
    *****
    '* Version to Show ASCII Only *
    *****
    PRINT "    ASCII = "; Ascii
    *****
    '* Version to Show ASCII and Character *
    *****
    '* Version shown commented out
    PRINT K$; "    ASCII = "; A
    *****
    '* Test for Exit *
    *****
    IF Key$ = "@" THEN I = 2 ELSE I = 1
NEXT I
*****
'* End Program *
*****
'* Restore Keys *
*****
POKE 63056, 0 '* Restore BREAK, PAUSE, functions. MENU also restores
POKE 64173, 1 '* Restore LABEL key. MENU also restores.
POKE 64228, 243 : POKE 64229, 127 '* Restore PRINT key, printer.
END

```

PICK/BASIC

Pick is a multiuser system. It will allow input not only from the IBM PC/XT keyboard, but also from remote terminals. The examples cover editing input into a Pick System from three different sources: the IBM PC/XT keyboard, a Wyse 60 terminal using the AT-style keyboard, and the Radio Shack Model 100, used as a terminal.

The following descriptions will illustrate techniques for resetting the keys on the IBM PC/XT keyboard, considered by Pick to be line zero, and the Wyse 60 terminal, which Pick will access through serial port one or two as line one or line two. The methods for resetting the IBM PC/XT keyboard and the Wyse 60 keyboard are completely different.

Under Pick, the ASCII number actually returned by a keystroke can differ from the number that Pick Systems lists in the KEYBOARDS file, shown later in this section. For example, pressing the ENTER or the CTRL-M key will cause the Pick BASIC INPUT statement to return ASCII 255 rather than the ASCII 13 that one would expect. The Pick System interprets ASCII 13 as ASCII 255, which under Pick is the segment mark, to signify the end of input. If a user exit is employed to capture the keystroke, ASCII 13 is returned. If ASCII 255 is deliberately assigned to a key, then pressing the key has no effect. Therefore, you should reset to ASCII 255 any key that you wish to disable. In the examples shown, CTRL-S has been reset to ASCII 255 on the IBM PC/XT keyboard to prevent the key from causing a pause. (The Wyse 60 will not allow CTRL-S to be reset.)

On the IBM PC/XT keyboard under Pick, CTRL-@ will return ASCII 253 rather than ASCII zero, whether input is through the INPUT statement or a user exit. But if the keyboard is reassigned, with the procedure shown below, then CTRL-@ will return ASCII zero. CTRL-@ will return ASCII zero on the Wyse 60 and on the Radio Shack 100 terminal whether input is through the INPUT statement or a user exit.

After reassignment, the keys reset to ASCII 251, 252, 253, and 254 on the IBM PC/XT keyboard will be used as lead keys. Pressing them has no effect until a follow key is pressed. That procedure is used to return foreign language characters, as shown in the description of foreign language characters later in this section. The procedure is not available for terminals.

Suppressing Control Character Echo

In PICK/BASIC for the IBM PC/XT, Release 1983, Version 2.0, the INPUT command will echo control characters to the screen, despite the ECHO OFF statement. To use control characters for editing in PICK/BASIC one of the following actions must be taken:

- 1) Apply the patch from Pick Systems described in the following section, and use the TERM command to reset the backspace function.
- 2) Employ a user exit to return a keystroke, rather than using INPUT. A user exit will not require the patch, but the patch will cause no harm.
- 3) Reset the ASCII values on the keyboard.

More than one method can be used in combination. You probably will wish to reset certain values on the keyboard even after employing the patch or a user exit. The method of resetting the keys will not work by itself unless all of the keys needed for editing can be reset. All of the keys can be reset on the IBM PC/XT keyboard, but not on most terminals.

The user exit method was employed to return the values shown in the tables under the subheading "PCKEYS User Exit." The PCKEYS user exit method was chosen because that utility program also was necessary in order to enable use of the cursor keys on the IBM PC/XT keyboard. The differences in the values returned by using INPUT rather than

the user exit are that keystrokes that would return ASCII 13 with the user exit will return ASCII 255 with INPUT, and the backspace and delete functions must be reset. An additional difference between the two methods is that ECHO OFF must be used with INPUT, but is not necessary with a user exit.

Patch the INPUT Command

The patch is necessary to make ECHO OFF function properly. Without the patch, the INPUT statement will echo control characters to the screen, despite the ECHO OFF command. After the patch, ECHO OFF will not be necessary to suppress the echoing of ASCII characters 1 through 31, but will be needed to prevent the echoing of all other characters. The editing routine probably will use some characters, such as ASCII 127 for delete, to perform editing functions, and these characters should not be echoed.

Although Pick allows resetting of the IBM PC/XT keyboard, the keys can be set only to single ASCII values (unlike the IBM PC DOS, which would use a combination such as ASCII 0 followed by another character). For the editing routine to use the function keys, for example, you would probably set the keys to values above ASCII 127. In the examples shown, you might set the keys F1 through F10 to ASCII values 176 through 185, which represent graphics characters that would not be used for data entry. ECHO OFF will be necessary to prevent the echoing of these characters to the screen.

The great disadvantage of the ECHO OFF command is that it makes the BASIC debugger much more difficult to use. If there is a break in the program, you must type blindly. There is no way to turn the echo back on from within the debugger.

If any mistakes are made while entering the patch, or if results are not satisfactory, reload the ABS section on the Pick System diskettes. Place System Diskette #1 in Drive A: and press CTRL-ALT-DEL. At the prompt:

```
OPTIONS (A,F,K)
```

Respond by entering:

```
A
```

To apply the patch so that ECHO OFF will function properly, enter the system debugger (not the BASIC debugger) by hitting the BREAK key while at Pick TCL level. An "I" for interrupt will appear followed by a number indicating the operating system frame at which the break occurred. On the next line will appear an exclamation mark, which is the system debugger's prompt character:

```
I 6.129  
!
```

At the exclamation mark, enter the frame location desired, followed by a semicolon and the number of bytes that will be changed:

```
!6.D5;14
```

Pick will display 14 hexadecimal bytes, followed by an equal sign. Enter a period, followed by 14 bytes of 90 (nine zero) to cancel out the Pick code:

.909090909090909090909090909090

Repeat the process for another location at the next exclamation mark prompt. Enter the location and length to change:

!6.185;10

Now enter a period and 10 bytes of 90 (nine zero):

.909090909090909090909090

Enter the next location and length to change:

!6.18F;16

Enter a period and 16 bytes of 90 (nine zero):

.90909090909090909090909090909090

Enter another location and length to change:

!6.2F;11

Enter a period and 11 bytes of 90 (nine zero):

.9090909090909090909090

To blank out the Line Feed to ASCII 255, so that CTRL-J will return ASCII 10 rather than 255, enter the final location and length to change:

!6.95;6

Enter a period and 6 bytes of 90 (nine zero):

.909090909090

Now end the session by entering:

END

The BACKSPACE key will work at Pick TCL level after the patch, but BACKSPACE will not return any ASCII value to the INPUT statement in an editing routine, because Pick will continue to echo the character. This aggravation applies not only to the IBM PC/XT machine, but also to terminals connected to it. The TERM Pick TCL command can be used to set the backspace function to another value, leaving the BACKSPACE key to return ASCII 8 to the editing routine. For example, if you set the backspace parameter in the

TERM command to ASCII 2, for CTRL-B, then pressing CTRL-B will not cause a backspace, rather it will echo the “happy face” graphics character for ASCII 2 to the screen, even if ECHO is off. The BACKSPACE key will no longer perform a backspace at Pick TCL, but the key will return an ASCII 8 to the editing routine. The result of resetting the parameter is that the backspace function has been disabled.

If the changed backspace function parameter is set to a number above 127, issuing the TERM command to view the parameters set will show a negative number. For example, setting the parameter to ASCII 219 will cause the parameter display to read -37. But if you type ASCII 219 on the keyboard (by holding down the ALT key and typing 219 on the numeric keypad) the graphics character for 219, which is a reverse blank, will show on the screen, despite an ECHO OFF command.

The disadvantage of resetting the backspace function for editing is that it will not be usable at the operating system level. Therefore the program should restore the backspace function to ASCII 8 (or to another value used by the terminal) before exiting. The backspace function also will not be available for use within the BASIC debugger.

To reset the backspace function, within a Pick BASIC program:

```
EXECUTE \ TERM ,,,, \ : New.Value; * 1 space, 5 commas after TERM
```

To restore the backspace function to the BACKSPACE key:

```
EXECUTE \ TERM ,,,, 8 \;
```

Rather than resetting the Backspace function with TERM, you might reset the value returned by the BACKSPACE key, as shown in the description of resetting the keyboard. For example, the BACKSPACE key could be reset to return ASCII 219, and the editing routine would look for ASCII 219 as signifying that a backspace should be performed. Before exiting, the BACKSPACE key would be reset to a normal ASCII 8. For use in the BASIC debugger, another character, such as CTRL-BACKSPACE, would retain the backspace function.

Resetting the PC/XT keyboard, however, will not solve the problem for terminals. The TERM command must be given from each terminal, or the value returned by the BACKSPACE and DELETE keys must be changed for each terminal. The Wyse 60 terminal will allow the values returned by these keys to be changed by downloading a Pick BASIC program, but the Radio Shack Model 100 will not. Therefore the TERM command must be given to change the backspace function on the Radio Shack 100.

Employ a User Exit in Place of the INPUT Command

The ECHO OFF command will not be necessary to suppress the echoing of control or other characters when a user exit is employed to return a keystroke. Those characters that should be shown, the normal data entry characters, must be displayed on the screen by CRT or PRINT statements in the program.

Several public-domain user exits are available for input. The routine is relatively short and simple, but it requires the use of the Pick Assembler, which is not an easy matter. The following Assembly Language routine will return a single character, without echo:

Frame	Frame.Num
MOV	TSBEG,TS
INC	TS
READ	TS
MCC	SM,TS
MOV	TS,TSEND
RTN	

In place of Frame.Num, provide the number of an unused frame. Frame 511 is an obvious choice, because it immediately follows the Pick Operating System. Issue the DUMP command in the form:

```
DUMP 511,X
```

to see if the frame appears to be filled with zeroes or another single repeating character. The CompuSheet spreadsheet program from Interactive Systems, Inc. already uses frame 511.

After a frame has been chosen and the program run through the Pick Assembler, the user exit would be employed in the following manner in a PICK/BASIC program. Note the specific reference to the frame number into which the routine was placed. If frame 511 was not used, the correct frame number must be applied. To preserve the portability of the program, the program might first read a configuration or parameters file that would tell the program what value to place in the variable Frame.Num.

```
* Set User Exit in the Assign Constants section of Basic program
Frame.Num = 511;                * Frame holding assembly routine
Input.User.Exit$ = "U2" : OCONV(Frame.Num, "MCDX") "R%3"
* Input keystroke
CRT "Press a key: " :
Key$ = OCONV(0, Input.User.Exit$)
```

Rather than paying \$300 for the Pick Assembler and mastering its use, you could employ the user exit included with the PC-KEYS utility package available for \$50 from the Modular Software Corporation. (The package is available from any Pick dealer or from Alycona Computer Consultants, 7080 Hollywood Blvd., Suite 718, Hollywood, CA 90028-6933, (213) 462-1549.) The PC-KEYS user exit would be employed in a PICK/BASIC program in exactly the same manner as the user exit just shown.

PC-KEYS also enables the use of the cursor keys on the IBM PC/XT and IBM PC/AT, and thus is needed for PC-type editing.

After transferring the PC-KEYS file, the following command must be run each time that the Pick Operating System is booted:

```
RUN PC-KEYS INSTALL
```

While INSTALL is running, you must not press any key, or the system might hang, requiring a restart.

The PC-KEYS installation, and any subsequent change in the PC-KEYS parameters, must be run from the SYSPROG account, because PC-KEYS employs the LOCK-

FRAME and MLOAD verbs that normally are available only in SYSPROG. On a PC/XT running release 1983 version 2.0, when attempts were made to install PC-KEYS on another account the following message appeared:

```
Invalid Lock-Frame address# 0E00
```

For convenience, PC-KEYS could be installed automatically every time the system was started by placing the following lines in the COLDSTART PROC, which resides in the file SYSPROG-PL in the SYSPROG account:

```
C Install PC-KEYS from Modular Software to enable cursor keys
C Must be installed from SYSPROG Account
HLOGTO SYSPROG
P
HRUN PC-KEYS INSTALL
P
```

(For more information on the COLDSTART PROC, see the section: "System: Booting, Start-Up.")

The first time that the INSTALL program is run, it will confirm the Pick version in use and ask for an available frame number, specifying frame 511 as the default choice.

During this first install, PC-KEYS also will allow the user to change the ASCII values returned by pressing the keys that PC-KEYS enables: the cursor keys, PAGE UP, PAGE DOWN, HOME, END, INSERT, DELETE, and SHIFT-PRINT SCREEN. PC-KEYS displays a list of the keys and asks the user if any are to be changed. The user should type in the new ASCII value desired. PC-KEYS will allow up to four characters to be returned by a single keystroke.

The table "ASCII Value Returned by Each Key," at the beginning of this section, which is "Keyboard: Change Keys," shows the original PC-KEYS values for each of these keys, and the suggested values to which the keys could be changed for maximum editing compatibility across different systems and terminals. Use the table to alter the PC-KEYS configuration to the suggested values.

To change the key values or other PC-KEYS parameters after an initial installation (on PC-KEYS version 1.6), enter the following command:

```
RUN PC-KEYS CONFIG
```

The same installation program run previously will appear. (On earlier versions of PC-KEYS, you would instead delete the CONFIG item in the PC-KEYS file and run INSTALL again. Earlier versions did not allow the user to reset the values returned by the keys.)

When installing a new version of PC-KEYS, it might be necessary to delete the old version first. In the SYSPROG account, enter the command:

```
DELETE-FILE PC-KEYS
```


Then follow the PC-KEYS instructions to load the program from a diskette. If the new version still will not install, check that the current account is SYSPROG with the WHO command:

WHO

The answer should appear:

0 SYSPROG

If the new PC-KEYS still will not take, there might be a problem with the frame in which PC-KEYS is trying to place its Assembly Language routines. To clear out frame 511, reload the ABS (for Absolute Frame Numbers) section of the Pick partition. Place Pick Systems Diskette number one in the A: Drive and either press CTRL-ALT-DELETE or give the command REBOOT. When Pick presents its list of options, choose "A" to reload the ABS only. Now delete the old PC-KEYS file, if it is still present; make sure that the current account is SYSPROG, and then load and install PC-KEYS again.

IN

Under the newer versions of Pick, the IN command will accomplish single character input without an echo in the following manner:

```
* Input keystroke
CRT "Press a key: " :
IN Ascii.Num
Key$ = CHAR(Ascii::Num)
```

Version 2.2 of Release 1983, and open architecture will, according to Pick Systems, enable the use of the cursor keys. However, any characters except the control characters below ASCII 32 will be echoed to the screen. ASCII 127, most used for the DELETE key, will echo to the screen a character that resembles a little house, unless ECHO OFF is used.

Reset the IBM PC/XT Keyboard for Use with Pick

The new ASCII values were chosen because the graphics characters in the IBM extended ASCII set run from 176 through 223. The user presumably would not enter the graphics characters as data. The lead keys for foreign language characters must be numbered ASCII 251 through 254. Values greater than ASCII 252 should not be allowed for input as data because Pick interprets these characters as system delimiters. Pick will treat ASCII 255 as a null and ignore the keystroke.

The Pick control keys return the same value whether or not SHIFT is pressed. Thus CTRL-: and CTRL-; return the same value. The "Pick Standard USA Keyboard" table shows the ASCII value, in hexadecimal, Pick lists for each key. As noted, the value returned to an editing routine might be different. For example, if a key is assigned to ASCII 255 (FF in hexadecimal) pressing the key has no effect.

The hexadecimal key values appear in a file supplied by Pick Systems for the IBM PC/XT. The filename is KEYBOARDS, the item ID is USA, and the file is contained in the SYSPROG account. Alternative keyboards appear in the same KEYBOARDS file in items with item IDs such as FRENCH, SPANISH, and BRITISH.

The hexadecimal values contained in each item can be used to reset the keyboard. You can reset the keys to the ASCII values of IBM extended characters that would return foreign characters such as “ñ” with a tilde.

Pick Standard USA Keyboard Table

													Keys
ESC	1	2	3	4	5	6	7	8	9	0	-	=	BK
TAB	Q	W	E	R	T	Y	U	I	O	P	[	]	EN
CT	A	S	D	F	G	H	J	K	L	;	'		
SH	\	Z	X	C	V	B	N	M	,	.	/	SH	PRTSC ALT SPC CAPS

Shifted Keys

27	33	64	35	36	37	94	38	42	40	41	95	43	8																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						

Unshifted Keys

27	49	50	51	52	53	54	55	56	57	48	45	61	8				Dec	
1B	31	32	33	34	35	36	37	38	39	30	2D	3D	08				Hex	
9	113	119	101	114	116	121	117	105	111	112	91	93	13				Dec	
09	71	77	65	72	74	79	75	69	6F	70	5B	5D	0D				Hex	
255	97	115	100	102	103	104	106	107	108	59	39	96					Dec	
FF	61	73	64	66	67	68	6A	6B	6C	3B	27	60					Hex	
255	92	122	120	99	118	98	110	109	44	46	47	255	42		255	32	255	Dec
FF	5C	7A	78	63	76	62	6E	6D	2C	2E	2F	FF	2A		FF	20	FF	Hex

Control Keys

27	255	0	255	255	255	30	255	255	255	255	31	255	127				Dec	
1B	FF	00	FF	FF	FF	1E	FF	FF	FF	FF	1F	FF	7F				Hex	
255	17	23	5	18	20	25	21	9	15	16	27	29	10				Dec	
FF	11	17	05	12	14	19	15	09	0F	10	1B	1D	0A				Hex	
255	1	19	4	6	7	8	10	11	12	255	255	255					Dec	
FF	01	13	04	06	07	08	0A	0B	0C	FF	FF	FF					Hex	
255	28	26	24	3	22	2	14	13	255	255	255	255	255		255	32	255	Dec
FF	1C	1A	18	03	16	02	0E	0D	FF	FF	FF	FF	FF		FF	20	FF	Hex

The keyboard can be reset with the SET-KBRD program supplied by Pick Systems in the SYSPROG-BP file in the SYSPROG account. The program is called by a PROC also named SET-KBRD in the MD. The format of the command is the following:

SET-KBRD Filename Item.ID

To reset the keyboard to the French alternative, in the SYSPROG account, enter:

SET-KBRD KEYBOARDS FRENCH

To return to the USA keyboard:

SET-KBRD KEYBOARDS USA

In the same manner the function keys can be reset from the SYSPROG account with the SET-FUNC program in the SYSPROG-BP file, called by the SET-FUNC PROC in the MD. Alternate sets of function key definitions are contained in the FUNCKEYS file. To reset the function keys enter the command:

SET-FUNC Filename Item.ID

To restore the function keys to null use the NULL item supplied in the file:

SET-FUNC FUNCKEYS NULL

The keyboard redefinition programs supplied by Pick are quite slow. They are not suitable for a data entry program that must reset the keys for editing purposes each time that the program begins, and then restore the keys to their normal values when the program concludes. Pick resets the keyboard and the function keys with user exits. Rather than using the SET-KBRD and SET-FUNC programs, a data entry program can employ the user exits directly to reset and then restore the keys.

The key-resetting user exits require a string containing the ASCII values, in hexadecimal, for all the keys on the keyboard and for all the function keys. Each hexadecimal value must be two digits in length, thus numbers under 10 must include a leading zero. The table shown for the "Pick Standard USA Keyboard" gives the hexadecimal values to be used, in the order in which they would appear.

To restore the standard USA keys after having changed them, you could place the hexadecimal characters shown in the Pick Standard USA Keyboard into a single long string without separating spaces or Carriage Returns, and apply the user exit "U80BA":

```
Keyboard$ = "1B21402324255E262A28295F2B080051574552545955494F507
B7D0DFF4153444647484A4B4C3A227EFF7C5A584356424E4D3C3E3FFF00F
F20FF1B313233343536373839302D3D080971776572747975696F
705B5D0DFF6173646667686A6B6C3B2760FF5C7A786376626E
6D2C2E2FFF2AFF20FF1BFF00FFFFFF1EFFFFFFF1FFF7FFF1117
0512141915090F101B1D0AFF0113040607080A0B0CFFFFFFFFF1
C1A180316020E0DFFFFFFFFFFFFFFFF20FF"
Restored.Keys = ICONV(Keyboard$, "U80BA")
CRT "USA standard keyboard restored"
```

To change the keys to new values, place the new values in the string instead.

The Pick System begins with all of the function keys defined as null (hexadecimal FF, decimal 255). To restore the function keys to null after changing them, place 40 repetitions of the hexadecimal number FF in a single string without separating spaces or Carriage Returns, and apply the User Exit "UA0BA":

```
Funcs$ = STR("FF", 40)
Restored.Funcs = ICONV(Funcs$, "UA0BA")
CRT "Function keys restored to null"
```

To change the function keys to new values, place the new values in the string instead.

Rather than typing long strings, it will be easier to use the following programs to build a file of key definitions, to change that file, and then to change the keys by applying the definitions. Before presenting the programs, however, the use of lead and follow keys for foreign language characters will be covered.

Foreign Language Characters on the IBM PC/XT Keyboard

The IBM PC extended character set includes foreign language characters in the range of ASCII 128 through 168 as well as Greek characters commonly used as mathematical symbols in the range ASCII 224 through 240. These characters can be entered on any IBM-compatible machine, and on the Wyse 60 terminal, by holding down the ALT key, typing the ASCII number on the numeric keypad, and then releasing the ALT key. That procedure, however, would become tedious if the characters were used often.

Pick facilitates the use of foreign language characters on the IBM PC/XT keyboard by providing lead keys that modify the action of the keystroke that follows. Four lead keys can be defined. They must be designated by resetting keys on the keyboard to ASCII values 251 through 254.

The lead keys provide a convenient method of using variations on the basic Roman characters. For example, the character "a" is sometimes rendered in various European languages with an accent above the key. The "á" acute accent slopes down from right to left, the "à" grave accent slopes down from left to right, the "â" circumflex resembles a hat over the key, and the "ä" umlaut consists of two dots.

You can define lead keys that resemble the accent marks, such as CTRL-´ for the acute accent, CTRL-` for the grave accent, CTRL-> for circumflex and CTRL-: for umlaut. When the user types the lead key CTRL-´ followed by the follow key "a" he will get "á" with an acute accent, with the lead key CTRL-` he will get "à" with a grave accent, with CTRL-> he will get "â" with a circumflex, and with CTRL-: he will get "ä" with an umlaut.

The control characters have been set to ASCII 251 through 254 in the examples shown for resetting the keyboard. Other miscellaneous foreign characters have also been included (Note that CTRL-> is actually CTRL-. and that CTRL-: is actually CTRL-;. The ASCII number does not vary if SHIFT is added to CTRL.)

The key definitions for foreign characters are added to the string shown above for redefining the keyboard with a user exit. Following the keyboard definition, 12 hexadecimal numbers will appear for the follow keys that are pressed after entering a lead key.

Then from one to four sets of 12 hexadecimal numbers will appear for each for the characters that are to be returned by the combination of lead and follow keys. Each set, both for follow keys and for characters returned, must contain exactly 12 hexadecimal numbers. If fewer than 12 are desired, the set must be padded with "00".

Examples of lead and follow keys with the characters that they return are shown in the table "Foreign Language Characters with Lead and Follow Keys." Each set of characters includes the ASCII number for the key, first in decimal and then in hexadecimal.

Foreign Language Characters with Lead and Follow Keys Table

Lead	CTRL-' 251 FB	CTRL-` 252 FC	CTRL-> 253 FD	CTRL-: 254 FE
Follow				
a 97 61	á : a acute 160 A0	à : a grave 133 85	â : a circumflex 131 83	ä : a umlaut 132 84
e 101 65	é : e acute 130 82	è : e grave 138 8A	ê : e circumflex 136 88	ë : e umlaut 137 89
i 105 69	í : i acute 161 A1	ì : i grave 141 8D	î : i circumflex 140 8C	ï : i umlaut 139 8B
o 111 6F	ó : o acute 162 A2	ò : o grave 149 95	ô : o circumflex 147 93	ö : o umlaut 148 94
u 117 75	ú : u acute 163 A3	ù : u grave 151 97	û : u circumflex 150 96	ü : u umlaut 129 81
A 65 41	- 0 00	- 0 00	- 0 00	Ä : A umlaut 142 8E
E 69 45	É : E acute 144 90	- 0 00	- 0 00	- - 00
O 79 4F	- 0 00	- 0 00	- 0 00	Ö : O umlaut 153 99
U 85 55	- 0 00	- 0 00	- 0 00	Ü : U umlaut 154 9A
c 99 63	ç : c cedilla 135 87	Ç : C cedilla 128 80	- - 00	- - 00

n	ñ : n tilde	Ñ : N tilde	-	-
110	164	165	-	-
6E	A4	A5	00	00
?	¿ : ? upside down	-	-	-
63	168	0	0	0
3F	A8	00	00	00

Even though a foreign character might appear on the screen, the printer used might not be capable of reproducing that character. To find out which foreign characters a printer will create, run the TEST.PRINT.FOREIGN.CHARS program.

Foreign language characters for Pick Release 2.2 and 3.0 are covered in the appendix.

TEST.PRINT.FOREIGN.CHARS Program

```
* TEST.PRINT.FOREIGN.CHARS   David L. Clark 05/06/1988
* Send foreign and Greek math symbols to the printer.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TForeign.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  PRINTER ON
  * Foreign characters
  FOR I = 128 TO 175
    PRINT I : " = " : CHAR(I)
  NEXT I
  * Greek math symbols
  FOR I = 224 TO 254
    PRINT I : " = " : CHAR(I)
  NEXT I
  PRINTER OFF
END
```

The series of programs presented here will reset the IBM PC/XT keyboard for Pick. In each program the parameters used are divided between those that would have to be altered for other PC models or Pick versions and those parameters that should remain the same. The programs to redefine the IBM PC/XT keyboard also designate the Pick release as "R83," while the redefining programs for the Wyse 60 AT do not specify the Pick release.

The BUILD.KEYS.FILE.IBM.PC.XT.R83 program will take the keyboard definitions out of the KEYBOARDS file (in the SYSPROG account under Pick release 1983 2.0), reformat them into single strings, and place them in the KEYBOARD-CHOICES.IBM.PC.XT.R83 file for use by the other programs that follow. The program will place the keyboard definitions in attribute 1, and the function key definition as null in attribute 2. The item will retain the original item ID it had in the KEYBOARDS file. If the item ID already exists in the KEYBOARD.CHOICES.IBM.PC.XT.R83 file, the program will ask if the item is to be over-written.

BUILD.KEYS.FILE.IBM.PC.XT.R83 Program

```
* BUILD.KEYS.FILE.IBM.PC.XT.R83   David L. Clark 08/24/1988
* Create the file KEYBOARD.CHOICES.IBM.PC.XT.R83 of alternative keyboards.
```

```

* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: BdkyFipC.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Reformat the Items in the KEYBOARDS file supplied by Pick Systems for the
* IBM PC-XT into single strings in the KEYBOARD.CHOICES.IBM.PC.XT.R83
* file used by the CHANGE.KEYS.IBM.PC.XT.R83 external subroutine.
*****
* Assign Constants, Parameters *
*****
*****
* Set Constants, Parameters Valid for All Models *
*****
PROMPT ""
EQU Attrib.Mark$ TO CHAR(254)
EQU False TO 0
File.Keys.Pick.Name$ = "KEYBOARDS"; * File supplied by Pick Systems
EQU True TO 1
*****
* Set Parameters by PC Model, Version *
*****
File.Keys.Name$ = "KEYBOARD.CHOICES.IBM.PC.XT.R83"
EQU Lead.Keys.End TO 234; * Include Lead - Follow keys, not function
* Set empty list for function keys
Keyboard$ = Attrib.Mark$ : STR("FF", 40); * Store 40 blank func keys
=====
*****
* Open Files *
*****
OPEN File.Keys.Pick.Name$ TO File.Keys.Pick$ ELSE
  CRT "Missing file: " : File.Keys.Pick.Name$
  GOTO 9999; *|End Program|
END
OPEN File.Keys.Name$ TO File.Keys$ ELSE
  CRT "Missing file: " : File.Keys.Name$
  GOTO 9999; *|End Program|
END
*****
* Get Pick Keyboards, Put in Choices File *
*****
EXECUTE \SSELECT \ : File.Keys.Pick.Name$ CAPTURING Capture.Output$
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
  READNEXT Keyboard.ID$ THEN
    * Check if same ID already exists in Choices file
    READV Dummy$ FROM File.Keys$, Keyboard.ID$, 1 THEN
      CRT
      CRT "Keyboard Item " : Keyboard.ID$ : " already exists.":
      CRT " Over-write old Item?"
      CRT "1. Yes 2. No Pick a Number [1]: " :
      INPUT Num.Choice
      IF Num.Choice # "2" THEN GOSUB 100; *|Store Keyboard|
    END ELSE
      GOSUB 100; *|Store Keyboard|
    END
  END ELSE
    Quit.Flag = True
  END
END

```

```

REPEAT
GOTO 9999;                                *|End Program|
*-----*
*****
100  * Store Keyboard *
*****
* Parameters: Keyboard.ID$, File.Keys.Pick$, File.Keys$
* Reformats and stores keyboard from File.Keys.Pick$ into File.Keys$
* Builds empty key lists for Lead - Follow keys and function keys.
READ Keyboard.Pick$ FROM File.Keys.Pick$, Keyboard.ID$ ELSE
    CRT "No keyboard in Item: " : Keyboard.ID$
RETURN
END
* Reformat Item into 1 string
Total.Attribs = DCOUNT(Keyboard.Pick$, Attrib.Mark$)
Keys$ = ""
FOR I = 5 TO Total.Attribs
    Attrib$ = TRIM(Keyboard.Pick$<I>)
    IF Attrib$[1, 1] # "*" THEN
        Keys$ = Keys$ : FIELD(Attrib$, " ", 1)
    END
NEXT I
* Add Lead - Follow keys, if not already present
Keys$ = Keys$ : STR("00", Lead.Keys.End - LEN(Keys$) / 2)
Keyboard$<1> = Keys$
* Function keys are in 2nd Attrib
* Store
WRITE Keyboard$ ON File.Keys$, Keyboard.ID$
RETURN
*=====*
*****
9999 * End Program *
*****
END

```

If you have Pick Release 2.2 or 3.0, use the BUILD.KEYS.FILE program listed in the appendix.

The CHANGE.KEYS.FILE.IBM.PC.XT.R83 program will alter the keyboard definitions held in the KEYBOARD.CHOICES.IBM.PC.XT.R83 file. The program first will display the definitions held in the file and ask which item is to be used as the source for the original key values. To use the standard USA keyboard as a starting point, choose the item "USA." The longest portion of the program is the that part that correlates the position of a key definition in the keyboard redefinition string with the alphabetical and numerical order that a user would find most natural.

The program will show one set of keys at a time: shifted, unshifted, control, lead-follow for foreign characters, and function keys. For each key the program will show the key pressed followed by the decimal ASCII number of the character returned. The user can change any of the ASCII numbers.

In the case of the lead-follow keys, the program shows the foreign character that results from the use of the lead-follow procedure. Use the following tables in this section as guides to changing the keyboard: the table "ASCII Value Returned by Each Key" at the beginning of the section, for the regular and function keys, and the table "Foreign Language Characters with Lead and Follow Keys," for the foreign language characters.

If a key is assigned as ASCII zero, the key will actually return that value when pressed. If a key is assigned as ASCII 255, pressing it will have no effect. Therefore to disable a key, assign it as ASCII 255.

After going through all of the key sets, or choosing to quit, the program will ask for a name under which to store the new keyboard. The original name will be shown as the default choice. You also can choose quit in order not to store the new definition. If the original USA keyboard is accidentally over-written, after leaving the program use the COPY command to give the new definition a different name, and then restore the original USA definition with the BUILD.KEYS.FILE.IBM.PC.XT.R83 program.

CHANGE.KEYS.FILE.IBM.PC.XT.R83 Program

```
* CHANGE.KEYS.FILE.IBM.PC.XT.R83  David L. Clark 08/24/1988
* Change keyboard in KEYBOARD.CHOICES.IBM.PC.XT.R83 file
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ChKyFiPC.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants, Parameters, Initial Values *
*****
* Set Constants, Parameters Valid for All PC Models *
*****
PROMPT ""
Clear.Screen$ = @(-1)
Erase.Screen.from.Pos$ = @(-3)
EQU False TO 0
EQU Keyboards.Max TO 20;      * Maximum number of different keyboards
DIM Keyboard.Name$(20)
Row.Bottom = SYSTEM(3)
DIM Sec.Begin(5), Sec.End(5)
DIM Title$(5)
Title$(1) = "SHIFTED KEYS"
Title$(2) = "UNSHIFTED KEYS"
Title$(3) = "CONTROL KEYS"
Title$(4) = "LEAD AND FOLLOW KEYS"
Title$(4) = Title$(4) : @(0, 3) : "ORIGINAL"
Title$(4) = Title$(4) : @(16) : "ASCII 251 +" : @(32) : "ASCII 252 +"
Title$(4) = Title$(4) : @(48) : "ASCII 253 +" : @(64) : "ASCII 254 +"
Title$(5) = "FUNCTION KEYS"
Title$(5) = Title$(5) : @(0, 3) : "UNSHIFTED" : @(20) : "SHIFTED"
Title$(5) = Title$(5) : @(40) : "CONTROL" : @(60) : "ALTERNATE"
EQU Total.Secs TO 5
EQU True TO 1
*****
* Set Parameters by PC Model, Version *
*****
* Change these for model other than IBM PC-XT R83
* File holding alternative keyboards
File.Keys.Name$ = "KEYBOARD.CHOICES.IBM.PC.XT.R83"
* Parameters for number of keys
* Key arrays include Lead - Follow keys, function keys
EQU Total.All.Keys TO 274; * Dimension for key arrays.
DIM Key$(274), Key.Ascii(274), Key.Pos(274), Key.Scrn$(274)
```

```

MAT Key$ = ""
MAT Key.Ascii = 0
* Total of Shifted, Unshifted, Control keys on main keyboard
EQU Main.Keys.End TO 174
* Total of each group of Shifted, Unshifted, Control keys
EQU Main.Keyboard TO 58
* Lead - Follow keys: Hold down ASCII 251 to 254 and press original
* key, get alternate ASCII character from 4 char sets.
EQU Lead.Keys.Begin TO 175; EQU Lead.Keys.End TO 234
* Function keys
EQU Func.Group.Size TO 10
EQU Total.Func.Keys TO 40
EQU Func.Keys.Begin TO 235; EQU Func.Keys.End TO 274
* Sections: Shifted, Unshifted, Control, Lead - Follow, Function
Sec.Begin(1) = 1; Sec.End(1) = Main.Keyboard
Sec.Begin(2) = Main.Keyboard + 1; Sec.End(2) = Main.Keyboard * 2
Sec.Begin(3) = Main.Keyboard * 2 + 1; Sec.End(3) = Main.Keys.End
Sec.Begin(4) = Lead.Keys.Begin; Sec.End(4) = Lead.Keys.End
Sec.Begin(5) = Func.Keys.Begin; Sec.End(5) = Func.Keys.End
* Set key names, position in string from KEYBOARDS, screen position
* Shifted keys are first in the string
Key$(1) = "A"; Key.Pos(1) = 59; Key.Scrn$(1) = @(0, 2)
Key$(2) = "B"; Key.Pos(2) = 95; Key.Scrn$(2) = @(0, 3)
Key$(3) = "C"; Key.Pos(3) = 91; Key.Scrn$(3) = @(0, 4)
Key$(4) = "D"; Key.Pos(4) = 63; Key.Scrn$(4) = @(0, 5)
Key$(5) = "E"; Key.Pos(5) = 35; Key.Scrn$(5) = @(0, 6)
Key$(6) = "F"; Key.Pos(6) = 65; Key.Scrn$(6) = @(0, 7)
Key$(7) = "G"; Key.Pos(7) = 67; Key.Scrn$(7) = @(0, 8)
Key$(8) = "H"; Key.Pos(8) = 69; Key.Scrn$(8) = @(0, 9)
Key$(9) = "I"; Key.Pos(9) = 45; Key.Scrn$(9) = @(0, 10)
Key$(10) = "J"; Key.Pos(10) = 71; Key.Scrn$(10) = @(0, 11)
Key$(11) = "K"; Key.Pos(11) = 73; Key.Scrn$(11) = @(0, 12)
Key$(12) = "L"; Key.Pos(12) = 75; Key.Scrn$(12) = @(0, 13)
Key$(13) = "M"; Key.Pos(13) = 99; Key.Scrn$(13) = @(0, 14)
Key$(14) = "N"; Key.Pos(14) = 97; Key.Scrn$(14) = @(0, 15)
Key$(15) = "O"; Key.Pos(15) = 47; Key.Scrn$(15) = @(0, 16)
Key$(16) = "P"; Key.Pos(16) = 49; Key.Scrn$(16) = @(0, 17)
Key$(17) = "Q"; Key.Pos(17) = 31; Key.Scrn$(17) = @(0, 18)
Key$(18) = "R"; Key.Pos(18) = 37; Key.Scrn$(18) = @(0, 19)
Key$(19) = "S"; Key.Pos(19) = 61; Key.Scrn$(19) = @(0, 20)
Key$(20) = "T"; Key.Pos(20) = 39; Key.Scrn$(20) = @(0, 21)
Key$(21) = "U"; Key.Pos(21) = 43; Key.Scrn$(21) = @(26, 2)
Key$(22) = "V"; Key.Pos(22) = 93; Key.Scrn$(22) = @(26, 3)
Key$(23) = "W"; Key.Pos(23) = 33; Key.Scrn$(23) = @(26, 4)
Key$(24) = "X"; Key.Pos(24) = 89; Key.Scrn$(24) = @(26, 5)
Key$(25) = "Y"; Key.Pos(25) = 41; Key.Scrn$(25) = @(26, 6)
Key$(26) = "Z"; Key.Pos(26) = 87; Key.Scrn$(26) = @(26, 7)
Key$(27) = "1"; Key.Pos(27) = 3; Key.Scrn$(27) = @(26, 8)
Key$(28) = "2"; Key.Pos(28) = 5; Key.Scrn$(28) = @(26, 9)
Key$(29) = "3"; Key.Pos(29) = 7; Key.Scrn$(29) = @(26, 10)
Key$(30) = "4"; Key.Pos(30) = 9; Key.Scrn$(30) = @(26, 11)
Key$(31) = "5"; Key.Pos(31) = 11; Key.Scrn$(31) = @(26, 12)
Key$(32) = "6"; Key.Pos(32) = 13; Key.Scrn$(32) = @(26, 13)
Key$(33) = "7"; Key.Pos(33) = 15; Key.Scrn$(33) = @(26, 14)
Key$(34) = "8"; Key.Pos(34) = 17; Key.Scrn$(34) = @(26, 15)
Key$(35) = "9"; Key.Pos(35) = 19; Key.Scrn$(35) = @(26, 16)
Key$(36) = "0"; Key.Pos(36) = 21; Key.Scrn$(36) = @(26, 17)
Key$(37) = "-"; Key.Pos(37) = 23; Key.Scrn$(37) = @(26, 18)
Key$(38) = "="; Key.Pos(38) = 25; Key.Scrn$(38) = @(26, 19)

```

```

Key$(39) = "BKSP"; Key.Pos(39) = 27; Key.Scrn$(39) = @(26, 20)
Key$(40) = "ENTER"; Key.Pos(40) = 55; Key.Scrn$(40) = @(26, 21)
Key$(41) = "ESC"; Key.Pos(41) = 1; Key.Scrn$(41) = @(52, 2)
Key$(42) = "TAB"; Key.Pos(42) = 29; Key.Scrn$(42) = @(52, 3)
Key$(43) = "["; Key.Pos(43) = 51; Key.Scrn$(43) = @(52, 4)
Key$(44) = "]"; Key.Pos(44) = 53; Key.Scrn$(44) = @(52, 5)
Key$(45) = ";"; Key.Pos(45) = 77; Key.Scrn$(45) = @(52, 6)
Key$(46) = "\'"; Key.Pos(46) = 79; Key.Scrn$(46) = @(52, 7)
Key$(47) = "`"; Key.Pos(47) = 81; Key.Scrn$(47) = @(52, 8)
Key$(48) = "\"; Key.Pos(48) = 85; Key.Scrn$(48) = @(52, 9)
Key$(49) = ","; Key.Pos(49) = 101; Key.Scrn$(49) = @(52, 10)
Key$(50) = "."; Key.Pos(50) = 103; Key.Scrn$(50) = @(52, 11)
Key$(51) = "/"; Key.Pos(51) = 105; Key.Scrn$(51) = @(52, 12)
Key$(52) = "ALT"; Key.Pos(52) = 111; Key.Scrn$(52) = @(52, 13)
Key$(53) = "CAPS"; Key.Pos(53) = 115; Key.Scrn$(53) = @(52, 14)
Key$(54) = "CTRL"; Key.Pos(54) = 57; Key.Scrn$(54) = @(52, 15)
Key$(55) = "PRTSC"; Key.Pos(55) = 109; Key.Scrn$(55) = @(52, 16)
Key$(56) = "SHIFT"; Key.Pos(56) = 83; Key.Scrn$(56) = @(52, 17)
Key$(57) = "SHIFT"; Key.Pos(57) = 107; Key.Scrn$(57) = @(52, 18)
Key$(58) = "SPACE"; Key.Pos(58) = 113; Key.Scrn$(58) = @(52, 19)
* Unshifted and Control keys
FOR I = Main.Keyboard + 1 TO Main.Keys.End
    Key$(I) = Key$(I - Main.Keyboard)
    Key.Pos(I) = Key.Pos(I - Main.Keyboard) + Main.Keyboard * 2
    Key.Scrn$(I) = Key.Scrn$(I - Main.Keyboard)
NEXT I
* Lead - Follow keys
FOR I = Lead.Keys.Begin TO Lead.Keys.End
    Key.Pos(I) = 1 + (I - 1) * 2
NEXT I
FOR I = 0 TO 4
    Start.Set = (Lead.Keys.Begin - 1) + (12 * I); * 5 sets of 12 keys
    Col = 16 * I; * Column for display
    FOR J = 1 TO 12
        Key.Scrn$(Start.Set + J) = @(Col, 5 + J)
    NEXT J
NEXT I
* Function keys. Key position within Funcs$.
FOR I = 0 TO 3
    Func.Group = Func.Group.Size * I
    Start.Set = (Func.Keys.Begin - 1) + Func.Group; * 4 grps 10 keys
    Col = 20 * I; * Column for display
    FOR J = 1 TO 10
        Key.Num = Start.Set + J
        Key.Scrn$(Key.Num) = @(Col, 5 + J)
        Key$(Key.Num) = "F" : J
        Key.Pos(Key.Num) = 1 + (Func.Group + J - 1) * 2
    NEXT J
NEXT I
*=====
*****
* Get Default Choices for Keys *
*****
OPEN File.Keys.Name$ TO File.Keys$ ELSE
    CRT "Missing file: " : File.Keys.Name$
    GOTO 9999; *|End Program|
END
EXECUTE \SSELECT \ : File.Keys.Name$ CAPTURING Capture.Output$
CRT Clear.Screen$ : @(20) : "CHOOSE KEYBOARD FOR BEGINNING VALUES" :

```

```

Quit.Flag = False
FOR I = 1 TO Keyboards.Max UNTIL Quit.Flag
    READNEXT Keyboard.Name$(I) THEN
        CRT @(30, I + 1) : I : ". " : Keyboard.Name$(I) :
    END ELSE
        Quit.Flag = True
    END
NEXT I
CRT @(20, Row.Bottom) : "Pick a Number or (Q)uit [1]: " :
INPUT Response$ :
IF NOT(NUM(Response$)) THEN GOTO 9999; *|End Program|
IF Response$ = "" THEN Num.Choice = 1 ELSE Num.Choice = Response$ + 0
Keyboard.ID$ = Keyboard.Name$(Num.Choice)
READ Keyboard$ FROM File.Keys$, Keyboard.ID$ ELSE
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "No keyboard found in Item: " : Keyboard.ID$ :
    GOTO 9999; *|End Program|
END
CRT @(0, Row.Bottom - 1) : Erase.Screen.from.Pos$ :
CRT "Reading keyboard " : Keyboard.ID$ : " in file ":File.Keys.Name$
CRT "Please wait ...":
Keys$ = Keyboard$<1>
Funcs$ = Keyboard$<2>
* Load arrays from strings by position
FOR I = 1 TO Lead.Keys.End
    Key.Ascii(I) = OCONV(Keys$[Key.Pos(I), 2], "MCXD")
NEXT I
* Lead - Follow keys. Assume not Control characters.
FOR I = Lead.Keys.Begin TO Lead.Keys.End
    Key$(I) = CHAR(Key.Ascii(I))
NEXT I
* Function keys
FOR I = Func.Keys.Begin TO Func.Keys.End
    Key.Ascii(I) = OCONV(Funcs$[Key.Pos(I), 2], "MCXD")
NEXT I
*****
* Set New Key Values *
*****
Quit.Flag = False
New.Sec = 1
LOOP UNTIL Quit.Flag DO
    Sec = New.Sec
    CRT Clear.Screen$ : @(30) : Title$(Sec) :
    FOR J = Sec.Begin(Sec) TO Sec.End(Sec)
        CRT Key.Scrn$(J) : J : ". " : Key$(J) : " = " : Key.Ascii(J):
    NEXT J
    LOOP; * While Section remains the same
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(15) :
    CRT "Pick a Number, (N)ext, (P)revious, or (Q)uit [N]: " :
    INPUT Response$ :
    BEGIN CASE
        CASE Response$ = "" OR Response$ = "N"; * Next, Default
            New.Sec = Sec + 1
            Quit.Flag = (New.Sec > Total.Secs)
        CASE Response$ = "P" AND Sec > 1; * Previous
            New.Sec = Sec - 1
        CASE Response$ = "Q"; * Quit
            Quit.Flag = True
    END CASE

```

```

WHILE Sec = New.Sec AND NOT(Quit.Flag) DO
  IF NUM(Response$) THEN
    N = Response$ + 0
    IF N >= Sec.Begin(Sec) AND N <= Sec.End(Sec) THEN
      CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(15):
      CRT Key$(N) : " = " : Key.Ascii(N) :
      CRT "      Enter new ASCII number: " :
      INPUT Response$ :
      * Default choice to keep same ASCII number
      IF Response$ = "" THEN Response$ = Key.Ascii(N)
      IF NUM(Response$) THEN
        Ascii = Response$ + 0
        IF Ascii > -1 AND Ascii < 256 THEN
          Key.Ascii(N) = Ascii
          * Lead - Follow keys
          IF N>Main.Keys.End AND N<Func.Keys.Begin THEN
            Key$(N) = CHAR(Ascii)
          END
          * Spaces on end in case old number was longer
          CRT Key.Scrn$(N) : N : ". " :
          CRT Key$(N) : " = " : Ascii : "   " :
          END; * If valid range for ASCII
        END; * If a number entered
      END; * If within range of section
    END; * If a number entered
  REPEAT; * While Section remains the same
REPEAT; * Until Quit.Flag
*****
* Store New Keyboard *
*****
CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(5) :
CRT "Name to store keyboard or (Q)uit [" : Keyboard.ID$ : "]: " :
INPUT Response$ :
IF Response$ = "Q" OR Response$ = "q" THEN GOTO 9999; *|End Program|
IF Response$ # "" THEN Keyboard.ID$ = Response$
CRT @(0, Row.Bottom - 1) : Erase.Screen.from.Pos$ :
CRT "Storing keyboard " : Keyboard.ID$ : " in file " : File.Keys.Name$
CRT "Please wait ..." :
* Load strings from arrays by position
FOR I = 1 TO Lead.Keys.End
  Pos = Key.Pos(I)
  Ascii.Hex = OCONV(Key.Ascii(I), "MCDX") "R(%2)"; * Make 2 digits
  Keys$ = Keys$[1, Pos - 1] : Ascii.Hex : Keys$[Pos + 2, 999]
NEXT I
FOR I = Func.Keys.Begin TO Func.Keys.End
  Pos = Key.Pos(I)
  Ascii.Hex = OCONV(Key.Ascii(I), "MCDX") "R(%2)"; * Make 2 digits
  Funcs$ = Funcs$[1, Pos - 1] : Ascii.Hex : Funcs$[Pos + 2, 999]
NEXT I
Keyboard$<1> = Keys$
Keyboard$<2> = Funcs$
WRITE Keyboard$ ON File.Keys$, Keyboard.ID$
=====
*****
9999 * End Program *
*****
END

```

If you have Pick Release 2.2 or 3.0, use the CHANGE.KEYS.FILE program in the appendix.

After the alternate keyboard definitions have been created and stored in the KEYBOARD.CHOICES.IBM.PC.XT.R83 file, the CHANGE.KEYS.IBM.PC.XT.R83 external subroutine will apply the redefinitions to change the current keyboard. Although the alternate definitions can be created from a terminal, the actual change can be accomplished only from an IBM PC/XT on port zero.

The CHANGE.KEYS.IBM.PC.XT.R83.SUB external subroutine receives the parameter Keybd.to.Apply\$ from the calling program.

The CHANGE.KEYS program is used to call the external subroutine in the next example. The CHANGE.KEYS program will first prompt the user for the keyboard ID of the terminal keyboard that is to be redefined, so that the user will not have to remember "IBM.PC.XT.R83" or "WYSE.60.AT.R83." CHANGE.KEYS will then use the keyboard ID to find the appropriate KEYBOARD.CHOICES.—file and CHANGE.KEYS.—external subroutine. The filename and the program name each end with the keyboard ID.

The CHANGE.KEYS program will show the keyboard definitions available in the KEYBOARD.CHOICES.—file and ask which is to be applied. CHANGE.KEYS will then call the appropriate CHANGE.KEYS.—subroutine, passing to it the parameter Keybd.to.Apply\$. The program also will pass the parameter Error.Flag, which will return true if no redefinition took place.

A later example in this section, of the external subroutine CHANGE.KEYS.WYSE.60.AT.SUB, shows how the same method can be used to change the keyboard definition on the Wyse 60 AT terminal.

The APPLY.TERMINAL.AND.KEY.IDS program later in this section shows the manner in which a data entry program that needed to change the keyboard for editing would call the CHANGE.KEYS.—subroutine without prompting the user for information. When the data entry program ended, it might call the subroutine again, passing to it the name of the old keyboard to restore.

CHANGE.KEYS Program

```
* CHANGE.KEYS David L. Clark 08/24/1988
* Call subroutine to redefine keyboard.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ChngKeys.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* For each type of keyboard which may be redefined, there will be a
*   separate File.Keys$ file holding the redefinitions available, and a
*   separate external Keybd.Subroutine$ to apply the redefinition.
*   File.Keys.Name$ = "KEYBOARD.CHOICES." + Keybd.ID$
*   Keybd.Subroutine$ = "CHANGE.KEYS." + Keybd.ID$
* Error checking, such as verifying Port zero for the IBM PC-XT, is left
*   to the called subroutine.
*   *****
*   * Assign Constants, Parameters *
*   *****
```

```

PROMPT ""
Clear.Screen$ = @(-1)
Erase.Screen.from.Pos$ = @(-3)
EQU False TO 0
File.Keys.Name.Begin$ = "KEYBOARD.CHOICES.";      * Will add Keybd.ID$
DIM Keybd.ID.Name$(20);      * Such as: IBM.PC.XT.R83
EQU Keybd.IDs.Max TO 20;      * Maximum different terminal keyboards
DIM Keybd.Def.Name$(20);      * Keyboard definitions
Keybd.Subroutine.Begin$ = "CHANGE.KEYS.";      * Will add Keybd.ID$
EQU Keybd.Defs.Max TO 20;      * Maximum keyboard definitions
Row.Bottom = SYSTEM(3)
EQU True TO 1
*=====
*****
* Name Terminal Keyboard *
*****
Command$ = \SSELECT MD = "\ : File.Keys.Name.Begin$ : \]" \
EXECUTE Command$ CAPTURING Capture.Output$
Test.Flag = False
Choices.Total = 0
Pos.Start = LEN(File.Keys.Name.Begin$) + 1
FOR I = 1 TO Keybd.IDs.Max UNTIL Test.Flag
  READNEXT Keybd.ID.Name$(I) THEN
    Choices.Total = Choices.Total + 1
    Keybd.ID.Name$(I) = Keybd.ID.Name$(I)[Pos.Start, 999]
  END ELSE
    Test.Flag = True
  END
NEXT I
IF Choices.Total THEN
  CRT Clear.Screen$ : @(25) : "NAME TERMINAL AND KEYBOARD" :
  FOR I = 1 TO Choices.Total
    CRT @(30, I + 1) : I : ". " : Keybd.ID.Name$(I) :
    NEXT I
    CRT @(25, Row.Bottom) : "Pick a Number or (Q)uit [1]: " :
    INPUT Response$ :
    IF Response$ = "" THEN Response$ = "1";      * Default
    IF NUM(Response$) THEN
      Num.Choice = Response$ + 0
      Keybd.ID$ = Keybd.ID.Name$(Num.Choice)
    END ELSE; * User chose Quit
    GOTO 9999;      *|End Program|
  END
END ELSE; * No choices available
  CRT "No Keyboard Choices files available"
  GOTO 9999;      *|End Program|
END
*****
* Choose Keyboard Definition to Apply *
*****
File.Keys.Name$ = File.Keys.Name.Begin$ : Keybd.ID$
EXECUTE \SSELECT \ : File.Keys.Name$ CAPTURING Capture.Output$
Test.Flag = False
Choices.Total = 0
FOR I = 1 TO Keybd.Defs.Max UNTIL Test.Flag
  READNEXT Keybd.Def.Name$(I) THEN
    Choices.Total = Choices.Total + 1
  END ELSE

```

```

        Test.Flag = True
    END
NEXT I
IF Choices.Total THEN
    CRT Clear.Screen$ : @(20) : "CHOOSE KEYBOARD DEFINITION TO APPLY":
    FOR I = 1 TO Choices.Total
        CRT @(30, I + 1) : I : ". " : Keybd.Def.Name$(I) :
    NEXT I
    CRT @(25, Row.Bottom) : "Pick a Number or (Q)uit [1]: " :
    INPUT Response$ :
    IF Response$ = "" THEN Response$ = "1"; * Default
    IF NUM(Response$) THEN
        Num.Choice = Response$ + 0
        Keybd.to.Apply$ = Keybd.Def.Name$(Num.Choice)
    END; * If user does not Quit
END; * If choices available
*****
* Call External Subroutine to Define New Keyboard *
*****
* Keybd.to.Apply$ in File.Keys$ is passed to the external subroutine.
Keybd.Subroutine$ = Keybd.Subroutine.Begin$ : Keybd.ID$ : ".SUB"
Error.Flag = False; * Will return True if keyboard not applied
CALL @Keybd.Subroutine$(Keybd.to.Apply$, Error.Flag)
*****
* Return from External Subroutine *
*****
CRT
IF NOT(Error.Flag) THEN
    CRT "Keyboard definition: " : Keybd.to.Apply$ : " was applied"
END ELSE
    CRT "Keyboard was not redefined"
END
=====
*****
9999 * End Program *
*****
END

```

CHANGE.KEYS.IBM.PC.XT.R83.SUB External Subroutine

```

SUBROUTINE CHANGE.KEYS.IBM.PC.XT.R83.SUB(Keybd.to.Apply$, Error.Flag)
* Apply keys from KEYBOARD.CHOICES.IBM.PC.XT.R83. David L. Clark 09/09/1989
* Pick Basic external subroutine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ChKeysPC.bps
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Read the Keybd.to.Apply$ Item from KEYBOARD.CHOICES.IBM.PC.XT.R83 file
* and apply with User Exit to change current keyboard.
* If no keyboard is applied, the Error.Flag returned to the caller as True.
* Alternative: Create Error.Num() array to show type of error.
*****
* Assign Constants, Parameters *
*****
PROMPT ""
* Error.Flag is received as False
EQU False TO 0
Port.Num$ = FIELD(OCONV(0, "U50BB"), " ", 1)

```



```

Port.Num = Port.Num$ + 0;          * Convert to number
EQU True TO 1
*****
* Set Parameters by PC Model and Version *
*****
* May have to change these for model other than IBM PC-XT R83 2.0
File.Keys.Name$ = "KEYBOARD.CHOICES.IBM.PC.XT.R83"
Keys.User.Exit$ = "U80BA"
Funcs.User.Exit$ = "UA0BA"
=====
*****
* Open Files *
*****
OPEN File.Keys.Name$ TO File.Keys$ ELSE
  CRT "Missing file: " : File.Keys.Name$
  Error.Flag = True
  GOTO 9999;          *|End Subroutine|
END
*****
* Check Passed Parameter *
*****
IF Keybd.to.Apply$ = "" THEN
  CRT "Keyboard definition to apply not designated"
  Error.Flag = True
  GOTO 9999;          *|End Subroutine|
END
*****
* Check for Port Zero *
*****
IF Port.Num # 0 THEN
  CRT "This program must be run from an IBM PC-XT on Port zero"
  Error.Flag = True
  GOTO 9999;          *|End Subroutine|
END
*****
* Read Keyboard *
*****
READ Keybd$ FROM File.Keys$, Keybd.to.Apply$ ELSE
  CRT "No keyboard found in Item: " : Keybd.to.Apply$ :
  Error.Flag = True
  GOTO 9999;          *|End Subroutine|
END
*****
* Apply Keyboard *
*****
Keys$      = Keybd$<1>
Funcs$     = Keybd$<2>
New.Keys   = ICONV(Keys$, Keys.User.Exit$)
New.Funcs  = ICONV(Funcs$, Funcs.User.Exit$)
=====
*****
9999 * End Subroutine *
*****
RETURN
END

```

→

If you have Pick Release 2.2 or 3.0, use the CHANGE.KEYS.IBM.PC subroutine in the appendix.

Changing the Keyboard on the Wyse 60 Terminal

The BUILD.KEYS.FILE.WYSE.60.AT program will create the items that will later be utilized to change the values returned by the keys on the Wyse 60 terminal. The Wyse 60 terminal can be purchased with four alternative keyboards. The AT keyboard should be employed for maximum compatibility with the IBM PC/XT and PC/AT.

In the file KEYS.AND.SYMBOLS.WYSE.60.AT the program will create an item with an item ID beginning with "KEYS", a period, and the name of the terminal and keyboard: "WYSE.60.AT". The program also will create an item with an item ID beginning with "SYMBOLS", a period, and the name of the terminal and keyboard. Each attribute in the KEYS item will contain the name that appears on the keyboard for a particular key and each matching attribute in the SYMBOLS item will contain the symbol that Wyse uses to designate that key during the redefinition process.

In the file KEYBOARD.CHOICES.WYSE.60.AT the program will create an item with the item ID "STANDARD", then a period, and then the terminal and keyboard designation. The item will contain the original default values that the terminal returns for each key. For each keystroke the program will create one attribute, matching in order the attributes in the KEYS and the SYMBOLS items. Multiple characters returned by a single keystroke will be separated by value marks (ASCII 253).

BUILD.KEYS.FILE.WYSE.60.AT Program

```
* BUILD.KEYS.FILE.WYSE.60.AT David L. Clark 08/25/1988
* Create a file for alternative keyboards and a file for Wyse's key symbols
* Pick Basic program.
* Pick IBM PC-XT R83 2.0, Wyse 60 AT terminal
* Filename under IBM PC DOS: BdKyFiWy.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* In the KEYS.AND.SYMBOLS.WYSE.60.AT file the program will create a
* KEYS.WYSE.60.AT Item in which each Attribute contains the name of a key
* and a SYMBOLS.WYSE.60.AT Item in which each Attribute contains Wyse's
* symbol for resetting the key.
* In the KEYBOARD.CHOICES.WYSE.60.AT file the program will create a
* STANDARD.WYSE.60.AT Item in which each Attribute contains the
* characters returned by the key before resetting. (A similar Standard
* Item is created for each type of keyboard that may be redefined.
* The name of the keyboard is always divided from "STANDARD" by an
* asterisk.)
* The Attributes of the Keys, Symbols, and Standard characters Items will
* match by order.
*****
* Assign Constants, Parameters *
*****
*****
* Set Constants, Parameters Valid for All Keyboards *
*****
EQU CR$ TO CHAR(13)
EQU Escape$ TO CHAR(27)
*****
* Set Parameters for AT Keyboard *
*****
* Change these for keyboard other than AT
DIM C$(56); * Default chars returned by keystroke
```

```

MAT C$ = ""
Chars.Item.ID$           = "STANDARD*WYSE.60.AT"
File.Keys.Name$          = "KEYBOARD.CHOICES.WYSE.60.AT"
File.Keys.Symbols.Name$  = "KEYS.AND.SYMBOLS.WYSE.60.AT"
DIM K$(56);              * Name of key
MAT K$ = ""
Keys.Item.ID$            = "KEYS.WYSE.60.AT"
DIM S$(56);              * Symbol used to identify key for redefinition
MAT S$ = ""
Symbols.Item.ID$         = "SYMBOLS.WYSE.60.AT"
EQU Total.All.Keys TO 56;      * Dimension for key arrays
=====
*****
* Open Files *
*****
OPEN File.Keys.Name$ TO File.Keys$ ELSE
    CRT "Missing file: " : File.Keys.Name$
    GOTO 9999;                *|End Program|
END
OPEN File.Keys.Symbols.Name$ TO File.Keys.Symbols$ ELSE
    CRT "Missing file: " : File.Keys.Symbols.Name$
    GOTO 9999;                *|End Program|
END
*****
* Assign Key, Symbol, Chars Arrays *
*****
* Returns:
*   K$() = Key      : Name for each key
*   S$() = Symbols  : Symbol used to reset each key
*   C$() = Chars    : Characters returned by keystroke
* The keys are assigned in the order in which they appear in the
* Wyse 60 User's Guide and the Wyse 60 Programmer's Guide, to make
* it easier to change the listing for other keyboards with the
* Wyse 60 or for other Wyse terminals.
K$(1) = "ESC";          S$(1) = " "; C$(1) = Escape$
K$(2) = "SH-ESC";       S$(2) = "&"; C$(2) = Escape$
K$(3) = "TAB";          S$(3) = "!"; C$(3) = CHAR(9)
K$(4) = "SH-TAB";       S$(4) = "&"; C$(4) = Escape$ : CHAR(73)
K$(5) = "BKSP";         S$(5) = "\"; C$(5) = CHAR(8)
K$(6) = "SH-BKSP";      S$(6) = "\'"; C$(6) = CHAR(8)
K$(7) = "DEL";          S$(7) = "#"; C$(7) = CHAR(127)
K$(8) = "SH-DEL";       S$(8) = "("; C$(8) = CHAR(127)
K$(9) = "HOME";         S$(9) = "*"; C$(9) = CHAR(30)
K$(10) = "SH-HOME";     S$(10) = "/"; C$(10) = Escape$ : CHAR(123)
K$(11) = "UP";          S$(11) = "+"; C$(11) = CHAR(1)
K$(12) = "SH-UP";       S$(12) = "0"; C$(12) = CHAR(11)
K$(13) = "DOWN";        S$(13) = ","; C$(13) = CHAR(2)
K$(14) = "SH-DOWN";     S$(14) = "1"; C$(14) = CHAR(10)
K$(15) = "LEFT";        S$(15) = "-"; C$(15) = CHAR(6)
K$(16) = "SH-LEFT";     S$(16) = "2"; C$(16) = CHAR(8)
K$(17) = "RIGHT";       S$(17) = "."; C$(17) = CHAR(4)
K$(18) = "SH-RIGHT";    S$(18) = "3"; C$(18) = CHAR(12)
K$(19) = "ENTER";       S$(19) = "s"; C$(19) = CHAR(13)
* SHIFT-ENTER on the Wyse 60 AT toggles key-click unless changed
K$(20) = "SH-ENTER";    S$(20) = "4"; C$(20) = ""
K$(21) = "INS";         S$(21) = "q"; C$(21) = Escape$ : CHAR(114)
K$(22) = "SH-INS";      S$(22) = "p"; C$(22) = Escape$ : CHAR(113)
K$(23) = "PG DN";       S$(23) = "r"; C$(23) = Escape$ : CHAR(75)

```

```

K$(24) = "SH-PG DN"; S$(24) = "w"; C$(24) = Escape$ : CHAR(75)
* In the Wyse manual shown as PRINT SCREEN rather than multiplication
K$(25) = "* Kypd"; S$(25) = "R"; C$(25) = CHAR(42)
K$(26) = "SH-PRT SC"; S$(26) = "X"; C$(26) = Escape$ : CHAR(80)
K$(27) = "END"; S$(27) = "\"; C$(27) = CHAR(49)
K$(28) = "SH-END"; S$(28) = "]"; C$(28) = CHAR(49)
K$(29) = "+ Kypd"; S$(29) = "^"; C$(29) = CHAR(43)
K$(30) = "SH-+ Kypd"; S$(30) = " "; C$(30) = CHAR(43)
K$(31) = "- Kypd"; S$(31) = "|"; C$(31) = CHAR(45)
K$(32) = "SH-- Kypd"; S$(32) = "y"; C$(32) = CHAR(45)
K$(33) = "PG UP"; S$(33) = ":"; C$(33) = Escape$ : CHAR(74)
K$(34) = "SH-PG UP"; S$(34) = ";"; C$(34) = Escape$ : CHAR(74)
K$(35) = "5 Kypd"; S$(35) = "<"; C$(35) = CHAR(53)
K$(36) = "SH-5 Kypd"; S$(36) = "="; C$(36) = CHAR(53)
* Function keys
FOR I = 1 TO 10
    N = I + 36; * Number in array
    C = 63 + I; * Both char and symbol
    K$(N) = "F" : I; S$(N) = CHAR(C); C$(N) = CHAR(1) : CHAR(C) : CR$
NEXT I
FOR I = 1 TO 10
    N = I + 46; * Number in array
    C = 95 + I; * Both char and symbol
    K$(N) = "SH-F":I; S$(N) = CHAR(C); C$(N) = CHAR(1) : CHAR(C) : CR$
NEXT I
*****
* Store Items with Keys, Symbols, Chars *
*****
MATWRITE K$ ON File.Keys.Symbols$, Keys.Item.ID$
MATWRITE S$ ON File.Keys.Symbols$, Symbols.Item.ID$
MATWRITE C$ ON File.Keys$, Chars.Item.ID$

*=====*
*****
9999 * End Program *
*****
END

```

The CHANGE.KEYS.FILE.WYSE.60.AT program will alter the keyboard definitions held in the KEYBOARD.CHOICES.WYSE.60.AT file. The program will first display the definitions held in the KEYBOARD.CHOICES.WYSE.60.AT file and ask which item is to be used as the source for the original key values. To use the standard default keyboard as a starting point, choose the item "STANDARD.WYSE.60.AT."

The program will display all of the editing and function keys on one screen. For each key the program will show the key pressed followed by the decimal ASCII number of the character returned. If pressing a key returns nothing, than no ASCII number will appear. If several characters are returned, the ASCII number for each will be shown, and the numbers will be connected by the plus sign "+." The user can change any of the ASCII numbers. Use the table "ASCII Value Returned by Each Key" at the beginning of this section as a guide to changing the keyboard.

The Wyse 60 will not return characters greater than ASCII 127, except in the case of the control-function keys, with CTRL-F1 beginning at ASCII 128. The editing routine will obtain ASCII values greater than 127 by taking in a two-character set and adding 128 to the second character, as is done in the SHOW.ASCII program at the end of this section.

If a key is assigned as ASCII zero, or as ASCII 255, or as null, the key will return its original Wyse 60 value.

When the user chooses to quit, the program will ask for a name under which to store the new keyboard. The original name will be shown as the default choice. You also can choose quit in order not to store the new definition. If the original STANDARD.WYSE.60.AT keyboard is accidentally over-written, after leaving the program use the COPY command to give the new definition a different name, and then restore the original standard definition with the BUILD.KEYS.FILE.WYSE.60.AT program.

CHANGE.KEYS.FILE.WYSE.60.AT Program

```
* CHANGE.KEYS.FILE.WYSE.60.AT David L. Clark 07/09/1988
* Change keyboard in KEYBOARD.CHOICES.WYSE.60.AT file.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ChKyFiWy.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* In the KEYS.AND.SYMBOLS.WYSE.60.AT file the program will read a
* KEYS.WYSE.60.AT Item in which each Attribute has the name of a key.
* In the KEYBOARD.CHOICES.WYSE.60.AT file the program will read an
* Item in which each Attribute contains the characters returned
* by the key before resetting.
* The Attributes of the Keys and Characters Items match by order.
*****
* Assign Constants, Parameters *
*****
*****
* Set Constants, Parameters Valid for All Keyboards *
*****
PROMPT ""
Clear.Screen$ = @(-1)
Erase.Screen.from.Pos$ = @(-3)
EQU False TO 0
EQU Keyboards.Max TO 20;          * Maximum number of different keyboards
DIM Keyboard.Name$(20)
Mask$ = "L(#9)";                * 9 spaces max for key name, ASCII returned
Row.Bottom = SYSTEM(3)
EQU True TO 1
*****
* Set Parameters for AT Keyboard *
*****
* Change these for keyboard other than AT
DIM C$(56);                      * Chars returned by keystroke
File.Keys.Name$ = "KEYBOARD.CHOICES.WYSE.60.AT"
File.Keys.Symbols.Name$ = "KEYS.AND.SYMBOLS.WYSE.60.AT"
DIM K$(56);                      * Name of key
DIM Key.Ascii$(56);              * Ascii numbers of chars returned
DIM Key.Scrn$(56);              * Screen position where displayed
Keys.Item.ID$ = "KEYS.WYSE.60.AT"
EQU Total.All.Keys TO 56;        * Dimension for key arrays.
*****
* Assign Screen Positions in 3 Cols *
*****
* Assume no more than will fit on a screen 3 columns across.
* The Wyse 60 allows 36 editing keys + 20 function keys to be reset
```

```

*   List begins on row 4, counting from title at row 0 at top.
FOR I = 1 TO 19
    Key.Scrn$(I) = @(0, I + 3)
NEXT I
FOR I = 20 TO 38
    Key.Scrn$(I) = @(26, I - 16)
NEXT I
FOR I = 39 TO 56
    Key.Scrn$(I) = @(52, I - 35)
NEXT I
=====
*****
* Get Default Choices for Keys *
*****
OPEN File.Keys.Name$ TO File.Keys$ ELSE
    CRT "Missing file: " : File.Keys.Name$
GOTO 9999;                *|End Program|
END
OPEN File.Keys.Symbols.Name$ TO File.Keys.Symbols$ ELSE
    CRT "Missing file: " : File.Keys.Name$
GOTO 9999;                *|End Program|
END
MATREAD K$ FROM File.Keys.Symbols$, Keys.Item.ID$ ELSE
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "No key names found in Item: " : Keys.Item.ID$ :
    GOTO 9999;                *|End Program|
END
EXECUTE \SSELECT \ : File.Keys.Name$ CAPTURING Capture.Output$
CRT Clear.Screen$ : @(20) : "CHOOSE KEYBOARD FOR BEGINNING VALUES" :
Quit.Flag = False
FOR I = 1 TO Keyboards.Max UNTIL Quit.Flag
    READNEXT Keyboard.Name$(I) THEN
        CRT @(20, I + 1) : I : ". " : Keyboard.Name$(I) :
    END ELSE
        Quit.Flag = True
    END
NEXT I
CRT @(20, Row.Bottom) : "Pick a Number or (Q)uit [1]: " :
INPUT Response$ :
IF NOT(NUM(Response$)) THEN GOTO 9999;    *|End Program|
IF Response$ = "" THEN Num.Choice = 1 ELSE Num.Choice = Response$ + 0
Chars.Item.ID$ = Keyboard.Name$(Num.Choice)
MATREAD C$ FROM File.Keys$, Chars.Item.ID$ ELSE
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "No keyboard found in Item: " : Chars.Item.ID$ :
    GOTO 9999;                *|End Program|
END
*****
* Convert Chars Returned by Key to ASCII Numbers *
*****
MAT Key.Ascii$ = ""
FOR I = 1 TO Total.All.Keys
    Length = LEN(C$(I))
    IF Length THEN
        Key.Ascii$(I) = SEQ(C$(I)[1, 1]); * 1st or only char
        *****
        * Check for More than 1 Char per Keystroke *
        *****
        FOR J = 2 TO Length

```

```

        Key.Ascii$(I) = Key.Ascii$(I) : "+" : SEQ(C$(I)[J, 1])
    NEXT J
END
NEXT I
*****
* Set New Key Values *
*****
CRT Clear.Screen$ : @(30) : "EDITING AND FUNCTION KEYS"
CRT \Enter ASCII numbers under 128 for characters \ :
CRT \returned by keys. To enter two\
CRT \numbers, connect with "+", such as: "27+67". To return \ :
CRT \ASCII 128 or greater,\
CRT \enter "27" and a number 128 below the final number desired.\
FOR I = 1 TO Total.All.Keys
    CRT Key.Scrn$(I) : I:". " : K$(I) Mask$ : " = " : Key.Ascii$(I) Mask$ :
NEXT I
LOOP; * Until Quit
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "Pick a Number or (Q)uit [1]: " :
    INPUT Response$ :
    IF Response$ = "" THEN Response$ = 1; * Default
WHILE NUM(Response$) DO
    N = Response$ + 0
    IF N > 0 AND N <= Total.All.Keys THEN
        CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(15) :
        CRT K$(N) : " = " : Key.Ascii$(N) :
        CRT "    Enter new ASCII numbers: " :
        INPUT Response$ :
        * Test for valid range for ASCII.
        * Default choice to keep same ASCII numbers.
        IF Response$ # "" THEN
            Total.Numbers = DCOUNT(Response$, "+")
            Test.Flag = True
            FOR I = 1 TO Total.Numbers WHILE Test.Flag
                Ascii = TRIM(FIELD(Response$, "+", I))
                IF NUM(Ascii) THEN
                    IF Ascii < 0 OR Ascii > 127 THEN
                        Test.Flag = False
                    END
                END ELSE
                    Test.Flag = False
                END
            NEXT I
            IF Test.Flag THEN
                Key.Ascii$(N) = Response$
                * Spaces from mask in case old number was longer
                CRT Key.Scrn$(N) : N : ". " :
                CRT K$(N) Mask$ : " = " : Key.Ascii$(N) Mask$ :
            END ELSE
                CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(5):
                CRT "ERROR: " : Ascii : " is not a number " :
                CRT "between 0 and 127. Press ENTER:" :
                INPUT Response$ :
            END; * If valid range for ASCII
        END; * If change in ASCII
    END; * If within range of number of keys
REPEAT; * Until Quit

```

```

*****
* Store New Keyboard *
*****
CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :
CRT "Name for keyboard or (Q)uit [" : Chars.Item.ID$ : "]: " :
INPUT Response$ :
IF Response$ = "Q" OR Response$ = "q" THEN GOTO 9999; *|End Program|
IF Response$ # "" THEN Chars.Item.ID$ = Response$
CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :
CRT "Storing keyboard " : Chars.Item.ID$ : " in file ":File.Keys.Name$:
*****
* Convert ASCII Numbers to Chars *
*****
MAT C$ = ""
FOR I = 1 TO Total.All.Keys
  IF Key.Ascii$(I) # "" THEN
    Total.Numbers = DCOUNT(Key.Ascii$(I), "+")
    FOR J = 1 TO Total.Numbers
      Ascii = TRIM(FIELD(Key.Ascii$(I), "+", J))
      C$(I) = C$(I) : CHAR(Ascii)
    NEXT J
  END
NEXT I
MATWRITE C$ ON File.Keys$, Chars.Item.ID$
*=====
*****
9999 * End Program *
*****
END

```

After the keyboard redefinitions have been created and stored in the KEYBOARD.CHCHOICES.WYSE.60.AT file, the CHANGE.KEYS.WYSE.60.AT.SUB external subroutine will apply the redefinitions to change the current keyboard. Although the redefinition process can be accomplished from any terminal or from port zero, which is the IBM PC/XT keyboard, the actual change can take place from a Wyse 60 AT terminal only. The program will check that the change is not being attempted from port zero.

The CHANGE.KEYS program will call the subroutine, after prompting the user for the keyboard ID and the name of the keyboard definition to apply. The APPLY.TERMINAL.AND.KEY.IDS program, shown later in this section, will call the routine without prompting the user for information. Each calling program will pass the name of the new keyboard definition in the parameter Keybd.to.Apply\$. The second parameter, Error.Flag, will be returned as true is no redefinition took place.

CHANGE.KEYS.WYSE.60.AT.SUB External Subroutine

```

SUBROUTINE CHANGE.KEYS.WYSE.60.AT.SUB(Keybd.to.Apply$, Error.Flag)
* Apply keys from KEYBOARD.CHCHOICES.WYSE.60.AT. David L. Clark 09/09/1989
* Pick Basic external subroutine.
* Pick IBM PC-XT R83 2.0, Wyse 60 AT terminal
* Filename under IBM PC DOS: ChKeysWy.bps
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* In the KEYS.AND.SYMBOLS.WYSE.60.AT file the subroutine will read a
* SYMBOLS.WYSE.60.AT Item in which each Attribute has a key symbol.

```



```

* In the KEYBOARD.CHOICES.WYSE.60.AT file the subroutine will read an
* Item in which each Attribute contains the characters returned
* by the key before resetting.
* The Attributes of the Symbols and Characters Items match by order.
* If no keyboard is applied, the Error.Flag returned to the caller as True.
* Alternative: Create Error.Num() array to show type of error.
* Unlike the IBM PC-XT, it is assumed that differences in Pick releases
* will not change the procedures to redefine the keyboard.

```

```

*****
* Assign Constants, Parameters *
*****
*****
* Set Constants, Parameters Valid for All Keyboards *
*****
PROMPT ""
EQU Delete$ TO CHAR(127)
EQU Direction TO 1;                * Assume remote direction for all
EQU Escape$ TO CHAR(27)
EQU False TO 0
Port.Num$ = FIELD(OCONV(0, "U50BB"), " ", 1)
Port.Num = Port.Num$ + 0;          * Convert to number
EQU True TO 1
EQU Value.Mark$ TO CHAR(253)
*****
* Set Parameters for AT Keyboard *
*****
* Change these for keyboard other than AT
DIM C$(56);                        * Chars returned by keystroke
File.Keys.Name$ = "KEYBOARD.CHOICES.WYSE.60.AT"
File.Keys.Symbols.Name$ = "KEYS.AND.SYMBOLS.WYSE.60.AT"
DIM K$(56);                        * Name of key
DIM S$(56);                        * Symbol for key
Symbols.Item.ID$ = "SYMBOLS.WYSE.60.AT"
EQU Total.All.Keys TO 56;          * Dimension for key arrays
*=====
*****
* Open Files *
*****
OPEN File.Keys.Name$ TO File.Keys$ ELSE
  CRT "Missing file: " : File.Keys.Name$
  Error.Flag = True
  GOTO 9999;                        *|End Subroutine|
END
OPEN File.Keys.Symbols.Name$ TO File.Keys.Symbols$ ELSE
  CRT "Missing file: " : File.Keys.Symbols.Name$
  Error.Flag = True
  GOTO 9999;                        *|End Subroutine|
END
*****
* Check Passed Parameter *
*****
IF Keybd.to.Apply$ = "" THEN
  CRT "Keyboard definition to apply not designated"
  Error.Flag = True
  GOTO 9999;                        *|End Subroutine|
END
*****
* Check for Not Port Zero *
*****

```

```

IF Port.Num = 0 THEN
  CRT "Only the Wyse 60 AT keys may be changed with this program"
  Error.Flag = True
  GOTO 9999;                                *|End Subroutine|
END
*****
* Read Symbols *
*****
MATREAD S$ FROM File.Keys.Symbols$, Symbols.Item.ID$ ELSE
  CRT "No key symbols found in Item: " : Symbols.Item.ID$
  Error.Flag = True
  GOTO 9999;                                *|End Subroutine|
END
*****
* Read Keyboard *
*****
MATREAD C$ FROM File.Keys$, Keybd.to.Apply$ ELSE
  CRT "No keyboard found in Item: " : Keybd.to.Apply$ :
  Error.Flag = True
  GOTO 9999;                                *|End Subroutine|
END
*****
* Apply Keyboard *
*****
FOR I = 1 TO Total.All.Keys
  PRINT Escape$ : "Z" : Direction : S$(I) : C$(I) : Delete$ :
NEXT I
*=====
*****
9999 * End Subroutine *
*****
RETURN
END

```

The SHOW.ASCII program was used with the IBM PC/XT keyboard, the Wyse 60 terminal, and the Radio Shack 100 in terminal mode, to obtain the values indicated in the tables of key values for PICK/BASIC. The program contains several alternatives. Comment out the alternatives selectively to observe their effect.

A keystroke from the IBM PC/XT running Pick will only return a single character, but a keystroke from a Pick terminal can return several characters. The program presents two versions for handling additional characters. The first version calls SYSTEM(14) with every keystroke and takes any number of additional characters. The second version avoids unnecessary system calls and looping by using a lead character to indicate the possible presence of a second character. If a two-character set has been sent, the routine returns only the second character, adding 128 to its ASCII value to distinguish it from regular keyboard values. This character could then be used in a branching statement of the form:

```
ON INDEX(Table.Ctrl.Chars$, Key$, 1) GOSUB 100, 200, 300, ...
```

For Pick Release 1983 2.2 and 3.0, a keystroke for a function key may return two characters: a lead character followed by a character for the specific key.

SHOW.ASCII Program

```
* SHOW.ASCII David L. Clark 04/25/1988
* Show ASCII number for keystroke.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ShowAsci.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* The program presents two versions of input:
*   Pick Basic INPUT statement, echoes Control characters despite ECHO OFF
*   unless a patch from Pick Systems is applied.
*   User Exit from PC-KEYS by Modular Software that does not echo.
* A keystroke may return more than 1 character. The SYSTEM(14) function
* will check the input buffer for the presence of an additional char.
* Disable the version that is not to be used by placing an asterisk at the
* start of the line.
*****
* Assign Constants, Initial Values *
*****
PROMPT ""
ECHO OFF
Key$ = ""; * Will contain keystroke
* Set lead character to indicate possible presence of a 2nd char.
* ASCII zero may not be used as the lead character.
Lead.Char$ = CHAR(27)
* Set PC-KEYS User Exit to input Key$
EQU Frame.Num TO 511; * Frame holding assembly routine
Input.User.Exit$ = "U2" : OCONV(Frame.Num, "MCDX") "R%3"
*****
* Display Exit Method *
*****
CRT \Enter "@" to end\
*****
* Get Keystroke Loop *
*****
LOOP UNTIL Key$ = "@" DO
  CRT "Type one character: " :
  * Pick input version
  INPUT Key$, 1 :
  * User Exit version
  Key$ = OCONV(0, Input.User.Exit$)
  Ascii.Num$ = SEQ(Key$); * 1st or only char
  *****
  * Check for More than 1 Char per Keystroke *
  *****
  * Two versions below: |Check Every Keystroke| and
  * |Check Only by Lead Char, Add 128 to ASCII|
  *****
  * Check Every Keystroke *
  *****
  * Check input buffer at every keystroke for any number of chars
  Length.Extra = SYSTEM(14)
  * Separate input statement required for each extra char
  FOR I = 1 TO Length.Extra
    * Pick Input version
    INPUT Key$, 1 :
    * User Exit version
    Key$ = OCONV(0, Input.User.Exit$)
```

```

        Ascii.Num$ = Ascii.Num$ : " + " : SEQ(Key$)
NEXT I
*****
* Check Only by Lead Char, Add 128 to ASCII *
*****
* Check for an additional char only when Lead.Char$ is returned.
* Check for only 1 additional char, not an unlimited number.
* Return only the 2nd char, after adding 128 to its ASCII value.
*   If programs attempts to set ASCII value over 255, Pick
*   will subtract 256.
* This version has been commented out.
IF Key$ = Lead.Char$ THEN
    IF SYSTEM(14) THEN
        * Pick Input version
        INPUT Key$, 1 :
        * User Exit version
        Key$ = OCONV(0, Input.User.Exit$)
        * Add 128 to ASCII value
        Key$ = CHAR(128 + SEQ(Key$))
        Ascii.Num$ = SEQ(Key$)
    END
END
*****
*Version to Show ASCII Only *
*****
CRT "      ASCII = " : Ascii.Num$
*****
* Version to Show ASCII and Character *
*****
* This version has been commented out
* CRT Key$ : " = ASCII " : Ascii.Num$
REPEAT; * Until Key$ = "e"
*****
* End Program *
*****
ECHO ON
END

```

Choose an ASCII Value for Each Editing Function

Because the various terminals and the IBM PC/XT running under Pick do not allow every key to be redefined, it will not be possible to make the keyboards conform by changing keys alone. The final step is to choose an ASCII value for each editing function, with different values assigned for different terminals and keyboards.

The values assigned for the keyboard will be concatenated into a table of control characters. The editing routine will branch to a particular function according to the position of the character input within the table of control characters:

```

Control.Char.Pos = (INDEX, Table.Control.Chars$, 1)
ON INDEX(Table.Ctrl.Chars$, Key$, 1) GOSUB 100, 200, 300, ...

```

The CHANGE.EDITING.KEYS.FILE program will create a table of control characters as one attribute in an item in the file EDITING.KEYS. The name for the item will be chosen by the user according to the terminal and keyboard in use. The terminal designa-

tion alone is not sufficient because some terminals, such as the Wyse 60, have different keyboards available.

The characters that are Pick System delimiters, ASCII 252 through 255, should not be stored in a file, therefore the program will not accept these characters as input. The data entry and editing program that uses the table of control characters also should filter out ASCII 252 through 255 as data input.

CHANGE.EDITING.KEYS.FILE Program

```
* CHANGE.EDITING.KEYS.FILE David L. Clark 08/24/1988
* Create a Table of Control Characters to use for branching when editing.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ChEdKyFi.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* In the file EDITING.KEYS create an Item with a Table of Control
* Characters as a single string in the first Attribute.
* Do not try to store Pick system delimiters, ASCII 252 - 255.
* The Item ID will be assigned by the user.
*****
* Assign Constants, Parameters *
*****
PROMPT ""
Clear.Screen$ = @(-1)
Edit.Keys.ID = "IBM.PC.XT.R83";          * Default choice for name
Erase.Screen.from.Pos$ = @(-3)
EQU False TO 0
File.Edit.Keys.Name$ = "EDITING.KEYS"
DIM Func$(35);                          * Description of each editing function
Func.Mask$ = "L(#20)"
DIM Key.Ascii(35);                       * ASCII char value set for each function
MAT Key.Ascii = 0
EQU Key.Tables.Max TO 20;                 * Maximum number of tables
DIM Key.Table.Name$(20)
DIM Key.Scrn$(35);                       * Screen position where displayed
Row.Bottom = SYSTEM(3)
Table.Ctrl.Chars$ = ""
EQU Total.Edit.Funcs TO 35;               * Dimension for arrays
EQU True TO 1
*****
* Assign Functions Array *
*****
* Must be changed for a program using different functions
Func$(1) = "Next      Field"
Func$(2) = "Previous Field"
Func$(3) = "Quit"
Func$(4) = "Backspace"
Func$(5) = "Delete"
Func$(6) = "Left  1 character"
Func$(7) = "Right 1 character"
Func$(8) = "Up    1 line"
Func$(9) = "Down 1 line"
Func$(10) = "Left  1 word"
Func$(11) = "Right 1 word"
```

```

Func$(12) = "Start of Field"
Func$(13) = "End   of Field"
Func$(14) = "Top    of Page"
Func$(15) = "Bottom of Page"
Func$(16) = "Previous Page"
Func$(17) = "Next     Page"
Func$(18) = "Start of Form"
Func$(19) = "End   of Form"
Func$(20) = "Jump to Field"
Func$(21) = "Page of Form"
Func$(22) = "Over-Type, Insert"
Func$(23) = "Clear Field"
Func$(24) = "Get Previous Field"
Func$(25) = "Select Text"
Func$(26) = "Copy  Selected Text"
Func$(27) = "Cut   Selected Text"
Func$(28) = "Paste Selected Text"
Func$(29) = "Erase Value"
Func$(30) = "New Item ID Number"
Func$(31) = "Next Item Short Cut"
Func$(32) = "Help"
Func$(33) = "Prev Item Short Cut"
Func$(34) = "Start of List"
Func$(35) = "Resume  List"
*****
* Assign Screen Positions in 2 Cols *
*****
* List begins on row 5, counting from title at row 0 at top.
FOR I = 1 TO 18
    Key.Scrn$(I) = @(0, I + 4)
NEXT I
FOR I = 19 TO Total.Edit.Funcs
    Key.Scrn$(I) = @(39, I - 14)
NEXT I
*=====*
*****
* Open Files *
*****
OPEN File.Edit.Keys.Name$ TO File.Edit.Keys$ ELSE
    CRT "Missing file: " : File.Edit.Keys.Name$
    GOTO 9999;                *|End Program|
END
*****
* Get Default Choices for Editing Keys, If Available *
*****
EXECUTE \SSELECT \ : File.Edit.Keys.Name$ CAPTURING Capture.Output$
IF SYSTEM(11) THEN;          * At least 1 Item in file
    CRT Clear.Screen$ :
    CRT @(20) : "CHOOSE KEY TABLE FOR BEGINNING VALUES" :
    Quit.Flag = False
    FOR I = 1 TO Key.Tables.Max UNTIL Quit.Flag
        READNEXT Key.Table.Name$(I) THEN
            CRT @(20, I + 1) : I : ". " : Key.Table.Name$(I) :
        END ELSE
            Quit.Flag = True
    END
NEXT I
CRT @(20, Row.Bottom) : "Pick a Number or (Q)uit [1]: " :
INPUT Response$ :

```

```

IF NOT(NUM(Response$)) THEN GOTO 9999;    *|End Program|
IF Response$ = "" THEN
    Num.Choice = 1
END ELSE
    Num.Choice = Response$ + 0
END
Edit.Keys.ID$ = Key.Table.Name$(Num.Choice)
READV Table.Ctrl.Chars$ FROM File.Edit.Keys$,Edit.Keys.ID$,1 ELSE
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "No key table found in Item: " : Edit.Keys.ID$ :
    GOTO 9999;    *|End Program|
END
*****
* Convert Table Control Chars to ASCII Numbers *
*****
FOR I = 1 TO Total.Edit.Funcs
    Key.Ascii(I) = SEQ(Table.Ctrl.Chars$[I, 1])
NEXT I
END; * If at least 1 Item in File.Edit.Keys
*****
* Set New ASCII Values for Editing Functions *
*****
CRT Clear.Screen$ : @(25) : "KEY VALUES FOR EDITING FUNCTIONS"
CRT @(5, 2) : "Enter an ASCII number below 252 for each editing " :
CRT "function." :
FOR I = 1 TO Total.Edit.Funcs
    CRT Key.Scrn$(I) : I:". ":Func$(I) Func.Mask$:" = ":Key.Ascii(I):
NEXT I
LOOP; * Until Quit
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "Pick a Number or (Q)uit [1]: " :
    INPUT Response$ :
    IF Response$ = "" THEN Response$ = 1;    * Default
WHILE NUM(Response$) DO
    N = Response$ + 0
    IF N > 0 AND N <= Total.Edit.Funcs THEN
        CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(10) :
        CRT Func$(N) : " = " : Key.Ascii(N) :
        CRT "      Enter one new ASCII number: " :
        INPUT Response$ :
        * Test for valid range for ASCII.
        * Do not allow pick system delimiters.
        * Default choice to keep same ASCII numbers.
        Response$ = TRIM(Response$)
        IF Response$ # "" THEN
            Test.Flag = True
            Ascii = Response$
            IF NUM(Ascii) THEN
                IF Ascii < 0 OR Ascii > 251 THEN Test.Flag = False
            END ELSE
                Test.Flag = False
            END
        END
        IF Test.Flag THEN
            Key.Ascii(N) = Ascii
            * Spaces from mask in case old number was longer
            CRT Key.Scrn$(N) : N : ". " :
            CRT Func$(N) Func.Mask$ : " = " :
            CRT Key.Ascii(N) "L(#3)"
        END
    END

```

```

        END ELSE
        CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(5):
        CRT "ERROR: " : Ascii : " is not a number " :
        CRT "between 0 and 251. Press ENTER:" :
        INPUT Response$ :
        END; * If valid number for ASCII
    END; * If change in ASCII
END; * If within range of number of editing functions
REPEAT; * Until Quit
*****
* Store New Table Control Chars *
*****
CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :
CRT "Name for key table or (Q)uit [" : Edit.Keys.ID$ : "]: " :
INPUT Response$ :
IF Response$ = "Q" OR Response$ = "q" THEN GOTO 9999; *|End Program|
IF Response$ # "" THEN Edit.Keys.ID$ = Response$
CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :
CRT "Storing key table " : Edit.Keys.ID$ :
CRT " in file " : File.Edit.Keys.Name$:
*****
* Convert ASCII Numbers to Table Control Chars *
*****
Table.Ctrl.Chars$ = ""
FOR I = 1 TO Total.Edit.Funcs
    Table.Ctrl.Chars$ = Table.Ctrl.Chars$ : CHAR(Key.Ascii(I))
NEXT I
WRITEV Table.Ctrl.Chars$ ON File.Edit.Keys$, Edit.Keys.ID$, 1
=====
*****
9999 * End Program *
*****
END

```

A data-entry program, rather than asking the user each time the program began for the name of the keyboard to apply, would take the name of the keyboard redefinition, as well as identifiers for the table of control characters for editing and for a set of video characteristics, out of a parameters file. The file `TERMINAL.AND.KEY.IDS` will store these identifiers, to be applied automatically by a program.

The item ID of each item in the file will consist of the terminal type, the Pick version, and the port number. The port number will be optional, except for port zero. Except in the case of port zero, if the data entry program cannot find an item ID with the current port number, the program will look for an item ID with no port number element. The elements of the item ID are connected with asterisks:

```
Item.ID$ = Term.Type$ : "*" : Pick.Version$ : "*" : Port.Num
```

The attributes in the item will contain the following information:

- 01 Terminal name
- 02 Keyboard name—some terminals, such as the Wyse 60, offer different keyboard options
- 03 New keyboard definition, if any, to set when the program begins
- 04 Old keyboard definition, if any, to restore when the program ends

The Pick version designation can be a part of the terminal and keyboard parameters. Thus an IBM PC/XT using Pick release 1983 could be indicated as "IBM.PC.XT.R83".

It probably will be necessary to include a version specification in the terminal name, to reflect differences in @ () functions between Pick versions. It probably will not be necessary to include the version in the keyboard name, except for the keyboard used as port zero, such as the IBM PC/XT, where Pick can change which keys it implements. The Wyse 60 terminal name might be stored as "WYSE.60.R83", including the version, but not the keyboard option, and the keyboard stored as "WYSE.60.AT", reflecting the keyboard option, but not the version. The facilities to create terminal and keyboard parameters make it easy to duplicate a set of parameters and then include slight variations.

The distinction between terminal type and terminal name arises mainly because, on Pick PC/XT version 2.0, terminal type is limited to a single letter. Under open architecture, a full name is allowed.

The STORE.TERMINAL.AND.KEY.IDS program will create items for the TERMINAL.AND.KEY.IDS file. The program will ask the user for the elements of the item ID and for the identifiers to store.

The program begins by displaying any existing item IDs in the TERMINAL.AND.KEY.IDS file, and asking which is to be used as the basis for a new or changed setup. If no items are present in the file, the program will create an item for the IBM PC/XT, using Pick version release 1983, on port zero, to employ for default identifier values.

The program then displays the current terminal and keyboard identifiers and asks which is to be changed. The choices for each identifier are taken from the item IDs of the appropriate file. The choices for terminal name are the item IDs of the TERMINAL.PARAMETERS file, which was created by the SET.TERMINAL.PARAMETERS program.

The choices for keyboard name are the item IDs of the EDITING.KEYS file, which holds the tables of control characters, and was set by the CHANGE.EDITING.KEYS.FILE program. The keyboard name also will be employed later to find the name of the external subroutine that will be used to redefine the keyboard, and the name of the file that holds the alternate keyboard definitions. The subroutine will be named "CHANGE.KEYS." plus the keyboard name, plus ".SUB". The file will be named "KEYBOARD-CHOICES." plus the keyboard name.

The choices for new and old keyboard definitions are the item IDs of the "KEYBOARD-CHOICES." (plus keyboard name) file. If no such file exists for the keyboard name previously selected, then the keyboard is not to be redefined, and these identifiers are null.

After the user is done changing identifiers, and indicates "Quit," the program asks if the new set of identifiers is to be saved. If it is, the program then asks if the same ID for the set is to be used, or a new ID. If a new ID is to be employed, the program asks in turn for the terminal type (a one-letter code under Pick IBM PC/XT R83 2.0), Pick version (such as R83 or OA), and port. For port, the user can give a number, or enter "ALL" to signify that the same parameters will apply to a terminal of this type on any port.

STORE.TERMINAL.AND.KEY.IDS Program

```

* STORE.TERMINAL.AND.KEY.IDS David L. Clark 08/25/1988
* Store identifiers for terminal and keyboard.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: StorTmKy.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Store in the TERMINAL.AND.KEY.IDS file identifiers for: terminal name,
* keyboard name, any new keyboard definition, and any old keyboard
* definition to restore.
* Note the distinction between terminal and keyboard. For the Wyse 60 and
* some other terminals, different keyboard options are available. It
* will also probably be necessary to indicate the Pick version in the
* terminal name, but not in the keyboard name.
* The Item ID in the TERMINAL.AND.KEY.IDS file will consist of terminal
* type, Pick version, and Port number.
* The first time that the program is used, it will create an Item
* for the IBM PC-XT running Pick Release 1983 on Port zero.
* The program may be run from any terminal or port. It is not necessary
* to run it from the terminal for which the parameters are to be set.
*****
* Assign Constants, Parameters *
*****
PROMPT ""
* Array of identifiers for terminals and keyboards must be
* dimensioned before equates
DIM Term.Key.ID$(4)
* Constants used to define other constants
Erase.Screen.from.Pos$ = @(-3)
Row.Bottom = 23
* Other constants
DIM Choice$(44)
EQU Choices.Max TO 44;           * Max choices for display on screen
Choices.Total = 0;              * Actual number of choices available
Clear.Screen$ = @(-1)
Erase.Row.Bottom$ = @(0, Row.Bottom) : Erase.Screen.from.Pos$
EQU False TO 0
File.Edit.Keys.Name$ = "EDITING.KEYS"
* See program: CHANGE.EDITING.KEYS.FILE
* File.Keys.Name$ : See program "CHANGE.KEYS.FILE." + Keybd.ID$
File.Keys.Name.Begin$ = "KEYBOARD.CHOICES."; * Will add Keybd.ID$
File.Term.Key.IDs.Name$ = "TERMINAL.AND.KEY.IDS"
File.Term.Params.Name$ = "TERMINAL.PARAMETERS"
* See program: SET.TERMINAL.PARAMETERS
* IDs.File.ID$ : ID of Item in File.Term.Keys.ID$ holding identifiers
EQU Keybd.ID$ TO Term.Key.ID$(2); * Terminal and keyboard
EQU Keybd.New$ TO Term.Key.ID$(3); * New keyboard definition
EQU Keybd.Old$ TO Term.Key.ID$(4); * Old keyboard to restore
Prompt.Mask$ = "L(#21)"
DIM Term.Key.Prompt$(4)
Term.Key.Prompt$(1) = "TERMINAL"
Term.Key.Prompt$(2) = "TERMINAL AND KEYBOARD"
Term.Key.Prompt$(3) = "NEW KEY DEFINITION"
Term.Key.Prompt$(4) = "OLD KEY DEFINITION"
EQU Term.ID$ TO Term.Key.ID$(1); * Name of terminal
EQU Total.Term.Key.IDs TO 4; * Dimension of array
EQU True TO 1
* Start identifiers at null

```

```

MAT Term.Key.ID$ = ""
*=====
*****
* Open Files *
*****
OPEN File.Term.Key.IDs.Name$ TO File.Term.Key.IDs$ ELSE
  CRT "Missing file: " : File.Term.Key.IDs.Name$
  GOTO 9999;          *|End Program|
END
*****
* Check for Files *
*****
File.Choices.Name$ = File.Edit.Keys.Name$;  * Table of Control Chars
GOSUB 110;          *|Read Choices from File|
IF Choices.Total = 0 THEN
  CRT "Missing Table of Control Characters in file: " :
  CRT File.Edit.Keys.Name$
  GOTO 9999;          *|End Program|
END
File.Choices.Name$ = File.Term.Params.Name$; * Video characteristics
GOSUB 110;          *|Read Choices from File|
IF Choices.Total = 0 THEN
  CRT "Missing video characteristics in file: " :
  CRT File.Term.Params.Name$
  GOTO 9999;          *|End Program|
END
*****
* Read Current Identifiers or Create Default Item *
*****
File.Choices.Name$ = File.Term.Key.IDs.Name$
GOSUB 110;          *|Read Choices from File|
IF Choices.Total = 0 THEN;          * No Items already in file
  *****
  * Create Default Terminal Key IDs Item *
  *****
  * Defaults for IBM PC-XT, Pick Release 1983 2.0, Port zero.
  * Change Keys.New$ for a data entry program with a
  *   name other than "HISTORY.DATABASE".
  IDs.File.ID$ = "I*R83*0"
  Term.ID$ = "IBM.PC.XT.R83"
  Keybd.ID$ = "IBM.PC.XT.R83"
  Keybd.New$ = "HISTORY.DATABASE*IBM.PC.XT.R83"
  Keybd.Old$ = "USA"
  MATWRITE Term.Key.ID$ ON File.Term.Key.IDs$, IDs.File.ID$
  * Assign single Choice$() for display
  Choices.Total = 1
  Choice$(1) = IDs.File.ID$
END; * If any terminal parameters item already exists
Title$ = "CODES FOR TERMINAL, VERSION, PORT"
GOSUB 120;          *|Display Choices|
GOSUB 130;          *|Pick a Number or Quit|
IF Quit.Flag THEN GOTO 9999;          *|End Program|
IDs.File.ID$ = Choice$(Num.Choice)
MATREAD Term.Key.ID$ FROM File.Term.Key.IDs$, IDs.File.ID$ ELSE
  CRT Erase.Row.Bottom$ : @(20) :
  CRT "No terminal and key identifiers found in Item: ":
  CRT IDs.File.ID$ :
  GOTO 9999;          *|End Program|
END

```

```

*****
* Change Terminal and Key Identifiers *
*****
*****
* Choose Identifiers to Change *
*****
* Sends to |Pick a Number or Quit|: Choices.Total
* Receives from |Pick a Number or Quit|: Num.Choice, Quit.Flag
LOOP;                                * Until Quit.Flag
    Title$ = "TERMINAL AND KEY IDENTIFIERS"
    Choices.Total = Total.Term.Key.IDs
    FOR I = 1 TO Choices.Total
        Choice$(I) = Term.Key.Prompt$(I) Prompt.Mask$ : " = "
        Choice$(I) = Choice$(I) : Term.Key.ID$(I)
    NEXT I
    GOSUB 120;                        *|Display Choices|
    GOSUB 130;                        *|Pick a Number or Quit|
UNTIL Quit.Flag DO
    ID.Num = Num.Choice
    *****
    * Change Identifier *
    *****
    Title$ = Term.Key.Prompt$(ID.Num)
    BEGIN CASE
        CASE ID.Num = 1; * Terminal Name
            File.Choices.Name$ = File.Term.Params.Name$
            GOSUB 100;                *|Choose from List of Item IDs|
        CASE ID.Num = 2; * Terminal and Keyboard Name
            * There must be a separate Table of Control Chars for
            * every different keyboard.
            File.Choices.Name$ = File.Edit.Keys.Name$
            GOSUB 100;                *|Choose from List of Item IDs|
        CASE ID.Num > 2; * Keyboard Definition
            * These two identifiers will be null if no redefinition
            CRT Erase.Row.Bottom$ : @(5) :
            CRT "Keyboard to be redefined? 1. Yes 2. No" :
            CRT " Pick a Number [1]: " :
            INPUT Response$ :
            IF Response$ # "2" THEN
                File.Keys.Name$ = File.Keys.Name.Begin$ : Keybd.ID$
                File.Choices.Name$ = File.Keys.Name$
                GOSUB 100;                *|Choose from List of Item IDs|
            END ELSE
                MAT Choice$ = ""
                Num.Choice = 1
            END
        END CASE
    IF NOT(Quit.Flag) THEN
        Term.Key.ID$(ID.Num) = Choice$(Num.Choice)
    END
REPEAT
    *****
    * Store New Identifiers *
    *****
    CRT Erase.Row.Bottom$ : @(5) :
    CRT "Store New Identifiers? 1. Yes 2. No" :
    CRT " Pick a Number [1]: " :
    INPUT Response$ :

```

```

IF Response$ = "2" THEN GOTO 9999; *|End Program|
CRT Erase.Row.Bottom$ :
CRT "Identifiers Name = " : IDs.File.ID$ : " 1. Keep 2. Change" :
CRT " Pick a Number [1]: " :
INPUT Response$ :
IF Response$ = "2" THEN
    *****
    * Create New Identifiers Name *
    *****
    Term.Type$ = FIELD(IDs.File.ID$, "*", 1)
    CRT Erase.Row.Bottom$ : @(15) :
    CRT "Enter Terminal Type [" : Term.Type$ : "]: " :
    INPUT Response$ :
    IF Response$ # "" THEN Term.Type$ = Response$
    Version$ = FIELD(IDs.File.ID$, "*", 2)
    CRT Erase.Row.Bottom$ : @(15) :
    CRT "Enter Pick Version Name [" : Version$ : "]: " :
    INPUT Response$ :
    IF Response$ # "" THEN Version$ = Response$
    Port.Num$ = FIELD(IDs.File.ID$, "*", 3)
    IF Port.Num$ = "" THEN Port.Num$ = "ALL"
    CRT Erase.Row.Bottom$ : @(15) :
    CRT \Enter Port Number or "ALL" [\ : Port.Num$ : "]: " :
    INPUT Response$ :
    IF Response$ # "" THEN Port.Num$ = Response$
    IF Port.Num$ = "ALL" THEN
        IDs.File.ID$ = Term.Type$ : "*" : Version$
    END ELSE
    IF NUM(Port.Num$) THEN
        Port.Num = Port.Num$ + 0
        IDs.File.ID$ = Term.Type$ : "*" : Version$ : "*" : Port.Num
    END ELSE
    CRT
    CRT \Entry for Port Number = \ : Port.Num$
    CRT \Port Number must be a number or "ALL"\
    GOTO 9999; *|End Program|
END
END
END; * If new IDs.File.ID$
CRT Erase.Row.Bottom$ : "Storing Identifiers: " : IDs.File.ID$ :
CRT " in file: " : File.Term.Key.IDs.Name$ :
MATWRITE Term.Key.ID$ ON File.Term.Key.IDs$, IDs.File.ID$
GOTO 9999; *|End Program|
*=====
*****
100 * Choose from List of Item IDs *
*****
* Parameters: File.Choices.Name$, Choices.Max, Title$
* Returns: Choice$(), Choices.Total, Num.Choice, Response$, Quit.Flag
GOSUB 110; *|Read Choices from File|
IF Choices.Total THEN
    GOSUB 120; *|Display Choices|
    GOSUB 130; *|Pick a Number or Quit|
END ELSE
    * Choice$() array set to null
    Num.Choice = 1; * Choice$(Num.Choice) will be null
    Quit.Flag = False
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :

```

```

        CRT "No Choices for: " : Title$ : ".    Press ENTER: " :
        INPUT Response$ :
    END
    RETURN
    *-----*
    *****
110  * Read Choices from File *
    *****
    * Parameters: File.Choices.Name$
    * Returns: Choice$(), Choices.Total
    EXECUTE \SSELECT \ : File.Choices.Name$ CAPTURING Capture.Output$
    Test.Flag = False
    Choices.Total = 0
    MAT Choice$ = ""
    FOR I = 1 TO Choices.Max UNTIL Test.Flag
        READNEXT Choice$(I) THEN
            Choices.Total = Choices.Total + 1
        END ELSE
            Test.Flag = True
    END
    NEXT I
    RETURN
    *-----*
    *****
120  * Display Choices *
    *****
    * Parameters: Choice$(), Choices.Total
    * Returns: nothing
    CRT Clear.Screen$ : @(25) : "CHOOSE: " : Title$
    IF Choices.Total < 23 THEN
        Row.Adjust = (Choices.Total < 22);    * Blank row if enough room
        FOR I = 1 TO Choices.Total
            CRT @(15, I + Row.Adjust) : I : ". " : Choice$(I) :
        NEXT I
    END ELSE
        Row.Adjust = (Choices.Total < 43);    * Blank row if enough room
        FOR I = 1 TO 22
            CRT @(0, I + Row.Adjust) : I : ". " : Choice$(I) :
        NEXT I
        Row.Adjust = Row.Adjust - 22
        FOR I = 23 TO Choices.Total
            CRT @(39, I + Row.Adjust) : I : ". " : Choice$(I) :
        NEXT I
    END
    RETURN
    *-----*
    *****
130  * Pick a Number or Quit *
    *****
    * Parameters: Choices.Total
    * Returns: Num.Choice, Response$, Quit.Flag
    LOOP;    * Until pass test for a valid response
        CRT Erase.Row.Bottom$ : @(20) :
        CRT "Pick a Number or (Q)uit [1]: " :
        INPUT Response$ :
        IF NUM(Response$) THEN
            Quit.Flag = False
            IF Response$ = "" THEN
                Num.Choice = 1

```

```

        Test.Flag = True
    END ELSE
        Num.Choice = Response$ + 0
        Test.Flag = (Num.Choice>0 AND Num.Choice<=Choices.Total)
    END
END ELSE;                * Assume non-numeric response = Quit
Quit.Flag = True
Test.Flag = True
END
UNTIL Test.Flag DO
REPEAT
RETURN
*=====
*****
9999 * End Program *
*****
END

```

The APPLY.TERMINAL.AND.KEY.IDS.SUB external subroutine would be called by a data entry program. The subroutine will apply the parameters stored in the TERMINAL.AND.KEY.IDS file by the STORE.TERMINAL.AND.KEY.IDS program.

To use the Pick version as part of the identification process, the subroutine must receive the version specification as a parameter from the caller, or the subroutine must find the version by using the SYSTEM(100) function or by prompting the user. The SYSTEM(100) function works only on the IBM PC/XT and PC/AT. The use of SYSTEM(100) is demonstrated in the routine "Get Version." For an IBM PC/XT running release 1983 2.0 the following message was returned:

PICK'S R83 PC/XT Ver 2.0 07 May 1986

If neither the characters "R83" (release 1983) or "OA" (open architecture) can be found in the returned message, then the program prompts the user to enter an identifier for the version. For each version of Pick used, check the SYSTEM(100) function to make sure that it returns the characters that the program will search for.

A more exact identifier of the version might be necessary, such as "R83-2.0" or "R83-2.2". It is assumed here that the user will be running the latest version available for a particular machine. The latest version currently available for the IBM PC/XT is "R83 2.0".

To redefine the keyboard an external subroutine must be called, because each type of terminal will require different programming for redefining its keyboard. The name of the subroutine will consist of "CHANGE.KEYS." plus the keyboard ID, plus ".SUB". The name of the alternate keyboard definition to apply will be passed to the subroutine. Keyboard definitions were created with the CHANGE.KEYS program.

The TEST.CALL.TERMINAL.AND.KEY.IDS program is intended only to demonstrate how a data entry program would make use of the APPLY.TERMINAL.AND.KEY.IDS.SUB external subroutine. Alternative methods would include incorporating the code from APPLY.TERMINAL.AND.KEY.IDS.SUB directly in a data entry program, or making APPLY.TERMINAL.AND.KEY.IDS.SUB the calling program and the data entry module the external subroutine.

If the program—external subroutine form is used, then the “Assign Constants, Parameters” section of both the program and the subroutine should include a COMMON statement to share certain variables in common. Using COMMON provides a faster method of sharing an array than passing it as a parameter. For an array, COMMON also serves as a dimension statement. It is not required that the variables have the same names between caller and subroutine, only that they appear in the same order in the COMMON statement.

The following should be declared COMMON:

- ☐ Error.Flag : To report if the terminal and keyboard parameters could not applied
- ☐ Keybd.to.Restore\$: If the old keyboard definition is to be restored
- ☐ Keybd.Subroutine\$: Call at end of data entry to restore old keyboard
- ☐ Table.Ctrl.Chars\$: The table of control characters for editing
- ☐ Term.Param\$() : Array that will be equated with screen constants

If Keybd.to.Restore\$ is not null, then the data entry program will have to call Keybd.Subroutine\$ to restore the old keyboard definition.

The caller will include EQUATE statements equating screen constants to Term.Param\$(). It is not necessary to include the EQUATEs in the subroutine, as long as the variables will not be used in the subroutine.

TEST.CALL.TERMINAL.AND.KEY.IDS Program

```
* TEST.CALL.TERMINAL.AND.KEY.IDS David L. Clark 08/25/1988
* Show how a data entry program would call APPLY.TERMINAL.AND.KEY.IDS.SUB
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TCalTmKy.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* External subroutine applies terminal and keyboard parameters.
* If an old keyboard definition must be restored, call Keybd.Subroutine$
* at the end.
*****
* COMMON Statements to Share Variables *
*****
* Same in program and external subroutine
COMMON Error.Flag, Keybd.to.Restore$, Keybd.Subroutine$
COMMON Table.Ctrl.Chars$, Term.Param$(26)
*****
* Equate Screen Constants to Terminal Parameters Array *
*****
EQU Term.Type.Param$      TO Term.Param$(1)
* Graphics characters
EQU Blank.Reverse$       TO Term.Param$(2)
EQU Cross$               TO Term.Param$(3)
EQU Horiz.Bar$           TO Term.Param$(4)
EQU Left.Down$           TO Term.Param$(5)
EQU Left.Up$             TO Term.Param$(6)
EQU Mid.Down$            TO Term.Param$(7)
EQU Mid.Left$            TO Term.Param$(8)
EQU Mid.Right$           TO Term.Param$(9)
```



```

EQU Mid.Up$           TO Term.Param$(10)
EQU Right.Down$       TO Term.Param$(11)
EQU Right.Up$         TO Term.Param$(12)
EQU Vert.Bar$         TO Term.Param$(13)
* Video characteristics
EQU Cursor.Off$       TO Term.Param$(14)
EQU Cursor.On$        TO Term.Param$(15)
EQU Screen.Lock$      TO Term.Param$(16)
EQU Undo.Screen.Lock$ TO Term.Param$(17)
EQU Video.Intensity.High$ TO Term.Param$(18)
EQU Video.Intensity.Low$ TO Term.Param$(19)
EQU Video.Protect$     TO Term.Param$(20)
EQU Undo.Video.Protect$ TO Term.Param$(21)
EQU Video.Reverse$     TO Term.Param$(22)
EQU Video.Reverse.Low$ TO Term.Param$(23)
EQU Undo.Video.Reverse$ TO Term.Param$(24)
EQU Video.Underline$  TO Term.Param$(25)
EQU Undo.Video.Underline$ TO Term.Param$(26)
*****
* Assign Constants, Parameters *
*****
PROMPT ""
EQU False TO 0
EQU True TO 1
*=====
*****
* Call |APPLY.TERMINAL.AND.KEY.IDS| External Subroutine *
*****
*****
* Initialize COMMON Parameters *
*****
Error.Flag = False; * Returns True if parameters not applied
Keybd.to.Restore$ = ""
Keybd.Subroutine$ = ""
Table.Ctrl.Chars$ = ""
MAT Term.Param$ = ""
CALL APPLY.TERMINAL.AND.KEY.IDS.SUB
*****
* Return from External Subroutine *
*****
IF Error.Flag THEN
    CRT
    CRT "Terminal and keyboard parameters not applied"
    GOTO 9999; *|End Program|
END
*****
* Body of Data Entry Program *
*****
* The body of a data entry program would be placed here, or a call to
* a data entry external subroutine.
*****
* Draw a Box *
*****
* Demonstrate graphics characters passed from external subroutine
CRT
CRT " " : Left.Down$ : STR(Horiz.Bar$, 13) : Right.Down$
CRT " " : Vert.Bar$ : " Box Drawing " : Vert.Bar$
CRT " " : Vert.Bar$ : " Characters " : Vert.Bar$
CRT " " : Left.Up$ : STR(Horiz.Bar$, 13) : Right.Up$

```

```

CRT
*****
* Choose Program to Execute *
*****
* For demonstration purposes, the program will ask for the name of a
* program to EXECUTE. Use the SHOW.ASCII program to see the
* effects of redefining the keyboard.
* No parameters are passed with the EXECUTE statement.
* If the program has not been cataloged, the required form is:
* RUN Filename Program.Name
CRT "Enter name of a program to execute: " :
INPUT Program.Name$
IF Program.Name$ # "" THEN
  CRT
  EXECUTE Program.Name$
END
*=====*
*****
9999 * End Program *
*****
* One reason for always exiting to an |End Program| routine, rather
* than placing STOP statements within a program, is to be sure
* that if any restoration or cleanup is necessary that it will
* be done, because it comes immediately before |End Program|.
*****
* Restore Old Keyboard *
*****
* Parameters from COMMON: Keybd.to.Restore$, Keybd.Subroutine$
* Will cause no harm to restore this keyboard, even if new one was
* not applied.
IF Keybd.to.Restore$ # "" AND Keybd.Subroutine$ # "" THEN
  Keybd.to.Apply$ = Keybd.to.Restore$
  Error.Flag = False; * Will return True if keyboard not applied

CALL @Keybd.Subroutine$(Keybd.to.Apply$, Error.Flag)
*****
* Return from Keyboard Restoration *
*****
IF Error.Flag THEN
  CRT "Keyboard was not restored"
  CRT "Keyboard to restore was: " : Keybd.Old$
END; * If keyboard not restored
END; * If keyboard defined for restoration
* End of program
END

```

APPLY.TERMINAL.AND.KEY.IDS.SUB External Subroutine

```

SUBROUTINE APPLY.TERMINAL.AND.KEY.IDS.SUB
* Apply IDs from file TERMINAL.AND.KEY.IDS. David L. Clark 08/25/1988
* Pick Basic External Subroutine
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: AppTmKy.bps
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Apply from TERMINAL.AND.KEY.IDS file identifiers for: terminal name,
* terminal plus any keyboard option (such as AT for Wyse 60) which will
* indicate the Table of Control Characters for editing, any new keyboard
* definition, and any old keyboard definition to restore.
* The Item ID consists of terminal type, Pick version, and Port number.

```

```

*****
* COMMON Statement to Share Variables *
*****
* Same in program and external subroutine
COMMON Error.Flag, Keybd.to.Restore$, Keybd.Subroutine$
COMMON Table.Ctrl.Chars$, Term.Param$(26)
*****
* Assign Constants, Parameters *
*****
PROMPT ""
* Array of identifiers for terminals and keyboards must be
* dimensioned before equates
DIM Term.Key.ID$(4)
* Other constants
EQU False TO 0
File.Edit.Keys.Name$ = "EDITING.KEYS"
* See program: CHANGE.EDITING.KEYS.FILE
* File.Keys.Name$ : See program "CHANGE.KEYS.FILE." + Keybd.ID$
File.Keys.Name.Begin$ = "KEYBOARD.CHOICES."; * Will add Keybd.ID$
File.Term.Key.IDs.Name$ = "TERMINAL.AND.KEY.IDS"
File.Term.Params.Name$ = "TERMINAL.PARAMETERS"
* See program: SET.TERMINAL.PARAMETERS
* IDs.File.ID$ : ID of Item in File.Term.Keys.ID$ holding identifiers
Keybd.Subroutine.Begin$ = "CHANGE.KEYS."; * Will add Keybd.ID$
EQU Keybd.ID$ TO Term.Key.ID$(2); * Terminal and keyboard
EQU Keybd.New$ TO Term.Key.ID$(3); * New keyboard definition
EQU Keybd.Old$ TO Term.Key.ID$(4); * Old keyboard to restore
Port.and.Account$ = OCONV(0, "U50BB")
Port.Num$ = FIELD(Port.and.Account$, " ", 1)
Port.Num = Port.Num$ + 0; * Convert to number
EQU Term.ID$ TO Term.Key.ID$(1); * Name of terminal
* Term.Param$() not dimensioned here, because in COMMON
Term.Type$ = SYSTEM(7)
EQU Total.Term.Key.IDs TO 4; * Dimension of array
EQU Total.Term.Params TO 26; * " " " "
EQU True TO 1
EQU Value.Mark$ TO CHAR(253)
=====
*****
* Open Files *
*****
OPEN File.Edit.Keys.Name$ TO File.Edit.Keys$ ELSE
  CRT "Missing file: " : File.Edit.Keys.Name$
  Error.Flag = True
  GOTO 9999; *|End Subroutine|
END
OPEN File.Term.Key.IDs.Name$ TO File.Term.Key.IDs$ ELSE
  CRT "Missing file: " : File.Term.Key.IDs.Name$
  Error.Flag = True
  GOTO 9999; *|End Subroutine|
END
OPEN File.Term.Params.Name$ TO File.Term.Params$ ELSE
  CRT "Missing file: " : File.Term.Params.Name$
  Error.Flag = True
  GOTO 9999; *|End Subroutine|
END
*****
* Get Version *
*****

```

```

* Version$ could be passed instead by the caller.
* SYSTEM(100) returns version on IBM PC-XT and PC-AT only.
* Check SYSTEM(100) on each Pick version used to verify the exact
* message returned, in case "OA" or "R83" is not included.
Version.Message$ = SYSTEM(100)
IF INDEX(Version.Message$, "R83", 1) THEN
    Version$ = "R83"
END ELSE
    IF INDEX(Version.Message$, "OA", 1) THEN
        Version$ = "OA"
    END ELSE
        CRT "Enter Version Identifier or (Q)uit: " :
        INPUT Version$ :
        IF Version$[1, 1] = "Q" OR Version$[1, 1] = "q" THEN
            Error.Flag = True
            GOTO 9999;
        *|End Subroutine|
    END
END
END
*****
* Read Identifiers *
*****
IDs.File.ID$ = Term.Type$ : "*" : Version$ : "*" : Port.Num
MATREAD Term.Key.ID$ FROM File.Term.Key.IDs$, IDs.File.ID$ ELSE
    * Maybe Port not specified
    IDs.File.ID$ = Term.Type$ : "*" : Version$
    MATREAD Term.Key.ID$ FROM File.Term.Key.IDs$, IDs.File.ID$ ELSE
        CRT "No identifiers found for Terminal Type: " : Term.Type$
        CRT "Version: " : Version$
        CRT "In file: " : File.Term.Key.IDs.Name$
        CRT "Use the program STORE.TERMINAL.AND.KEY.IDS" :
        CRT " to create identifiers"
        Error.Flag = True
        GOTO 9999;
    *|End Subroutine|
END
END
*****
* Read Terminal Parameters by Terminal ID *
*****
MATREAD Term.Param$ FROM File.Term.Params$, Term.ID$ ELSE
    CRT "Missing Terminal Parameters for Terminal: " : Term.ID$
    CRT "in file: " : File.Term.Params.Name$
    Error.Flag = True
    GOTO 9999;
    *|End Subroutine|
END
*****
* Convert Video ASCII or Function Numbers to Chars *
*****
* Local variables:
*   Num.Func      : Negative number represents an @() function
*                   If number not negative, it is an ASCII char
*   Param$        : Holds new parameter while old is being converted
*   Total.Values  : Total separate values in parameter
*   Term.Param$(1) = Terminal type code. Could use to check.
*   When parameters were created with program SET.TERMINAL.PARAMETERS
*   Pick system delimiters (ASCII 252-255) were not allowed.
FOR I = 2 TO Total.Term.Params
    IF Term.Param$(I) # "" THEN

```

```

Param$ = ""
Total.Values = DCOUNT(Term.Param$(I), Value.Mark$)
FOR J = 1 TO Total.Values
    Num.Func = Term.Param$(I)<1, J> + 0
    IF Num.Func < 0 THEN;                * @() function
        Param$ = Param$ : @(Num.Func)
    END ELSE;                            * ASCII Char
        Param$ = Param$ : CHAR(Num.Func)
    END
NEXT J
Term.Param$(I) = Param$
END
NEXT I
*****
* Read Table of Control Chars by Keyboard ID *
*****
* There must be a separate Table of Control Chars for every keyboard.
READV Table.Ctrl.Chars$ FROM File.Edit.Keys$, Keybd.ID$, 1 ELSE
    CRT "Missing Table of Editing Keys for keyboard: " : Keybd.ID$
    CRT "in file: " : File.Edit.Keys.Name$
    Error.Flag = True
    GOTO 9999;                          *|End Subroutine|
END
*****
* Call External Subroutine to Redefine Keyboard *
*****
* Keybd.to.Apply$ in File.Keys$ is passed to the external subroutine.
Keybd.Subroutine$ = Keybd.Subroutine.Begin$ : Keybd.ID$ : ".SUB"
IF Keybd.New$ # "" THEN
    Keybd.to.Apply$ = Keybd.New$
    Error.Flag = False;                * Will return True if keyboard not applied
    CALL @Keybd.Subroutine$(Keybd.to.Apply$, Error.Flag)
    *****
    * Return from Keyboard Redefinition *
    *****
    IF Error.Flag THEN
        CRT "Keyboard was not redefined"
        CRT "Keyboard to apply was: " : Keybd.to.Apply$
        * Error.Flag will be passed back to caller as True
        GOTO 9999;                    *|End Subroutine|
    END
END; * If Keybd.New$ to be applied
*****
* Pass COMMON Parameter to Restore Old Keyboard *
*****
* Keybd.to.Restore$ is in COMMON statement for use by caller.
* Keybd.Old$ is EQUATED to Term.Key.ID$().
Keybd.to.Restore$ = Keybd.Old$
=====
*****
9999 * End Subroutine *
*****
RETURN
END

```

KEYBOARD: CONTROL CHARACTERS FOR EDITING

IBM PC BASIC

In the routine “Branch According to Key,” INSTR and ON...GOSUB are used with a table of control characters. The program branches to an appropriate editing routine according to which key has been pressed.

(For setting the values of the function keys, see the section: “Keyboard: Change Keys.” Note again that all of the comments should be removed from the executable code, and variable names abbreviated to one or two characters, with no periods.)

Branch According to Key Routine

Branch According to Key: Dictionary of Constants and Variables

The short version of the name is used in the executable program, the long version in the documentation:

```
K$ = Key$           : Key pressed, can be a control character
ZT$= Table.Ctrl.Chars$ : Table of control characters

' *****
' * Branch According to Key *
' *****
' * Branch to editing routine according to Control Key pressed by user
' * IBM PC Basic routine, documentation version
' * IBM PC BASICA 3.0, Radio Shack 100
' * Stored under filename: BrnchKey.brd
' * Parameters: Key$
' * Returns: Call to Editing Routine
ON INSTR(Table.Ctrl.Chars$, Key$) GOSUB 100, 200, 300 ...
RETURN
```

PICK/BASIC

In the following “Branch According to Key” routine, INDEX and ON . . . GOSUB are used with a table of control characters. The program branches to an appropriate editing routine according to which key has been pressed.

Branch According to Key Routine

```
*****
* Branch According to Key *
*****
* Parameters: Key$
* Returns: Call to Editing Routine
INDEX(Table.Ctrl.Chars$, Key$, 1) ON GOSUB 100, 200, 300 ...
RETURN
*_____*
```

You might wish to test that the Key\$ pressed did in fact appear within the table of control characters, with a construction such as:

```
Command.Number = INDEX(Table.Ctrl.Chars$, Key$, 1)
IF Command.Number THEN ON Command.Number GOSUB 100, 200, 300 ...
```

The Boolean Flag test is safe, because Command.Number will definitely be a number, and not a string. However, the routine will be faster if this extra test is not necessary. A previous routine would have determined if Key\$ represented a printable or a control character.

The same result also could be obtained with the CASE statement, but ON...GOSUB is faster.

KEYBOARD: ESCAPE KEY TO INTERRUPT

IBM PC BASIC

INKEY\$

For example:

```
Escape = CHR$(27)
IF RIGHT$(INKEY$, 1) = Escape$ THEN ... '* IBM PC or Radio Shack 100
IF      INKEY$      = Escape$ THEN ... '* Radio Shack 100 only
```

INKEY\$ will return any keystroke waiting in the keyboard buffer. An INKEY\$ routine, to check if the user had pressed the ESCAPE key, can provide a method to exit from a process such as printing or transferring a file. The INKEY\$ statement should be placed within the loop that accomplishes the printing or transfer. The computer will check for an ESCAPE at that point in the loop. The response will not be instantaneous. The escape only will occur when the INKEY\$ statement is reached.

On the IBM PC, control codes are returned with a length of two characters. On the Radio Shack 100, the code will have a length of one character. However, the IBM PC version will also work on the 100.

To ascertain whether any key had been pressed, not only ESCAPE, the program would check:

```
IF INKEY$ <> "" THEN ...
```

(The function keys also can be used to interrupt a process, as shown in the section: "Keyboard: Function Keys.")

The following program contains a loop with an inner loop that counts from 0 to 319, displaying the last digit of each number. The numbers fill four rows on the IBM PC or the entire screen on the Radio Shack 100. As the outer loop ends each time, it branches to a subroutine that checks to see if the ESCAPE key has been pressed. If it has, the program presents a choice of continuing or quitting. The user might employ this pause to put more paper in the printer, and then continue the process. Many different processes in a program could use the same subroutine to check for the ESCAPE key.

Boolean tests are faster and shorter than comparisons, positive tests are faster and shorter than negative tests, the choice used most frequently should always be placed first, and a FOR...NEXT loop is faster than "IF...GOTO Line.Number."

Test Escape Program, Documentation Version

Test Escape Program: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

F	= False	: Boolean value: 0 in IBM PC Basic
QF	= Quit.Flag	: Boolean Flag: IF True, stop the process
T	= True	: Boolean value: -1 in IBM PC BASIC
ZZ\$	= Escape\$	: ESCAPE key, control code ASCII 27

```

1  '* TEscape.bsd: Test Escape  David L. Clark  05/04/1987
2  '* Test ESCAPE key to interrupt.
3  '* IBM PC Basic program. Documentation version.
4  '* IBM PC BASICA 3.0, Radio Shack 100
5  '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10 '* GOTO 9500                      '*|Assign Constants, Parameters|
   *=====*
   '* Subroutines:
   '*****
100 '* Display Numbers *
   '*****
   '* Parameters: Quit.Flag
CLS
:   FOR I = 1 TO 1000
:       FOR J = 0 TO 319
:           PRINT RIGHT$(STR$(J), 1);
:       NEXT
:       GOSUB 200                      '*|Check for ESCAPE Key|
:       I = I + 1000 * -Quit.Flag '* If Quit.Flag = True, end of loop
:   NEXT
:   RETURN
   '* End line
   '-----*
200 '* Check for ESCAPE Key *
   '*****
   '* Parameters: Escape$
   '* Returns: Quit.Flag
   '* Local variables: X$
   '* Begin line
   '* Use only 1 of the next 2 statements:
   '* Faster alternative for Radio Shack 100: IF INKEY$ <> Escape$
   IF RIGHT$(INKEY$, 1) <> Escape$ THEN '* IBM PC or Radio Shack 100
       RETURN                      '*** Exit #1 ***
   ELSE
       CLS

```



```

:      PRINT "1.Continue  2.Quit"
:      LINE INPUT "Pick a Number [1]: "; X$
:      Quit.Flag = VAL(X$) = 2
:      '*** If pressed anything but 2, Quit.Flag will be False
:      RETURN                                     '*** Exit #2 ***
:      '* End: If Inkey$ ...
'* End line
'*****
6000  '* Main Program *
'*****
Quit.Flag = False                                '* Set parameter for subroutine
:      GOSUB 100                                '*|Display Numbers|
:      GOTO 9990                                '*|End Program|
'*****
9500  '* Assign Constants, Parameters *
'*****
DEFINT A-Z
'* Constants:
:      Escape$ = CHR$(27)
:      False   = 0
:      True    = -1
:      GOTO 6000                                '*|Main Program|
'* End line
'*****
9990  '* End Program *
'*****
END

```

PICK/BASIC

SYSTEM(14)

For example:

```
IF SYSTEM(14) THEN ...
```

The SYSTEM function with parameter 14 will return the number of keystrokes in the input buffer. The function can be used to determine if a key has been pressed during a process, and then give the user the opportunity to end the process early.

Note the use of the INPUT one keystroke function to absorb the ESCAPE character:

```
INPUT Escape.Char$, 1 :
```

If the ESCAPE character had not been absorbed in this way, it would have been included in the input taken into the Response\$ variable. You could use the INPUT statement to check that it was the ESCAPE key that was pressed, and not some other key:

```
Escape.Key$ = CHAR(251)
INPUT Key$, 1 :
IF Key$ = Escape.Key$ THEN ...
```

The next program will clear the screen and then loop 1,000 times, displaying digits 0 through 9 on each loop, unless it is interrupted by the pressing of a key.

TEST.ESCAPE Program

```
* TEST.ESCAPE David L. Clark 11/09/1987
* Test ESCAPE key to interrupt.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TEscape.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants, Parameters *
*****
PROMPT ""
EQU False TO 0, True TO 1
Backspace$ = @(-9)
Clear.Screen$ = @(-1)
Quit.Flag = False
*=====
*****
* Main Program *
*****
GOSUB 100;                      *|Display Numbers|
GOTO 9990;                      *|End Program|
*=====
*****
100 * Display Numbers *
*****
* Parameters: Quit.Flag
* Local variables: I, J
CRT Clear.Screen$ :
FOR I = 1 TO 1000 UNTIL Quit.Flag
  FOR J = 0 TO 9
    CRT J : " " :
  NEXT J
  GOSUB 200;                      *|Check for ESCAPE Key|
NEXT I
RETURN
*-----
*****
200 * Check for ESCAPE Key *
*****
* Returns: Quit.Flag
* Local variables:
*   Escape.Char$
*   Response$
* Checks to see if any key has been pressed, not just ESCAPE
* Backspace over the display of the ESCAPE character "]"
IF SYSTEM(14) THEN
  CRT
  * Absorb ESCAPE character, so that it will not be in Response$
  INPUT Escape.Char$, 1 :
  * Backspace over and erase ESCAPE character display "["
  CRT Backspace$ : " " : Backspace$ :
```

```

      CRT "1.Continue      2.Quit      Pick a Number [1]:" :
      INPUT Response$ :
      Quit.Flag = Response$ = "2"
              * If pressed anything but 2, Quit.Flag will remain False
      CRT
END
RETURN
=====
*****
9990  * End Program *
*****
END

```

KEYBOARD: FUNCTION KEYS

For changing the values returned by the Function keys, see the section: “Keyboard Input: Change Keys.”

IBM PC BASIC

ON KEY GOSUB Line.Number { , Line.Number, ... }

For example:

ON KEY GOSUB 100, 200, 300, , 500 ...

Pressing a function key interrupts the program and causes it to branch to the line in a list of line numbers that corresponds to the number of the function key. Thus if you press F3, the program will branch to the third number in the list. The line numbers must be literals, not variables. The line numbers are separated by commas. To skip a position in the list, place a comma at that point in the list, with no number.

The function key interrupt could be used to allow the user to interrupt a process when encountering difficulties, such as during printing. Unlike the interrupt method using the ESCAPE key (shown in the section “Keyboard: ESCAPE Key to Interrupt”), the Function key method would not require that the program continually check to see if the key had been pressed. In addition, the program could branch to different recovery routines depending on which key was pressed.

KEY ON { Key Number }, KEY OFF, KEY STOP

The function key interrupt first must be enabled with the KEY ON statement, which can include a number for a specific function key to be enabled. Thus the statement:

KEY ON

will enable an interrupt for all the function keys, while:

KEY (8) ON

will enable only function key F8.

KEY OFF disables the interrupt. KEY STOP also disables the interrupt, but stores the information about any function key pressed, so that a subsequent KEY ON command will cause branching to the designated line number.

(On the Radio Shack 100, the method of disabling the BREAK key shown in the section: "Keyboard: BREAK Key" also can be used to disable the function key interrupt, and set the Function keys F1-F8 to return ASCII values 248-255.)

KEYBOARD: ONE KEYSTROKE

You can input a single keystroke without having to press ENTER for data entry and editing.

IBM PC BASIC

INKEY\$

For example:

```
10 Key$ = INKEY$ : IF Key$ = "" GOTO 10
```

INKEY\$ inputs one character without pressing the ENTER key. INKEY\$ does not wait for a response, so the routine must loop back until a response is returned. The character typed is not echoed to the screen. If ENTER is pressed without typing a character, a null string is assigned to the variable.

For an example of the use of INKEY\$, see the program "Character to ASCII" in the section: "Characters: Convert: Character to ASCII."

INPUT\$

For example:

```
Keys.Entered$ = INPUT$(Num.Chars)
```

The INPUT\$ statement will take the number of keystrokes designated with the Num.Chars parameter from the keyboard and place them in a string variable without requiring that ENTER be pressed. The Num.Chars parameters can be a variable or a literal, and it must be within IBM PC BASIC's string limit of 255 characters. The computer will wait for a response. The characters typed are not echoed to the screen. If ENTER is pressed without typing any characters, a null string is assigned to the variable. The principal use for "INPUT\$ Num.Chars" is to take in a single keystroke. Thus the statement:

```
Key$ = INPUT$(1)
```

will accept a single keystroke and place it in the variable Key\$.

PICK/BASIC

INPUT

For example:

```
INPUT Key$, 1 :
```

This INPUT statement takes in one character without requiring that the user press ENTER. The character typed is assigned to the variable, which in the example is named Key\$. The computer waits for a response. The colon prevents a Carriage Return—Line Feed after the input. The command:

```
PROMPT ""
```

should be used at the beginning of the program or routine to suppress the automatic prompt character of "?".

If the ECHO OFF command has been issued the character typed should not be echoed to the screen.

(For PROMPT and ECHO OFF, see the beginning of the section: "Keyboard." To avoid echoing control characters in PICK/BASIC, and for other methods of inputting one character, see the section: "Keyboard: Change Keys.")

KEYBOARD: SELECT A CHOICE FROM A MENU

The main program will first display a list of choices. The main program will put the total number of legal choices into the variable Num.Choices.Total, and call the routine "Pick a Number" to accept the user's input.

The Pick a Number routine will put the prompt to the user at the bottom of the screen:

```
Pick a Number [1]:
```

The user will type a number selected from a list of choices, then press the ENTER key. To accept the default choice, which is always the first item on the list, the user can press the ENTER key without typing a number, thereby entering a null value that the Pick a Number routine will convert into choice number "1." The routine will handle the incorrect input of a letter or a number greater than the total number of choices available.

In the documentation some conditional tests are shown stretching over several lines. For example:

```
UNTIL Num.Choice < = Num.Choices.Total AND Num.Choice > 0  
  AND Test.Flag DO
```

In an executable program, such a test would not be split between lines by Carriage Return—Line Feeds.

(For details on checking that only numeric digits have been entered, see the section: "Numbers: Convert: String to Number.")

IBM PC BASIC

Pick a Number Routine

Pick a Number Routine: Dictionary of Constants and Variables

The short version of the name is used in the executable program, the long version in the documentation:

N = Num.Choice : Number of choice selected from menu
NT = Num.Choices.Total : Total choices, to insure correct input

```
100 *****
   * Pick a Number *
   *****
   * David L. Clark 09/22/1988
   * Pick a number from choices on a menu.
   * IBM PC Basic routine. Documentation version.
   * IBM PC BASICA 3.0, Radio Shack 100
   * Stored under filename: PickNum.brd
   * Copyright 1988 David L. Clark -- TAB BOOKS Inc.
   * Parameters: Num.Choices.Total
   * Returns: Num.Choice
   * Local variables:
   * X$ = Response$ : Input entered by user
PRINT " Pick a Number [1]: ";
: LINE INPUT Response$
: IF Response$ = "" THEN          '* ENTER only for Default choice
    Num.Choice = 1
: RETURN
ELSE
    '* Before assigning to Integer, must test with VAL. Could be
    '* out of Integer range of - 32768 to 32767.
    '* If Response$ was not null, but Num.Choice is 0,
    '* the user pressed a character other than a number.
    '* Also prevent a negative number that would pass the
    '* Boolean Flag test "IF Num.Choice THEN..."
: IF VAL(Response$) > 0
    AND VAL(Response$) <= Num.Choice.Total THEN
    Num.Choice = VAL(Response$)
    RETURN
ELSE
    GOTO 100                      '* Loop back if response not valid
    '* End: If VAL(Response$)
    '* End: If Response$ = ""
'* End line
```

PICK/BASIC

Pick a Number Routine

```
*****
* Pick a Number *
*****
* David L. Clark 09/22/1988
* Pick a number from choices on a menu.
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: PickNum.bpr
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants *
*****
PROMPT ""
EQU False TO 0
EQU True TO 1
*****
* Pick a Number *
*****
* Parameters: Num.Choices.Total
* Returns: Num.Choice
* Local variables:
*   Response$ : Input entered by user
*   Test.Flag : Test for valid number
LOOP;                                * Repeat until valid number or null
  CRT " Pick a Number [1]: " :
  INPUT Response$ :
  IF Response$ = "" THEN
    Num.Choice = 1;                  * ENTER only for Default choice
    Test.Flag = True
  END ELSE
    Num.Choice = Response$;          * Pick does not require conversion
    Test.Flag = NUM(Num.Choice);      * Check that it is a number
  END
UNTIL Num.Choice <= Num.Choices.Total AND Num.Choice > 0
  AND Test.Flag DO
REPEAT; * End: Loop Until Num.Choice
RETURN
```

Numbers



NUMBERS: CONVERT: DECIMAL, HEXADECIMAL AND OTHER BASES

Pick TCL

DTR

For example:

DTR {New.Base} Number.to.Convert

To convert a decimal number to another radix (base) use the DTR (Decimal to Radix) command. The default radix is hexadecimal (base 16).

RTD

For example:

RTD {New.Base} Number.to.Convert

To convert a number with a nondecimal radix (base) to decimal form use the RTD (Radix to Decimal) command. The default radix is hexadecimal (base 16).

NUMBERS: CONVERT: NEGATIVE TO POSITIVE

IBM PC BASIC

ABS

For example:

Positive.Number = ABS(Numeric.Expression)

ABS will make Positive.Number positive, whether Numeric.Expression is positive or negative.

IBM PC BASIC uses -1 as a Boolean true value. (See the section: “Boolean Flags.”) The ABS statement would convert the flag to the +1, which is the true value used by PICK/BASIC:

Test.Flag = ABS(A = B) '* Returns 0 or 1

PICK/BASIC

ABS

For example:

Absolute.Value = ABS(Numeric.Expression)

Absolute.Value will be an unsigned number, whether Numeric.Expression is positive or negative.

NUMBERS: CONVERT: NUMBER TO STRING

To convert a number into a form where it can be manipulated as a string, use the String command.

IBM PC BASIC

STR\$

For example:

A.Number = 2
A.String\$ = STR\$(A.Number)

When a positive integer is converted, its positive “sign” will become a blank, thus A.String\$ would = “ 2”. To eliminate the blank, you must extract, beginning with the second character:

A.String\$ = MID\$(STR\$(A.Number), 2)

(For an explanation of MID\$, see the section: “Strings: Extract.”)

PICK/BASIC

It is not necessary to convert a number to a string in PICK/BASIC, because strings and numbers are not held as distinct types:

```
A.String$ = A.Number
```

ALPHA

For example:

```
IF ALPHA(String$) THEN ...
```

To test for valid alphabetic characters, use the ALPHA function:

```
Test.Alpha.Flag = ALPHA(A. Number)
IF Test.Alpha.Flag THEN
    A.String$ = A.Number : ""
ELSE
    A.String$ = ""
END
```

ALPHA will test that all the characters are valid alphabetic characters. If there are any digits, the result will be zero for false. Thus "12C" would not be accepted. If there are any symbols such as "#" or "?" the result will be zero for false.

A string should be stored as a string, and not as a number, otherwise the system will have to perform repeated conversion when string usage is called for. To store a number as a string, concatenate a null to it:

```
A.String$ = A.Number : ""
```

NUMBERS: CONVERT: STRING TO NUMBER

To convert a string into a form where it can be manipulated as a number, use the Value command. It is best to take input from the keyboard as a string and then convert it into a number, rather than having the program crash if a proper number is not entered.

IBM PC BASIC

VAL

For example:

```
A.String$ = "2"
A.Number = VAL(A.String$)
```

To take input in string form from the keyboard with LINE INPUT, and then use the input as a number, such as for a selection from a list of choices:

```

LINE INPUT "Pick a Number: "; Choice$
Num.Choice = VAL(Choice$)

```

VAL ignores leading blanks and trailing nondigits. If there are no digits or the string is null, VAL returns 0. Thus if the user only pressed Carriage Return in response to the request above, Num.Choice would = 0.

To make sure that an entire string is numeric, extract one character at a time and test if the ASCII value of the character is between 48 and 57 inclusive, for digits 0-9. In the example, note that the colon indicates the continuation of a line, and that remarks would not actually appear in the middle of a line.

```

Length = LEN(Object$)
:   FOR I = 1 TO Length
:       Ascii = ASC(MID$(Object$, I, 1))
:       IF Ascii > 47 AND Ascii < 58 THEN                '* Number 0 - 9
:           NEXT
:
:               Test.Num.Flag = True
:       ELSE
:           I = Length                                '*Break Loop if not numeric
:           NEXT
:           Test.Num.Flag = False
:       '*End If Ascii for a digit
:   '* End: For I
'* End Line

```

PICK/BASIC

It is not necessary to convert a string to a number in PICK/BASIC because strings and numbers are not held as distinct types;

```
A.Number = A.String$
```

Although Pick does not have formal type distinctions, Pick will give the error message "Non-numeric used when numeric expected" if you try to use a string where only a number would be appropriate. For example, you can multiply "2" by "3," but you cannot multiply "dog" by "cat." To test for a valid number, use the function NUM:

NUM

```

Test.Num.Flag = NUM(A.String$)
IF Test.Num.Flag THEN
    A.Number = A.String$ + 0
ELSE
    A.Number = 0
END

```

NUM will not accept a number with nondigits. Thus from the string "12C" the IBM PC BASIC VAL function would take the number 12. Pick's NUM function would declare

that there was no number, and any attempt to perform addition, multiplication, or other arithmetic operations with it would return an error message. NUM will also not accept nonalphabetic symbols such as “#” or “?”.

A number should be stored as a number, and not as a string. Leading zeros that take up extra space will be stored for a string, but not for a number. In addition, the system will not have to perform the conversion repeatedly when numerical usage is called for. To cause the string to be stored as a number, add zero:

```
Number = Number$ + 0
```

You cannot add zero with an element of a dynamic array as the recipient. Thus the following operation would be illegal:

```
Number<Attrib.Num> = Number<Attrib.Num> + 0
```

Elements of a dynamic array can only be stored as strings. Pick considers a dynamic array to be a string divided into elements by delimiters.

Null, the empty string “”, can be treated differently on different Pick implementations. On the IBM PC/XT, a null value passes the numeric test NUM (), identifying it as a number, and fails the alphabetic test ALPHA (), which would identify it as an alphabetic character. Although null passes the numeric test, null is not considered equal to zero, and will fail that equality test. However, null can be made equal to zero by adding a zero to it. The following TEST.NULL.ZERO program tests whether null is considered equivalent to zero on a particular implementation.

TEST.NULL.ZERO Program

```
* TEST.NULL.ZERO David L. Clark 11/11/1987
* Test if Null is considered equivalent to Zero.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TNulZero.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  String$ = ""
  Number = 0
  IF NUM(String$) THEN
    CRT "Null passes Numeric test"
  END ELSE
    CRT "Null fails Numeric test"
  END
  IF ALPHA(String$) THEN
    CRT "Null passes Alphabetic test"
  END ELSE
    CRT "Null fails Alphabetic test"
  END
  IF String$ = Number THEN
    CRT "Null = Zero"
  END ELSE
    CRT "Null does not equal Zero automatically"
  END
  IF String$ + Number = 0 THEN CRT "Null may be made equal to Zero"
* End Program
END
```

Operators



THE VARIOUS LANGUAGES FOR THE MOST PART USE THE SAME OPERATORS, HOWEVER, the precedence of operators differs among languages, causing what would appear to be the same operation to have different results. The expression:

$A * -B$

will succeed in IBM PC BASIC and PICK/BASIC, in which the unary minus takes precedence, but would not work in other languages such as Pascal, in which multiplication takes precedence over the unary minus. To be safe, use parentheses, which always have the highest precedence. The expression:

$A * (-B)$

will always have the same meaning.

When operators in an expression have the same level of precedence, the expression is evaluated from left from right.

The following lists show the operators for each language, in order of precedence, with notations for any operators whose meaning is not obvious. Indentation indicates continuation of the same level of precedence.

(For more details on the arithmetic operators, see the section: "Arithmetic." For more details on the string operators, see the section: "Strings.")

IBM PC BASIC

()
^ '* Exponent
Unary +, Unary -
*, /
\ '* Integer division
MOD
+, -
<, >, =, < >, < =, = >
NOT
AND
OR
XOR '* Exclusive OR: conditions not the same
EQV '* Equivalence: both conditions true or false
IMP '* First condition implies the second. Result is
 '* true unless first condition true and second false

IBM PC BASIC's "AND" might cause confusion with bit operations. Thus the expression:

IF (A < B) AND (C > D)

would be made more reliable with parentheses:

IF (A < B) AND (C > D)

PICK/BASIC

(), NOT(Expression) * Parentheses required for function
[] * Extract a substring from a string
Unary =, Unary -
^ * Exponent
*, /
+, -
: or CAT * Concatenate two strings
= or EQ, < > or > < or # or NE,
 < or LT, > or GT,
 < = or LE, > = or GE,
MATCH OR MATCHES * Test if string matches a pattern
AND or &, OR or !

At Pick TCL level, for Access search statements, BEFORE can be used for less than, AFTER can be used for greater than, in the absence of an operator "=" is assumed, and in the absence of a connective OR is assumed. (For details see the section: "Files, Items: Search with the Pick Access Processor.")

Note Pick's optional use of one-character operators where more characters are the norm, as in "#" for Not Equal, "!" for OR, and "&" for AND.

Pick allows the combination of string and numeric expressions. Thus Pick would allow you to extract a number from a string, or to form a number by concatenating strings, and then use that number in arithmetic. (For the string operators, see the section: "Strings.")

Although the Pick manual includes " $= <$ " for less than or equal and " $= >$ " for greater than or equal, those operators do not work on Pick PC/XT release 2.0. The following Test Operators program will test the operators. Each operator that was not accepted has been noted and "commented out" by placing an asterisk at the beginning of the line.

The Pick manual also is incorrect in stating that concatenation has precedence over arithmetic. The example used of " $5 * 2 : 0$ " would yield the same result of 100 no matter whether multiplication or concatenation had precedence. The remainder of the Test Operators program tests the precedence of various operators.

TEST.OPERATORS Program

```
* TEST.OPERATORS David L. Clark 11/11/1987
* Test the operators and their precedence.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TOperatr.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Test Each Relational Operator *
*****
* The operations that were not accepted by the Pick Basic IBM PC-XT
* compiler have an asterisk in front. To test if an operator works
* on a different system, remove the asterisk.
CRT "Test Relational Operators"
Small = 1
Big = 10
*
IF Small = 1 THEN CRT "= worked"
IF Small # Big THEN CRT "# worked"
IF Small <> Big THEN CRT "<> worked"
IF Small NE Big THEN CRT "NE worked"
IF Small >< Big THEN CRT ">< worked"
IF Small < Big THEN CRT "< worked"
IF Small <= Big THEN CRT "<= worked"
IF Small LE Big THEN CRT "LE worked"
* IF Small =< Big THEN CRT "=< worked"; * Not accepted by compiler
IF Big > Small THEN CRT "> worked"
IF Big >= Small THEN CRT ">= worked"
IF Big GE Small THEN CRT "GE worked"
* IF Big => Small THEN CRT "=> worked"; * Not accepted by compiler
IF NOT(Small = Big) THEN CRT "NOT worked"
*****
* Test the Precedence of Operators *
*****
Result.Concat = 4 * 2 : 5
CRT "4 * 2 : 5 = " : Result.Concat
```

```

IF Result.Concat = 100 THEN
    CRT "Concatenation has precedence over arithmetic"
END ELSE
    CRT "Arithmetic has precedence over concatenation"
END
Number$ = 125
Result.Extract = 4 * Number$[2, 2]
CRT "Number$ = " : Number$
CRT "4 * Number$[2, 2] = " : Result.Extract
IF Result.Extract = 100 THEN
    CRT "Extraction has precedence over arithmetic"
END ELSE
    CRT "Arithmetic has precedence over extraction"
END
Result.Divide = 1000 / 2 * 5
CRT "1000 / 2 * 5 = " : Result.Divide
IF Result.Divide = 100 THEN
    CRT "Multiplication has precedence over division"
END ELSE
    CRT "Multiplication and division have equal precedence"
END
* End program
END

```


Printer



THE FOLLOWING COMMANDS ARE USED TO SEND OUTPUT TO THE PRINTER.

IBM PC BASIC

LPRINT

For example:

```
LPRINT X$; Y$; Z$; ' * No Carriage Return  
LPRINT X$; Y$; Z$ ' * With Carriage Return
```

An ending semicolon suppresses the Carriage Return—Line Feed. To send an extra Carriage Return—Line Feed, use “LPRINT” alone:

```
LPRINT X$  
LPRINT  
IPRINT Y$
```

You also could put a Carriage Return—Line Feed in a variable, rather than starting a new LPRINT statement:

```
CRLF$ = CHR$(13) + CHR$(10)  
LPRINT X$; CRLF$; CRLF$; Y$
```

If commas are used as separators rather than semicolons, the element will be printed at preset tab positions beginning with zero for the left-hand column and increasing by 14 positions across the page.

Positive numbers print with a blank in front. (See the section: "Numbers: Convert: Number to String.")

TAB()

For example:

```
LPRINT TAB(Column - 1); U; X$; ...
```

LPRINT TAB(Column) actually starts printing at Column + 1, therefore set the TAB number for one less than the column where printing should begin. TAB also comes up short on very long lines. It will not allow the total of the spaces to skip and the characters to print to exceed 255. If the print position is already greater than the column number, TAB has no effect.

The following simple Print Label program will print labels. It will first ask the user for the maximum number of lines per label and the left margin of spaces to skip. This information will be used to set the position to begin printing, so that it will not be necessary for the user to adjust the printer. The number of lines per label also will be used to calculate the lines to skip between labels. Thus if the maximum length is six lines, and the user types two lines to print on the label, the program knows to skip four lines before starting on the next label. The following prompts will appear:

Maximum lines per label [6]:

Left margin [25]:

The prompts will offer default choices in brackets. To accept the default choice, simply press ENTER without typing anything. To indicate something other than the default, type the number desired and press ENTER.

The program will request each line that it is to print, up to the maximum number of lines indicated previously by the user. When all of the lines desired have been entered, press ENTER without typing anything to indicate that the entry stage is finished. For example, to enter two lines for printing and then to quit entry, a user might type the following. The user's input appears after the colon:

Enter Line#1: LADWP-15-1

Enter Line#2: Water-supply

Enter Line#3:

If the user presses ENTER without typing anything at the first line, the program will end, assuming that no more labels are to be printed.

If text has been entered for labels, the program will ask how many labels are to be printed. The default choice is one label only:

Number of labels to print [1]:

The program will print the labels, and then return to the prompt for the first line of a label, to await more input. To end the program, press ENTER without typing any text:

Enter Line#1:

PrtLabel.bsd Program

Print Label: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

LM	= Left.Margin	: Spaces to skip
LN\$()	= Line\$()	: Test to print in lines
LS	= Lines.to.Skip	: Skip lines between labels.
		Equals Lines.Max - Lines.Total.
LT	= Lines.Total	: Actual number to print
LX	= Lines.Max	: Maximum on a label
NL	= Number.of.Labels	: Repeat the lines on each label

```

1      '* PrtLabel.bsd: Print Labels. David L. Clark 08/26/1988
2      '* Print information entered by user on multiple labels.
3      '* IBM PC Basic program. Documentation version.
4      '* IBM PC BASICA 3.0, Radio Shack 100
5      '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
20     '* Get maximum lines per label
30     LINE INPUT "Maximum lines per label [6]: "; Lines.Max$
40     IF Lines.Max$ = "" THEN
        Lines.Max = 6                '* Default
    ELSE
        Lines.Max = VAL(Lines.Max$) '* Convert to number
    '* End
50     IF Lines.Max = 0 THEN          '* Error: No lines
        PRINT "Maximum lines requested was: "; Lines.Max$
        : GOTO 30                    '* Loop back
    '* End
60     DIM Lines$(Lines.Max)
70     LINE INPUT "Left margin [25]: "; Left.Margin$
80     IF Left.Margin$ = "" THEN
        Left.Margin = 25             '* Default
    ELSE
        Left.Margin = VAL(Left.Margin$) '* Convert to number
    '* End
90     IF Left.Margin < 0 THEN        '* Error: Negative number
        PRINT "Left margin requested was: "; Left.Margin$
        : GOTO 70                    '* Loop back
    '* End
100    '* Top of Loop to Enter and Print
110    FOR I = 1 TO Lines.Max
120        PRINT "Enter Line#"; MID$(STR$(I), 2, 1); ": ";
130        LINE INPUT Line$(I)
140        IF Line$(I) = "" THEN
            Lines.Total = I-1
        : I = Lines.Max             '* Break loop

```

```

150     '* End
160 NEXT I
170 IF Lines.Total = 0 GOTO 9999           '* Exit if no lines to print
180 Lines.to.Skip = Lines.Max - Lines.Total '* Skip lines between labels
190 LINE INPUT "Number of labels to print [1]: "; Number.of.Labels$
200 IF Number.of.Labels$ = "" THEN
210     Number.of.Labels = 1               '* Default
220 ELSE
230     Number.of.Labels = VAL(Number.of.Labels$) '* Convert to number
240 '* End
250 IF Number.of.Labels = 0 THEN           '* User changed mind or error
260     PRINT "Number of labels requested was: "; Number.of.Labels$
270     : GOTO 100                         '* Loop back
280 '* End
290 '* Print Labels
300 FOR I = 1 TO Number.of.Labels
310     FOR J = 1 TO Lines.Total
320         LPRINT TAB(Left.Margin); Line$(J)
330     NEXT J
340     FOR J = 1 TO Lines.to.Skip
350         LPRINT
360     NEXT J
370 NEXT I
380 GOTO 100                             '* Loop back
390 '* Radio Shack 100 exit = MENU
400 '* IBM PC DOS exit = SYSTEM
410 SYSTEM

```

Pick Basic

```

PRINT, PRINTER ON, PRINTER OFF
PRINTER ON
PRINT X$ : Y$ : Z$ :           * No Carriage Return
PRINT X$ : Y$ : Z$           * With Carriage Return
PRINTER OFF

```

The PRINTER ON statement causes the output of the next PRINT command to be sent to the printer rather than to the screen. PRINTER OFF directs output back to the screen.

The colon concatenates together different values and variables to be printed. An ending colon suppresses the Carriage Return—Line Feed. To send a Carriage Return—Line Feed, use PRINT alone:

PRINT

If commas are used as separators, the elements will be printed at pre-set tab positions beginning with zero for the left-hand column and increasing by 18 positions across the page.

PRINT ON File. Number

For example:

```
PRINT ON 1 X$ : Y$ : Z$
```

PRINT ON will cause output to be sent to a particular numbered print file. The File.Number can be a variable or a literal.

PRINTER CLOSE

For example:

PRINTER CLOSE

PRINTER CLOSE will cause the data to be printed immediately, rather than for the spooler to wait for the termination of the process.

Pick TCL

LIST-LABEL, SORT-LABEL

For example:

LIST – LABEL Filename {Item.IDs} {Conditions} {Options}

SORT – LABEL Filename {Item.IDs} {Conditions} {By Attribs} {(Options)}

In Pick, the LIST-LABEL and SORT-LABEL commands make label-printing programs unnecessary. After the command is issued, Pick gives a question mark prompt. The user must then provide the following parameters, on a single line, each separated by a comma:

- 1) Number of labels across each page
- 2) Number of lines on each label
- 3) Number of lines to skip between labels
- 4) Left margin indentation
- 5) Width of each label, in columns
- 6) Number of columns to skip between labels across a page
- 7) The “C” option will suppress spaces for null attributes

Page headings should be suppressed with the “(H)” for “Heading-Suppress” option, for continuous printing down pages of labels. The “(P)” for “Printer” option sends output to the printer rather than to the screen.

If the left margin indentation is not zero, Pick will prompt for line headings, requesting a heading for each line of the labels. To suppress line headings, give a left margin of zero, or press ENTER without typing anything in response to the line heading prompts.

The next example would print labels from the data in a file of photographs, showing the accession number and folder heading of each photo. The labels will be printed in the order of item IDs in the file.

SORT-LABEL Photo.File Folder.Heading Accession.Number (HP)

? 3,4,1,0,20,2,C

The syntax and choices available for the LIST-LABEL and SORT-LABEL commands are the same as those of the other access commands LIST and SORT. (For details on use of the access commands for searching, see the section: "Files, Items: Input: Search with Pick Access Processor.")

PRINTER: ACCESS SYSTEM INFORMATION, CHANGE SETTINGS

These commands can be used to obtain information about the printer currently in use and to change settings on the printer.

PICK/BASIC

For example:

```
SYSTEM( )
Page.Length      = SYSTEM (3) + 1
Page.Width       = SYSTEM (2) + 1
Lines.Remaining  = SYSTEM (4)      * Used with HEADING or FOOTING
Page.Num         = SYSTEM (5)      * " " " " " "
Line.Number      = SYSTEM (6)      * " " " " " "
```

From within a PICK/BASIC program, the SYSTEM function will return the information shown above, when the printer rather than screen has been designated for output with the PRINTER ON statement. (For more details on the SYSTEM function, see the section: "System: Access System Information.")

PICK TCL

TERM, SET-TERM

For example:

```
TERM
TERM Screen.Width, Screen.Height, Line.Skip, Line.Feed.Delay,
    Form.Feed.Delay, Backspace.Char, Print.Page.Width,
    Print.Page.Length, Terminal.Type
```

The TERM command, with no parameters, will display the current terminal and printer characteristics indicated above.

The TERM or SET-TERM command, with parameters, will change those characteristics. The parameters are separated by commas. A comma alone will suffice as a place holder for any parameter that is to remain the same.

SP-ASSIGN { options }

For example:

```
SP-ASSIGN ?
```

The SP-ASSIGN (for Spooler-Assign) verb is used to set the characteristics of the spooler. With a question mark, the command will return the current settings.

The “M” (Message) option, which is not documented in the Pick manual, will suppress display of the spooler message “ENTRY #” that normally appears when material is sent to the spooler:

```
SP-ASSIGN M
```

The “M” option can be added to any other options used. For example, the “M” option might be combined with the “I” (Immediate) option, which sends the output to the printer immediately, rather than waiting for the end of the process:

```
SP-ASSIGN MI
```

PRINTER: FORMAT MASKS

This command is used to format entries for printing by placing them in format masks.

PICK/BASIC

For example:

```
PRINT Variable$ Format.Mask$
```

The colon, used to concatenate elements for printing, does not appear between the variable to be printed and the mask.

The same procedure can be used for screen formatting, but direct cursor control with the “CRT @(Column, Row)” statement is easier and more precise.

The following characters are available to fill the mask field:

```
# = Space  
* = Asterisk  
% = Zero
```

A format mask can be placed in a variable or used as a literal. If any element of the format mask is repeated, a number can be used rather than showing the repetition. Thus a seven digit number can be displayed in telephone number format with any of the following commands:

```
PRINT Phone.Num "(###-####)"
```

or

```
Phone.Format$ = "(###-####)"  
PRINT Phone.Num Phone.Format$
```

or

```
Phone.Format$ = "(3#-4#)"  
PRINT Phone.Num Phone.Format$
```

Note carefully that there is no concatenating colon between the variable and the format mask.

A justification parameter can appear at the beginning of the mask. Thus to left-justify the variable Title\$ within a field of 40 blanks:

```
PRINT Title$ "L(#40)"
```

The length parameter of the mask can be a variable rather than a literal. The following routine will print four numbers across the page, each 20 spaces apart:

```
Length = 20
Mask.20$ = "L(#" : Length : ")"
N1 = 1; N2 = 2; N3 = 3; N4 = 4
PRINTER ON
PRINT N1 Mask.20$ : N2 Mask.20$ : N3 Mask.20$ : N4
PRINTER OFF
```

Although the Pick manual declares that any character can be used as a field fill, in practice a character other than the standard space, asterisk, or zero covers up the entry, rather than only masking it in a field of fill characters. In the TEST.FORMAT.MASK program the IBM extended character ASCII 219, a reverse space, is designated as the fill character. But when displayed on the IBM PC/XT, only the fill character remains, overwriting the entry.

TEST.FORMAT.MASK Program

```
* TEST.FORMAT.MASK David L. Clark 11/03/1987
* Put format mask into a variable
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TFormat.bp
  EQU Blank$ TO CHAR(219)
  Length = 20
  Mask.Blank$ = "L(" : Blank$ : Length : ")"
  Mask.Space$ = "L(#" : Length : ")"
  N1 = 1; N2 = 2; N3 = 3; N4 = 4
  PRINT N1 Mask.Blank$ : N2 Mask.Blank$ : N3 Mask.Blank$ : N4
  PRINT N1 Mask.Space$ : N2 Mask.Space$ : N3 Mask.Space$ : N4
  END
```

PRINTER: PAGING, HEADINGS, AND FOOTINGS

PICK/BASIC

PAGE, HEADING, FOOTING

For example:

```
* Assign setting for report ...
HEADING "Security Pacific Photo Collection 'CT' Page 'PLL'"
* Loop to print report ...
PAGE
```


The **HEADING** statement places a heading at the top of a page of output each time that the **PAGE** command is given, whether output is to the screen or to a printer. The **FOOTING** statement operates in the same manner as **HEADING**, except that the material is placed at the bottom of the page. The entire heading or footing is enclosed in double quotes, with options enclosed in single quotes. The example shown would place the heading "Security Pacific Photo Collection" at the top of the page. The heading would be centered on the line (C option), it would include the time and date (T option), the word "Page," the page number (P option), and be followed by two Carriage Returns—Line Feeds (L option used twice).

The following options are available for printing:

- ☐ C Center
- ☐ D Date in format DD MMM YYYY
- ☐ L Line Feed, Carriage Return
- ☐ P Page number, right justified in a field of four blanks
- ☐ T Time and Date in format HH:MM:SS DD MMM YYYY

The disadvantage of using the **HEADING** and **PAGE** commands is that **PICK/BASIC** will always throw a blank page first before printing. That procedure might be desirable in a large data processing shop in which the printer is kept in a separate room. Throwing the first page will ensure that different reports are not mixed together. However, the standard Pick method will not be considered acceptable by those accustomed to a single-user, IBM PC DOS environment. It will be difficult to sell Pick as a system that is more powerful and flexible than **dBASE** and **R:BASE**, while telling the potential customer that there is no way to prevent wasting a page with every printout.

The **TEST.HEADING** program provides a method of preventing a program from throwing the first page, without requiring that the program take total control over the paging process.

The following tests were made in the course of arriving at the final procedure used. If different results are obtained on a system other than the IBM PC/XT, then the tests should be repeated:

When the heading and footing were defined in the normal way, before the first item was printed, then a blank page was thrown before printing.

When the heading was not defined at the time of the first **PAGE** command, then the second page printed with no heading.

When the heading was defined at the time of the first **PAGE** command, then no footing was printed on the first page, and an extra page followed with heading and footing but no item.

When the program handled the printing of a heading on the first and second pages, then no blank pages were thrown, and every page had both a heading and a footing. However, the first page and the second page each must be given special handling. The main advantage of using the **PAGE** statement, rather than using total program control, is that the program does not have to count lines nor position the footing on the last line of the page.

TEST.HEADING Program

```
* TEST.HEADING David L. Clark 11/17/1987
* Test Heading and Footing commands.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: THeading.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  * Assign Constants
  EQU True TO 1, False TO 0
  Page.1.Flag = True
  PRINTER ON
  Page.Width = SYSTEM(2)
  Prt.Head$ = "This is the heading Page "
  Prt.Head$ = SPACE((Page.Width - LEN(Prt.Head$)) / 2) : Prt.Head$
  Prt.Foot$ = "'LC'" : "This is the footing" : "'L'"; *1 up from bottom
  Prt.Item$ = "This is the Item to print"
  * Imitate a program loop to print a report. The program would not
  * know the number of Items in advance.
  FOR I = 1 TO 3
    * Special handling for 1st 2 pages
    IF Page.1.Flag THEN
      PRINT Prt.Head$ : 1 "R(#4)";          * Add page number
      FOOTING Prt.Foot$
      Page.1.Flag = False
      Page.2.Flag = True;                  * For the next pass
    END ELSE
      IF Page.2.Flag THEN
        PAGE 2;
        PRINT Prt.Head$ : 2 "R(#4)";        * Set correct page number
        HEADING Prt.Head$ : "'P'";          * Add page number
        Page.2.Flag = False                * Add page number
      END ELSE
        PAGE
      END
    END
    * Print the Item
    PRINT Prt.Item$
  NEXT I
  * End program
END
```

Program Running and Compiling



IBM PC BASIC

The IBM PC BASIC language reviewed here is interpreted rather than compiled, so no compiling operation is necessary. Although a compiled version is available, a program sufficiently large and complex to require compilation for adequate speed should be compiled instead in the very similar but also more powerful and more structured language, Microsoft QuickBASIC. The IBM PC BASIC programs presented in this guide will compile in QuickBASIC.

IBM PC Advanced BASIC is contained in the file "BASICA.COM" on the IBM PC DOS system diskette. To use IBM PC BASIC, copy the BASICA.COM file onto the disk and directory desired.

To create a special directory of BASIC programming, use the MD (Make Directory) command, at the IBM PC DOS operating system level, from the root directory:

```
MD \ Basic
```

Then copy the BASICA.COM file from the diskette in the A drive to the BASIC directory on the hard disk in the C drive:

```
COPY A: BASICA.COM C: \ Basic
```

BASICA, LOAD, RUN, SAVE, SYSTEM

To enter Advanced BASIC programming environment from the IBM PC DOS operating system level, in the directory that contains BASICA.COM, enter the command:

BASICA

The "OK" prompt will appear to indicate that the BASIC programming environment has been entered. You can type in the source code of a program while in BASIC, but it will be much easier, and also encourage more structured programming and more complete documentation, to type the program with a good IBM PC DOS word processor and then **LOAD** the program into IBM PC BASIC.

To load a program that already has been written, enter the command:

LOAD "ProgName"

To run the program after loading:

RUN

To load and run simultaneously:

RUN "ProgName"

The program name must correspond to IBM PC DOS file name conventions. The name must not be longer than eight characters, nor have an extension longer than three characters. Within the BASIC environment, if no extension is specified, the extension ".bas" will be applied automatically.

To save a program that has been typed or changed while in BASIC:

SAVE "ProgName"

The program will be saved in BASIC condensed format. To save the program in ASCII format, for editing with a word processor, append a comma and an "A" for ASCII to the **SAVE** command:

SAVE "ProgName", A

Note the display of function key labels at the bottom of IBM PC BASIC's screen. The **LOAD**, **RUN**, and **SAVE** commands all can be entered with function keys.

To exit from the BASIC programming environment enter the command:

SYSTEM

The **SYSTEM** command can be used within a program, to exit from the program directly to the DOS operating system level, rather than into BASIC.

The Basic.bat batch file shown here will allow you to enter the BASIC environment from the root directory, and then return to the root directory after exiting BASIC.

Basic.bat Batch File

```
ECHO OFF
REM Basic.bat David L. Clark 04/23/1988
REM Change to Basic directory, enter BASICA, return to Root directory
REM IBM PC DOS Batch file
CD \Basic
BASICA
CD \
```

A BASIC program can be executed directly from the DOS operating system level with the command BASICA followed by the name of the program:

```
BASICA ProgName
```

The Run.bat Batch file shown here will allow the user to enter the command:

```
Run ProgName
```

at the IBM PC DOS operating system level in order to run a BASIC program. Note the difference from the PICK/BASIC RUN command, which would require a filename before the program name.

The IBM PC BASIC program should end with the SYSTEM command, to return control to the operating system level.

Run.bat Batch File

```
ECHO OFF
REM Run.bat David L. Clark 04/23/1988
REM Change to Basic directory, run BASICA program, return to Root
CD \Basic
BASICA %1
CD \
```

To make the process still easier for the user, a separate batch file could be created for each IBM PC BASIC program, allowing the user to enter simply the program name in order to run a program. This usage would correspond to the procedure in PICK/BASIC for running a cataloged program, for which only a program name is required. If the user will be employing both Pick and DOS, the operations required should be made to appear identical whenever possible. As an example, the Example.bat Batch file shown here will run the Example.bas program, and then return the user to the root directory.

Example.bat Batch File

```
ECHO OFF
REM Example.bat David L. Clark 04/23/1988
REM Change to Basic directory, run Example program, return to Root
CD \Basic
```

BASICA Example

CD \

If a Menu.bat batch file has been created to display a menu of options at the IBM PC DOS operating system level, then a final line would be added to the file containing the batch file command "Menu." (For more information on batch files, see the section: "Batch Files and Procs.")

PICK TCL

BASIC

For example:

BASIC Filename Program.Name {(Options)}

The BASIC command to compile a program is issued at TCL level. Pick offers the following options for program compilation with the BASIC statement:

- ☐ **A** Output a listing of the object code in Assembly format.
- ☐ **C** Cut out all end-of-line marks. One byte per line will be saved, but error messages will ascribe all errors to line 1.
- ☐ **M** Create a map of the compiled program.
- ☐ **P** Send the output listing to the printer.
- ☐ **S** Suppress the symbol table. The table is used for discovering the values held by variables during debugging. Doing without the table cuts down the code size by about 25 percent.
- ☐ **X** Create a cross-reference table for the variables, to identify typing mistakes in variable names and to spot seldom-used global variables that could be made into local variables. See next example.

While still testing the program, compile with the "(X)" option to create a cross-reference table. A file named "BSYM" must first be created to hold the table. A modulo of 1 will be adequate for this file:

CREATE-FILE BSYM 1 1

(For creating files, see the section: "Files, Items: Open, Close.")

After the compile has finished, the BSYM will contain an item for every variable and statement label. The first attribute of the item will contain all of the line numbers on which the variable or label appears. Create the following two attribute definition items in the dictionary level of the BSYM file. (Do not include the material in braces.):

Item ID: Line.Number

01 A (Identifies Item as an Attribute Definition)

02	1	(Attribute Mark Count)
09	R	(Right justification for a number)
10	10	(Display width)

Item ID: 1		
01	A	(Identifies Item as an Attribute Definition)
02	1	(Identifies Item as an Attribute Definition)
03	Line.Number	(Name for display)
09	R	(Right justification for a number)
10	10	(Display width)

The definition item with an ID of “1” provides a default attribute list, which will be displayed automatically with the LIST or SORT commands. (For attribute definition items, see the beginning of the section: “Files, Items.” For default attribute lists, see the section: “Files, Items: Input: Search with the Pick Access Processor”.)

At Pick TCL level, display the cross-reference table in order by label number and variable name with the command:

`SORT BSYM`

Display the cross table in order by line number, and secondarily by label number and variable name, with the command:

`SORT BSYM BY-EXP Line.Number`

An asterisk will be included with the numbers of those lines on which the variable is assigned a value. Use the cross-reference table to eliminate variables that are never used or are used only within a single routine, and thus could be made into local rather than global variables. (For drawing a distinction between local and global variables, see the section: “Size, Speed of Program.”)

The “(A)” option for Assembly listing would provide a comparison for the relative efficiency of different methods of performing an operation. To be useful, the listing should be saved for further reference by combining the printer option with the Assembly option, or by using the SCREEN.TO.FILE program shown in the section “Screen: Redirect Output” to save the Assembly listing as an item.

Pick previous to open architecture limits the size of a single item to 32K. However, a pointer-list can have a size of 64K. To avoid the 32K limit on the size of the source code for a program, file the code as a list. First change attribute 1 of the file definition item for the data level of the file from “D” to “DC”. This item is contained in the dictionary level of the file, with an item ID that is the same as the filename.

At TCL type:

`ED DICT File.Program File.Program`

The Editor will respond with:

`TOP`

Type:

R/D/DC/
FI

Then store the source code for the program with the Pick Editor using “FIL” to file a list rather than “FI” to file an item.

At TCL type:

ED File.Program Item.Program

The Editor responds:

TOP

Type:

FIL

RUN

For example:

RUN Filename Program.Name

To run a PICK/BASIC program, issue the RUN command at Pick TCL level, including the name of the file in which the program is stored as an item, and the item ID of the program.

CATALOG

For example:

CATALOG Filename Program.Name

To avoid having to remember and enter the filename, CATALOG the program at Pick TCL. Pick will respond that the program has been cataloged. Then simply type the program name at Pick TCL to run the program.

DECATALOG

For example:

DECATALOG Filename Program.Name

The DECATALOG command will reverse the CATALOG process.

Repetition



THE “FOR . . . NEXT” STRUCTURE, TO REPEAT A LOOP A SET NUMBER OF TIMES, IS part of all of the languages treated here. The different implementations of the common structure have certain similarities and differences that are best understood by examining them together.

The following routine is presented in the syntax of IBM PC BASIC and PICK/BASIC:

```
FOR I = Start TO End  
  Array(I) = I  
Next I
```

The example uses the variable “I” as an index counter, and assigns to each element of the fixed numeric array “Array()” the value of the counter. Assigning elements of a fixed array is a common usage of FOR...NEXT.

The expression “I = Start” shown in the example is evaluated only once. The expressions “TO End” and “NEXT I” must be evaluated at each repetition of the loop, hence it is undesirable to require extra computation in the end statement, such as:

```
For I = Start TO A + B
```

Instead use:

```
End = A + B  
FOR I = Start TO End
```

The NEXT statement causes the counter to be increased and for program control to branch back to the beginning of the loop.

The loop ends when the counter exceeds the end value, not when the counter equals the end value. Thus if the starting and ending values are equal, the loop will still take place once, as in the statement:

```
FOR I = 1 TO 1
```

Loops can be nested, but they cannot cross each other's boundaries. One loop must be nested entirely within another, for example:

```
FOR I = 1 TO 10
  FOR J = 1 TO 10
    PRINT Array(I,J)
  NEXT J
NEXT I
```

The routine would print 100 different elements from the two-dimensional fixed array.

The "FOR...NEXT" structure will perform differently in different languages according to whether it tests the index value at the beginning of the loop, or tests at the end, after all the actions in the loop have been performed. If FOR...NEXT does not test until the end, it will execute at least once even if the index exceeds the final value.

Languages and implementations also differ in regard to whether the counter or index variable can be changed in its value within the loop, causing the loop to end prematurely, and whether the variable has a value after the loop has terminated, or is considered undefined. If the counter can be changed, or has a value after the loop terminates, care must be taken for unintended side effects.

IBM PC BASIC

FOR...NEXT

For example:

```
FOR I = Start TO End (STEP Step.Size) : Array(I) = I : NEXT {I}
```

It is not necessary to repeat the index variable "I" shown in the example "NEXT" statement. The loop will execute faster without the final "I."

Step.Size can be negative, to move from a larger to a smaller index value:

```
FOR I = 10 TO 1 STEP -1
  PRINT Array$(I)
NEXT
```

IBM PC BASIC performs evaluation for FOR...NEXT at the top of the loop, therefore there is no execution if start is greater than end. Microsoft BASIC on the Radio Shack 100 performs evaluation for FOR...NEXT at the end of the loop, therefore the loop executes at least once even if start is greater than end.

The following TForNext (Test FOR...NEXT) program tests whether an implementation of FOR...NEXT tests the index value at the start or end of the loop.

The program also demonstrates that in IBM PC BASIC (and in similar versions of Microsoft BASIC), you can change the value of the index variable or counter within the loop, and that this variable has a value after the loop has terminated. The variable will be equal to the end value specified in the FOR statement, plus one step.

TForNext Program, Documentation Version

```

1  '* TForNext.bsd: Test For...Next. David L. Clark 07/05/1987
2  '* Test where counter evaluated, if can be changed, and value at end.
3  '* IBM PC Basic program. Documentation version.
4  '* IBM PC BASICA 3.0, Radio Shack 100
5  '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10 DEFINT A-Z
20  '* Begin line
    FOR I = 1 TO 0
    :      PRINT "END OF LOOP"
    :      NEXT I
    :      PRINT "If the phrase END OF LOOP appears,"
    :      PRINT "then this FOR...NEXT test was "
    :      PRINT "made at the end of the loop"
    :      PRINT
    '* End line
30  PRINT "Counter =";
40  '* Begin line
    FOR I = 1 TO 10
    :      PRINT I;
    :      IF I = 5 THEN
    :          I = 8
    :          PRINT
    :          PRINT " Counter changed from 5 to " : I : ", "
    :          '* End: If I
    :          End: For I - on next line
    '* End line
50  NEXT
60  PRINT
    :      PRINT "After Loop completed, Counter ="; I
    '* End line
9999 END

```

You can break out of a FOR...NEXT loop by resetting the index variable, thereby simulating the WHILE or UNTIL constructions of Pick, QuickBASIC, Pascal, and other languages. The result is faster than a comparable "IF...GOTO Line.Number," and avoids an extra line number reference. Double NEXT statements for a single FOR are allowed, as long as only one NEXT per loop is encountered:

```

10 '* Begin line
    FOR Index = Start TO End
    :      IF Data.Array$(Index) = "" THEN
    :          Index = End          '* Terminate early at first null
    :          NEXT
    :      ELSE
    :          PRINT #1, Data.Array$(Index)
    :          NEXT
    :          '* End: If Data.Array$(Index) = ""
    '* End: For...Next Index
    '* End line

```

The loop will execute faster if the entire loop is on one line and as compact as possible, hence the value of using Boolean flags in place of “IF...THEN” conditions. The inclusion of a conditional clause in the loop will require ending the line, otherwise the loop will not reach the NEXT statement when the condition is untrue. If an “IF...THEN” condition is necessary, NEXT can be repeated. (To avoid the “IF...THEN” construction, see the section: “Boolean Flags: Used in Place of IF...THEN.”)

In the following example, the elements of string array Y\$() are passed to array X\$(), with the condition that only those elements that fit within the designated Max.Length (for maximum length) are allowed. If Y\$(I) is too long for the allowed Max.Length, then X\$(I) is made null rather than truncating the string:

```
10 '* Begin line
    FOR I = Start TO End
      : IF LEN(Y$(I)) <= Max.Length THEN
          X$(I) = Y$(I)
      : NEXT
    ELSE
      X$(I) = ""
    : NEXT
    '* End: If LEN(Y$(I)) ...
    '* End: For...Next I
  '* End line
```

The IF...THEN test has broken the continuous logic of the line. Only the flow of logic that agrees with the ELSE condition will continue. The flow that agrees with the IF...THEN condition will drop down to the next line. To avoid breaking the logic flow:

```
10 '* Begin line
    FOR I = Start TO End
      : Test.Flag = LEN(Y$(I)) <= Max.Length
      : Num.Chars = Max.Length * - Test.Flag
      : '* Num.Chars will = Max.Length or zero
      : X$(I) = LEFT$(Y$(I), Num.Chars)
      : NEXT
      '* Additional statements on the same line ...
    '* End line
```

or:

```
10 '* Begin line
    FOR I = Start TO End
      : X$(I) = LEFT$(Y$, (Max.Length * -(LEN(Y$(I)) <= Max.Length)))
      : NEXT
      '* Additional statements on the same line ...
    '* End line
```

An index variable could be changed with a Boolean operation when reading to the end of the file:

```
FOR I = 1 TO 1000
: LINE INPUT #1, Line.From.File$
: PRINT #2, Line.From.File$
: I = I + 1000 * - EOF(1)
: NEXT
```

When the EOF (end-of-file) condition is true, the index variable "I" will be set to a figure more than 1000, thus terminating the loop. (For input from a file, see the section: "Files, Items: Input.")

WHILE...WEND

For Example:

```
WHILE Test.Flag
    . . .
WEND
```

IBM PC BASIC provides an equivalent for the LOOP...WHILE construction of PICK/BASIC. Note that the ending statement is "WEND," not "END."

Simpler versions of Microsoft BASIC do not include the WHILE...WEND construction.

PICK/BASIC

FOR...NEXT

For example:

```
FOR I = Start TO End {STEP Step.Size}
    Array(I) = I
NEXT I
```

Step.Size can be negative, to move from a larger to a smaller index value:

```
FOR I = 10 TO STEP - 1
    PRINT Array$(I)
NEXT I
```

PICK/BASIC makes its FOR...NEXT evaluation at the top of the loop, therefore there is no execution if start is greater than end.

The following TEST.FOR.NEXT program tests whether an implementation of FOR...NEXT tests the index value at the start or end of the loop.

The program also demonstrates that in PICK/BASIC, you can change the value of the index variable or counter within the loop, and that this variable has a value after the loop has terminated. The variable will be equal to the end value specified in the FOR statement, plus one step.

TEST.FOR.NEXT Program

```
* TEST.FOR.NEXT David L. Clark 07/06/1987
* Test where counter evaluated, if can be changed, and value at end.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TForNext.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
    FOR I = 1 TO 0
```

```

        CRT "END OF LOOP"
NEXT I
CRT "If the phrase END OF LOOP appears, then this"
CRT "FOR...NEXT test was made at the end of the loop"
CRT
CRT "Counter = " :
FOR I = 1 TO 10
    CRT I : " " :
    IF I = 5 THEN
        I = 8
        CRT
        CRT "Counter changed from 5 to " : I
    END
NEXT I
CRT
CRT "After Loop completed, Counter = " : I
END

```

A FOR...NEXT loop will execute faster if the entire loop is placed on one line. However, the inclusion of a conditional clause in the loop will require ending the line, otherwise the loop will not reach the NEXT or REPEAT statement when the condition is untrue. A possible alternative to avoid breaking the line would be to use a Boolean statement to make an assignment. (For conditional clauses, see the section: "Conditions." For the use of Boolean statements, see the section: "Boolean Flags: In place of IF...THEN," and the examples with IBM PC BASIC within this section.)

FOR...UNTIL...NEXT, FOR...WHILE...NEXT

You can stop the loop before reaching end with UNTIL or WHILE:

```

FOR I = Start TO End {STEP Step.Size} UNTIL Array$(I) = ""
NEXT I

```

or:

```

FOR I = Start TO End {STEP Step.Size} WHILE Array$(I) # ""
NEXT I

```

In both examples, the loop will act as a FOR...NEXT construction, unless it encounters a null value in the array, which will cause the loop to end.

LOOP...WHILE...DO...REPEAT, LOOP...UNTIL...DO...REPEAT

Pick LOOP constructions permit eliminating the FOR...NEXT;

```

I = 0
LOOP
    I = I + 1
WHILE I < J DO
    CRT I
REPEAT

I = 0
LOOP

```

```

      I = I + 1
UNTIL I >= J DO
  CRT I
REPEAT

```

In each case in these examples, if J equals 5, the values 1 through 4 will be displayed for I. Pick's syntax provides two separate places in the loop where statements can be executed. Statements are not required at either location.

The statements, if any, immediately following "LOOP" will always be executed at least the first time. Therefore you should place those statements that must be executed at least once in the first location, after "LOOP," and place the statements that might never be executed in the second location, following the WHILE or UNTIL clause. Thus to display each character in the string Entry\$, without displaying a null for an empty string:

```

Length.Entry = LEN(Entry$)
Pos.in.Entry = 0
LOOP
  Pos.in.Entry = Pos.in.Entry + 1
WHILE Pos.in.Entry <= Length.Entry DO
  CRT Entry$ [Pos.in.Entry, 1] :
REPEAT

```

The WHILE clause could have been written instead as:

```

UNTIL Pos.in.Entry > Length.Entry

```

with the same result.

The use of end-of-statement punctuation can become somewhat confusing with conditional structures such as IF...THEN...ELSE or LOOP. The following TEST.LOOP program gives 14 examples of the use of Carriages Returns and semicolons to end statements.

All of the tests below the line of dashes required additional semicolons to be accepted by the compiler.

The keywords for the LOOP structure are LOOP, UNTIL...DO or WHILE...DO, and REPEAT. In all the tests above the dashes, a keyword ended the line. It did not matter if the keyword was alone on the line or preceded by commands. For example, in Test #7, the UNTIL...DO clause ended the line, preceded by multiple commands.

In the tests below the dashes, additional semicolons were required because one or more commands followed a keyword on the same line, and the line did not end with a keyword.

TEST.LOOP Program

```

* TEST.LOOP David L. Clark 08/04/1987
* Test accepted syntax for LOOP.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TestLoop.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
EQU True TO 1
* Each keyword or command alone on a separate line

```

```

LOOP
    CRT "Test #1"
UNTIL True DO
    CRT "Never"
REPEAT
* LOOP, UNTIL, REPEAT each alone on line
LOOP
    CRT "Test #2"; I = 1
UNTIL True DO
    CRT "Never"; J = 1
REPEAT
* All in 1 line, ended by REPEAT
LOOP CRT "Test #3" UNTIL True DO CRT "Never" REPEAT
* Line starts with Label, ends with REPEAT
LOOP CRT "Test #4" UNTIL True DO CRT "Never" REPEAT
* All in 1 line, multiple commands in 1st clause, ends with REPEAT
LOOP CRT "Test #5"; I = 1 UNTIL True DO CRT "Never" REPEAT
* All in 1 line, multiple commands, ends with REPEAT
LOOP CRT "Test #6"; I = 1 UNTIL True DO J = 1; CRT "Never" REPEAT
* LOOP and UNTIL on same line, ended by UNTIL...DO
LOOP CRT "Test #7"; I = 1 UNTIL True DO
    J = 1; CRT "Never"
REPEAT
* All in 1 line, ended by REPEAT. Same as Test #3 above, except that
*   Test #3 included no semicolons. The semicolons are not required,
*   but they are allowed when in doubt.
LOOP; CRT "Test #8"; UNTIL True DO; CRT "Never"; REPEAT
*-----*
* The statements below this point required extra semicolons.
*-----*
* Command ends line after LOOP. UNTIL starts a new line.
LOOP; CRT "Test #9"
    I = 1
UNTIL True DO
    J = 1; CRT "Never"
REPEAT
* Two commands end line after LOOP. UNTIL starts a new line.
LOOP; CRT "Test #10"; I = 1
UNTIL True DO
    J = 1; CRT "Never"
REPEAT
* Command ends line after UNTIL. REPEAT starts a new line.
LOOP; CRT "Test #11"; I = 1; UNTIL True DO; J = 1; CRT "Never"
REPEAT
* Commands end lines after LOOP, UNTIL.
LOOP; CRT "Test #12"; I = 1
UNTIL True DO; J = 1; CRT "Never"
REPEAT
* Compare Tests #13, #14:
*   Two Commands end line after LOOP.
*   UNTIL starts a new line, ended by REPEAT.
*   On the UNTIL...DO...REPEAT line, extra semicolons are not
*   required after DO and before REPEAT. However, if one extra
*   semicolon is included, then both must be included.
LOOP; CRT "Test #13"; I = 1
UNTIL True DO J = 1; CRT "Never" REPEAT
LOOP; CRT "Test #14"; I = 1
UNTIL True DO; J = 1; CRT "Never"; REPEAT
END

```


Screen



SCREEN COMMANDS ARE USED TO SEND OUTPUT TO THE TERMINAL SCREEN.

IBM PC BASIC

PRINT

For example:

```
PRINT X$; Y$; Z$;    '* No Carriage Return
PRINT X$; Y$; Z$     '* With Carriage Return
```

An ending semicolon suppresses the Carriage Return—Line Feed. To send an extra Carriage Return—Line Feed, use “PRINT” alone:

```
PRINT
```

You also could put a Carriage Return—Line Feed in a variable, rather than issuing a new **PRINT** statement.

```
CRLF$ = CHR$(13) + CHR$(10)
PRINT X$; CRLF$; CRLF$; Y$
```

If commas are used as separators rather than semicolons, the elements will be displayed at preset tab positions beginning with zero for the left-hand column and increasing by 14 positions across the screen.

Positive numbers print with a blank in front. (For more details see the section: “Numbers: Convert: Number to String.”)

TAB()

For example:

```
PRINT TAB(Column - 1); U; X$; . . .
```

Since PRINT TAB(Number) actually skips Number spaces and starts printing at column Number + 1, set the TAB for one less than the column where printing should begin. TAB also comes up short on very long lines. It will not allow the total of the spaces to skip and the characters to print to exceed 255. If the print position is already greater than the column number, TAB has no effect.

LOCATE

For example:

```
LOCATE Row, Col  
PRINT U; V; X$; Y$; . . .
```

Use LOCATE to place the cursor at a position on the screen, prior to display. (For more details see the section: “Screen: Cursor: Place the Cursor.”)

Microsoft BASIC (Radio Shack 100)

PRINT @

For example:

```
PRINT @ Pos.Screen, U; V; X$; Y$;
```

The major interpreted BASIC variant to the LOCATE statement for placing the cursor is the “PRINT @” construction. (For more details see the section: “Screen: Cursor: Place the Cursor.”)

PICK/BASIC

CRT, PRINT

For example:

```
CRT U : V : X$ : Y$ :      * No Carriage Return  
CRT U : V : X$ : Y$      * With Carriage Return
```

The colon concatenates together different values and variables to be displayed. An ending colon suppresses the Carriage Return—Line Feed. To send just a Carriage Return—Line Feed, use CRT or PRINT alone:

```
CRT
```

If commas are used as separators, the elements will be displayed at preset tab positions beginning with zero for the left-hand column and increasing by 18 positions across the screen.

Pick allows either CRT or PRINT for screen display. However, PRINT also is used for sending material to the printer, after a "PRINTER ON" command. For clarity, CRT will be used in this guide for all Pick screen display commands.

SYSTEM()

For example:

```
Printer.On.Flag = SYSTEM(1)
```

The SYSTEM(1) function will show whether the screen or printer has currently been designated for output. The assigned flag will hold "1" if the printer is currently designated for output from the PRINT statement, and zero if the screen is designated.

CRT @ (), PRINT @()

For example:

```
Clear.Screen$ = @(-1)  
CRT Clear.Screen$
```

With regard to the compatibility of Pick implementations between different terminals, note that the Pick "@ ()" expressions presented in the sections below should be placed in variables assigned with "=" rather than constants assigned with the EQUATE statement. As variables, they will be adapted to the type of terminal in use at the time. As constants, they would not be adapted.

Pick TCL

ECHO or P

For example:

```
ECHO  
P
```

The "ECHO" or "P" command toggles output to the screen off and on. When either command is first issued at Pick TCL level, output to the screen is suppressed. If a second "ECHO" or "P" command is given, output to the screen is restored.

The "P" command is not documented in the Pick manual, and is mentioned here only to explain what will occur if the user accidentally enters "P" at the terminal. Unfortunately, that error is quite common, because the "P" command in the Pick Editor is used to display the next page of information. A user will exit from the Editor, hit "P" without thinking, and then not understand why the commands that he types do not appear on the screen.

SCREEN: ACCESS SYSTEM INFORMATION

This section shows how to obtain information about the characteristics of the screen currently in use. Screen characteristics vary greatly from one machine to another. If a program is to present a uniform appearance on different machines, the program must obtain information from the system as to the screen characteristics, and then adapt the screen output to the features available.

To discover which keystrokes return which ASCII numbers, and which ASCII numbers produce what characters on the screen, see the section: "Characters." The extended ASCII characters, beyond the normal letters, numbers, and punctuation of ASCII 32 through 126, are different on different machines.

IBM PC

The screen height of the IBM PC is 25 rows, the screen width is 80 columns.

Radio Shack 100

The Radio Shack 100 uses an LCD screen. The screen height is 8 rows, the screen width is 40 columns, the only available color is black and white.

PICK/BASIC

SYSTEM()

For example:

Col.Max	= SYSTEM(2)	* Counted from 0. Maximum column.
Screen.Width	= SYSTEM(2) + 1	* Counted from 1
Row.Bottom	= SYSTEM(3)	* Counted from 0
Screen.Height	= SYSTEM(3) + 1	* Counted from 1
Rows.Remaining	= SYSTEM(4)	* Used with HEADING or FOOTING
Page.Num	= SYSTEM(5)	* " " " " "
Row.Number	= SYSTEM(6)	* " " " " "
Terminal.Type\$	= SYSTEM(7)	

Pick can make use of many different kinds of terminals, which will have different characteristics. From within a PICK/BASIC program, the SYSTEM function will return the information shown above. (For more details on the SYSTEM function, see the section: "System: Access System Information.")

Pick TCL

TERM, SET-TERM

For example:

TERM
TERM Screen.Width, Screen.Height, Line.Skip, Line.Feed.Delay,
Form.Feed.Delay, Backspace.Char, Print.Page.Width,
Print.Page.Length, Terminal.Type

The TERM command, with no parameters, will display the current terminal and printer characteristics indicated above.

The TERM or SET-TERM command, with parameters, will change those characteristics. The parameters are separated by commas. A comma alone will suffice as a placeholder for any parameter that is to remain the same. Because all of the parameters are numbers, with the exception of the terminal type, which is a letter, a letter will be accepted as a terminal type designation without preceding commas.

SCREEN: ASSIGN SCREEN CONSTANTS

In PICK/BASIC, in many other forms of BASIC (but not IBM PC BASIC), and for most video display terminals, a screen position or a video characteristic such as reverse or high intensity can be defined as a string constant or variable. This string then can be concatenated with other strings, including text messages, additional video characteristics, and screen positions also defined as strings, and sound. The strings can be combined to define a total message or even a total screen for display. For example:

```
Message$ = @(Col.Row) : Video.Reverse$ : Bell$ : Error.Text$  
Message$ = Message$ : Undo.Video.Reverse$  
CRT Message$
```

The advantage of the method of defining video characteristics and screen positions as strings is that it avoids issuing a separate command each time that a screen position or characteristic must be changed. The method of concatenating screen constants is generally faster and gives a better appearance than issuing a series of separate display statements.

A screen characteristic such as Video.Reverse\$ will remain on until it is specifically undone with Undo.Video.Reverse\$. Therefore if a screen constant includes the former, it should end with the latter.

There is considerable variation among language versions and machines in regard to screen display. The most widely applicable method should be used to define each screen constant or variable. For example, when defining a constant as Clear.Screen\$ for use with PICK/BASIC, as shown in the screen constants tables, you should use the PICK/BASIC definition rather than separate definitions for each type of terminal. The @() functions are supplied by PICK/BASIC to avoid having to alter a program for each type of machine.

The DEFINE-TERMINAL program supplied by Pick, and demonstrated in the section "Screen: Assign Terminals for Pick," will allow the use of standard Pick definitions for video characteristics, provided that both Pick and the terminal employed support the given characteristic. To test whether a character or code under Pick has the effect desired, see the section: "Screen: Video Characteristics" for the programs TEST.VIDEO, which displays a message with different video settings, and TEST.VIDEO.FUNCTIONS, which allows the user to enter a character or code and observe the results. The operations of the "@()" functions always should be verified for each new machine and Pick release. Do not trust the manuals.

Two screen constants tables are presented here, each with the same list of constants. The first table shows which of these constants can be defined using PICK/BASIC "@()" functions, and the variations among Pick versions. The second table shows how the constants can be defined on each machine, for PICK/BASIC, IBM PC BASIC, and the Radio

Shack Model 100. If a PICK/BASIC function is available, use it rather than the machine-specific control characters and escape codes.

Screen Constants: Functions for Versions of Pick Table

"@ ()" Functions for Versions of Pick

	Pick R83	Pick R83 2.0 PC-XT	Pick R83 2.2 PC-AT, 386	Pick Open Arch 2.0
Backspace\$	= @(-9)	@(-9)	@(-9)	@(-9)
Bell\$	=			
Carriage.Return\$	=			
Clear.Screen\$	= @(-1)	@(-1)	@(-1)	@(-1)
Clear.Whole.Line\$	=			
Cursor.Off\$	=			
Cursor.On\$	=			
Delete.Current.Char\$	=			
Delete.Current.Line\$	=			
Embedded.Attrib.Flag	=		@(-99)	@(-99)
Erase.Line.from.Pos\$	= @(-4)	@(-4)	@(-4)	@(-4)
Erase.Screen.from.Pos\$	= @(-3)	@(-3)	@(-3)	@(-3)
Insert.Blank.Line\$	=			
Insert.Char\$	=			
Keys.Reset.to.Default\$	=			
Pos.Down.w.Scroll\$	=		@(-101)	@(-101)
{Line Feed}				
Pos.Down.No.Scroll\$	=			
Pos.Home\$	= @(-2)	@(-2)	@(-2)	@(-2)
Pos.Lt.to.Start.Line\$	=			
Pos.Rt.to.End.Line\$	=			
Pos.Rt.to.Next.Line\$	=		@(-100)	@(-100)
Pos.Screen\$	= @ (Col, Row)	@ (Col, Row)	@ (Col, Row)	@ (Col, Row)
Pos.Screen.Col\$	= @ (Col)	@ (Col)	@ (Col)	@ (Col)
Pos.Up.w.Scroll\$	= @(-10)	@(-10)	@(-10)	@(-10)
Pos.Up.No.Scroll\$	=			
Protect.Disable\$	=	@(-12)	@(-12)	
Protect.Enable\$	=	@(-11)	@(-11)	
Protect.Start\$	= @(-7)	@(-7)	@(-7)	@(-7)
Protect.Stop\$	= @(-8)	@(-8)	@(-8)	@(-8)
Screen.Lock\$	=			@(-11)
Undo.Screen.Lock\$	=			@(-12)
Space.Char\$	=			
Status.Line.Erase\$	=		@(-28)	@(-28)
Status.Line.Lock\$	=			
Undo.Status.Line.Lock\$	=			
Status.Line.Off\$	=			
Status.Line.On\$	=			
Status.Line.Write\$	=		@(-27)	@(-27)
Tab\$	=			
Video.Blinking\$	= @(-5)	@(-5)	@(-5)	@(-5)
Undo.Video.Blinking\$	= @(-6)	@(-6)	@(-6)	@(-6)
Video.Intensity.High\$	=	@(-8)		@(-17)
Video.Intensity.High\$	=		@(-101)	
(Not on memory monitor)				
Video.Intensity.Low\$	=	@(-7)		@(-18)

"@ ()" Functions for Versions of Pick

	Pick R83	Pick R83 2.0 PC-XT	Pick R83 2.2 PC-AT, 386	Pick Open Arch 2.0
Video.Intensity.Low\$ = (Not on memory monitor)			@(-100)	
Video.Normal\$ =				
Video.Reverse\$ =		@(-13)	@(-13)	@(-13)
Video.Reverse.Low\$ =				
Undo.Video.Reverse\$ =		@(-14)	@(-14)	@(-14)
Video.Underline\$ =		@(-15)	@(-15)	@(-15)
Undo.Video.Underline\$ =		@(-16)	@(-16)	@(-16)

The definitions shown in the next table of control characters and escape codes assume the prior assignment of the constant Esc\$, representing the Escape character, to ASCII 27:

Esc\$ = CHAR(27) ;	* PICK/BASIC
Esc\$ = CHR\$(27) ;	* IBM PC, Radio Shack 100

Screen Constants: Codes for Different Machines Table

	IBM PC IBM Basic	IBM PC Pick Basic	Wyse 60	Radio Shack 100
Backspace\$	= CHR\$(8)	CHAR(8)	CHAR(8)	CHR\$(8)
Bell\$	= CHR\$(7)	CHAR(7)	CHAR(7)	CHR\$(7)
Carriage.Return\$	= CHR\$(13)	CHAR(13)	CHAR(13)	CHR\$(13)
Clear.Screen\$	= CHR\$(12)	CHAR(12)	CHAR(26)	CHR\$(12)
""			Esc\$: ";"	Esc\$+"E"
Clear.Whole.Line\$	=			Esc\$+"l"
Cursor.Off\$	=		Esc\$: "`0"	Esc\$+"Q"
Cursor.On\$	=		Esc\$: "`1"	Esc\$+"P"
Delete.Current.Char\$	=		Esc\$: "w"	
Delete.Current.Line\$	=		Esc\$: "R"	Esc\$+"M"
Embedded.Attrib.Flag	=			
Erase.Line.from.Pos\$	=		Esc\$: "T"	Esc\$+"K"
Erase.Screen.from.Pos\$	=		Esc\$: "Y"	Esc\$+"J"
Insert.Blank.Line\$	=		Esc\$: "E"	Esc\$+"L"
Insert.Char\$	=		Esc\$: "Q"	
Keys.Reset.to.Default\$	=		Esc\$: "cU"	
Pos.Down.w.Scroll\$	= CHR\$(10)	CHAR(10)	CHAR(10)	CHR\$(10)
{Line Feed}				
Pos.Down.No.Scroll\$	= CHR\$(31)			Esc\$+"B"
Pos.Home\$	= CHR\$(11)		CHAR(30)	CHR\$(11)
""			Esc\$: "("	Esc\$+"H"
Pos.Lt.to.Start.Line\$	= CHR\$(29)			Esc\$+"D"
Pos.Rt.to.End.Line\$	= CHR\$(28)			Esc\$+"C"
Pos.Rt.to.Next.Line\$	= CHR\$(28)			
Pos.Screen\$	=		CHAR(12)	
			Esc\$: "a"	Esc\$+"Y"
			:Row: "R"	+CHR\$(Row+32)
			:Col: "C"	+CHR\$(Col+32)
Pos.Screen.Col\$	=		CHAR(13)	
			:STR(CHAR	
			(12),Col)	
			CHAR(11)	
Pos.Up.w.Scroll\$	=			
Pos.Up.No.Scroll\$	= CHR\$(30)			Esc\$+"A"
Protect.Disable\$	=		Esc\$: "''"	

	IBM PC IBM Basic	IBM PC Pick Basic	Wyse 60	Radio Shack 100
Protect.Enable\$	=		Esc\$: "&"	
Protect.Start\$	=		Esc\$: ")"	
Protect.Stop\$	=		Esc\$: "("	
Screen.Lock\$	=		Esc\$: "N"	Esc\$+"V"
Undo.Screen.Lock\$	=		Esc\$: "O"	Esc\$+"W"
Space.Char\$	= CHR\$(32)	CHAR(32)	CHAR(32)	CHR\$(32)
Status.Line.Erase\$	=			
Status.Line.Lock\$	=			Esc\$+"T"
Undo.Status.Line.Lock\$	=			Esc\$+"U"
Status.Line.Off\$	=		Esc\$: "`c"	
Status.Line.On\$	=		Esc\$: "`b"	
Status.Line.Write\$	=			
Tab\$	= CHR\$(9)	CHAR(9)	CHAR(9)	CHR\$(9)
Video.Blinking\$	=		Esc\$: "G2"	
Undo.Video.Blinking\$	=		Esc\$: "G0"	
Video.Intensity.High\$	=		Esc\$: "G0"	
Video.Intensity.Low\$	=		Esc\$: "Gp"	
Video.Normal\$	=		Esc\$: "G0"	
Video.Reverse\$	=		Esc\$: "G4"	Esc\$+"p"
Video.Reverse.Low\$	=		Esc\$: "Gt"	
Undo.Video.Reverse\$	=		Esc\$: "G0"	Esc\$+"q"
Video.Underline\$	=		Esc\$: "G8"	
Undo.Video.Underline\$	=		Esc\$: "G0"	

Graphics characters also differ among machines, and there are no PICK/BASIC definitions for the graphic characters. The graphics characters table presents the character for a video reverse blank, which would most often be used to indicate the space available in a data entry field, and the characters that a program would utilize for drawing boxes.

The IBM PC Extended ASCII double-line characters give a much more attractive appearance than single-line characters or pluses and dashes. Note the difference in appearance of these data entry forms:

DATA ENTRY FORM
NAME:
ADDRESS:
PHONE-NUMBER:

DATA ENTRY FORM
NAME:
ADDRESS:
PHONE-NUMBER:

Graphics Characters Table

Graphics Characters

IBM PC Extended ASCII			Radio Shack 100		No Graphics	
Blank.Reverse\$	=	CHAR(219)		CHR\$(239)		Video.Reverse\$: " "
Cross\$	=	CHAR(206) ‡		CHR\$(250) †		:Undo.Video.Reverse\$
Horiz.Bar\$	=	CHAR(205) =		CHR\$(241) -		CHAR(43) +
Left.Down\$	=	CHAR(201) ⌋		CHR\$(240) [		CHAR(45) -
Left.Up\$	=	CHAR(200) ⌌		CHR\$(246) [		CHAR(43) +
Mid.Down\$	=	CHAR(203) ⌋		CHR\$(243)]		CHAR(43) +
Mid.Left\$	=	CHAR(185) ⌋		CHR\$(249)]		CHAR(43) +
Mid.Right\$	=	CHAR(204) ⌋		CHR\$(244)]		CHAR(43) +
Mid.Up\$	=	CHAR(202) ⌋		CHR\$(248)]		CHAR(43) +
Right.Down\$	=	CHAR(187) ⌋		CHR\$(242)]		CHAR(43) +
Right.Up\$	=	CHAR(188) ⌋		CHR\$(247)]		CHAR(43) +
Vert.Bar\$	=	CHAR(186)		CHR\$(245)		CHAR(124)

The Wyse 60 will display the IBM PC Extended ASCII set of graphics characters, but the Wyse 30 and Wyse 50 terminals will not. Wyse provides its own method of display for graphic characters through Escape sequences. The following Escape sequences will display the box-drawing and other characters on Wyse terminals.

Graphics for Wyse Terminals without IBM Extended ASCII Table

Graphic	Wyse Terminal Code	
Blank.Reverse\$	=	Escape\$: "H7"
Cross\$	=	Escape\$: "H8" †
Horiz.Bar\$	=	Escape\$: "H:" -
Left.Down\$	=	Escape\$: "H2" [
Left.Up\$	=	Escape\$: "H1" [
Mid.Down\$	=	Escape\$: "H0"]
Mid.Left\$	=	Escape\$: "H9"]
Mid.Right\$	=	Escape\$: "H4"]
Mid.Up\$	=	Escape\$: "H="]
Right.Down\$	=	Escape\$: "H3"]
Right.Up\$	=	Escape\$: "H5"]
Vert.Bar\$	=	Escape\$: "H6"

Some video characteristics work in combination. Under PICK/BASIC on the Wyse 60, Video.Normal\$ will undo all of the special characteristics such as reverse, underline, and blinking. The Wyse 60 also offers a single escape code for reverse video with low intensity, while PICK/BASIC does not. In PICK/BASIC the combined characteristic would be created by concatenation:

Video.Reverse.Low\$ = Video.Reverse\$: Video.Intensity.Low\$

The most widely used combined screen constant is Carriage Return—Line Feed:

```
CRLF$ = CHAR(13) : CHAR(10) ;      * Pick BASIC
CRLF$ = CHR$(13) + CHR$(10)      /* IBM PC BASIC
```

On the IBM PC/XT under Pick release 1983 2.0, the @(-7) function is defined for both Video.Intensity.Low\$ and for Protect.Start\$. If Protect.Enable\$ has been activated, the function accomplishes both. If Protect.Enable\$ is not activated, only low intensity results. (For more information see the section: “Screen Output: Protected.”)

Store Message Routine

In the “Store Message” routine shown here, Message.Error\$ is stored as a single constant, including its position for display. The display can be called up as a single constant by any routine, and concatenated with other similar screen constants.

```
*****
* Store Message *
*****
* David L. Clark 09/22/1988
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: StoreMsg.bpr
* Store screen positions and characteristics as constants.
* Include with the |Assign Constants| section of program
*****
* Video, Sound Constants *
*****
Bell$      = CHAR(7)
Error.Text$ = " ERROR "
Row.Bottom = SYSTEM(3)
Col.Max    = SYSTEM(2)
Undo.Video.Reverse$ = @(-14)
Video.Reverse$ = @(-13)
*****
* Set Position *
*****
Col.Center = (Col.Max - LEN(Error.Text$)) / 2
Pos.Message$ = @(Col.Center, Row.Bottom)
*****
* Assign Message *
*****
Message$ = Pos.Message$ : Bell$ : Video.Reverse$ : Error.Text$
Message$ = Message$ : Undo.Video.Reverse$
```

Microsoft BASIC (Radio Shack 100)

The following version of the “Store Message” routine demonstrates the use of the same methods with other forms of BASIC as were employed with PICK/BASIC. Many other forms of BASIC and many terminals use escape sequences for positioning. The example shown is for the Radio Shack Model 100, which employs escape plus “Y”, followed by the row and column.

Store Message Routine

```
'*****
'* Store.Message *
'*****
'* Store screen positions and characteristics as constants.
'* Use Escape sequence to store a screen position as a constant.
'* Include with the |Assign Constants| section of program
'*****
'* Video, Sound Constants *
'*****
'* Row, Col counted from 0.
Bell$           = CHR$(7)
Col.Max         = 39
Error.Text$     = " ERROR "
Escape$        = CHR$(27)
Pos.Code$      = Escape$ + "Y"      '* Row, Column to be added
Row.Bottom     = 7
Undo.Video.Reverse$ = Escape$ + "q"
Video.Reverse$  = Escape$ + "p"
'*****
'* Set Position *
'*****
Col.Center = (Col.Max - LEN(Error.Text$)) \ 2
Pos.Message$ = CHR$(Row.Bottom + 32) + CHR$(Column.Center + 32)
Pos.Message$ = Pos.Code$ + Pos.Message$
'*****
'* Assign Message *
'*****
Message$ = Pos.Message$ + Bell$ + Video.Reverse$ + Error.Text$
Message$ = Message$ + Undo.Video.Reverse$
```

SCREEN: ASSIGN TERMINAL

Pick

The Pick multiuser operating system allows the use of many different types of terminals, as well as the use of stand-alone computers as terminals. If the definition for a particular type of terminal is already on file, that definition can be called up, otherwise you can create a new definition.

TERM, SET-TERM

For example:

```
TERM Screen.Width, Screen.Height, Line.Skip, Line.Feed.Delay,
Form.Feed.Delay, Backspace.Char, Print.Page.Width,
Print.Page.Length, Terminal.Type
```

The TERM or SET-TERM command will change the screen parameters. The parameters are separated by commas. A comma alone will suffice as a place holder for any parameter that is to remain the same. Because all of the parameters are numbers, with the exception of the terminal type, which is a letter, a letter will be accepted as a terminal type designation without preceding commas.

In regard to the screen dimensions, note that the count starts from zero, not one. Thus a screen size of 79,24 will display 25 lines of 80 columns. To reduce the screen height to 24 lines, in order to allow for a terminal status line at the bottom, the following command would suffice:

```
TERM ,23
```

To change only the terminal type, to “W” for a Wyse-brand terminal, without changing any other parameters:

```
TERM W
```

TERM-TYPE

For example:

```
TERM-TYPE
```

The verb TERM-TYPE (for terminal-type) will call up a default definition stored for a particular serial port. The TERM-TYPE command is often included in the Logon PROC for each account, so that the terminal will be ready immediately for the user. The procedure is demonstrated in the example “Logon PROC to Set Terminal.”

Logon PROC to Set Terminal

```
Item ID: Account.Name
01  PQ
02  C Logon Proc to Set Terminal Type for This Account
03  HTERM-TYPE
04  P
```

(For more information on Logon PROCS, see the section: “System: Booting, Start-up.”)

The terminal port (or line) definition called up by TERM-TYPE is stored in an item in the ACC file, in the SYSPROG account. (ACC stands for accounting, SYSPROG for system programmer.) The item ID for the definition is the port number, written as two digits (under open architecture, three digits). The first attribute in the item is a description of the port. The second attribute contains the parameters for a TERM command, in the same format as those shown with the TERM command explanation.

The example of “Port Definitions” demonstrates default port definitions for an IBM PC/XT with two serial ports (also known as asynchronous communication ports). The definitions are stored as items in the dictionary level of the ACC file of the SYSPROG account.

The first definition, for port zero, is for the IBM PC/XT itself. Port 01 is set for a Wyse terminal. Because the Wyse brand of terminal is widely used, the Pick system provides a standard definition for that type of terminal, designated here as type “W.”

Port 02 is set for a very different sort of machine: a stand-alone computer, normally employed in a single-user mode, now placed into service as a terminal on the multiuser

Pick system. The example given is for the Radio Shack 100. Note the screen size of 40 by 8, rather than 80 by 25. The terminal type "Z" is a new type that has been defined by the user with the DEFINE-TERMINAL verb, as will be demonstrated shortly.

Port Definition Items in ACC File

```
Item ID: 00
01  Memory Mapped Monitor
02  79,24,0,2,2,8,80,59,I
```

```
Item ID: 01
01  Async Comm Port #1
02  79,24,0,2,2,8,80,59,W
```

```
Item ID: 02
01  Async Comm Port #2
02  39,7,0,2,2,8,80,59,Z
```

The default printer page length of 60 has been changed to 59 (last numerical parameter in attribute 2). Laser printers, which cannot print on the entire page, will throw an extra page for each one printed with a page length of 60.

As an alternative to the use of default port definitions with TERM-TYPE, you could ask the user, as in the CHOOSE.TERMINAL program. The program matches each terminal type with a name that will be understood by the user.

The program reads the names and types defined on the system from the CURSOR file in the SYSPROG account. (Under open architecture the file is named TERMS.) The program assumes the following format for the item ID and first two attributes for items in the file:

```
Item ID: Terminal.Name
01  Terminal.Type
02  Screen.Width, Screen.Height
```

To check the format of the items, at Pick TCL use the command:

```
SORT CURSOR *A1 *A2
```

or:

```
SORT TERMS *A1 *A2
```

Although there should be no more than one definition per alphabet letter in the file, more than one actually can exist. Should such duplication appear in the display of choices, change the terminal-type letter of one of the definitions with the DEFINE-TERMINAL procedure.

Although the screen size is included in the terminal definition, the dimensions are not applied unless they are included explicitly in the TERM command.

The CHOOSE.TERMINAL program and the Pick-supplied DEFINE-TERMINAL command, both assume access to the CURSOR file in which the terminal definitions are held. Therefore the program and the command either must be executed from the SYSPROG account, or a file synonym must be created pointing to that account. Such a synonym, also called a Q-pointer, would reside in the MD of the current account, and have the following form:

```
Item ID: CURSOR
01 Q
02 SYSPROG      {Original Account name}
03 CURSOR       {Original file name}
```

(For more information on file synonyms, see the section: "Files, Items: Synonyms.")

As an alternative, rather than reading the CURSOR file and displaying all of the available choices, the program could read items in a special file containing a much more limited and therefore less confusing list.

CHOOSE.TERMINAL Program

```
* CHOOSE.TERMINAL David L. Clark 04/15/1988
* Ask user for terminal type.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ChosTerm.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Match the terminal type with a name that will be understood by
* the user. Read the names and types defined on the system from
* the CURSOR file in the SYSPROG Account. (Under Open Architecture the
* file is named TERMS).
*****
* Assign Constants *
*****
* Do not assign any screen constants until after terminal is defined.
PROMPT ""
EQU False TO 0
File.Terms.Name$ = "CURSOR";      * Open Arch: = "TERMS"
EQU Terms.Max TO 26;              * 1 per letter of alphabet
DIM Term.Name$(26);              * Terminal name understood by user
DIM Term.Size$(26);              * Screen dimensions: Width, Height
DIM Term.Type$(26);              * Single letter used by TERM
* Set to null
MAT Term.Name$ = ""
MAT Term.Size$ = ""
MAT Term.Type$ = ""
EQU True TO 1
*****
*****
* Fill Matching Terminal Arrays *
*****
OPEN File.Terms.Name$ TO File.Terms$ ELSE
  CRT " Cannot find file: " : File.Terms.Name$
  GOTO 9999;                      *|End Program|
END
* Get Item IDs starting in alphabetical order
```

```

Command$ = \SSELECT \ : File.Terms.Name$
EXECUTE Command$ CAPTURING Capture.Output$
End.of.Terms.Flag = False
Terms.Total = Terms.Max
FOR I = 1 TO Terms.Max UNTIL End.of.Terms.Flag
  READNEXT Term.Name$(I) THEN
    READ Term.Item$ FROM File.Terms$, Term.Name$(I) THEN
      Term.Type$(I) = Term.Item$<1>
      Term.Size$(I) = Term.Item$<2>
    END ELSE
      Term.Type$(I) = "Type not defined"
      Term.Size$(I) = "80,24";      * Most common size
    END
  END ELSE
    End.of.Terms.Flag = True
    Terms.Total = I - 1;          * Total less than max allowed
  END
NEXT I
*****
* Choose from List *
*****
*****
* Display List of Choices *
*****
* No screen constants or cursor placement defined before terminal.
Space.30$ = SPACE(30)
CRT; CRT; CRT; CRT
CRT Space.30$ : "TERMINAL TYPES"
* Assume minimum screen height of 24 rows, minus 2 for heading
* and prompt.
IF Terms.Total < 22 THEN;          * 1 column
  FOR I = 1 TO Terms.Total
    CRT Space.30$ : I : ". " : Term.Type$(I) : ": " : Term.Name$(I)
  NEXT I
END ELSE;                          * 2 columns
  FOR I = 1 TO 13
    * Column 1 display
    Col.1.Dsp$ = I : ". " : Term.Type$(I) : ": " : Term.Name$(I)
    CRT Col.1.Dsp$ :
    * Column 2 display
    Col.2.Index = I + 13;          * Array index for column 2
    IF Terms.Total >= Col.2.Index THEN
      Col.2.Dsp$ = SPACE(40 - LEN(Col.1.Dsp$)) : Col.2.Index
      Col.2.Dsp$ = Col.2.Dsp$ : ". " : Term.Type$(Col.2.Index)
      Col.2.Dsp$ = Col.2.Dsp$ : ": " : Term.Name$(Col.2.Index)
      CRT Col.2.Dsp$
    END ELSE
      CRT
    END
  NEXT I
END
*****
* Pick a Number *
*****
LOOP
  CRT Space.30$ : "Pick a Number [1]: " :
  INPUT Response$
  IF Response$ = "" THEN

```

```

        Num.Choice = 1;                * Default
        Test.Flag = True
    END ELSE
        * Must test if numerical before testing size of number
        IF NUM(Response$) THEN
            Num.Choice = Response$
            IF Num.Choice > 0 AND Num.Choice <= Terms.Total THEN
                Test.Flag = True
            END ELSE
                Test.Flag = False
            END
        END ELSE
            Test.Flag = False
        END
    END
END
UNTIL Test.Flag DO
REPEAT
*****
* Set Terminal Type *
*****
* Screen height and width in terminal definition counted from 1.
*   For TERM command, count from zero. So must subtract 1.
Screen.Width = FIELD(Term.Size$(Num.Choice), ",", 1) - 1
Screen.Height = FIELD(Term.Size$(Num.Choice), ",", 2) - 1
Command$ = \TERM \ : Screen.Width : \, \ : Screen.Height : \, \
Command$ = Command$ : Term.Type$(Num.Choice)
EXECUTE Command$
*=====
*****
9999 * End Program *
*****
END

```

DEFINE-TERMINAL

For example:

```
DEFINE-TERMINAL
```

The DEFINE-TERMINAL command can be used to create definitions for terminals not already defined on the system. The DEFINE-TERMINAL procedure also will allow stand-alone computers, intended for single-user employment, to be put into service as live terminals connected to the Pick multiuser operating system.

The terminal definitions are stored in the CURSOR file in the SYSPROG account. Do not try to change terminal characteristics by modifying the CURSOR file with the editor or by other means. Changes must be made through the DEFINE-TERMINAL procedure. The CURSOR data file is actually a compiled form of the CURSOR, SOURCE file.

The section "Screen: Assign Screen Constants" includes a list of escape codes and control characters that could be employed on the Wyse 60 terminal and the Radio Shack 100 computer to set screen characteristics such as reverse video and to position the cursor. Many of the same codes can be used in the DEFINE-TERMINAL procedure, so that video characteristics can be set or the cursor positioned with standard PICK/BASIC statements such as "CRT @(Column, Row)" or "CRT @(-13)." In this manner new types of terminals can be employed without writing separate routines for each terminal.

To begin the procedure, at Pick TCL give the command:

DEFINE-TERMINAL

The system will take half a minute to round up all of the terminal definitions available, and then will display DEFINE-TERMINAL's main menu.

DEFINE-TERMINAL Menu

System Cursor Definition Utility

The following terminals are defined. Terminals marked with an asterisk (*) are selected to be included in your system cursor definition.

*A ADDS580	*F TV910	*M AMPEX.D80	*W WYSE50
*D DATAMEDIA	*I IBMPC	*R REGENT	*Y WYSE.60
*E ESPRIT6310	*J VT100	*T TV950	*Z RADIO.SHACK.100

- 1) Create Terminal Definition
- 2) Modify Terminal Definition
- 3) Delete Terminal Definition
- 4) Add Terminal to Selected Definitions
- 5) Delete Terminal from Selected Definitions
- EX Exit without updating System-Cursor
- FI Update System-Cursor to selected terminals

Enter Selection (1-5) or EX or FI:

To change or review a terminal choose "Modify Terminal Definition." The procedure then requests the name (not the one-letter code) of the terminal to be modified.

After the name is entered, the modify terminal screen is presented with the named terminal's characteristics. Only one terminal is displayed at a time. For this example three terminals have been shown together. The IBM PC definition was supplied by the Pick system, the WYSE.60 and RADIO.SHACK.100 definitions were created with the DEFINE-TERMINAL procedure. To create a new terminal definition, fill in the categories with the appropriate escape codes for the terminal.

Modify Terminal Screen

Terminal =	IBMPC	WYSE.60	RADIO SHACK.100
1. TYPE.....	I	Y	Z
2. SCREEN SIZE.....	80,25	132,24	40,08
3. CURSOR ADDRESS CODE.....	H	D+	L
4. @(X) CURSOR POSITIONING...	ESC "=" X DEL	CR STR(CHAR(12) ,X) BS	CR STR(ESC "C" ,X)
5. @(X,Y) CURSOR ADDRESSING..	ESC "=" X Y	ESC "a" Y "R" X "C"	ESC "Y" Y X
6. @(-1) CLEAR SCREEN & HOME.	ESC "*" CHAR(1)	CHAR(26)	ESC "E"
7. @(-2) CURSOR HOME.....	ESC "*" CHAR(2)	CHAR(30)	ESC "H"
8. @(-3) CLEAR TO END OF PAGE	ESC "*" CHAR(3)	ESC "Y"	ESC "J"
9. @(-4) CLEAR TO END OF LINE	ESC "*" CHAR(4)	ESC "T"	ESC "K"

Terminal %	IBMPC =	WYSE.60	RADIO SHACK.100
10. @(-5) START BLINK.....	ESC "*" CHAR(5)	ESC "G2"	NUL
11. @(-6) STOP BLINK.....	ESC "*" CHAR(6)	ESC "G0"	NUL
12. @(-7) START PROTECT.....	ESC "*" CHAR(7)	ESC ")"	NUL
13. @(-8) STOP PROTECT.....	ESC "*" CHAR(8)	ESC "("	NUL
14. @(-9) CURSOR BACK.....	ESC "*" CHAR(9)	ESC BS	ESC "D"
15. @(-10) CURSOR UP.....	ESC "*" CHAR(10)	ESC VT	ESC "A"
16. @(-11) ENABLE PROTECT MODE	ESC "*" CHAR(11)	ESC "&"	NUL
17. @(-12) DISABLE PROTECT MODE	ESC "*" CHAR(12)	ESC "'"	NUL
18. @(-13) START REVERSE VIDEO	ESC "*" CHAR(13)	ESC "G4"	ESC "p"
19. @(-14) STOP REVERSE VIDEO.	ESC "*" CHAR(14)	ESC "G0"	ESC "q"
20. @(-15) START UNDERLINE....	ESC "*" CHAR(15)	ESC "G8"	ESC "p"
21. @(-16) STOP UNDERLINE.....	ESC "*" CHAR(16)	ESC "G0"	ESC "q"
22. @(-17) SLAVE ON.....	ESC "*" CHAR(17)	ESC CHAR(24)	
23. @(-18) SLAVE OFF.....	ESC "*" CHAR(18)	ESC CHAR(20)	
24. @(-19) CURSOR FORWARD.....		FF	ESC "C"
25. @(-20) CURSOR DOWN.....		LF	ESC "B"
26. @(-99) EMBEDDED VISUAL ATTRIBUTES?		NO	

The list of choices is divided between two screens. After each screen is displayed, the DEFINE-TERMINAL procedure prompts:

Modify lines?

When the answer to the DEFINE-TERMINAL program's "Modify lines?" question is finally "NO," the program asks if the table is correct. If the answer is "YES" the procedure tries to compile the table. If there is an error, an error message appears. Otherwise, the procedure returns to the first screen. To store the new configuration, enter "FI" for file item.

If a function is not supported by the particular Pick version employed and so cannot be implemented through the DEFINE-TERMINAL procedure, you could set the function under DEFINE-TERMINAL to whatever video characteristic came closest. Thus in the example responses shown for DEFINE-TERMINAL, because the Radio Shack 100 does not support underlining, video reverse has been used as synonymous with underlining. In the case of the blinking function, which the Radio Shack 100 also does not support, that characteristic has simply been left null.

Rather than assigning a near alternative, you could create code, or perhaps definitions stored in a parameters file, for the video characteristics supported by different types of hardware. (For more information, see the section: "Screen: Assign Screen Constants.")

Column-Only Positioning

Line number four in the DEFINE-TERMINAL display shown earlier defines the "@(X) CURSOR POSITIONING" code to position the cursor, with only the column given. Not all terminals directly support this feature. If column-only positioning is not supported by a terminal, the feature can be simulated in two ways: by performing a Carriage Return to get to the left end of the row, followed by a number of cursor-right moves equal to the column designation, or by storing the previous row reference in the "Z" parameter, and then using "Z" as the row designator. In many cases the "Z" parameter

method will not work, so the cursor-right moves must be employed instead. In the examples shown, for the Wyse 60, CHAR(12) performs a cursor-right move; for the Radio Shack 100, the escape character plus "C" performs a cursor-right.

The column-only positioning instruction supplied by Pick Systems with version R83 2.0 for the Wyse terminal is incorrect. A BS (backspace) must be added, to compensate for the difference between Wyse and Pick column counting. Pick counts the first column on the left side of the screen as zero, Wyse counts it as one. This difference was noted in the cursor address code, where the "+" in "D+" signifies that column counting begins with one. As a result, Pick adds one when it sends column positions to the Wyse terminal. But for column-only cursor addressing, the cursor is not being placed by sending a column position, but by making a series of cursor-right moves. There will be one move too many, because Pick has added one to the requested cursor designation. Hence the need for a correcting backspace. Minor glitches of this type are the most aggravating aspect of giving a multiuser system like Pick an appearance that will match the smooth user interface of single-user systems.

To test if the cursor-only positioning function is performing correctly with a terminal, run the TEST.SCREEN.COL program shown here. Try the "Z" parameter method of column-only positioning. If it does not work, use a series of cursor-right moves instead.

TEST.SCREEN.COL Program

```
* TEST.SCREEN.COL David L. Clark 08/22/1988
* Test terminal's capability for column-only cursor placement with @(Col).
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TScrnCol.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* If the function works, a series of asterisks will appear on row 5.
* If the function does not work, the first asterisk will appear on row 5,
* the rest will appear on the top row.
  * Assign Constants
  PROMPT ""
  Clear.Screen$ = @(-1)
  * Show Column positioning
  CRT Clear.Screen$ : "On Row 5, stars at columns 0, 10, 20, 30" :
  CRT @(0, 5) : "*" : @(10) : "*" : @(20) : "*" : @(30) : "*" :
  * End Program
END
```

SCREEN: ASSIGN TERMINAL PARAMETERS

PICK

The settings for some video characteristics differ under the various versions of Pick. For example, on the IBM PC/XT under Pick release 1983 2.0 video intensity high and low are set by "@(-8)" and "@(-7)" and under open architecture by "@(-17)" and "@(-18)." A program intended for different versions of Pick will have to account for such differences.

This section includes detailed examples of the use of a parameters file to store video parameters for use by different terminals. A simple version of the procedure could be employed just to account for the different implementations of Pick.

In this first example, each item in the file `TERMINAL.PARAMETERS` represents a different version of Pick, and each attribute in an item represents a particular video characteristic, such as underlining, reverse video, or low intensity. The low intensity attribute for the IBM PC/XT release 1983 would contain "-7." For open architecture the same attribute would hold "-18." If these attributes were the only two per item, the items would contain the following:

Item ID	IBM.PC.XT.R83	Open.Arch
01	-7	-18
02	-8	-17

Note the method used to combine a variable with the "@()" function. The variable name is placed inside the function:

```
Video.Low.Intensity.Param = -7
Video.Low.Intensity.Func$ = @(Video.Low.Intensity.Param)
```

The routine "Assign Screen Constants for Pick Versions" will read the appropriate item in the `TERMINAL.PARAMETERS` file and then assign screen constants according to the parameters.

Assign Screen Constants for Pick Versions Routine

```
*****
* Assign Screen Constants for Pick Versions *
*****
* David L. Clark 09/22/1988
* Adjust video characteristics for different Pick implementations.
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: AsScPkVr.bpr
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
File.Term.Params.Name$ = "TERMINAL.PARAMETERS"
Term.ID$ = "IBM.PC.XT.R83"; * This could be taken from file or user
DIM Term.Param$(2)
EQU Total.Term.Params TO 2; * Dimension for array
EQU Video.Intensity.Low$ TO Term.Param$(1)
EQU Video.Intensity.High$ TO Term.Param$(2)
MAT Term.Param$ = ""
* Read from Parameters file
OPEN File.Term.Params.Name$ TO File.Term.Params$ ELSE STOP
MATREAD Term.Param$ FROM File.Term.Params$, Term.ID$ ELSE STOP
* Set Parameters
FOR I = 1 TO Total.Term.Params
* Note method of applying "@()" with variable inside
Term.Param$(I) = @(Term.Param$(I))
NEXT I
```

If a function is not supported by the particular Pick version and so cannot be implemented through the `DEFINE-TERMINAL` procedure, the function still can be employed if it can be expressed as an escape code or as a character, but at the cost of customized

assignment definitions for that function. For example, Pick does not directly support cursor off to turn off the blinking cursor. Many terminals do offer this feature, however. To employ cursor off on a Pick system, the assign constants section of a program could interrogate the terminal type definition and assign the function for those terminals that supported it, while creating either an empty constant or some sort of equivalent for those terminals that did not.

In the example "Assign Cursor Off" routine, the cursor off function is assigned for the Wyse 60 terminal and the Radio Shack Model 100, each of which supports that function. Because Pick does not support the function, as an alternative the cursor will be sent to position home (the upper left corner of the screen), so that at least the cursor will be out of the way. You might also set the cursor to low intensity. Another example of an alternative actions is to substitute reverse video for underlining for a terminal such as the Radio Shack 100 that will display in reverse but that will not underline.

Assign Cursor Off Routine

```
*****
* Assign Cursor Off *
*****
* David L. Clark 09/22/1988
* Cursor Off function supported by Wyse 60, Radio Shack 100, not Pick
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: CursrOff.bpr
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Terminal type definitions are created with the DEFINE-TERMINAL
* program supplied by Pick Systems.
Escape$ = CHAR(27)
Pos.Home$ = @(-2)
Term.Type$ = SYSTEM(7)
EQU True TO 1
BEGIN CASE
    CASE Term.Type$ = "W"; * User has applied this type for Wyse 60
        Cursor.Off$ = Escape$ : "`0"
        Cursor.On$ = Escape$ : "`1"
    CASE Term.Type$ = "Z"; * User created this for Radio Shack 100
        Cursor.Off$ = Escape$ : "V"
        Cursor.On$ = Escape$ : "W"
    CASE True; * Any other type of terminal
        Cursor.Off$ = Pos.Home$
        Cursor.On$ = ""
END CASE
```

Terminal functions that require a command, rather than an escape code or a control character, cannot be assigned if the command is not a valid PICK/BASIC statement. Thus the IBM PC BASIC line-drawing commands cannot be duplicated in Pick.

Rather than creating special code for each piece of hardware, terminal functions not supported by Pick could be read from a parameters file, using the terminal name as the item ID and placing each function in a particular attribute. In the file `TERMINAL.PARAMETERS` you would store an item for each type of terminal to be used. Every item would

have the same format, with each attribute containing the instructions for a particular video characteristic.

Most video characteristics are defined with control characters that cannot be displayed on the screen in the same manner as normal letters and numbers. The most commonly used is the escape character, ASCII 27. Control characters could be stored with several alternative methods. You could store the characters directly, by writing CHAR(27) and other control characters to the item, as long as the only interaction with the item was to take place through programming. If you intended to change the item with the Pick Editor or use verbs that would cause the item to be displayed on the screen or sent to a printer, then the program should store the ASCII numbers rather than the actual characters.

In this example, each item in the file TERMINAL.PARAMETERS represents a particular video characteristic, such as cursor off and cursor on. For those terminals that support the feature, the attributes contain the control codes. For those terminals that do not support the feature, the attributes are null.

The name "STANDARD" is used as a catch-all designation for a terminal that supports only the standard Pick features. The STANDARD item will hold alternative actions, such as moving the cursor to position home when the cursor cannot be turned off. Pick "@()" functions such as Pos.Home\$ will appear as negative numbers.

In each attribute the decimal ASCII number for each character is held as a separate value. (A value mark appears as "]" and is produced within the editor on the keyboard by pressing CTRL-].) In the example, the first attribute contains the control for cursor off and the second attribute holds the codes for cursor on.

Item ID	WYSE.60	RADIO.SHACK.100	STANDARD
01	27]96]48	27]86	-2
02	27]96]49	27]87	(Null)

The following "Assign Screen Constants Not Supported by Pick" routine will read the appropriate item in the TERMINAL.PARAMETERS file and then assign screen constants according to the parameters.

Assign Screen Constants Not Supported by Pick Routine

```
*****
* Assign Screen Constants Not Supported by Pick *
*****
* David L. Clark 09/22/1988
* Provide video characteristics not supported by Pick.
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: AsScNtPk.bpr
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
File.Term.Params.Name$ = "TERMINAL.PARAMETERS"
Term.ID$ = "WYSE.60";* Terminal name could be taken from file or user
DIM Term.Param$(2)
EQU Total.Term.Params TO 2;          * Dimension for array
EQU Cursor.Off$ TO Term.Param$(1)
EQU Cursor.On$ TO Term.Param$(2)
MAT Term.Param$ = ""
EQU Value.Mark$ TO CHAR(253)
OPEN File.Term.Params.Name$ AS File.Term.Params$ ELSE STOP
```

```

* Read from Parameters file
MATREAD Term.Param$ FROM File.Term.Params$, Term.ID$ ELSE STOP
* Set Parameters
FOR I = 1 TO Total.Term.Params
  IF Term.Param$(I) # "" THEN
    Param$ = ""
    Total.Values = DCOUNT(Term.Param$(I), Value.Mark$)
    FOR J = 1 TO Total.Values
      Value = TRIM(Term.Param$(I)<J>); * ASCII or Pick func num
      IF Ascii < 0 THEN; * Pick @( ) function
        Param$ = Param$ : @(Value)
      END ELSE; * ASCII control char
        Param$ = Param$ : CHAR(Value)
      END
    NEXT J
  Term.Param$(I) = Param$
NEXT I

```

A more complete solution is to allow the user to set the parameters for any type of terminal or Pick implementation in use. In addition to video characteristics the parameters would include graphics characters used in screen displays, such as lines, box corners, and a video reverse blank.

The parameters will be written to the parameters file as ASCII or function numbers, rather than as characters. The routine that uses the parameters to assign screen constants will convert the numbers to “@ ()” functions, escape sequences, or control codes. Because the SET.TERMINAL.PARAMETERS program stores numbers rather than characters, it is not necessary to run the program on the terminal that it is meant to define. Also avoided are problems with trying to store characters that Pick regards as system delimiters.

The SET.TERMINAL.PARAMETERS program displays descriptions for video characteristics and for graphic characters. Each description is followed by the decimal numbers of the control characters or “@ ()” functions used to create that video characteristic or graphics symbol.

The parameters are stored in the TERMINAL.PARAMETERS file, with an item ID set by the user. The item ID should include the Pick version in use, because “@ ()” functions differ among versions. Thus there might be two items stored for the Wyse 60 terminal, with one item named “WYSE.60.R83” for release 1983 and the other “WYSE.60.OA” for open architecture. It is necessary to include different keyboard options for the same terminal.

The SET.TERMINAL.PARAMETERS program includes the intended terminal type among the parameters. A program using the parameters might check this parameter against the type actually in use.

The item in the parameters file named “STANDARD.R83” is used to store alternatives to functions or characters not supported. For example, to create a reverse blank on a terminal that does not support graphics characters but that supports reverse video, you use the Pick reverse video function, a space, and then the Pick undo reverse function.

SET.TERMINAL.PARAMETERS Program

(For more information on how to apply the terminal parameters in a data entry program, see the section: “Keyboard: Change Keys.”)

```

* SET.TERMINAL.PARAMETERS David L. Clark 08/24/1988
* In the file TERMINAL.PARAMETERS create parameters Item for each terminal
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: SetTrmPm.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Each Attribute in the Item will hold an ASCII number for a
* graphics character or a function number to implement a video
* characteristic which is not directly supported by Pick or which varies
* among Pick implementations.
* The Terminal ID will be assigned by the user.
* The first time that the program is used, it will create parameters Items
* for the IBM PC-XT running Pick Release 1983 and a standard terminal
* without graphics characters.
*****
* Assign Constants, Parameters *
*****
PROMPT ""
Clear.Screen$ = @(-1)
Erase.Screen.from.Pos$ = @(-3)
EQU False TO 0
File.Term.Params.Name$ = "TERMINAL.PARAMETERS"
Num.Mask$ = "L(#17)"
DIM Name$(26); * Name for video function or graphics char
Name.Mask$ = "L(#14)"
Null.Symbol$ = "#"; * Allow user to set parameter to null
DIM Pos.Scrn$(26); * Screen position where displayed
Row.Bottom = 23
Term.ID$ = "IBM.PC.XT.R83"; * Default terminal
DIM Term.Name$(44)
EQU Term.Names.Max TO 44; * Maximum number of terminal names
Term.Names.Total = 0; * Actual number of terminal names
DIM Term.Param$(26)
MAT Term.Param$ = ""
* 1st column for terminal type, graphics characters.
* 2nd column for functions like reverse, screen lock
EQU Total.1st.Col.Params TO 13; * 1st col display
EQU Total.Term.Params TO 26; * Dimension of arrays
EQU True TO 1
EQU VM$ TO CHAR(253)
*****
* Assign Descriptive Names, Default Parameters *
*****
* Defaults are for IBM PC-XT on Pick Release 1983. Those parameters
* not shown are left null.
Name$(1) = "Terminal type"; Term.Param$(1) = "I"
* Graphics characters
Name$(2) = "Blank Reverse"; Term.Param$(2) = 219
Name$(3) = "Cross"; Term.Param$(3) = 206
Name$(4) = "Horizontal Bar"; Term.Param$(4) = 205
Name$(5) = "Left Down"; Term.Param$(5) = 201
Name$(6) = "Left Up"; Term.Param$(6) = 200
Name$(7) = "Middle Down"; Term.Param$(7) = 203
Name$(8) = "Middle Left"; Term.Param$(8) = 185
Name$(9) = "Middle Right"; Term.Param$(9) = 204
Name$(10) = "Middle Up"; Term.Param$(10) = 202
Name$(11) = "Right Down"; Term.Param$(11) = 187
Name$(12) = "Right Up"; Term.Param$(12) = 188
Name$(13) = "Vertical Bar"; Term.Param$(13) = 186

```



```

* Video functions
Name$(14) = "Cursor Off";          Term.Param$(14) = -2 : VM$ : -7
                                   * Pos Home, Low Intensity
Name$(15) = "Cursor On";          Term.Param$(15) = -8
                                   * High Intensity

Name$(16) = "Screen Lock"
Name$(17) = "Undo Lock"
Name$(18) = "Intensity High";      Term.Param$(18) = -8
Name$(19) = "Intensity Low";       Term.Param$(19) = -7
Name$(20) = "Protect";             Term.Param$(20) = -11
Name$(21) = "Undo Protect";        Term.Param$(21) = -12
Name$(22) = "Reverse";             Term.Param$(22) = -13
Name$(23) = "Reverse Low";         Term.Param$(23) = -13 : VM$ : -7
Name$(24) = "Undo Reverse";        Term.Param$(24) = -14
Name$(25) = "Underline";           Term.Param$(25) = -15
Name$(26) = "Undo Underline";      Term.Param$(26) = -16
*****
* Assign Screen Positions in 2 Cols *
*****
* List begins on row 7, counting from title at row 0 at top.
FOR I = 1 TO Total.1st.Col.Params
    Pos.Scrn$(I) = @(0, I + 6)
NEXT I
Start = Total.1st.Col.Params + 1
Row.Subtract = Start - 7;          * To begin column display at row 7
FOR I = Start TO Total.Term.Params
    Pos.Scrn$(I) = @(39, I - Row.Subtract)
NEXT I
*=====
*****
* Open Files *
*****
OPEN File.Term.Params.Name$ TO File.Term.Params$ ELSE
    CRT "Missing file: " : File.Term.Params.Name$
    GOTO 9999;                      *|End Program|
END
*****
* Read Current Parameters or Create Default Items *
*****
* Display terminal names in 1 or 2 columns, depending on number
* of names.
EXECUTE \SSELECT \ : File.Term.Params.Name$ CAPTURING Capture.Output$
Quit.Flag = False
FOR I = 1 TO Term.Names.Max UNTIL Quit.Flag
    READNEXT Term.Name$(I) THEN
        Term.Names.Total = Term.Names.Total + 1
    END ELSE
        Quit.Flag = True
    END
NEXT I
IF Term.Names.Total = 0 THEN;        * No Items already in file
    *****
    * Create Default Terminal Parameters Items *
    *****
    * Defaults for IBM PC-XT on Pick Release 1983.
    MATWRITE Term.Param$ ON File.Term.Params$, Term.ID$
    Term.Name$(1) = Term.ID$
    * Defaults for standard Release 1983 terminal without graphics.
    * Assumes use of Release 1983 @() functions for IBM PC-XT.

```

```

*   If the function is left null with the DEFINE-TERMINAL
*   procedure, the result will simply be a null parameter.
Term.Param$(1) = "R";           * Terminal Type
Term.Param$(2) = -13 : VM$ : 32 : VM$ : -14; * Blank Reverse
Term.Param$(3) = "+";           * Cross
Term.Param$(4) = "-";           * Horizontal Bar
FOR I = 5 TO 12
    Term.Param$(I) = "+";       * Box corners
NEXT I
Term.Param$(13) = "|";          * Vertical Bar
MATWRITE Term.Param$ ON File.Term.Params$, "STANDARD.R83"
Term.Name$(2) = "STANDARD.R83"
Term.Names.Total = 2
END; * If any terminal parameters item already exists
CRT Clear.Screen$ :
CRT @(20) : "CHOOSE TERMINAL NAME FOR BEGINNING VALUES" :
Quit.Flag = False
IF Term.Names.Total < 23 THEN
    FOR I = 1 TO Term.Names.Total
        CRT @(20, I + 1) : I : ". " : Term.Name$(I) :
    NEXT I
END ELSE
    FOR I = 1 TO 22
        CRT @(0, I + 1) : I : ". " : Term.Name$(I) :
    NEXT I
    FOR I = 23 TO Term.Names.Total
        CRT @(39, I - 21) : I : ". " : Term.Name$(I) :
    NEXT I
END
CRT @(20, Row.Bottom) : "Pick a Number or (Q)uit [1]: " :
INPUT Response$ :
IF NOT(NUM(Response$)) THEN GOTO 9999; *|End Program|
IF Response$ = "" THEN
    Num.Choice = 1
END ELSE
    Num.Choice = Response$ + 0
END
Term.ID$ = Term.Name$(Num.Choice)
MATREAD Term.Param$ FROM File.Term.Params$, Term.ID$ ELSE
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "No terminal parameters found in Item: " : Term.ID$ :
    GOTO 9999; *|End Program|
END
*****
* Convert Multi-Values for Display *
*****
* Term.Param$(1) is the Terminal Type
FOR I = 2 TO Total.Term.Params
    Total.Values = DCOUNT(Term.Param$(I), VM$)
    IF Total.Values > 1 THEN
        Param$ = FIELD(Term.Param$(I), VM$, 1)
        FOR J = 2 TO Total.Values
            Param$ = Param$ : "+" : FIELD(Term.Param$(I), VM$, J)
        NEXT J
        Term.Param$(I) = Param$
    END
NEXT I
*****
* Set New Terminal Parameters *

```

```

*****
CRT Clear.Screen$ : @(20) : "GRAPHICS CHARACTERS AND VIDEO FUNCTIONS":
CRT @(0, 2) : \Enter 0 to 255 for graphics characters and Escape \ :
CRT \sequences or negative numbers for Pick functions. Connect a \ :
CRT \series of numbers with "+", such as: "27+96+48".\
CRT \Enter "#" to make null.\ :
FOR I = 1 TO Total.Term.Params
    CRT Pos.Scrn$(I):I:". " : Name$(I) Name.Mask$ : " = " : Term.Param$(I) :
NEXT I
LOOP; * Until Quit
    CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
    CRT "Pick a Number or (Q)uit [1]: " :
    INPUT Response$ :
    IF Response$ = "" THEN Response$ = 1; * Default
WHILE NUM(Response$) DO
    N = Response$ + 0
    IF N > 0 AND N <= Total.Term.Params THEN
        CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :
        CRT Name$(N) : " = " : Term.Param$(N) : " New entry: " :
        INPUT Response$ :
        * Use mask in case old series of numbers was longer
        * Test for valid range for ASCII or function.
        * Default choice to keep same numbers.
        * Term.Param$(1) = terminal type. Allow anything.
        IF Response$ # "" AND Response$ # Null.Symbol$ THEN
            Test.Flag = True
            IF N > 1 THEN; * Not terminal type. Check for numbers.
                Total.Numbers = DCOUNT(Response$, "+")
                FOR I = 1 TO Total.Numbers WHILE Test.Flag
                    Num.Func = TRIM(FIELD(Response$, "+", I))
                    IF NUM(Num.Func) THEN
                        IF Num.Func > 255 THEN
                            Test.Flag = False
                        END
                    END ELSE
                        Test.Flag = False
                    END
                NEXT I
            END; * If not terminal type parameter
            IF Test.Flag THEN
                Term.Param$(N) = Response$
                CRT Pos.Scrn$(N) : N : ". " : Name$(N) Name.Mask$ :
                CRT " = " : Term.Param$(N) Num.Mask$ :
            END ELSE
                CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(5):
                CRT "ERROR: " : Num.Func : " is not a number " :
                CRT "below 256. Press ENTER:" :
                INPUT Response$ :
            END; * If valid number
        END ELSE
            IF Response$ = "#" THEN; * To make parameter null
                Term.Param$(N) = ""
                CRT Pos.Scrn$(N) : N : ". " : Name$(N) Name.Mask$ :
                CRT " = " : Term.Param$(N) Num.Mask$ :
            END; * If to be set to null
        END
    END; * If within range of number of terminal parameters
REPEAT; * Until Quit

```

```

*****
* Store New Parameters *
*****
CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :
CRT "Name for terminal or (Q)uit [" : Term.ID$ : "]: " :
INPUT Response$ :
IF Response$ = "Q" OR Response$ = "q" THEN GOTO 9999; *|End Program|
IF Response$ # "" THEN Term.ID$ = Response$
CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :
CRT "Storing parameters for " : Term.ID$ :
CRT " in file " : File.Term.Params.Name$ :
*****
* Convert Display to Multi-Values *
*****
* Term.Param$(1) = Terminal type
FOR I = 2 TO Total.Term.Params
    Total.Values = DCOUNT(Term.Param$(I), "+")
    IF Total.Values > 1 THEN
        Param$ = FIELD(Term.Param$(I), "+", 1)
        FOR J = 2 TO Total.Values
            Param$ = Param$ : VM$ : FIELD(Term.Param$(I), "+", J)
        NEXT J
        Term.Param$(I) = Param$
    END
NEXT I
MATWRITE Term.Param$ ON File.Term.Params$, Term.ID$
*=====*
*****
9999 * End Program *
*****
END

```

SCREEN: CLEAR SCREEN

IBM PC BASIC

CLS

For example:

CLS

Use the CLS command to clear the entire screen and put the cursor in the upper left corner. The same command can be used at IBM PC DOS terminal command level.

Microsoft BASIC (Radio Shack 100)

CLS or PRINT CHR\$(27) +

The functions to clear the entire screen will leave the cursor at the "Home" position of the upper left corner of the screen. The other clearing functions will leave the cursor where it was previously positioned, unless otherwise noted.

To clear the entire screen and put the cursor in the upper left corner:

CLS

or:

```
Escape$ = CHR$(27)
Clear.Screen$ = Escape + "E"
PRINT Clear.Screen$
```

To erase from the current cursor position to the end of the screen, leaving the cursor at its current position:

```
Erase.Screen.from.Pos$ = Escape$ + "J"
PRINT Erase.Screen.from.Pos$;
```

To erase from the current cursor position to the end of the line, leaving the cursor at its current position:

```
Erase.Line.from.Pos$ = Escape$ + "K"
PRINT Erase.Line.from.Pos$;
```

To clear the entire line on which the cursor rests, even if the cursor is not at the beginning of the line. The function will leave the cursor at its current position:

```
Clear.Whole.Line$ = Escape$ + "l" '* Lower-case "L", not number 1
PRINT Clear.Whole.Line$
```

Note the use of the clear whole line function to clear the entire line and display a message in the center, without the need for a second cursor placement command:

```
Pos.Message = 287 '*Bottom line, indented 8 spaces
PRINT @Pos.Message, Clear.Whole.Line$, "Pick a Number: ";
```

To insert a blank line, moving down the current line and the lines below it (The function will leave the cursor at the start of the line):

```
Insert.Blank.Line$ = Escape$ + "L"
PRINT Insert.Blank.Line$;
```

To delete the current line, moving up the lines below it (the function will leave the cursor at the start of the line):

```
Delete.Current.Line$ = Escape$ + "M"
PRINT Current.Line$;
```

PICK/BASIC

CRT @(-) or PRINT @(-)

The function to clear the entire screen will leave the cursor at the "Home" position of the upper left corner of the screen. The other clearing functions will leave the cursor where it was previously positioned.

To clear the entire screen and put the cursor in the upper left corner:

```
Clear.Screen $ = @(-1)
CRT Clear.Screen $ :
```

To erase from the current cursor position to the end of the screen leaving the cursor at its current position:

```
Erase.Screen.from.Pos$ = @(-3)
CRT Erase.Screen.from.Pos$;
```

To erase from the current cursor position to the end of the line, leaving the cursor at its current position:

```
Erase.Line.from.Pos$ = @(-4)
CRT Erase.Line.from.Pos$;
```

SCREEN: CURSOR: PLACE THE CURSOR

IBM PC BASIC

LOCATE

For example:

```
LOCATE Row, Col           '* Place the cursor
PRINT Message.1$; Message.2$ '* Print the message
```

Rows are numbered from 1 to 25, columns from 1 to 80. Trying to assign either parameter as zero causes an error.

To eliminate any existing function key display on row 25, use the command KEY OFF.

Two separate operations are always required, first to place, second to print. In PICK/BASIC both operations can be combined into a single statement.

TAB()

For example:

```
PRINT TAB(Column); Message$
```

TAB skips the number of spaces specified in parentheses, to begin display at the column position following. The left-most column is considered to be "1." (If a program treated the left-most column as zero rather than 1, then TAB would place the message at the column named.)

A zero column number does not cause an error. A negative column specification causes a function call error.

The following “Place Prompt Message” routine will clear the entire last row of the screen, and then prompt the user for input. Because screen positions cannot be assigned as strings and concatenated, the IBM PC BASIC version of the routine includes more separate statements than are necessary in the PICK/BASIC version.

Place Prompt Message Routine

```

*****
'* Place Prompt Message *
*****
'* David L. Clark 09/22/1988
'* Clear bottom row, prompt user.
'* IBM PC Basic routine. Documentation version.
'* IBM PC BASICA 3.0
'* Stored under filename: PlacPrmp.brd
*****
'* Set Position *
*****
'* Cannot set screen position as a string in IBM PC Basic
Text$      = "Pick a Number: "
Row.Bottom = 25
Col.Max    = 80
Col.Message = (Col.Max - LEN(Text$)) \ 2
*****
'* Clear and Prompt *
*****
LOCATE Row.Bottom, 1
PRINT SPACE$(Col.Max - 1);      '* Print to last column causes scroll
LOCATE Row.Bottom, Col.Message
PRINT Text$;
LINE INPUT; Entry$              '* Semicolon prevents scroll

```

Microsoft BASIC (Radio Shack 100)

PRINT @

For example:

```

PRINT @ Pos.Screen, "";          '* Place the cursor
PRINT @ Pos.Screen, Message.1$; Message.2$  '* Place and display

```

The “PRINT @” function is the alternative to LOCATE used on some versions of Microsoft Basic. Unlike LOCATE, “PRINT @” allows placement and display in a single command. An ending semicolon will prevent a Carriage Return—Line Feed when placing the cursor with “PRINT @”. Otherwise, the cursor would drop down to the start of the next row.

The Radio Shack 100 uses screen positions numbered from 0 to 319, for 8 rows of 40 columns each. To translate rows and columns to screen position, counting rows from top to bottom as 0–7 and Columns from left to right as 0–39, use the formula:

```

Screen.Width = 40
Pos.Screen = Row * Screen.Width + Col

```

Because the Radio Shack 100's screen position is a single number, it can be added to or subtracted from. The following routine will clear the entire last row of the screen and prompt the user for input:

```
Col.Message = 12
Pos.Bottom.Row = 280
Pos.Message = Pos.Bottom.Row + Col.Message
Escape$ = CHR$(27)
Erase.Screen.from.Pos$ = Escape$ + "J"
Message$ = "Pick a Number:"
PRINT @Pos.Bottom.Row, Erase.Screen.from.Pos$;
PRINT @Pos.Message, Message$;
LINE INPUT Entry$
```

PRINT CHR\$(27) +

The Radio Shack 100 also allows cursor placement using escape codes, which can be stored in a string variable and concatenated with video characteristics and the message to display. The screen position set with escape codes is treated here as a string because it cannot be added to or subtracted from.

To place the cursor as a specified row and column, counting rows and columns from zero:

```
Pos.Screen$ = Escape$ + "Y" + CHR$(Row + 32) + CHR$(Column + 32)
PRINT Pos.Screen$;
```

When defining a constant, the escape procedure for cursor placement can be shortened by using directly the characters that would be produced by the CHR\$() function. For example, the following two sequences each will place the cursor at row 3, column 33:

```
Row = 3
Column = 33
Pos.Screen$ = Escape$ + "Y" + CHR$(Row + 32) + (Column + 32)
```

or:

```
Pos.Screen$ = Escape$ + "Y#A"
```

The hash mark character "#" equals CHR\$(35), equivalent to CHR\$(3 + 33), and the character "A" equals CHR\$(65), equivalent to CHR\$(33 + 32).

A clear advantage of the escape code method of cursor placement, within the context of this multilingual guide, is that the escape procedure is compatible with PICK/BASIC, and can be executed in a PICK/BASIC program when using the Radio Shack 100 as a terminal. The escape codes also can be specified for Pick's "@(Col, Row)" functions, set with the DEFINE-TERMINAL procedure, in order to assign the Radio Shack 100 as a Pick terminal, allowing the use of the device without the need for special program code. (For details on DEFINE-TERMINAL, see the section: "Screen: Assign Terminal.")

When using the portable device on a stand-alone basis the choice between the two methods of cursor placement would depend on whether the mathematics of positioning

would be easier to perform on a row and column basis with escape codes or on a continuous numbering basis with "PRINT @".

The following "Place Prompt Message" routine will clear the entire last row of the screen, and then prompt the user for input. Note how similar the routine is to the PICK/BASIC version, presented in the PICK/BASIC portion of this section.

Place Prompt Message Routine

```
*****
'* Place Prompt Message *
*****
'* David L. Clark 09/22/1988
'* Clear bottom row, prompt user.
'* Mircrosoft Basic (Radio Shack 100) routine. Documentation version.
'* Radio Shack 100
'* Stored under filename: PlcPrp.do
Escape$ = CHR$(27)
Erase.Screen.from.Pos$ = Escape$ + "J"
Scroll.Lock$ = Escape$ + "V"
Scroll.Unlock$ = Escape$ + "W"
*****
'* Set Position *
*****
Text$ = "Pick a Number:"
Col.Max = 39
Col.Message = (Col.Max - LEN(Text$)) \ 2
Row.Bottom = 7
Pos.Code$ = Escape$ + "Y"
Pos.Bottom.Row$ = Pos.Code$ + CHR$(Row.Bottom + 32) + CHR$(32)
Pos.Message$ = Pos.Code$ + CHR$(Row.Bottom+32) + CHR$(Col.Message+32)
*****
'* Assign Message *
*****
Message$ = Scroll.Lock$ + Pos.Bottom.Row$ + Erase.Screen.from.Pos$
+ Pos.Message$ + Text$
*****
'* Clear and Prompt *
*****
PRINT Message$;
LINE INPUT Entry$
PRINT Scroll.Unlock$;
```

Microsoft BASIC on the Radio Shack 100 also provides the following special cursor placement functions using escape codes and control characters.

To place the cursor at the "Home" position at the upper left corner of the screen:

```
Escape$ = CHR$(27)
Pos.Home$ = Escape$ + "H"
PRINT Pos.Home$;
```

or:

```
Pos.Home$ = CHR$(11)
PRINT Pos.Home$;
```

To move the cursor up one line, retaining the same column position, without scrolling the screen:

```
Up.1.Line$ = Escape$ + "A"  
PRINT Up.1.Line$;
```

To move the cursor down one line, retaining the same column position, without scrolling the screen:

```
Down.1.Line$ = Escape$ + "B"  
PRINT Down.1.Line$;
```

To move the cursor down one line, retaining the same column position, and scrolling the screen after reaching the bottom line, unless scroll lock has been activated:

```
Line.Feed$ = CHR$(10)  
PRINT Line.Feed$;
```

(For Scroll Lock, see the section: "Screen Output: Scroll Lock.")

To move the cursor left one character:

```
Left.1.Char$ = Escape$ + "D"  
PRINT Left.1.Char$;
```

The cursor will stop at the left margin, rather than backspacing to the end of the previous line.

To backspace the cursor one character:

```
Backspace$ = CHR$(8)  
PRINT Backspace$;
```

The cursor will not stop at the left margin, rather the cursor will continue backspacing to the end of the previous line. No character will be erased.

To move the cursor to the left margin, without dropping down a line:

```
Carriage.Return$ = CHR$(13)  
PRINT Carriage.Return$;
```

To move the cursor right one character:

```
Right.1.Char$ = Escape$ + "C"  
PRINT Right.1.Char$;
```

The cursor will stop at the right margin, rather than continuing down to the next line.

PICK/BASIC

CRT @() or PRINT @()

For example:

```

CRT @(Column, {Row} )
CRT @(Column, Row) : "" :      * Just place the cursor
CRT @(Column, Row) : Message$
Pos.Screen$ = @(Column, Row)
CRT Pos.Screen$ : Message$
Col.Screen$ = @(Column)
CRT Col.Screen$ : Message$

```

Specifying the row is optional. Rows and columns are numbered from zero.

The ending colon prevents a Carriage Return—Line Feed when placing the cursor. Otherwise, the cursor would drop to the start of the next row.

To translate from a single-number screen position (used by the Radio Shack 100) to two numbers for rows and columns, use the formula:

```

Row = INT(Pos.Screen / Screen.Width)
Col = MOD(Pos.Screen, Screen.Width)

```

In PICK/BASIC the screen position can be stored in a string variable and concatenated with video characteristics and the message to display. The screen position is treated here as a string because it cannot be added to or subtracted from. The following “Place Prompt Message” routine will clear the entire last row of the screen, and then prompt the user for input.

Place Prompt Message Routine

```

*****
* Place Prompt Message *
*****
* David L. Clark 09/22/1988
* Clear bottom row, prompt user.
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: PlacPrmpt.bpr
Erase.Screen.from.Pos$ = @(-3)
*****
* Set Position *
*****
Text$           = "Pick a Number:"
Col.Max         = SYSTEM(2)
Col.Message     = (Col.Max - LEN(Text$)) / 2
Row.Bottom      = SYSTEM(3)
Pos.Bottom.Row$ = @(0, Row.Bottom)
Pos.Message$    = @(Col.Message, Row.Bottom)
*****
* Assign Message *
*****
Message$ = Pos.Bottom.Row$ : Erase.Screen.from.Pos$ : Pos.Message$
Message$ = Message$ : Text$
*****
* Clear and Prompt *
*****
CRT Message$ :
INPUT Entry$ :

```

CRT @(-) or PRINT @(-)

PICK/BASIC also provides the following special cursor placement functions.

To place the cursor at the “Home” position at the upper left corner of the screen:

```
Pos.Home$ = @(-2)
CRT Pos.Home$ :
```

To move the cursor up one line:

```
Up.1.Line$ = @(-10)
PRINT Up.1.Line$;
```

To backspace the cursor one character:

```
Backspace$ = @(-9)
CRT Backspace$ :
```

or:

```
Backspace$ = CHAR(8)
CRT Backspace$ :
```

With either backspace definition, the cursor will not stop at the left margin, rather the cursor will continue backspacing to the end of the previous line. No character will be erased.

SCREEN: CURSOR: RETURN POSITION

This command is used to get the row and column positions of the cursor on the screen.

IBM PC BASIC

CSRLIN, POS(0)

Cursor.Row	= CSRLIN	'* Row position of cursor
Cursor.Column	= POS(0)	'* Column position of cursor

SCREEN: CURSOR: TURN OFF

These commands are used to turn off the blinking cursor, to avoid distraction while displaying information.

IBM PC BASIC

LOCATE

For example:

```
LOCATE {Row} {, {Column} {, {Cursor.On.Off} } }
```

To turn the cursor off and on, without affecting the cursor position:

```
LOCATE ,,0      '* Cursor off
LOCATE ,,1      '* Cursor on
```

Microsoft BASIC (Radio Shack 100)

PRINT CHR\$(27) +

To turn the cursor off and on, without affecting the cursor position:

```
Cursor.Off$ = Escape$ + "Q"
PRINT Cursor.Off$;
Cursor.On$ = Escape$ + "P"
PRINT Cursor.On$;
```

SCREEN: FORMAT MASKS

Format masks are used in the same manner whether output is to a screen or to a printer. In IBM PC BASIC, format masks also will apply when “printing” to a file. In Pick, format masks also can be contained in conversions and correlatives.

Because direct cursor control is the preferred method for screen output, format masks are more necessary for printer than for terminal use. For the use of format masks see the section: “Printer: Format Masks.”

SCREEN: PAGING, HEADINGS, AND FOOTINGS

PICK/BASIC

PAGE, HEADING, FOOTING

For example:

```
* Assign settings for report . . .
HEADING "Security Pacific Photo Collection 'CT' Page 'PLL' "
* Loop to print report . . .
PAGE
```

The HEADING statement places a heading at the top of a page of output each time that the PAGE command is given, whether output is to the screen or to a printer. The FOOTING statement operates in the same manner as HEADING, except that the material is placed at the bottom of the page. The entire heading or footing is enclosed in double quotes, with options enclosed in single quotes. The previous example would place the heading “Security Pacific Photo Collection” at the top of the page. The heading would be centered on the line (C option), it would include the time and date (T option), the word “Page,” the page number (P option), and be followed by two Carriage Returns—Line Feeds (L option used twice).

The following options are available for use on the screen:

```
C Center
D Date in format DD MMM YYYY
```

L Line Feed, Carriage Return
N No stop at end of page, on screen
P Page number, right-justified in a field of 4 blanks
T Time and Date in format HH:MM:SS DD MMM YYYY

SCREEN: PROTECT

PICK/BASIC

CRT @(-) or PRINT @(-)

For example:

Protect.Enable\$ = @(-11);	* Must enable before using Protect.Start\$
Protect.Disable\$ = @(-12);	* Now Protect.Start\$ has no effect
Protect.Start\$ = @(-7);	* Sets Low Intensity, even if no protect
Protect. Stop\$ = @(-8);	* Sets High Intensity

PICK/BASIC offers a means of protecting information on the screen from being erased or overwritten. The protect function is used most often to display a data entry form with the attribute (field) names, prompts, and information for the user set in protected mode. As each item (record) is entered or edited the screen must be prepared for the next item. Because the information that stays the same is protected, all of the old data can be erased with the clear screen command without affecting the rest of the screen. The form stays on the screen while the data changes.

The protected information is displayed in low intensity, and the data is high intensity, to help the user distinguish between the two different types of information on the screen. If the protect function has not first been enabled by the command:

CRT Protect.Enable\$:

then the result of placing Protect.Start\$ at the start of a display command will be to cause display in low intensity.

Be sure to disable the protect mode and clear the screen at the end of a program that uses protect mode. Otherwise, the subsequent screen will be a mess, with the information in protected mode continuing to show. Note here the utility of always exiting to an "End Program" routine at the end of a program. Any necessary clean up operations will be included in that routine.

The following TEST.PROTECT sample program will create a Form.Page\$ of attribute names and a prompt in protected mode and low intensity. Data will then be displayed and cleared without affecting the form.

TEST.PROTECT Program

```
* TEST.PROTECT David L. Clark 08/27/1988
* Test video protect function.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TProtect.bp
```

```

* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* The Field names and prompt will be protected - they will not be erased
*   by the clear screen command. Only the data and the response to the
*   prompt will be erased.
* The clear screen is much slower when some information must be protected.
*****
* Assign Constants, Parameters, Initial Values *
*****
PROMPT ""
Clear.Screen$ = @(-1)
DIM Data$(8)
EQU False TO 0
DIM Name$(8)
DIM Pos.Data$(8)
DIM Pos.Name$(8)
Protect.Enable$ = @(-11); * Must enable before using Protect.Start$
Protect.Disable$ = @(-12); * Now Protect.Start$ has no effect
Protect.Start$ = @(-7); * Sets Low Intensity, even if no protect
Protect.Stop$ = @(-8); * Sets High Intensity
Row.Bottom = SYSTEM(3)
Title$ = @(30,0) : Protect.Start$ : "DATA ENTRY FORM" : Protect.Stop$
EQU True TO 1
EQU Total.Attribs TO 8; * Dimension of arrays
*****
* Set Prompt *
*****
Prompt$ = "1. Clear, Display Next Data 2. Clear Data 3. Quit"
Prompt$ = Prompt$ : " Pick a Number [1]:"
Pos.Response$ = @(LEN(Prompt$) + 1, Row.Bottom); * Place for response
Prompt$ = @(0, Row.Bottom) : Protect.Start$ : Prompt$ : Protect.Stop$
*****
* Assign and Display Form, Enable Protect *
*****
Form.Page$ = Clear.Screen$ : Protect.Enable$ : Title$
FOR I = 1 TO Total.Attribs
    Name$(I) = Protect.Start$ : "FIELD #" : I : " : " : Protect.Stop$
    Pos.Name$(I) = @(10, I * 2 + 2)
    Pos.Data$(I) = @(19, I * 2 + 2)
    Form.Page$ = Form.Page$ : Pos.Name$(I) : Name$(I)
NEXT I
Form.Page$ = Form.Page$ : Prompt$
CRT Form.Page$ :
*****
* Display Test Data #1 *
*****
Test.Num = 1
GOSUB 100; *|Assign and Display Test Data|
*****
* Loop to Display and Clear Data *
*****
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
    *****
    * Input and Act on Choice *
    *****
CRT Pos.Response$ :
INPUT Response$ :
IF Response$ = "" THEN Response$ = "1"; * Default
IF NUM(Response$) THEN

```

```

    Num.Choice = Response$ + 0;                * Convert to number
    BEGIN CASE
        CASE Num.Choice = 1;                * Clear, Display New Data
            CRT Clear.Screen$ :
            Test.Num = Test.Num + 1
            GOSUB 100;                *|Assign and Display Test Data|
        CASE Num.Choice = 2;                * Clear Data
            CRT Clear.Screen$ :
        CASE True;                * Quit
            Quit.Flag = True
    END CASE
    END ELSE
    Quit.Flag = True
    END; * If entered a number
REPEAT
GOTO 9999;                *|End Program|
*=====*
*****
100 * Assign and Display Test Data *
*****
* Parameters: Test.Num for data set to display
Data.Page$ = ""
FOR I = 1 TO Total.Attribs
    Data$(I) = "Test #" : Test.Num
    Data.Page$ = Data.Page$ : Pos.Data$(I) : Data$(I)
NEXT I
CRT Data.Page$ :
RETURN
*=====*
*****
9999 * End Program *
*****
*****
* Cleanup: Disable Protect, Clear Screen *
*****
CRT Protect.Disable$ : Clear.Screen$ :
END

```

SCREEN: REDIRECT OUTPUT

This command is used to redirect output that normally would go to one device to another. The most common use of redirection is to send output formatted for screen display to a printer or to a file to store the information for later reference.

IBM PC DOS Command Level

At IBM PC DOS command level, you can redirect output very simply by using the greater than symbol (">") as a pointer to the intended device. For example, to send the output of the DIR command, which will be a list of the files in the directory, to the printer:

```
DIR > PRN
```

To store the directory listing as a file, with the filename "Dir.doc":

```
DIR > Dir.doc
```


Under IBM PC DOS, the PRINT-SCREEN key also will send the current screen display to the printer. For output such as a directory listing, which might cover more than a single screen, redirection is the preferred method.

Most IBM PC DOS word programs also have a “Print to File” feature that allows output formatted for printing to be sent to a file. This feature is employed for electronic mail or computer source code where a Carriage Return—Line Feed at the end of every line is expected.

PICK/BASIC

The SCREEN.TO.FILE program will redirect output formatted for screen display or for printing into a Pick item. The program uses the CAPTURING clause of the EXECUTE command to accomplish the same purpose as redirection of output at the IBM DOS operating system level or a “Print to File” operation in an IBM PC DOS word processor.

The SCREEN.TO.FILE program could be used to save output for further manipulation by a word processor, or to store information that would otherwise appear only briefly. For example, compiling BASIC program code with the “(A)” option produces a listing of the object code in assembly language format. The listing could be sent to a printer with the option list “(AP)”, or the listing could be saved as an item by invoking the SCREEN.TO.FILE program, and then giving the compilation command when prompted by SCREEN.TO.FILE for a command:

Enter the command to execute [None to Quit]: BASIC Filename ID (A)

The same save-to-file operation will work if printer output is specified with the “(P)” option or the LPTR modifier, but screen format is preferred, to avoid the page break commands that will be included in printer output unless the TERM command is first used at Pick TCL level to set a printer page length of zero:

```
TERM ,,,,,,0
```

After the end of the program, TERM should be reset:

```
TERM ,,,,,,60
```

The output stored by SCREEN.TO.FILE can be custom trimmed for specific commands and purposes. To trim the output, set the parameters Attribs.Delete.Begin and Attribs.Delete.End in the assign constants, initial values section. Alternately, you could set the parameters through user input or a calling PROC.

Differences in the output format of various commands prevent the use of one trimming routine for all. Other differences will probably exist between the various Pick implementations.

The first four attributes of Capture.Output\$ usually will contain a combination of nulls, an echo of the command, or a heading. If the command is echoed, as in the case of the LISTFILES, the echoed command will appear in the first attribute. If output is formatted for the printer, a page break command can be included before the heading.

For a command that displays a list on the screen, the last attribute will contain the “Number of Items listed” message, and the next-to-last attribute will be null.

SCREEN.TO.FILE Program

```

* SCREEN.TO.FILE David L. Clark 04/01/1988
* Store in a Pick Item output formatted for screen display or for printing.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: ScrnFile.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants, Initial Values *
*****
PROMPT ""
EQU Attrib.Mark$ TO CHAR(254)
* Attributes to delete from start, end of output
Attribs.Delete.Begin = 0
Attribs.Delete.End = 0
Clear.Screen$ = @(-1)
*****
* Get User Request *
*****
CRT Clear.Screen$ : @(30) : "STORE OUTPUT IN A FILE"
CRT
CRT "Enter the command to execute [None to Quit]: " :
INPUT Command$
Command$ = OCONV(Command$, "MCU"); * Convert to upper case
IF Command$ = "" OR Command$ = "NONE" THEN GOTO 9999; *|End Program|
CRT
CRT "Enter name of file to store output: " :
INPUT File.Storage.Name$
CRT "Enter Item ID: " :
INPUT Item.ID$
*****
* Open Storage File *
*****
OPEN File.Storage.Name$ TO File.Storage$ ELSE
    CRT "Cannot find file: " : File.Storage.Name$
    GOTO 9999; *|End Program|
END
*****
* Execute the Command, Capture Output *
*****
EXECUTE Command$ CAPTURING Capture.Output$
*****
* Trim the Output *
*****
* Parameters: Attribs.Delete.Begin, Attribs.Delete.End,
* Capture.Output$
* Returns: Capture.Output$
* Temporary variables:
* Total.Lines : Total lines equals total Attributes Capture.Output$
* To trim the output, set Attribs.Delete.Begin and Attribs.Delete.End
* Delete Beginning Attributes
FOR I = 1 TO Attribs.Delete.Begin
    Capture.Output$ = DELETE(Capture.Output$, 1)
NEXT I
IF Attribs.Delete.End THEN
    * Delete Ending Attributes
    Total.Lines = DCOUNT(Capture.Output$, Attrib.Mark$)

```

```

        FOR I = 1 TO Attribs.Delete.End
            Capture.Output$ = DELETE(Capture.Output$, Total.Lines+1 - I)
        NEXT I
    END
    *****
    * Store Output as an Item *
    *****
    WRITE Capture.Output$ ON File.Storage$, Item.ID$
    *****
9999  * End Program *
    *****
    END

```

SCREEN: SCROLL LOCK

Scroll lock is used to prevent information from scrolling off the top of the screen.

Microsoft BASIC (Radio Shack 100)

PRINT CHR\$(27) +

For example:

```

Escape$ = CHR$(27)
Screen.Lock$ = Escape$ + "V"
Screen.Unlock$ = Escape$ + "W"

```

Screen lock should be turned on at the start of the program, in the assign constants section, and turned on only when needed.

SCREEN: VIDEO CHARACTERISTICS

IBM PC BASIC

COLOR

For example:

```
COLOR {Foreground} {, {Background} {, Border - Graphics mode only} }
```

Video characteristics such as reverse video are set with the **COLOR** statement, with separate foreground and background designations. Numbers can be added to the existing foreground setting and background setting to set and to combine various characteristics. Thus, 16 added to the current foreground setting will cause blinking. Video characteristics cannot be assigned to variables and concatenated, as in PICK/BASIC.

To turn on reverse video:

```
COLOR 0, 7
```

To undo reverse video:

```
COLOR 7, 0
```

Display Prompt in Video Reverse Routine

The following “Display Prompt in Video Reverse” routine will display the prompt “Pick a Number” at a screen position designated as Col.Message, Row.Message. The screen is first erased from that position to the end of the line. Reverse video is turned on to highlight the message, and then turned off for the user’s input.

```
*****
'* Display Prompt in Video Reverse *
*****
'* David L. Clark 09/22/1988
'* Clear bottom row, display prompt message in video reverse
'* IBM PC Basic routine. Documentation version.
'* IBM PC BASICA 3.0
'* Stored under filename: VidRevrs.brd
*****
'* Set Position *
*****
'* Cannot set screen position as a string in IBM PC Basic
Text$      = "Pick a Number"
Row.Bottom = 25
Col.Max    = 80
Col.Message = (Col.Max - LEN(Text$)) \ 2
*****
'* Display Message *
*****
LOCATE Row.Bottom, 1
PRINT SPACE$(Col.Max - 1);      '* Print to last column causes scroll
LOCATE Row.Bottom, Col.Message
COLOR 0, 7                      '* Video Reverse
PRINT Text$;
COLOR 7, 0                      '* Undo Video Reverse
PRINT ": ";                     '* Colon not in video reverse
LINE INPUT; Entry$             '* Semicolon prevents scroll
```

Microsoft BASIC (Radio Shack 100)

PRINT CHR\$(27) +

The IBM PC BASIC COLOR command is not available in Microsoft BASIC on the Radio Shack 100 and other simpler machines. Instead, video characteristics such as reverse video are set with the PRINT command, using the escape character ASCII 27 plus a character parameter that has been designated to set a certain characteristic.

To turn on reverse video:

```
PRINT CHR$(27) + "p"
```

To undo reverse video:

```
PRINT CHR$(27) + "q"
```

A variable can be used to set the characteristic. For example:

```
Escape$ = CHR$(27)
Video.Reverse$ = Escape$ + "p"
Undo.Video.Reverse$ = Escape$ + "q"
```

Parameters are not added or subtracted to combine characteristics. However variables can be concatenated rather than set individually.

The following "Display Prompt in Video Reverse" routine will display the prompt "Pick a Number" at a screen position designated as Pos.Message\$. The screen is first erased from that position to the end of the screen. Reverse video is turned on to highlight the message, and then turned off for the user's input.

Display Prompt in Video Reverse Routine

```

*****
* Display Prompt in Video Reverse *
*****
* David L. Clark 09/22/1988
* Clear bottom row, display prompt message in video reverse
* Microsoft Basic (Radio Shack 100) routine. Documentation version.
* Radio Shack 100.
* Stored under filename: VidRev.do
Escape$      = CHR$(27)
Erase.Screen.from.Pos$ = Escape$ + "E"
Scroll.Lock$  = Escape$ + "V"
Scroll.Unlock$ = Escape$ + "W"
Undo.Video.Reverse$ = Escape$ + "q"
Video.Reverse$ = Escape$ + "p"
*****
* Set Position *
*****
Text$        = "Pick a Number"
Col.Max      = 39
Col.Message  = (Col.Max - LEN(Text$)) \ 2
Row.Bottom   = 7
Pos.Code$    = Escape$ + "Y"
Pos.Bottom.Row$ = Pos.Code$ + CHR$(Row.Bottom + 32) + CHR$(32)
Pos.Message$ = Pos.Code$ + CHR$(Row.Bottom+32) + CHR$(Col.Message+32)
*****
* Assign Message *
*****
Message$ = Scroll.Lock$ + Pos.Bottom.Row$ + Erase.Screen.from.Pos$
+ Pos.Message$ + Video.Reverse$ + Text$ + Undo.Video.Reverse$
+ ": "
* Colon not in video reverse
*****
* Display Message *
*****
PRINT Message$;
LINE INPUT Entry$
PRINT Scroll.Unlock$;

```

On the Radio Shack 100 the only characteristic that can be changed to highlight a character is video reverse. Blinking and underline are not available.

PICK/BASIC

CRT @(-) or PRINT @(-)

Video characteristics such as reverse video are set with the CRT @() or PRINT @() command. Rather than placing a column and row position in the parentheses, a negative number is used as a parameter, to specify a video characteristic.

The parameters used for the “@()” functions differ among the various Pick implementations. Those shown in this section apply to IBM PC/XT release 1983 Version 2.0. To set parameters for other versions, within the “Screen” section, see the subsections for: “Assign Screen Constants,” “Assign Terminal,” and “Assign Terminal Parameters.” To test the functions for a particular implementation of Pick, use the TEST.VIDEO.FUNCTIONS program in this section.

To turn on reverse video:

```
CRT @(-13)
```

To undo reverse video:

```
CRT @(-14)
```

A variable can be used to set the characteristic. For example:

```
Video.Reverse$      = @(-13)
Undo.Video.Reverse$ = @(-14)
```

The variable assignment statement with “=” should be used rather than constant assignment with “EQUATE,” Variable assignment will allow the Pick System to modify the control characters that it sends to the terminal according to the type of terminal used.

Parameters are not added or subtracted to combine characteristics. However, variables can be concatenated rather than set individually.

The following “Display Prompt in Video Reverse” routine will display the prompt “Pick a Number” at a screen position designated as Pos.Message\$. The screen is first erased from that position to the end of the screen. Reverse video is turned on to highlight the message, and then turned off for the user’s input.

Display Prompt in Video Reverse Routine

```
*****
* Display Prompt in Video Reverse *
*****
!* David L. Clark 09/22/1988
* Clear bottom row, display prompt message in video reverse
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: VidRevrs.bpr
Erase.Screen.from.Pos$ = @(-3)
Undo.Video.Reverse$    = @(-14)
Video.Reverse$         = @(-13)
*****
* Set Position *
*****
Text$                  = "Pick a Number"
Col.Max                = SYSTEM(2)
Col.Message            = (Col.Max - LEN(Text$)) / 2
Row.Bottom             = SYSTEM(3)
Pos.Bottom.Row$        = @ (0, Row.Bottom)
Pos.Message$           = @ (Col.Message, Row.Bottom)
```

```

*****
* Assign Message *
*****
* Colon at end of message not in video reverse
Message$ = Pos.Bottom.Row$ : Erase.Screen.from.Pos$ : Pos.Message$
Message$ = Message$ : Video.Reverse$ : Text$:Undo.Video.Reverse$: " : "
*****
* Display Message *
*****
CRT Message$ :
INPUT Entry$ :

```

The following TEST.VIDEO program will display examples of the various video characteristics, and combinations of characteristics, that might be used to display a prompt message.

TEST.VIDEO Program

```

* TEST.VIDEO: David L. Clark. 07/24/1987.
* Test video characteristics used to emphasize a prompt message.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TVideo.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* The descriptions pertain to Pick PC-XT 2.0 on the IBM Monochrome.
* Show different methods of presenting the prompt message "Pick a Number"
* followed by a colon separator and the default entry of "1".
* The selections will be in low intensity.
* The default entry and entry from the user will be in high intensity.
* For a data entry routine, if reverse video blank is used to show the area
* for entry, and the prompt is reverse video, without a colon separator
* it will be difficult to distinguish the prompt from the space for the
* entry.
*****
* Assign Constants, Parameters, Initial Values *
*****
PROMPT ""
* Change next 2 lines for more alternatives
EQU Total.Alternatives TO 9
DIM Alternative$(9), Message$(9)
EQU False TO 0, True TO 1
Clear.Screen$ = @(-1)
EQU Col.Message TO 50
Prompt$ = "Pick a Number"
Col.Entry = Col.Message + LEN(Prompt$) + 1;          * Allow for colon
Col.Entry$ = @(Col.Entry)
Default$ = "1"
*****
* Blinking *
*****
* Very distracting and annoying
* Use Video Reverse instead
Video.Blinking$ = @(-5)
Undo.Video.Blinking$ = @(-6)
*****
* Intensity *
*****

```

```

* Use Low for normal, High for emphasis
Video.Intensity.High$ = @(-8)
Video.Intensity.Low$  = @(-7)
*****
* Underline *
*****
* Works better with Low Intensity
Video.Underline$      = @(-15)
Undo.Video.Underline$ = @(-16)
*****
* Video Reverse *
*****
* Works better with Low Intensity
* Cannot combine with Underline
Video.Reverse$        = @(-13)
Undo.Video.Reverse$   = @(-14)
*****
* Build Video Alternatives *
*****
Alternative$(1) = "Low Intensity"
Message$(1) = Prompt$
Alternative$(2) = "High Intensity"
Message$(2) = Video.Intensity.High$ : Prompt$
Alternative$(3) = "Video Reverse, Low Intensity"
Message$(3) = Video.Reverse$
Message$(3) = Message$(3) : Prompt$ : Undo.Video.Reverse$
Alternative$(4) = "Video Reverse, High Intensity"
Message$(4) = Video.Intensity.High$ : Video.Reverse$
Message$(4) = Message$(4) : Prompt$ : Undo.Video.Reverse$
Alternative$(5) = "Underlined, Low Intensity"
Message$(5) = Video.Underline$ : Prompt$ : Undo.Video.Underline$
Alternative$(6) = "Underlined, High Intensity"
Message$(6) = Video.Intensity.High$ : Video.Underline$
Message$(6) = Message$(6) : Prompt$ : Undo.Video.Underline$
Alternative$(7) = "Blinking, High Intensity"
Message$(7) = Video.Intensity.High$ : Video.Blinking$
Message$(7) = Message$(7) : Prompt$ : Undo.Video.Blinking$
Alternative$(8) = "Video Reverse, Low Intensity, Underlined"
Message$(8) = Video.Reverse$ : Video.Underline$ : Prompt$
Message$(8) = Message$(8) : Undo.Video.Reverse$
Message$(8) = Message$(8) : Undo.Video.Underline$
Alternative$(9) = "Video Reverse, Low Intensity, Blinking"
Message$(9) = Video.Reverse$ : Video.Blinking$
Message$(9) = Message$(9) : Prompt$ : Undo.Video.Reverse$
Message$(9) = Message$(9) : Undo.Video.Blinking$
*=====*
*****
* Main Program *
*****
CRT Clear.Screen$ : Video.Intensity.Low$ : "Select alternative:"
FOR Alt.Num = 1 TO Total.Alternatives
    GOSUB 100;                *|Display Video Alternatives|
NEXT Alt.Num
End.Num = Total.Alternatives + 1
Row = End.Num * 2
CRT @(0, Row) : Video.Intensity.Low$ : End.Num : ". " : "End Program"
Quit.Flag = False
Alt.Num = 1
LOOP UNTIL Quit.Flag DO

```



```

Row = Alt.Num * 2
Pos.Entry$ = @(Col.Entry, Row)
CRT Pos.Entry$ : Video.Intensity.High$ : Default$ :
CRT Pos.Entry$ :
INPUT Entry$ :
IF Entry$ = "" THEN Entry$ = Default$
IF NUM(Entry$) THEN
    Alt.Num = Entry$ * 1
    IF Alt.Num < 1 OR Alt.Num > Total.Alternatives THEN
        Quit.Flag = True
    END
END ELSE
    Quit.Flag = True
END
REPEAT
*****
* Exit *
*****
GOTO 9990;                                *|End Program|
*-----*
*****
100  * Display Video Alternatives *
*****
* Parameters: Alt.Num
* Other values are constants
Row = Alt.Num * 2
CRT @(0, Row) : Video.Intensity.Low$ : Alt.Num : ". " :
CRT Alternative$(Alt.Num) :
CRT @(Col.Message) : Message$(Alt.Num) : ":" :
CRT Video.Intensity.High$ : Default$ :
RETURN
*=====*
9990 * End Program *
*****
* Exit at high intensity, which is treated by Pick as the default
* mode. If exit at low intensity, the cursor will still be at high
* intensity. The cursor will switch to low when backspace, then
* to high again when move forward past last position.
Row.Bottom = SYSTEM(3)
CRT @(0, Row.Bottom - 1) :
END

```

Use the TEST.VIDEO.FUNCTIONS program to test the result of any character or code, including graphics characters, video characteristics, and screen positioning. The program will prompt for input, and then either display the character or activate the code. The user should then press ENTER to repeat the action, enter "NEXT" to try something else, or "QUIT" to quit. If the user enters the code for reverse video, for example, no result will at first be apparent. But when the user types "NEXT" the word will appear in reverse video, as will the next prompt message. Entering the ASCII number for the backspace character (usually 8) will cause the cursor to backspace across the prompt message.

The effect of a previously entered code is not canceled until the user enters another code for that purpose. As different combinations are tried, the screen display might become messy. Remember that "NEXT" or "QUIT" can be entered at any point, no matter where the cursor happens to be.

TEST.VIDEO.FUNCTIONS Program

```

* TEST.VIDEO.FUNCTIONS David L. Clark 04/22/1988
* Test terminal, Pick video codes.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TvidFunc.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* If unsure of the ESCAPE key, type in "ESCAPE".
* User may enter NEXT or QUIT at any input point.
* To test the erase line or erase screen functions, set the cursor to Home
*   to get above other displayed material.
* Set two different terminals side-by-side and compare.
* Reset video to normal before quitting.
*****
* Assign Constants, Initial Values *
*****
PROMPT ""
Clear.Screen$ = @(-1)
CRLF$ = CHAR(13) : CHAR(10)
Escape$ = CHAR(27)
Escape.Key.Pick$ = CHAR(251);      * Standard Pick. May vary.
EQU False TO 0, True TO 1
Next.Flag = False
Quit.Flag = False
Message$ = CRLF$ : \Enter ASCII number for character, use @() format\
Message$ = Message$ : \ for function or position, or press\ : CRLF$
Message$ = Message$ : \ESCAPE then enter code. Press ENTER again to\
Message$ = Message$ : \ repeat. Enter NEXT to start again\ : CRLF$
Message$ = Message$ : \or QUIT to quit: \
=====
*****
* Test Functions Loop *
*****
LOOP UNTIL Quit.Flag DO
*****
* Get Request *
*****
CRT Message$ :
INPUT Request$ :
Next.Flag = (Request$ = "NEXT" OR Request$ = "next")
Quit.Flag = (Request$ = "QUIT" OR Request$ = "quit")
IF NOT(Quit.Flag) AND NOT(Next.Flag) THEN
*****
* Assign Video Code *
*****
Ch.1$ = Request$[1, 1]
Ch.2$ = Request$[1, 2]
Ch.6$ = Request$[1, 6]
Length = LEN(Request$)
Ch.Last$ = Request$[Length, 1]
BEGIN CASE
CASE Ch.6$ = "ESCAPE" OR Ch.6$ = "escape";      * Escape
Video.Code$ = Escape$ : Request$[7, 99]
CASE Ch.1$ = Escape$ OR Ch.1$=Escape.Key.Pick$; * Escape
Video.Code$ = Escape$ : Request$[2, 99]
CASE NUM(Request$);      * ASCII Char
Video.Code$ = CHAR(Request$)
CASE Ch.2$ = "@"(" AND Ch.Last$ = ")";      * @()

```

```

        Code$ = Request$[3, Length - 3]
        * Comma in Request$?
        *   Yes: Column, Row.
        *   No: Column or negative number function
        Row = FIELD(Code$, ",", 2)
        IF Row THEN;                                * @(Col, Row)
            Col = FIELD(Code$, ",", 1)
            Video.Code$ = @(Col, Row)
        END ELSE;                                    * Col only or func
            Video.Code$ = @(Code$)
        END
    CASE True;                                        * Literal
        Video.Code$ = Request$
    END CASE
LOOP
    *****
    * Apply Video Code *
    *****
    CRT Video.Code$ :
    *****
    * Repeat or Next *
    *****
    INPUT Request$ :
    Next.Flag = (Request$ = "NEXT" OR Request$ = "next")
    Quit.Flag = (Request$ = "QUIT" OR Request$ = "quit")
    UNTIL Next.Flag OR Quit.Flag DO
    REPEAT
        END; * If Not Quit.Flag and Not Next.Flag
    REPEAT; * Until Quit.Flag
    *****
    *****
    9999 * End Program *
    *****
    END

```

SCREEN: VIDEO CHARACTERISTICS: BLINKING

IBM PC BASIC

COLOR

For example:

```
COLOR {Foreground} {, {Background} }
```

Adding 16 to the foreground expression produces blinking.

To display a blinking error message:

```

Foreground.White.Normal = 7
Foreground.White.Blinking = Foreground.White.Normal + 16
Background.Black = 0
COLOR Foreground.White.Blinking, Background.Black
PRINT "ERROR";
COLOR Foreground.White.Normal

```

PICK/BASIC

CRT @(-) or PRINT @(-)

To display a blinking error message:

```
Video.Blinking$      = @(-5)
Undo.Video.Blinking$ = @(-6)
Message$ = Video.Blinking$ : "ERROR" : Undo.Video.Blinking$
CRT Message$
```

SCREEN: VIDEO CHARACTERISTICS: INTENSITY

The effect of the intensity setting varies with the context, and with adjustments to the screen. Pick tends to treat the high intensity setting as the norm, and to display “protected areas” that might not be overwritten by the user in low intensity. IBM PC DOS, however, treats low intensity as the norm, and uses high intensity for emphasis. For uniformity of appearance between systems, you should decide upon a single philosophy regarding intensity.

IBM PC BASIC

COLOR

For example:

```
COLOR {Foreground} {, {Background} }
```

Adding 8 to the foreground expression produces high intensity.

To display an error message in high intensity:

```
Foreground.White.Normal = 7      '* Low Intensity is normal for PC
Foreground.White.Intense = Foreground.White.Normal + 8
Background.Black = 0
COLOR Foreground.White.Intense, Background.Black
PRINT "ERROR";
COLOR Foreground.White.Normal
```

PICK/BASIC

CRT @(-) or PRINT @(-)

To display an error message with high intensity, and then return the setting to low intensity:

```
Video.Intensity.High$ = @(-8)
Video.Intensity.Low$ = @(-7)
Message$ = Video.Intensity.High$ : "ERROR" : Video.Intensity.Low$
CRT Message$
```

SCREEN: VIDEO CHARACTERISTICS: UNDERLINE

IBM PC BASIC

COLOR

For example:

```
COLOR {Foreground} {, {Background} }
```

A foreground expression of 1 produces an underlined foreground.

To display an error message underlined:

```
Foreground.White.Normal = 7      '* Low Intensity is normal for PC
Foreground.White.Underlined = 1
Background.Black = 0
COLOR Foreground.White.Underlined, Background.Black
PRINT "ERROR";
COLOR Foreground.White.Normal
```

Starting with a foreground of 1, adding 8 would produce underlined, high intensity. Adding 16 would produce underlined, blinking. Adding both 8 and 16 would produce underlined, high-intensity, blinking.

PICK/BASIC

CRT @(-) or PRINT @(-)

To display an underlined error message:

```
Video.Underline$      = @(-15)
Undo.Video.Underline$  = @(-16)
Message$ = Video.Underline$ : "ERROR" : Undo.Video.Underline$
CRT Message$
```

SCREEN: VIDEO CHARACTERISTICS: VIDEO REVERSE

Video reverse is used to display characters in the reverse of the normal video mode, for emphasis. The normal mode of the IBM PC is light characters against a dark background. Video reverse will show dark characters on a light background.

IBM PC BASIC

COLOR

For example:

```
COLOR {Foreground} {, {Background} }
```

A foreground expression of 0 and background expression of 7 produces video reverse.

To display an error message in video reverse:

```
Foreground.White = 7
Foreground.Black = 0
Background.Black = 0
Background.White = 7
COLOR Foreground.Black, Background.White  '* Video Reverse
PRINT "ERROR";
COLOR Foreground.White, Background.Black  '* Undo Video Reverse
```

Setting both foreground and background to black causes the characters to become invisible. Adding 16 to the foreground produces video reverse, blinking.

To create a reverse blank character for use on an editing screen to show the difference between the present length of an entry and the maximum allowed length, you could use the reverse video function:

```
Num.Blanks = Max.Length.Entry - LEN(Entry$)
PRINT Entry$;
COLOR Foreground.Black, Background.White
PRINT SPACE$(Num.Blanks);
COLOR Foreground.White, Background.Black
```

You could use the reverse blank character defined in the ASCII character set for the machine. The reverse blank is ASCII 219 on the IBM PC:

```
Blank$ = CHR$(219)
PRINT Entry$; STRING$(Num.Blanks, Blank$);
```

Microsoft BASIC (Radio Shack 100)

The COLOR command is not available in Microsoft BASIC on the Radio Shack 100.

For the Radio Shack 100, the normal display is black characters on a white background. Reverse video will show white on black. On screens with few pixels per character, such as the Radio Shack 100's LCD screen, video reverse looks better if a space is included before and after the message.

```
PRINT CHR$(27) +
```

To display an error message in reverse video:

```
Escape$ = CHR$(27)
Video.Reverse$ = Escape$ + "p"
Undo.Video.Reverse$ = Escape$ + "q"
Message$ = Video.Reverse$ + " ERROR " + Undo.Video.Reverse$
PRINT Message$
```

Rather than concatenating the elements, the reverse video mode can be turned on, a series of messages printed, and then the mode turned off:

```
PRINT Video.Reverse$; Message.1$; Message.2$; Undo.Video.Reverse$
```

To create a reverse blank character for use on an editing screen, to show the difference between the present length of an entry and the maximum allowed length, you could use the reverse video function:

```
Num.Blanks = Max.Length.Entry - LEN(Entry$)
PRINT Entry$; Video.Reverse$; SPACE$(Num.Blanks);Undo.Video.Reverse$;
```

Avoid turning the reverse video function on and off repeatedly, with a construction such as:

```
Blank$ = Video.Reverse$ + " " + Undo.Video.Reverse$
PRINT Entry$; STRING$(Num.Blanks, Blank$);
```

For the last statement above, printing 100 blanks would require turning the reverse video function on and off 100 times.

You could use the reverse blank character defined in the ASCII character set for the machine. The reverse blank is ASCII 239 on the Radio Shack 100:

```
Blank$ = CHR$(239)
PRINT Entry$; STRING$(Num.Blanks, Blank$);
```

PICK/BASIC

CRT @(-) or PRINT @(-)

To display an error message in reverse video:

```
Video.Reverse$      = @(-13)
Undo.Video.Reverse$ = @(-14)
Message$ = Video.Reverse$ : " ERROR " : Undo.Video.Reverse$
CRT Message$
```

Rather than concatenating the elements, the reverse video mode can be turned on, a series of messages printed, and then the mode turned off:

```
CRT Video.Reverse$ : Message.1$ : Message.2$ : Undo.Video.Reverse$ :
```

To create a reverse blank character for use on an editing screen to show the difference between the present length of an entry and the maximum allowed length, you could use the reverse video function:

```
Num.Blanks = Max.Length.Entry - LEN(Entry$)
CRT Entry$ : Video.Reverse$ : STR(" ",Num.Blanks):Undo.Video.Reverse$
```

Avoid turning the reverse video function on and off repeatedly, with a construction such as:

```
Blank$ = Video.Reverse$ : " " : Undo.Video.Reverse$  
CRT Entry$; STR(Blank$, Num.Blanks) :
```

For the last statement above, printing 100 blanks would require turning the reverse video on and off 100 times.

You could use the reverse blank character defined in the ASCII character set for the machine. The reverse blank is ASCII 219 on the IBM PC:

```
EQUATE Blank$ TO CHAR(219)  
CRT Entry$ : STR(Blank$, Num.Blanks) :
```

For portability, you could interrogate the terminal type of the machine with the **SYSTEM** function, and set the reverse blank character accordingly. The IBM PC Terminal is type "I":

```
Term.Type$ = SYSTEM(7)  
IF Term.Type$ = "I" THEN  
    EQUATE Blank$ TO CHAR(219)  
END ELSE  
    Blank$ = Video.Reverse$ : " " : Undo.Video.Reverse$  
END
```

(For use of the **SYSTEM** function with the screen, see the section: "Screen: Access System Information.")

Size, Speed of Program: Advice



THE FOLLOWING ADVICE WILL HELP YOU TO REDUCE THE SIZE AND INCREASE the speed of your programs.

All Languages

Structure and Documentation

It is important that the files, data structures, and programs have the proper design. You should thoroughly document the intended use of each file and routine. The system that is well structured and documented will be faster and tighter in the long run, because changes and additions can be made in a logical way, rather than appended to sections that the programmer is afraid to change because he does not understand how they work.

Avoid cluttering the disk with unneeded files. The fuller the disk, the more fragmented files tend to become. Keep documentation that explains the purpose of each file. For programs, PROCs, and batch files, put a note explaining the purpose in the second line. Review the documentation and the filenames displayed with the LISTFILES command at Pick TCL or the DIR command in IBM PC DOS, and remove the files that are no longer needed.

Repetition

For loops that will execute many times, test different versions with the Test Speed program in the section "Size, Speed of Program: Test Speed."

Avoid any unnecessary operations within a loop that will execute many times. Rather than:

```
FOR I = 1 TO 100
    Search.Char$ = MID$(Search$, I, 1)
    IF Search.Char$ = "x" THEN ...
NEXT
```

use the following code that is less clear to read, but avoids the extra assignment statement:

```
FOR I = 1 TO 100
    IF MID$(Search$, I, 1) = "x" THEN ...
NEXT
```

If the code was not in a loop, the first version would be preferable, for clarity of documentation.

FOR...NEXT loops are faster than using IF ... GOTO, WHILE or UNTIL structures for the same purpose.

Test the particular implementation of FOR...NEXT to find out if the condition is evaluated at the top or bottom of the loop, to ascertain whether the loop will execute once when the starting value is already greater than the ending value. (See the section: "Repetition.")

Testing Conditions

Testing a Boolean flag for true or false is faster and usually shorter than comparing a variable to a literal or to another variable. Any test that must be performed repeatedly should therefore be done by testing a Boolean flag if this is possible. For example, rather than asking:

```
IF Error.Num > 0 THEN ...
```

have the error recovery procedure set Error.Flag to true if there is an error, and test:

```
IF Error.Flag THEN ...
```

Boolean flags also can be used in place of IF...THEN statements to avoid breaking the flow of a line. (See the section: "Boolean Flags.")

A positive test is faster and shorter than a negative test. Other factors being equal, it is better to ask:

```
IF Error.Flag THEN ...
```

than to ask:

```
IF NOT Error.Flag THEN ...
```

Comparisons of numbers generally take less time than comparisons of strings.

To perform a series of tests, rather than using the IF...THEN...ELSE structure, use the CASE statement (in Pick or QuickBASIC) or a computed GOSUB (in Pick or IBM PC BASIC). (See the section: "Conditions.")

Whenever a series of tests is required, place the most likely occurrences at the beginning.

Consider using separate tests rather than combining tests with "AND" or "OR." For example:

```
IF Test.Flag.1 AND Test.Flag.2 AND Test.Flag.3 THEN ...
```

The program might evaluate all three flags, rather than stopping if Test.Flag.1 is false. (Some compilers will "optimize" the process automatically to avoid the unnecessary tests, if the tests are not too complicated.) If Test.Flag.1 were false 90% of the time, the program might be doing a great deal of unnecessary work. The statement could be rewritten as:

```
IF Test.Flag.1 THEN IF Test.Flag.2 THEN IF Test.Flag.3 THEN ...
```

The test that would most often avoid execution of the remainder of the statement should be placed first. If two tests are approximately equal in their likelihood, place the test that takes the least time first. (See the section: "Conditions.")

Avoid Repeating Functions

Avoid using the same function twice. Thus rather than obtaining the length of the string Entry\$ twice, place the result in a variable, and use the variable:

```
Length.Entry = LEN(Entry$)
```

Make sure that the variable Length.Entry is not used after the actual length of Entry\$ has changed without a new assignment to get the correct length.

Screen Display

Screen display is generally slower than processing (particularly when using an LCD display or a serial port terminal), so avoid any unnecessary printing to the screen. For a data entry routine, when changing one character in an entry, redisplay only that character, at its position within the entry, rather than the entire entry. For editing, over-type mode will be faster than insert mode, because Over-Type will only require redisplay of a single character at a time.

When deleting characters, which will require bringing forward the remaining part of the entry to the current cursor position, redisplay only the portion of the field that is affected. The variable Length.Entry in the example was obtained before the deletion:

In IBM PC BASIC:

```
PRINT MID$(Entry$, Pos.in.Entry + 1, Length.Entry - Pos.in.Entry);
```

In PICK/BASIC:

CRT Entry\$[Pos.in.Entry + 1, Length.Entry - Pos.in.Entry] :

If screen output and processing are to be performed at the same time, perform the output first, so that the user can be reading the screen while the processing is taking place.

Variables

Give initial values to variables at the beginning of the program, in the section entitled "Assign Constants, Parameters, Initial Values" in the program examples. In programs with many variables, test for speed differences if the most speed-critical variables are initialized first, using the Test Speed program in the section "Size, Speed of Program: Test Speed."

Do not create new variables unnecessarily. If a variable is used only within a single routine, replace the global variable with a local variable.

A global variable must be valid globally, throughout the program, because a global variable is passed between routines. Therefore each global variable must have a distinct name. If the same name is used for different purposes, unintended "side effects" will occur when the value of the variable is changed by different routines. Consequently, as a program grows in size, the number of distinct variables grows larger.

A local variable need be valid only within a single routine, because a local variable is never passed from one routine to another. Therefore the same names can be reused repeatedly for local variables, reducing the total number of distinct variables used by the program. Pascal will restrict the scope of a local variable to the subroutine in which it is assigned, therefore in Pascal there is no reason to limit the names used for local variables. If a language does not have that capability, reserve strictly for local use some variable names such as I and J for counters, U and V for numbers, and X\$, Y\$, and Z\$ for strings. These letters were chosen because few words that would be used in meaningful variable names would begin with these characters. Never, never pass the designated local variables between routines.

Make sure that all local variables are assigned values each time before they are used. Thus do not begin a routine with a statement such as:

```
X$ = X$ + Added.Value$  
U = U + 1
```

You first would have to set the string variable X\$ to null and the numeric variable U to zero.

In the documentation, state the purpose of the local variable within each separate routine. The following "Split String by Delimiter" routine is presented in IBM PC BASIC.

Split String by Delimiter Routine

```
*****  
* Split String by Delimiter *  
*****  
* David L. Clark 09/22/1988
```

```

'* IBM PC Basic routine. Documentation version.
'* IBM PC BASICA 3.0, Radio Shack 100
'* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
'* Parameters: Delim$, Search$, Pos.Start
'* Returns: Before.Delim$, After.Delim$, Delim.Found.Flag
'* Local variables:
'*   U = Pos.Delim : Position of the delimiter Delim$ within Search$
Pos.Delim = INSTR(Pos.Start, Search$, Delim$)
:   Delim.Found.Flag = (Pos.Delim > 0)
:   IF Delim.Found.Flag THEN
:       Before.Delim$ = LEFT$(Search$, Pos.Delim - 1)
:       After.Delim$  = MID$(Search$, Pos.Delim + 1, 255)
:   ELSE
:       Before.Delim$ = Search$
:       After.Delim$  = ""
'* End
RETURN

```

In the executable code, the variable name “U” would be used in place of the “Pos.Delim” shown in the documentation.

With regard to string variables especially, it is very wasteful to create a new global variable, and assign a long string to it, if that variable is needed only in the routine in which it is assigned. For any string variable that might hold a long value, it is essential to decide if the variable can be kept purely local, with each value it holds used only in the routine in which that value is assigned, or if the variable must be global, to be used for only one purpose throughout the program. Clear large string variables to null when they are not needed by a program.

Constants

If the same string or numeric value will be used repeatedly, assign it to a constant rather than repeating the literal value.

String Functions

Simple addition and subtraction generally take less time than string functions, and certainly less time than any function that requires reading data from a file. Thus if two strings of known length are concatenated, add their lengths together, rather than using the LEN function to find the length of the new string.

Concatenation

Avoid the concatenation of strings, which is slow, when other methods are possible, such as the replacement of individual characters within a string.

Internal and External Subroutines

When a language offers a choice between internal and external subroutines, branching to an internal subroutine will be faster. Therefore speed-critical operations such as data

entry and editing should be consolidated into a single module that requires no external branching. Operations that are limited by hardware speed, such as printing, are the best candidates for external placement.

IBM PC BASIC

Program Size

Reducing the size of an interpreted (rather than compiled) program will increase its speed. Therefore remove all remarks except the program heading from the final code. Keep a separate copy of the documentation for the program.

When considering the size of a program, recall that the BASIC interpreter uses special symbols for keywords. Therefore although the command "PRINT" takes five characters in a text file, in the BASIC program it will only take one character.

Variables

Make all variable names as short as possible. Each character in the name takes a byte. The names used most often should be one character with no type ending, as explained in the next point.

Use definition statements for all the variables. Define all the variables, except for a few strings, as integers:

```
DEFINT A-W  
DEFSTR X-Z
```

Because the variables are predefined, it will not be necessary to use the type endings A%, X\$, and so on.

Use integers, which require two bytes for storage, rather than single-precision reals, which require four bytes, or double-precision reals, which take eight bytes. The default assumption for any variable without a type ending or definition statement is that it is double-precision. Therefore define all the variables. Integers are much faster than reals. The loop "FOR I = 1 TO Total" will execute 2½ times more quickly if the counter "I" and the final value "Total" are integers.

Place any string literal used more than once, and more than two characters long, into a variable.

IBM PC BASIC searches for variables in the order in which they were created, therefore initialize all single speed-critical string and numeric variables when the program first begins. The routine in which the variables are initialized need not occur at the physical beginning of the program. The first line of the program can be a GOTO or GOSUB statement to an assignment section placed at the physical end of the program. Since the assignment section will run only once, it makes sense to place it after all the sections that run repeatedly.

If the program has many variables, access will be three or more times faster to those initialized first. It is not necessary to reserve space for a string variable. Numeric variables of a given type always take the same amount of space.

To initialize those variables that simply can be set to zero or null, rather than using:

```
U   = 0
X$  = ""
```

instead use the much shorter DIM statement, normally employed to dimension fixed arrays. DIM will also initialize simple variables to null. Thus a program might begin with the following definition and initialization of the most frequently used variables:

```
DEFINT A-W
DEFSTR X-Z
DIM I, J, U, V, X, Y, Z
```

You might initialize all the simple variables used in a program in this way, greatly aiding in the documentation of the program. In addition, some speed advantage would be gained, because when a new simple numeric variable is created all of the existing arrays must be moved over to make room for it. However, initialization of all simple variables can take up more space than the program can afford, particularly on the Radio Shack 100.

The temporary variables I and J, generally used as loop counters, should be among those given priority initialization. The counters will not be affected, strictly speaking, in their role as counters, but through the other uses to which each counter is put. Thus there would be no speed difference in the loop:

```
FOR I = Start TO Total
  X = X + Y
NEXT
```

but there would be a difference in:

```
FOR I = Start TO Total
  X(I) = Y(I)
NEXT
```

In the FOR...NEXT examples, the speed advantage to be gained by giving the variables start and total priority initialization probably would not be significant because the gain would take place only once, when the loop began, rather than on each pass through the loop. Of course, if this loop were to be nested within another loop, the importance of fast access to these variables would increase.

The significance of a speed advantage for a variable depends upon the number of times that the variable will be accessed.

Variables initialized at the start of a program and literal numbers or strings give the same speed results. A literal should only be used, however, when it is certain that you will never need to change its value. Otherwise, the programmer will have to hunt through the program for every occurrence of the literal value to change.

Use the Test Speed program in the section "Size, Speed of Program: Test Speed" to confirm the speed differences from variable initialization that affect a given program. In the assign constants portion of the Test Speed program insert the corresponding part of the

program to be tested. The greater the number of variables, the greater the speed difference between early and late initialization. Place identical copies of the speed-critical code in the Test Speed areas designated for program versions #1 and #2. Then change a single variable in the code. For version #1, use a variable initialized before the others. In version #2, use a variable initialized later.

Arrays

An array variable is initialized when it is dimensioned with the DIM statement. An array variable will perform faster if its array is dimensioned before the other arrays rather than after. However, the difference is usually less significant than for single variables because there are generally far more single variables than there are arrays.

If both are initialized at the beginning of the program, a single variable will be nearly twice as fast as an array variable. However, a single variable initialized after many others might be only half as fast as an array variable that was not preceded by many other arrays.

Use the ERASE statement to erase from memory arrays that are no longer needed and to redimension arrays. The ERASE command is not available on the Radio Shack 100. (See the section: "Data Structures: Arrays, Fixed: Dimension or Declare.")

Line Numbers

Put as many statements as possible on a single line, up to a maximum length of 255 characters. Each line number takes five bytes, while the colon to separate multiple statements on a line takes only one byte. (For methods to avoid breaking the logic flow of a line see the section: "Boolean Flags: In Place of IF...THEN.")

Keep the line numbers low, because each digit in a statement takes a byte. Thus GOSUB 1000 takes two bytes more than GOSUB 10. The required line numbers that appear at the left margin at the beginning of each IBM PC BASIC statement always take the same amount of space, regardless of the number of digits.

Placement of Subroutines

Place speed-critical code at the physical beginning of the program. Branching with GOTO or GOSUB is fastest to portions of code at the beginning and to routines that immediately follow the caller. In a long program, in which the starting portion is already filled with code, place any routine that is called frequently after those routines that call it.

Looping

Remove the last index variable in FOR...NEXT:

```
FOR I = Start TO End
  Print X$(I)
NEXT
```


A FOR...NEXT loop is faster than a loop of IF...GOTO Line.Number, and avoids an extra line number reference. You can break out of a FOR...NEXT loop by resetting the counter variable to the ending value. (See the section: "Repetition.")

Use a FOR...NEXT loop rather than separate statements whenever possible. A loop counter easily can be used to set up a table of characters that follow each other in ASCII sequence. Thus to initialize a table containing the control characters ASCII 0 to 31:

```
FOR I = 0 to 31
    Table.Ctrl.Chars$ = Table.Ctrl.Chars$ + CHR$(I)
NEXT
```

When creating a table, if the characters do not follow each other in immediate sequence, use the DATA and READ statements. Thus to create a table for the keys that move the cursor:

```
DATA 1,4,5,6,12,17,18,19,22,24,25,28,29,30,31
FOR I = 1 TO 15
    READ Value
    Table.Cursor.Keys$ = Table.Cursor.Keys$ + CHR$(Value)
NEXT
```

Note a subtle difference between creating a table entirely through the use of FOR...NEXT, and creating a table with DATA and READ. In the case of DATA and READ, the variable used in the character assignment is not the counter variable for the loop. In the DATA and READ example just shown, the counter variable is "I," but the character value is "Value." If "I" had been used to READ the DATA values, the loop would have aborted as soon as a value was read in greater than 15, the end value for the FOR...NEXT loop.

You are not required to initialize variables with IBM PC BASIC, as in Pick. A routine similar to the preceding one in PICK/BASIC would have required an initial statement setting the Table.Cursor.Keys\$ to null.

Formatting Output

Place formatting in variables, and use the variables rather than repeating the formatting commands. For example, to send data to a file, ending each element with a Carriage Return—Line Feed, rather than:

```
PRINT #1, X$
PRINT #1, Y$
PRINT #1, Z$
```

use:

```
L$ = CHR$(13) + CHR$(10)
PRINT #1, X$, L$, Y$, L$, Z$
```

Separating Elements

IBM PC BASIC requires spaces except between parts of an arithmetic or string expression. Elements to PRINT or LPRINT can be separated by spaces, commas, or semicolons. More than one separator is not required:

```
A = B + C: X$ = LEFT$(Z$, 2): PRINT D$; E$: IF G$ = "End" OR H$ = "Done"  
GOTO 200
```

The last quote mark on a line is unnecessary:

```
PRINT "This is a test  
A$ = "Test
```

THEN and GOTO

When the phrase "THEN GOTO" or "ELSE GOTO" appears, both words are not required. For example:

```
IF Test.Flag GOTO 100 ELSE 200
```

(For details, see the section: "Conditions.")

Spaces and Strings

Use SPACE\$(Num.Chars) to create a string of spaces rather than STRING\$(Num.Chars, 32).

Microsoft BASIC (Radio Shack 100)

Separating Elements

Remove all spaces from the program. However, make sure that two variable names do not inadvertently form a reserved word. For example, eliminating a space between an "O" and "F" produces the reserved word "OF," or between "X" and "OR" creates "XOR." Rather than:

```
IFXORY
```

use:

```
IFYORX
```

When displaying or printing a series of variables and literals, remove the semicolons separating the elements whenever a quote mark or a variable type ending will serve as a separator:

```
PRINT A"Test"B$C$
```

If the variable B\$ did not have a type ending, and was simply B, it would be necessary to separate it from C\$ with a space or semicolon, so that the Radio Shack 100 would know that B and C\$ was not BC\$:

```
PRINT A"Test"B;C$
```

Plan the variable names so that those that will be printed get the type endings.

Clear String Space

On the Radio Shack 100, the CLEAR statement must be used to clear string space memory for a program, if the string variables will total more than 255 characters in length. If the program runs out of string space, Error 14 "Out of String Space" will occur. To save memory, clear only the amount of space that the program will need. To speed up the program, clear more, to reduce the frequency with which the string space must be reorganized.

Variables

Creating a new variable, rather than reassigning an existing variable, takes extra memory space. Assigning a new integer takes five more bytes than using an old one, a new string six more, a new single precision number seven more, and a new double-precision number eleven more. Arrays are even more expensive. Creating a new integer array of 10 elements takes 30 bytes, a new string array 41, a new single-precision array 52, and a new double precision array 96.

Files

The maximum number of files that can be held open at the same time for input and output is declared with the "MAXFILES = Max.Number.of.Files" statement. (See the section: "Files, Items: Open, Close.") Each file buffer requires an extra 267 bytes of memory, therefore keep the number as low as possible. The MAXFILES command causes a CLEAR—all the variables are set to null, and there is no RETURN from a subroutine whose GOSUB preceded the MAXFILES.

Pick

File Size, Modulo

The most important factor affecting speed on a Pick System is the proper sizing of files by setting an appropriate modulo (number of groups into which the items are placed—see the section: "Files, Items: Open, Close.") Test the file modulus periodically with the ISTAT and HASH-TEST commands at Pick TCL level.

Clean Up the Disk

Do a file save and file restore whenever the files have grown significantly in size. The new modulo, placed in attribute 13 of the file definition item, will be applied as part of the file restore process. In addition, the information on the disk will be recompactd, leaving a larger amount of contiguous free space. (For the file-save procedure, see the section: “Files, Items: Copy, Backup.” For choosing a modulo, see the section: “Files, Items: Modulo.”)

Accounting History File

Review the ACC (accounting history) file to see which processes and accounts are taking the most CPU cycles. If some processes take a disproportionate share of processing power, try to make those procedures more efficient, or schedule them so that they do not occur when users are entering or editing data, and are apt to complain about slow system response.

Opening Files

Open a file only once. PICK/BASIC allows file variables to be passed to external subroutines, therefore the subroutines do not need to reopen the files. There is no limit to the number of files that can be open at the same time. (For a routine to avoid reopening data files, see the section: “Files, Items: Open, Close.”)

Master Dictionary

The MD is accessed more often than any other file, so make sure it has an adequate modulo. Keep the number and length of items in the MD of the account to a minimum. Do not place PROCs, particularly those more than a few lines long, in the MD. Instead place in the MD a pointer to a library of PROCs. For example, for a PROC named “Sample” and a PROC library named “Proc.Library,” the MD would contain the following item, transferring control to the PROC library:

```
Item ID: Sample  
01 PQ  
02 (Proc.Library)
```

The PROC library will contain an item with the item ID “Sample” that holds the actual PROC to execute.

Attributes

To optimize the speed of searching a database, place the attributes that are shorter and that are interrogated most frequently at the beginning of the item, and the longer attributes that are examined less often at the end.

Use multiple values in an attribute only for repeated information of the same type, such as subject terms, many of which might apply to a single item.

Index Files

If most searches depend on one or two attributes, create index files. For example, if most searching is done by Library of Congress subject headings, create at least a simple index file of items in which each subject term is used as the item ID of an item that contains as separate attributes (or as separate values within an attribute) the item IDs of the items in the data file in which that subject heading appears in the "Subjects" attribute.

Keep a list of the item IDs being processed at any one time in sorted order in a dynamic array, and use LOCATE to retrieve the IDs. Employ the list to allow a user to move back and forth among the items processed, to reedit the previous item, rather than having to conduct a separate search. (For LOCATE, see the section: "Data Structures: Pick: Arrays, Dynamic: Locate.")

Active and Inactive Information

If the distinction can be made easily, put information that is under very active use into a file separate from static information that is not accessed often.

Item Size

Files are most efficient if all the items have the same length. If items have very different lengths, as they are apt to have if they contain textual descriptions, distribution among the groups will be less even, causing more disk reads to find some items. The optimum Pick file would have items of no more than one-half the frame size (500 bytes before open architecture, 1,000 bytes under open architecture).

Avoid very large items. To keep down item size, to avoid redundant information, and for consistency, keep information that tends to stay the same in a separate file, to be accessed by a correlative file translation. For example, for notes taken from sources such as books and archival records, reference the bibliographical source information by a bibliography number, perhaps a multipart number that would indicate the depository, collection, box number, file folder, and so forth. Rather than repeating author, title, or depository, collection, and so on, for each description item, include only the bibliography number.

In pre-open architecture systems, large items cause frame faults as they cross frame boundaries of 500 bytes. Under open architecture the data file will not actually store items more than about 750 bytes, but rather will store a pointer to another disk location. In either case more than one disk read will be necessary to retrieve the item.

Although large items generally are less efficient than small ones, to every rule there is an exception. If an entire series of values must all be read into a program at the same time, one large item will require fewer disk reads than several small items.

Only lock an item when the program is about to change it, particularly if the system locks entire groups rather than individual items.

Item ID

Keep the minimum amount of necessary information in the item ID, which is the key to accessing the item. Structure the item ID for maximum hashing efficiency. Use sequen-

tial numbers for item IDs, if possible, to provide good item distribution among the groups of a file. (Pick decides which group into which to place an item by applying a mathematical formula to the item ID.) Avoid redundant information, especially at the end of the item ID, which will tend to place too many items in the same group. For example, if storing photographs with sizes of 4×5 , 5×7 , and 8×10 , do not place the photo size at the end of the item ID.

If multiple pieces of information are stored in the item ID, place the information in the order that would most often be used for sorting. For example, if photo negatives are stored by collection and then by size, place collection number and then the size of the photo at the beginning of the item ID to produce easily a sorted list of the photos in the best order for retrieval of the negatives.

The following tests were made on a file of 268 items, with four attributes each and an average item size of 69 characters. The modulo of the file was 107.

The file was first tested with item IDs formed by concatenating the first and third attributes, and using the group extract correlative to extract those elements from the item ID. Then the file was tested with sequential numbers for item IDs, and with all of the attributes held in the item, rather than in the item ID. The item IDs made of sequential numbers were no more than three characters in length, whereas the concatenated item IDs ranged from 20 to 30 characters.

The sequential numbers for item IDs produced a much more even distribution of items between groups in the file. The sequential item ID also produced a faster sort, although the sort was less useful because it did not reflect the data held in the item. However, even a sort that included the first attribute was still faster than the concatenated item ID sort.

Time Tests of SSELECT Statement at Pick TCL

- ☐ 9 secs : Sort by Item ID concatenated from first and third Attributes
- ☐ 6 secs : Sort by Item ID of sequential numbers
- ☐ 8 secs : Sort by first Attribute, sequential Item ID
- ☐ 12 secs : Sort by first and third Attributes, sequential Item ID

Searching with Access

The Access SELECT statement is faster than BASIC at searching item IDs.

When using Access to search a file, Access goes through and performs all of the cor-relatives before handling the simple attributes that do not require calculation. Therefore, it might be faster to perform two searches, first on the simple attributes, to produce a select list of a reduced number of items, and finally the search involving attributes that use cor-relatives. (See the sections: "Conversions and Correlatives in Pick" and "Files, Items: Input: Search with Pick Access Processor.")

When performing multiple searches with Access, sorting should be left to the last search statement.

Select Lists

Create select lists for specific purposes. For example, when entering data originally on a portable computer and transferring the information later to a Pick database, store a list of item IDs of the new items. Review the items for approval, using the Pick database facilities not available on the smaller computer, such as checking whether the subject terms used correspond to the approved list of subject terms. As each item is reviewed and approved, remove its item ID from the list. In this manner you can quickly call up the new items for checking without having to search the database.

Save select lists for future use. Keep a library of such lists with documentation on the purpose and date of each list.

Conversions

Be sure to put conversions for output formatting in attribute 7, not 8, where correlatives belong. Correlative operations are performed during a search. Conversions are performed only during output. For example, for a date, just put a "D" in the correlative attribute, not "D/" or "D/2."

Program Size and Frame Faults

For PICK/BASIC programs, the shorter the compiled code, the less often frame faults are likely to occur. (A frame fault is the requirement that a new frame be read from the disk into memory.)

Comments in Code

If a comment appears on a line by itself, the end-of-line mark in the object code requires an extra byte.

Compiler Options

The "A" option of the command to compile a program will output the program code in assembly language format:

```
BASIC Filename Program.Name (A)
```

The assembly code will allow the programmer to compare the relative efficiency of different methods of performing an operation. More instructions usually means slower speed.

All end-of-line marks can be eliminated from the object code by compiling with the "C" option:

```
BASIC File.Name Program.Name (C)
```

However, error messages will be less useful, because they will ascribe all errors to Line 1. As an alternative, keep the end-of-line marks, but put several instructions on each line.

After the program has been thoroughly tested and debugged, compile it with the “S” option:

BASIC Program.Name (S)

The “S” option suppresses creation of the symbol table, which is a table of variables assigned by the compiler to symbolic addresses. The symbol table allows you to discover the values held by variables while debugging, but has no other purpose. Eliminating the table slims down the code by about 25%.

(For the compilation command, see the section: “Program Running and Compiling.” To redirect the output from the screen to an item, for later reference, see the section “Screen Output: Redirection.”)

Placement of Internal Subroutines

Place subroutines close to the routines that call them, to increase the likelihood that both the caller and the subroutine will be in the same frame, thereby avoiding having to load in a new frame when the subroutine is called. The same applies whether branching is by GOSUB or GOTO.

External Subroutines

Branching to an internal subroutine, contained within the program, with GOSUB is 3 to 10 times faster than calling an external subroutine with CALL. The CALL statement also exacts a large overhead cost. Therefore, avoid external calls to routines that are used repeatedly. Use the CALL statement to reduce the size of the compiled code for large programs. Place infrequently used modules, particularly those that are shared between several programs, in external subroutines.

Use the COMMON statement to share variables between a main program and external subroutine rather than passing more than a few parameters. The COMMON statement exacts no overhead cost. Although COMMON allows the variables in each module to have different names, give them the same names unless there is a compelling reason otherwise.

Avoid Repeating Functions

Avoid making the same function call twice, as often occurs with ICONV, OCONV and SYSTEM(). For example, to get the account name and port number of the user:

```
Port.and.Account$ = OCONV(0, U50BB)
Port.Num = FIELD(Port.and.Account$, " ", 1)
Account$ = FIELD(Port.and.Account$, " ", 2)
```

(For the SYSTEM function, see the section: “System: Access System Information.”)

Multiple Statements on One Line

For speed, include statements that use the stack on a single line, separated by the end-of-statement semicolon. In practice, because many statements use the stack, try to include as many statements as practicable on a single line.

Note, however, that in the following test only a small amount of extra speed was gained by placing statements on the same line. The loop executed slightly faster on one line rather than three, taking 32.8 seconds rather than 34 for 10,000 repetitions. The machine used was an IBM PC/XT, with only a single user. The speed difference might be greater with many users.

One line:

```
FOR I = 1 TO 10000; J = I; NEXT I
```

Three lines:

```
FOR I = 1 TO 10000
  J = I
NEXT I
```

See the section on “Conditions” for the differences in IF...THEN...ELSE statements between single line and multiple line forms. Many other statements, such as those used to open or read from a file, use the IF...THEN...ELSE structure.

The DIM, COMMON, EQUATE, and DATA statements take multiple objects and easily can be placed on a single line. For example:

```
COMMON A, B, C(10), ....
```

Variables, Constants, and Literals

Thrashing refers to a situation where the virtual memory requirements of a program and its variables exceed the actual memory of the machine. When this situation occurs, Pick must remove a frame from active memory every time that it needs a new frame, with the result that the machine is slowed down to the speed of disk access. Plan the usage of variables, literals, and constants assigned with the EQUATE statement to minimize memory requirements.

All variables that might not be assigned a value in their first appearance should be initialized, in alphabetical order, in the routine named “Assign Constants, Parameters, Initial Values.” This portion should be run at the start of the program. If a run-time error message of “Variable Not Assigned a Value: Zero Used” is received, the programmer only has to check the initialization routine and add the variable to it. This procedure will avoid redundant initialization statements scattered throughout the program.

Use variables in place of repeated literals, especially long strings. In the compiled program, string literals take up two bytes plus the length of the string. Numeric literals are converted to a six-byte binary form, and take up seven bytes of code. A reference to a variable requires only three bytes. There is some disadvantage in speed to variables, because reading in a different frame is often necessary to access a variable. This disadvantage might be offset by the greater speed efficiency of shorter program length. In addition, the program will be far easier to understand and maintain (and hence more efficient in the long run) if literals are avoided.

Each variable requires 10 bytes in the descriptor table of variables. More variables require more frames for the table, thereby increasing the memory overhead of the program

and the paging of frames to and from disk. Reducing the number of variables might therefore increase the speed of the program. Use the "X" option when compiling to generate a cross-reference listing to spot the variables used only once or twice, as candidates for elimination. (For the compilation command, see the section: "Program Running and Compiling.")

Within a BASIC program, keep data structures as small and simple as possible. Avoid keeping string variables or dynamic arrays hundreds or thousands of bytes long, unless absolutely necessary. Split the string or dynamic array into a dimensioned array. Another technique is to perform a SELECT on a long dynamic array variable or item. The select list created can be used with the READNEXT statement to take in the item or dynamic array one attribute at a time. READNEXT will be faster than reading through a dynamic array, asking for each of 100 attributes.

Maintain the type distinction between numbers and strings, although this is not required by PICK/BASIC. Numbers and strings are stored differently. If a variable is stored as a string but used as a number, or vice versa, the system must make a conversion each time that the variable is used. All input comes originally in string form. To store a string as a number, add zero to it. To store a number as a string, concatenate a null to it:

```
Number = String$ + 0
String$ = Number : " "
```

(To convert between numbers and strings, see the section: "Numbers: Convert.")

To make the distinction between strings and numbers obvious in a program, use the standard BASIC convention of ending the names of string variables with a dollar sign, or define some letters for use only as string variables.

The lengths of the variable names has no effect on object code in PICK/BASIC. Use fully descriptive names.

When a constant created with the EQUATE statement is used in a program, the literal value of that constant is compiled into the code. The result is faster access than would be the case with a variable. However, long strings should not be EQUATED, because each character would take up space. (For the EQUATE statement, see the section: "Constants.")

SPACE and STR

For groups of six spaces or fewer, use a literal, for more, use SPACE(Number.of. Spaces). For a repeating character with nine repetitions or fewer, use a literal, for more, use STR(Char\$, Number.Repeats).

Reading from a File

When inputting data from a file, READV is faster than READ, and READ is faster than MATREAD, therefore use the slower statements only when necessary. If only one attribute is to be retrieved, or only the existence of an item need be tested, use READV. To access more than one attribute use READ and place the item in a dynamic array, rather than employing multiple READV statements. If the information will be accessed frequently, use MATREAD to read the item into a fixed array, which will be faster to access

than a dynamic array. The length of time required to MATREAD an item will vary according to the total size of the item and the number of attributes that it contains. The bigger the item, the slower the MATREAD. (For READV, READ, and MATREAD see the section: "Files, Items: Input.")

When deciding to use READV or READ, consider how much unnecessary information will be loaded into memory by READ. Normally, READ is recommended if you need to access more than two attributes, but consider whether the item contains a large attribute several hundred bytes long that will fill up the active memory.

The READU statement requires more disk activity than READ, because of the locking flags that READU sets. Therefore do not use READU unless the item will be changed.

Arrays, Fixed and Dynamic

A fixed (dimensioned) array is faster than a dynamic array, particularly if the dynamic array is large, because the computer must scan through the dynamic array to find the delimiters. For frequently accessed values, use a dimensioned rather than dynamic array.

A dynamic array is stored as a single string variable in the descriptor table. To find an attribute in a dynamic array, the processor must read through the string, examining characters and counting system delimiters, until the processor finds the correct attribute mark.

Disadvantages of the dimensioned array include extra overhead and greater compiled program size, the need to loop through each element to measure the total size of the array, and the additional time required to read and write the array from and to a file.

A simple variable is even faster than a fixed array variable. Tables of values used frequently by the program, or used for speed-critical operations such as editing, should be placed in dimensioned arrays. For individual values within a table that will be accessed frequently, EQUATE a simple variable to the array element.

When dealing with large dynamic arrays, use the REPLACE statement to replace several values at one time.

With dynamic array operations as with other functions, place the result of an operation into a variable rather than performing the same operation several times. For example, to access individual values within an attribute held in a dynamic array, extract the attribute from the array just once, placing it into a simple variable, and then extract the individual values:

```
Attrib$ = Dyn.Ay<16>
Total.Values = DCOUNT(Attrib$, Value.Mark$)
FOR I = 1 TO Total.Values
    Value$(I) = Attrib$<I>
NEXT I
```

(For more information on fixed and dynamic arrays, see the section: "Data Structures.")

Screen Display

Combine text, screen positions, and video characteristics into a string to display with a single CRT command, rather than using a series of separate CRT statements.

The row and column numbers in the standard cursor placement command “CRT @(Column, Row)” require seven bytes each in the compiled program. Consider alternatives to repeating the column and row designations. For example, if a certain screen position is used often, it can be placed in a variable. If messages to the user are to be displayed on the bottom row, at column 30, that position can be stored in a variable and concatenated into the message:

```
Row.Bottom    = SYSTEM(3)
Col.Message   = 30
Pos.Message$  = @(Col.Message, Row.Bottom)
Message$      = Pos.Message$ : Message.Text$
```

Less space will be required in the program to reference the position variable than would be taken by the literal numbers for the position. Using variables for screen positions is also preferable to literals to maintain consistency and to allow changes throughout the program when necessary. Note that the “@(Column, Row)” position variable in PICK/BASIC should be considered a string, not a number.

An alternative to stating column position for each element to be displayed is to use commas as tab stops. Commas in a CRT statement will cause the elements to be displayed as preset tab positions beginning with zero for the left-hand columns and increasing by 18 positions across the screen.

Another alternative to stating the position is to include Carriage Returns—Line Feeds and buffers made up of a set column position. For example, if an array of choices 1 through 10, held in the variable Choice\$(), are each to be displayed on a separate line, starting at column 12:

```
CRLF$ = CHAR(13) : CHAR(10)
Buffer.Space$ = @(12)
Line.Buffer$ = CRLF$ : Buffer.Space$
Display$ = ""
FOR I = 1 TO 10
    Display$ = Display$ : Line.Buffer$ : Choice$(I)
NEXT I
CRT Display$
```

Put a colon at the end of every CRT command unless a Carriage Return—Line Feed is actually needed. Otherwise, two extra characters will be added to the program code.

If the screen display is unacceptably slow, experiment with hard-coding the formatting instructions for specific machinery by using escape sequences and control characters employed by a particular terminal or by storing formatting instructions as constants with the EQUATE statement rather than as variables with the “=” assignment statement.

Alternative Computer

It is common practice in a multiuser environment to establish a schedule of when the most time-consuming operations that will slow down the entire system can be performed. Consider using a cheap, portable device such as Radio Shack’s 100 to let users enter data whenever (and wherever) they wish, as an alternative to abiding by a schedule.

SIZE, SPEED OF PROGRAM: TEST SPEED

The following program, in each language, will test the speed of two versions of an operation and display the results.

Each version is placed in a separate subroutine. To use the program, do the following:

- 1) Place the two versions of an operation in subroutines 100 (first version) and 200 (second version).
- 2) Set the number of FOR...NEXT repetitions in each subroutine to an amount that will show a significant time difference, without taking all day. In the example shown, 10,000 repetitions are used for a very simple operation that took about 30 seconds in IBM PC BASIC and PICK/BASIC.
- 3) Add any constants or type definitions that the operations need at the locations specified. Make sure not to duplicate the name of a constant or variable already in use.
- 4) Place the results of the test in a special file for later reference. The program comments should include the name and location of the reference file.
- 5) To test more than two versions at one time, increase the constant Total.Versions, increase the dimension of the array Time.Passed(), add extra subroutines in the same format, and place calls to those subroutines in the main program.

IBM PC BASIC

In a long interpreted rather than compiled program, one-time assignments should be placed at the end, speed-critical routines placed at the beginning.

All the remarks should be removed from the executable program, and statements placed behind the line numbers, with no break in the line.

The timing operation in the TSpeed (Test Speed) program assumes that midnight will not pass while the program is running.

TSpeed Program, Documentation Version

Test Speed Program: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

GT!	= Get.Time!	: current time returned by Get Time routine
H!	= Hour!	
I,J	= I,J	: Temporary counters
M	= Minute	
S	= Second	
T\$	= Time.String\$	: Current time as a string "HH:MM:SS"
TB!	= Time.Begin!	
TE!	= Time.End!	
TP()	= Time.Passed()	: Amount of time, in seconds, taken by the routine
TV	= Total.Versions	: A constant: number of versions to be compared
V	= Version	

```

1      '* TSpeed.bsd: Test Speed David L. Clark 05/02/1987
2      '* Test Speed of 2 versions of program code.
3      '* IBM PC Basic program. Documentation version.
4      '* IBM PC BASICA 3.0, Radio Shack 100
5      '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10     GOTO 9500      '*|Assign Constants, Parameters, Initial Values|
      '*=====*
      '******
      '* Subroutines *
      '******
20     '* Measure Time *
      '******
      '* Parameters: Time.Begin, Version
      '* Returns: Time.Begin, Time.Passed(Version)
      '* Local variables: Time.End
      GOSUB 60      '*|Get Time|
      :   Time.End! = Get.Time!
      :   Time.Passed(Version) = Time.End! - Time.Begin!
      :   Time.Begin! = Time.End!      '* For next call to routine
      :   RETURN
      '* End line
      '*-----*
60     '* Get Time *
      '******
      '* Returns: Get.Time (current time in seconds)
      '* Local variables: Hour, Minute, Second, Time.String$
      Time.String$ = TIME$      '* Time as "HH:MM:SS"
      :   Hour! = VAL(LEFT$(Time.String$, 2)) * 3600
      :   Minute = VAL(MID$(Time.String$, 4, 2)) * 60
      :   Second = VAL(RIGHT$(Time.String$, 2))
      :   Get.Time! = Hour! + Minute + Second
      :   RETURN
      '* End line
      '*=====*
      '******
      '* Versions to Test for Speed *
      '******
      '******
100    '* First Version *
      '******
      '* Explain operation here:
      '*   For...Next on 1 line
      '* Begin line
      :   FOR I = 1 TO 10000
      :       J = I
      :   NEXT
      '* End line
190    RETURN
      '*-----*
200    '* Second Version *
      '******
      '* Explain operation here:
      '*   For...Next over 3 lines
      FOR I = 1 TO 10000
210        J = I
220    NEXT

```

```

290 RETURN
    '*****
    '*****
6000 '* Main Program *
    '*****
    '* Get Beginning Time
    GOSUB 60 '*|Get Time|
    : Time.Begin! = Get.Time!
    '* End line
6010 '* Test First Version
    Version = 1
    : GOSUB 100 '*|First Version|
    : GOSUB 20 '*|Measure Time|
    '* End line
6020 '* Test Second Version
    Version = 2
    : GOSUB 200; '*|Second Version|
    : GOSUB 20; '*|Measure Time|
    '* End line
6100 '* Display Results
    '* Begin line
    FOR Version = 1 TO Total.Versions
    : PRINT "Time in seconds for Version#"; Version; " = ";
    : Time.Passed(Version) '* Same line continued
    : NEXT
    : GOTO 9990 '*|End Program|
    '* End line
    '*****
9500 '* Assign Constants, Parameters, Initial Values *
    '*****
    DEFINT A - Z '* Unmarked variables are Integer
    : Total.Versions = 2; '* Change for more than 2 versions
    : DIM Time.Passed(Total.Versions)
    : '* Place additional Constants, Definitions:
    : GOTO 6000 '*|Main Program|
    '* End line
    '*****
9990 '* End Program *
    '*****
END

```

The Executable Version of the Program

```

1 'TSpeed.bas: Test Speed David L. Clark 05/02/1987
5 'Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10 GOTO 9500
20 GOSUB 60 : TE! = GT! : TP(V) = TE! - TB! : TB! = TE! : RETURN
60 T$ = TIME$ : H! = VAL(LEFT$(T$,2)) * 3600 : M = VAL(MID$(T$,4,2)) * 60 : S =
100 FOR I = 1 TO 10000 : J = I : NEXT
190 RETURN
200 FOR I = 1 TO 10000
210 J = I
220 NEXT
290 RETURN
6000 GOSUB 60 : TB! = GT!
6010 V = 1 : GOSUB 100 : GOSUB 20
6020 V = 2 : GOSUB 200 : GOSUB 20

```

```

6100 FOR V = 1 TO TV : PRINT "Time in seconds for Version#"; V; " = ";
      TP(V) : N
9500 DEFINT A-Z : TV = 2 : DIM TP(TV) : GOTO 6000
9990 END

```

PICK/BASIC

The time taken for an operation under the virtual memory system utilized by Pick can vary considerably according to whether a frame holding needed information already resides in active memory (RAM) or must be read in from the disk. Therefore a speed test should be run twice in succession, because the information needed by the routines tested might be in active memory on the first run, but not on the second. If two routines are tested comparatively within a single program, the sequence of the routines also might make a difference. Information might be in active memory for the second routine that was not there for the first.

In a multiuser environment, speed also will vary with the number of other users on the system, and the operations that they are performing. For a useful speed test, try to replicate typical conditions as closely as possible.

TEST.SPEED Program

```

* TEST.SPEED David L. Clark 04/20/1987
* Test Speed of 2 versions of program code.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TSpeed.bp.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Assign Constants, Parameters, Initial Values *
*****
EQU Format$ TO "27";          * 2 decimal places, div by 1000
                               * Default justification is Left
EQU Total.Versions TO 2;      * Change for more than 2 versions
DIM Time.Passed(2);          * Change for more than 2 versions
=====
*****
* Main Program *
*****
* Get Beginning Time
GOSUB 30;                      *|Get Beginning Time|
* Test First Version
Version = 1
GOSUB 100;                     *|First Version|
GOSUB 20;                      *|Measure Time|
* Test Second Version
Version = 2
GOSUB 200;                     *|Second Version|
GOSUB 20;                      *|Measure Time|
* Display Results
FOR Version = 1 TO Total.Versions
  CRT "Time in seconds for Version#" : Version : " = " :
  CRT Time.Passed(Version) Format$
NEXT Version
GOTO 9990;                     *|End Program|
=====

```



```

*****
20  * Measure Time *
*****
* Parameters: Time.Begin, Version
* Returns: Time.Begin, Time.Passed(Version)
* Local variables: Time.End
Time.End = SYSTEM(12)
Time.Passed(Version) = Time.End - Time.Begin
Time.Begin = Time.End;      * For next call to routine
RETURN
*-----*
*****
30  * Get Beginning Time *
*****
* Returns: Time.Begin
Time.Begin = SYSTEM(12)
RETURN
*=====*
*****
100 * First Version *
*****
* Explain operation here:
*   For...Next on 1 line
FOR I = 1 TO 10000; J = I; NEXT I
RETURN
*-----*
*****
200 * Second Version *
*****
* Explain operation here:
*   For...Next over 3 lines
FOR I = 1 TO 10000
    J = I
NEXT I
RETURN
*=====*
*****
9990 * End Program *
*****
END

```

Strings

STRINGS: ASSIGN

IBM PC BASIC

" '* Double Quote

For example:

```
X$ = "This is a string."
```

To make use of the double quote for input or output, use CHR\$(34):

```
Quote$ = CHR$(34)
```

```
PRINT Quote$; "AND"; Quote$; " is a reserved word."
```

The system will display:

```
"AND" is a reserved word.
```

PICK/BASIC

" or ' or \ ; Double quote, right single quote, backslash

For example:

```
X$ = "This is a string."
```

```
Y$ = 'This is a string.'
```

```
Z$ = \This is a string. \
```

When one of the three choices is used as a delimiter, either of the others can be contained within the string:

CRT \ "AND" is a reserved word. \

The system will display:

"AND" is a reserved word.

STRINGS: COMPARE

The same comparison operators can be used with strings as with numbers. The strings will be compared in alphabetical order, from left to right, rather than in numerical sequence, from right to left. The strings will be compared according to the ASCII values of their characters, therefore all of the uppercase letters will be considered "Less Than" any of the lowercase letters, because the uppercase letters have lower ASCII numbers. (For a list of all the operators, see the section: "Operators.")

STRINGS: COMPARE: MATCH PATTERNS

PICK/BASIC

MATCH or MATCHES

For example:

```
Phone.Pattern$ = "3N'-4N"  
IF NOT(Phone.Num$ MATCHES Phone.Pattern$) THEN  
    CRT "Invalid phone number"  
END
```

MATCH and MATCHES have exactly the same usage in testing if a string matches a given pattern. The pattern can be composed of a combination of length and type requirements with literals. The entire pattern is enclosed in double quote marks.

A null pattern requires a null entry. In the last example, if Phone.Pattern\$ were null, a null entry for Phone.Number\$ would be required for the match to succeed.

If a null entry is to be allowed but not required, then the pattern must provide for that possibility with a separate multivalue of null:

```
Pattern$ = 6A]';      * 6 alphabetic characters or null
```

A literal to match can be any literal string enclosed within single quote marks.

The type and length requirements are composed of a literal number followed by a type designation, with no space between:

LengthA	Alphabetic characters only
LengthN	Numeric characters only
LengthX	Any character accepted

If the specified length is zero, any length is allowed of the designated type. A pattern of "0A" would require that the entry be a string composed entirely of alphabetic characters, and thus would serve the same purpose as the ALPHA function. A pattern of "0N" would require a string composed entirely of numeric characters, and thus would serve approximately the same purpose as the NUM function, although the pattern "0N" would not allow a leading minus sign for a negative number, nor would it permit a decimal point. The pattern "0N" would restrict the number to an unsigned integer. The pattern "0N'.0N" would require that the number contain a decimal point. A match requirement composed only of "0X" would serve no purpose, because it would allow any length of any character.

Pattern matching can be applied to numbers treated as strings. Thus a match pattern of "3N" would require the entry of a three-digit number. However, it would be preferable to use range-checking mechanisms instead, to check if the entry truly fell between "100" and "999."

Numerical patterns are usually checked only when they must be divided by literals such as dashes, as in the case of the phone number, or with Social Security numbers, which match the pattern of "3N'-'2N'-'4N."

If no delimiter separates the alphabetical and numerical characters, there would be no delimiter in the pattern specification. Thus a requirement for one alphabetical character followed by four numbers would be rendered "1A4N."

Several alternative possibilities can be included, delimited by value marks. (The Pick System value mark is ASCII 253, and is entered at the keyboard by pressing CTRL-]. The value mark appears on the screen as a right bracket.) For example, you might wish to allow the option of including an area code surrounded by parentheses with a phone number:

```
Phone.Pattern$ = "3N'-'4N]('3N')3N'-'4N"
```

Note the right bracket, representing the value mark, which separates the alternative patterns.

STRINGS: CONCATENATE

Concatenate to attach one string of characters to the end of another.

IBM PC BASIC

+

For example:

```
New$ = Old.1$ + Old.2$
```

To concatenate a number with a string, first convert the number to a string. (See the section: "Numbers: Convert: Number to String.")

PICK/BASIC

: or CAT

For example:

```
New$ = Old.1$ : Old.2$
```

or:

```
New$ = Old.1$ CAT Old.2$
```

Pick allows you to concatenate a number with a string, without first converting the number:

```
New$ = Old.1$ CAT Old.Num
```

(See the section: “Numbers: Convert: Number to String.”)

STRINGS: EXTRACT

Extract commands are used to extract a part of a string. In the following examples, presented in IBM PC BASIC, the variable New\$ will contain the characters extracted from Old\$. For example:

```
New$ = LEFT$(Old$, 3)
```

The variable New\$ will contain the three left-most characters of Old\$.

In both IBM PC BASIC and PICK/BASIC, New\$ and Old\$ can be different variables, or the same variable. A new variable would be used if one wanted to preserve the value of the old variable, to use it for other purposes.

If the variable Time.Now\$ contained the time in the format “HH:MM:SS,” you could eliminate the seconds from the string by taking just the left-most five characters, which contain the hours and minutes. This shortened time could be stored in a new variable:

```
Hours.Mins$ = LEFT$(Time.Now$, 5)
```

or in the same variable, which is thereby changed:

```
Time.Now$ = LEFT$(Time.Now$, 5)
```

IBM PC BASIC

For all the string extraction functions, if the Old\$ string from which the extraction is to be made is null, or if the Num.Chars to extract is zero, there is no error, and a null string is returned. If Num.Chars is negative, there is a function call error.

MID\$

For example:

```
New$ = MID$(Old$, Pos.Start {, Num.Chars} )
```

New\$ will contain characters from Old\$, beginning at Pos.Start, up to the optional Num.Chars.

If Pos.Start is zero or negative or beyond the 255 character limit for the length of a string, there is a function call error. If Pos.Start is greater than the length of Old\$, but within the 255 character limit, a null string is returned without error. Note however, that this null string might produce unexpected results if it is next used in a search, because every string is considered to contain the null string. (For the INSTR searching statement, see the section: "Strings: Find Position of Substring.")

LEFT\$

For example:

```
New$ = LEFT$(Old$, Num.Chars)
```

New\$ will contain the left-most characters of Old\$, up to Num.Chars

RIGHT\$

For example:

```
New$ = RIGHT$(Old$, Num.Chars)
```

New\$ will contain the right-most characters of Old\$, starting at the right end up to Num.Chars.

PICK/BASIC

[]

For example:

```
New$ = Old$[Pos.Start, Num.Chars]
```

Like IBM PC BASIC's MID\$ function, New\$ will contain characters from Old\$, beginning at Pos.Start, up to Num.Chars. If Pos.Start is greater than the length of Old\$, New\$ will be null. If Pos.Start is negative or zero, a Pos.Start of one will be used. If Num.Chars is negative or zero, New\$ will be null, without an error. If Num.Chars is greater than the remaining number of characters in Old\$, all of remaining characters will be returned.

To simulate IBM PC BASIC's LEFT\$ function:

```
New$ = Old$[1, Num.Chars]
```

To simulate IBM PC BASIC's RIGHT\$ function:

```
New$ = Old$[LEN(Old$) - (Num.Chars - 1), Num.Chars]
```

To find if a string contains a space and, if so, to extract the portion of the string after the space:

```
Pos.Space = INDEX(Search, " ", 1)  
IF Pos.Space THEN New$ = Search$[Pos.Space + 1, 9999]
```

With IBM PC BASIC's MID\$ function, the length parameter would have been eliminated from the last example. In PICK/BASIC, it is common to substitute a length parameter greater than the possible length of the string.

FIELD, COL1(), COL2()

For example:

```
Search$      = "First,Second,Third"  
Delimiter$   = ", "  
Element$     = FIELD(Search$, Delimiter$, Occur.Num)  
Pos.Before   = COL1( )  
Pos.After    = COL2( )
```

To extract a series of elements separated by a delimiter use the PICK/BASIC FIELD function, rather than extracting by position, as you would have to do in IBM PC BASIC. The FIELD function will extract a substring from a string according to a designated single character delimiter. In the previous example, Element\$ will contain the string marked off by Delimiter\$, according to the Occur.Num (occurrence number) of the delimiter.

In the example, Search\$ contains three elements divided by two delimiters. If the specified Occur.Num is one, Element\$ will contain the characters up to the first occurrence of the delimiter, thus Element\$ = "First". If Occur.Num is two, Element\$ will contain the characters between the first and second occurrences of the delimiter, thus Element\$ = "Second". If Occur.Num is three, Element\$ will contain the characters after the second occurrence of the delimiter, thus Element\$ = "Third". If Occur.Num is greater than the number of elements, FIELD will return a null string, without causing an error.

COL1() will return the position immediately before the start of the element extracted. If there were a preceding delimiter, COL1() would give its position. In the example, if Occur.Num is 1, COL1() will return 0 as the position before the element.

COL2() will return the position immediately after the element extracted. If there was a following delimiter, COL2() would give its position. In the example, if Occur.Num is three, COL2() will return 19, a number one greater than the total length of Search\$, as the position after the element.

COUNT, DCOUNT

COUNT and DCOUNT are often used with FIELD. COUNT will determine the total number of occurrences of a delimiter or substring within a string. DCOUNT will determine the total number of elements that lie between the delimiters:

```
Search$      = "Words and spaces"  
Delimiter$   = " "  
Total.Delimiters = COUNT(Search$, Delimiter$)  
Total.Elements  = DCOUNT(Search$, Delimiter$)
```

In this example, Total.Delimiters will be set to the number of spaces, which equal two, while Total.Elements will be set to the number of words, which equal three. Elements are counted before the first delimiter and after the last delimiter.

With regard to COUNT, if the delimiter does not exist within Search\$, or Search\$ is null, COUNT returns zero. If Delimiter\$ is null, COUNT will return the length of Search\$, because a null string is considered to match any character.

DCOUNT functions somewhat differently than COUNT. If Search\$ is null, DCOUNT will return zero just as COUNT would. However, if the delimiter is not found within Search\$, and Search\$ is not null, DCOUNT will return one. With no delimiter, DCOUNT interprets the entire string as one element. If Search\$ ends with a delimiter, DCOUNT will count an additional null element after the ending delimiter. If Search\$ begins with a delimiter, DCOUNT will count a null element before the starting delimiter. For example:

```
Delimiter$      = ","
Search$         = ",Second,Third,"
Total.Elements = DCOUNT(Search$, Delimiter$)
First.Element$ = FIELD(Search$, Delimiter$, 1)
Last.Element$  = FIELD(Search$, Delimiter$, Total.Elements)
```

Total.Elements will equal four. First.Element\$ and Last.Element\$ both will be null. Note that this usage corresponds to the interpretation of the comma delimiter in IBM PC BASIC, which will find a null element after a delimiter that ends a line and before a delimiter that begins a line. Thus DCOUNT and FIELD can be used together to convert an IBM PC BASIC file, with elements delimited by commas, for use by PICK/BASIC. (For more details, see the section: "Files, Items: Input: Convert from IBM PC BASIC to Pick.")

DCOUNT often is used to find the number of elements in a dynamic array. (See the section: "Data Structures: Pick: Arrays, Dynamic: Count Elements.")

Note carefully that the FIELD function accepts only a single character as a delimiter. In the following example, elements delimited by a semicolon and space in the string Entry\$ are loaded into a fixed array Fix.Ay\$. Because the FIELD function, used to extract the element, accepts only a single character as delimiter, an additional TRIM function is required to remove the space. If you included the space in the delimiter as ";" rather than " ;," the compiler would report no error, but the elements, after the first, would each include a leading space.

```
Delim$ = "; ";      * Single character delimiter
Total.Elements = DCOUNT(Entry$, Delim$)
FOR I = 1 TO Total.Elements
    Element$ = TRIM(FIELD(Entry$, Delim$, I)); * Trim leading space
    Fix.Ay$(I) = Element$
NEXT I
```

The TRIM function also will remove any trailing spaces or repeating spaces in each element. To avoid those side effects a second FIELD statement could be used instead. However, the first element in the string would have to be treated separately, because the first element would not have a leading space:

```
Delim.1$ = "; ";    * Single character delimiter
Delim.2$ = " ";     * " " " "
Total.Elements = DCOUNT(Entry$, Delim.1$)
Fix.Ay$(1) = FIELD(Entry$, Delim.1$, 1); * No leading space on 1st
FOR I = 2 TO Total.Elements
```



```

      Element$ = FIELD(Entry$, Delim.1$, 1)
      Element$ = FIELD(Element$, Delim.2$, 2);      * After leading space
      Fix.Ay$(I) = Element$
NEXT I

```

The first method, using TRIM to remove the leading space, normally would be preferred. That method would work even if the operator forgot to include a space or included unwanted redundant spaces.

After a FIELD operation the system variable COL1() holds the position of the delimiter preceding the substring and COL2() holds the position of the delimiter trailing the substring. The first character of the substring will be found at COL1() + 1, and the last position of the substring will be found at COL2() - 1.

Using COL1(), COL2() and FIELD, you could delete a substring from a string in the following manner, if the delimiter were a single character, and the occurrence number of the substring were known:

```

      Element$ = FIELD(Entry$, Delim$, Occur.Num) ;      * Extract
      Entry$ = Entry$[1,COL1( ) : Entry$[COL2( ) + 1, 9999];      * Delete

```

The operation included the preceding delimiter in the new Entry\$, and excluded the trailing delimiter.

The following routine "Set Name without Spaces" takes a name such as "Description of Photos," which describes the kind of data held in a file, and changes the spaces to periods, yielding "Description.of.Photos". This name code then could be used as a filename or as part of a filename or item ID that indicated the relation.

Set Name without Spaces Routine

```

*****
* Set Name without Spaces *
*****
* David L. Clark 09/22/1988
* Change spaces in Name$ to periods in Name.Code$.
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: SetwoSpc.bpr
* Parameters: Name$
* Returns: Name.Code$
* Temporary variables:
*   Total.Elements : Elements delimited by spaces in Name$.
*                   If no spaces = 1.
Total.Elements = DCOUNT(Name$, " ")
* Must not put a period before the 1st element or after the last.
* 1st element, no period.
Name.Code$ = FIELD(Name$, " ", 1)
* Remaining elements, each preceded by a period.
* If Total.Elements = 1, Pick Basic will not execute the loop.
FOR I = 2 TO Total.Elements
  Name.Code$ = "." : FIELD(Name$, " ", I)
NEXT I
RETURN

```

The next routine, "Delete Control Characters," will delete control characters from a string. First the "MCP" conversion will convert the control characters to periods. ("MCP" does not stand for "Male Chauvinist Pig." It represents "Mask Character Period.")

After conversion the routine will search for periods and remove them. To avoid removing real periods from the original string Orig\$, the corresponding character position in Orig\$ will be checked for a period. For the character positions to correspond between the changed string Changed\$ and Orig\$, the number of control characters already cut must be taken into account, for the position in Changed\$ will differ from the position in Orig\$ by that amount.

Delete Control Characters Routine

```
*****
* Delete Control Characters *
*****
* David L. Clark 09/09/1989
* Delete control characters from a string.
* Pick Basic routine.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: DelCtrCh.bpr
* Parameters: Orig$ : the original string
* Returns: Changed$ : without control characters
Changed$ = OCONV(Orig$, "MCP"); * Change control chars to periods
Length = LEN(Changed$); * Length still same for both
Occur.Num = 1
Chars.Cut = 0; * Keep track of number of chars cut
LOOP
  Pos = INDEX(Changed$, ".", Occur.Num); * Position of period
  WHILE Pos DO
    IF Orig$(Pos + Chars.Cut, 1) # "." THEN
      Chars.Cut = Chars.Cut + 1
      Changed$ = Changed$[1, Pos - 1] : Changed$[Pos + 1, Length]
    END ELSE
      Occur.Num = Occur.Num + 1
    END
  REPEAT
```

STRINGS: FIND LENGTH

This section will cover ways on how to find the length of a string.

Video characteristics stored in variables and concatenated with a string can be included in the length, giving spurious results. For example, a prompt message in reverse video might be created in PICK/BASIC in the following manner:

```
* Set Message
Video.Reverse$      = @(-13)
Undo.Video.Reverse$ = @(-14)
Message$ = Video.Reverse$ : "Pick a Number:" : Undo.Video.Reverse$
```

The program then might branch to a data entry routine to accept the input. The entry routine would need to measure the length of the message, to determine the base position for data entry cursor control:

```
* Data Entry
Length.Message = LEN(Message$)
Base.Pos      = Col.Message + Length.Message
```

The measured length would be greater than it should be, because the escape codes used to set and undo video reverse would be included with the text characters in the message. The length of the escape codes would vary between terminals, and so could not easily be accounted for.

Because the length measurement was incorrect, the data entry routine would begin with the cursor positioned several extra spaces after the end of the message. The length of a message should be taken before any video characteristics have been concatenated with the text.

IBM PC BASIC

LEN

For example:

```
Length.Entry = LEN(Entry$)
```

Addition is about 25 percent faster than the LEN function. Thus if you were adding a single character to the string Entry\$ in a data entry routine, it would be faster to compute the new length with:

```
Length.Entry = Length.Entry + 1
```

than by using:

```
Length.Entry = LEN(Entry$)
```

PICK/BASIC

LEN

For example:

```
Length.Entry = LEN(Entry$)
```

Addition is about 25 percent faster than the LEN function. Thus if you were adding a single character to the string Entry\$ in a data entry routine, it would be faster to compute the new length with:

```
Length.Entry = Length.Entry + 1
```

than by using:

```
Length.Entry = LEN(Entry$)
```

STRINGS: FIND POSITION OF SUBSTRING

IBM PC BASIC

INSTR

For example:

```
Pos.Match = INSTR( {Pos.Start,} Search$, Match$)
```

If Match\$ is contained in Search\$, Pos.Match will contain the starting position. If Match\$ is not found, Pos.Match will be zero. Pos.Start, from which to start the search, is optional.

If Match\$ is null and Search\$ is not null, INSTR will always return a value of one, because any string is considered to contain a null. If both strings are null, INSTR returns a zero.

A null Match\$ string can produce incorrect results if it is not anticipated that Match\$ can be null. For example, a null Match\$ can occur because of the prior use of a MID\$ statement in which a search extended beyond the length of the string. To determine if a given character in Test.String\$ exists within Search\$:

```
Search$ = "String to search"
Test.String$ = "1234"
Pos.Start = 5      '* Beyond length of Test.String$
Num.Chars = 1
Match$ = MID$(Test.String$, Pos.Start, Num.Chars) '* Match$ = Null
Test.Flag = INSTR(Search$, Match$)      '* Test.Flag = True
```

The Pos.Start for MID\$ should be limited to the actual length of the string.

The problem occurs most frequently when checking for a pair of characters. For example, the following "Find Word to Right" routine looks for the next word to the right within the string Entry\$, beginning at the current Pos.in.Entry.

Find Word to Right Routine

```
'*****
'* Find Word to Right *
'*****
'* David L. Clark 09/22/1988
'* IBM PC Basic routine. Documentation version.
'* IBM PC BASICA 3.0, Radio Shack 100
'* Stored under filename: FndWrdRt.brd
'* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
'* Parameters: Entry$, Pos.in.Entry
'* Returns: Pos.Word.to.Rt : Starting position of next word to right
'* Local variables:
'*   Length.Entry : Length of Entry$
'*   Search.Limit : Length.Entry - 1. Don't try to look beyond.
'*   Delim.Chars$ : Space or Dash as delimiters between words
Delim.Chars$ = " -"
:   Length.Entry = LEN(Entry$)
```

```

:   Search.Limit = Length.Entry - 1
:   FOR I = Pos.in.Entry to Search.Limit
:       '* Find a Delimiter that is not followed by a Delimiter.
:       IF INSTR(Delim.Chars$, MID$(Entry$, I, 1 ))
:           AND INSTR(Delim.Chars$, MID$(Entry$, I+1, 1 )) = 0 THEN
:           Pos.Word.to.Rt = I + 1
:           I = Search.Limit           '* Break loop
:       NEXT I
:       RETURN
:   ELSE
:       NEXT I
:       Pos.Word.to.Rt = 0
:       RETURN
:   '* End: If Instr ...
: '* End: For I
'* End line

```

PICK/BASIC

INDEX

For example:

```
Pos.Match = INDEX(Search$, Match$, Occur.Num)
```

If the occurrence specified for Match\$ within Search\$ exists, Pos.Match will contain the starting position, otherwise Pos.Match will be zero.

FIELD, COL1(), COL2()

For example:

```

Element$ = FIELD(Search$, Delimiter$, Occur.Num)
Pos.Before.Element = COL1( )
Pos.After.Element  = COL2( )

```

Element\$ will contain the substring marked off by the Delimiter\$, according to the Occur.Num (occurrence number) of the Delimiter\$. If the specified Occur.Num is one, Element\$ will contain the characters up to the first occurrence of Delimiter\$. COL1() will return the position immediately before the start of Element\$; COL2() will return the position immediately after Element\$.

STRINGS: REPLACE, INSERT

This command can be used to replace or insert characters in the middle of a string. Direct replacement or insertion is faster and uses less memory space than to divide the string and then concatenate the parts.

IBM PC BASIC

MID\$ =

For example:

```
MID$(Old$, Pos.Start {, Num.Chars} ) = New.Chars$
```

When MID\$ appears on the left side of an expression, it replaces characters in Old\$ with New.Char\$, starting at Pos.Start. Old\$ keeps its original length, therefore MID\$ cannot be used to delete, or to insert a larger string. If New.Chars\$ is shorter than the Num.Chars, then the extra places will be filled with spaces. If New.Chars\$ is longer than the space available within Old\$, then New.Chars\$ will be truncated.

The Num.Chars specification is optional. If present, Num.Chars will limit how much of New.Chars\$ is inserted. Pos.Start must not be larger than the length of Old\$, therefore MID\$ cannot be used to concatenate New.Chars\$ to the end of Old\$.

PICK/BASIC

The MID\$ direct assignment function of IBM PC BASIC is not available in PICK/BASIC. You must divide the string, extracting characters with “[]”, and then concatenate the parts. Dynamic arrays have functions for direct replacement and insertion. (For dynamic arrays, see the section: “Data Structures: Pick: Arrays, Dynamic.”)

The following routine will replace every instance of the substring Old\$ with New\$ in the variable S\$:

```
* Replace Old$ with New$
Occur.Num = 1
Length.Old = LEN(Old$)
LOOP
    Pos.Old = INDEX(S$, Old$, Occur.Num)
    WHILE Pos.Old DO
        S$ = S$[1, Pos.Old - 1] : New$ : S$[Pos.Old + Length.Old, 9999]
        Occur.Num = Occur.Num + 1
    REPEAT
```

STRINGS: SPACES, REPEATING CHARACTER

IBM PC BASIC

SPACE\$

For example:

```
String.of.Spaces$ = SPACE$(Num.Chars)
```

String.of.Spaces\$ will be filled with Num.Chars spaces. If Num.Chars is zero, a null will result without error. If Num.Chars is negative, a function call error will result.

STRINGS

For example:

```
New$ = STRING$(Num.Chars, Fill.Char$)
```

New\$ will be filled with Fill.Char\$. Num.Chars can range from 0 to 255.

If Fill.Char\$ is null, a function call error will result. Fill.Char\$ must be only one character, not several.

PICK/BASIC

SPACE

For example:

```
String.of.Spaces$ = SPACE(Num.Chars)
```

String.of.Spaces\$ will be filled with a Num.Chars space.

TRIM

For example:

```
No.Extra.Spaces$ = TRIM(String$)
```

The TRIM function will delete leading and ending spaces, and more than one space in sequence.

STR

For example:

```
New$ = STR(Fill.String$, Num.Repeats)
```

New\$ will be filled with Num.Repeats of Fill.String\$. Unlike the IBM PC BASIC STRING\$ function, the PICK/BASIC STR function will accept a Fill.String\$ more than one character in length. For example:

```
New$ = STR("Extra ", 3)
```

New\$ would equal "Extra Extra Extra".

The STR function will accept a null Fill.String\$ and a zero or negative number for Num.Chars without error. The result will be null.

Subroutines



SUBROUTINES: INTERNAL

IBM PC BASIC

GOSUB Line.Number, GOTO Line.Number

For example:

```
GOSUB 100  
GOTO 100
```

IBM PC BASIC uses a line number as the subroutine label. The line number cannot be a variable, it must be a literal number, without a decimal point. The lines must be in numerical order, and every line in the program must have a number.

Line numbers are the greatest aggravation in using IBM PC BASIC. They have been eliminated from Microsoft's QuickBASIC.

RETURN

For example:

```
RETURN
```

If GOSUB is used to call a subroutine, you can return to the statement following the calling statement with RETURN. It does not matter if the statement that follows the call is on the next line or the same line. The GOSUB...RETURN method is more structured,

easier to understand, and requires fewer line number references to keep track of than using a GOTO to branch to the subroutine and a second GOTO to get back to the main program.

The following program entitled TSubInt (Test Subroutine Internal) uses GOSUB and RETURN to add two numbers and to display the result.

TSubInt.bas Program

```

1      '* TSubInt.bas: Test Subroutine Internal. David L. Clark 09/08/1988
2      '* Show use of subroutines to add 2 numbers and display result.
3      '* IBM PC Basic program. Executable version.
4      '* IBM PC BASICA 3.0, Radio Shack 100
5      '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10     '* Main program *
11     '* Sends to |Add 2 Numbers|: N1, N2
12     '* Receives from |Add 2 Numbers|: Sum
13     '* Sends to |Display Result|: N1, N2, Sum
20     N1 = 1
30     N2 = 2
40     GOSUB 100                      '* |Add 2 Numbers|
50     GOSUB 200                      '* |Display Result|
99     GOTO 9999                      '* |End Program|
100    '* Add 2 Numbers *
101    '* Parameters: N1, N2
102    '* Returns: Sum
110    Sum = N1 + N2
120    RETURN
199    *-----*
200    '* Display Result *
201    '* Parameters: N1, N2, Sum
202    '* Returns: Nothing. Displays the parameters.
210    PRINT N1; " + "; N2; " = "; Sum
220    RETURN
299    '*=====*
9990   '* End Program *
9999   END

```

All the variables in an IBM PC BASIC program are global, therefore there is no formal method of parameter passing for internal subroutines, as there is in Pascal. The global nature of the variables can create severe problems of side effects if you do not keep strict control over which routines use and change which variables. Therefore each subroutine should begin with an approximation of the Pascal form, to show what parameters a subroutine uses, and what it returns, as shown in the last example.

It is preferable that there be only one entry point into a subroutine, and only one exit point. If speed and memory considerations dictate otherwise, these variations from structured programming should be documented. The convention used in this book is that each subroutine with a separate RETURN is shown separated by a line of dashes. (Major portions of a program are each separated by a line of equal signs.) If one routine is allowed to drop into another, the line of dashes does not appear, and the subroutine title bears the notation {Drop or Call} to indicate that the routine can be entered in two different ways. If there is more than one exit from the routine, the exits are numbered and a notation for each exit is placed against the right margin of the page, in order to stand apart from the code:

```

90      '* Drops into next routine
      '******
100     '* Subroutine * {Drop or Call}
      '******
110     IF ... THEN RETURN          '*** Exit #1 ***}
120     ...
130     ...
140     RETURN                      '*** Exit #2 ***}

```

GOSUB...RETURN subroutines can be nested. You can branch to a subroutine, from which another subroutine is called. Each return line number is placed on a stack that has a Last-In-First-Out structure. Each RETURN command returns to the statement following the most recent GOSUB. Thus you can climb down a ladder of six GOSUB statements, and climb back up the ladder with six RETURN commands. If more GOSUBs than RETURNS are executed, for example, if the program ends after six GOSUBs but only three RETURNS, there is no error. This procedure is not recommended, however, except for early termination caused by a fatal error condition.

Certain statements in a subroutine will clear the portion of memory holding the return address after a GOSUB, such that the program will not remember where to return to. These amnesia-causing commands are CLEAR, RUN, LOAD, MERGE, NEW, CHAIN (on the IBM PC), and MAXFILES (on the Radio Shack 100).

```

RETURN Line.Number
IF Error.Flag THEN RETURN 9000      '*|Error Routines|

```

On the IBM PC the command RETURN Line.Number is an alternative form of subroutine exit. (The line number cannot be specified in Microsoft BASIC on the Radio Shack 100.) The line number, as always, must be a literal, not a variable. From the subroutine, you could RETURN to one line under certain conditions or RETURN to another line under other conditions.

You also could erase the most recent returning line number from the stack with a dummy RETURN Line.Number, where the line number specified was simply the next line in the subroutine. In that case the next RETURN would return, not to the most recent GOSUB, but to the one previous.

The use of RETURN Line.Number is undesirable for the same reasons as GOTO. However, it might be used to handle error conditions when the subroutine would not be able to proceed further.

PICK/BASIC

GOSUB Label.Number, GOTO Label.Number

For example:

```

GOSUB 100
GOTO 100

```

PICK/BASIC uses a number as the subroutine label. (Some Pick implementations allow names rather than numbers to be used.)

The label number cannot be a variable, it must be a literal number. Unlike IBM PC BASIC, the number can be a decimal, such as 100.1 or 100.2, and the labels do not need to be in numerical order. Also unlike IBM PC BASIC, the only lines in a PICK/BASIC program that require numbers are the entry points for subroutines.

RETURN

For example:

```
RETURN
```

If GOSUB is used to branch to the subroutine, you can return to the statement following the calling statement with RETURN. It does not matter if the statement that follows the call is on the next line or the same line. The GOSUB...RETURN method is more structured, easier to understand, and requires fewer label number references to keep track of than using a GOTO to branch to the subroutine and a second GOTO to get back to the main program.

The following TEST.SUBROUTINE.INTERNAL program uses GOSUB and RETURN to add two numbers and to display the result.

TEST.SUBROUTINE.INTERNAL Program

```
* TEST.SUBROUTINE.INTERNAL
* Show use of internal subroutines to add 2 numbers and display result.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TSubInt.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
*****
* Main program *
*****
* Sends to |Add 2 Numbers|: N1, N2
* Receives from |Add 2 Numbers|: Sum
* Sends to |Display Result|: N1, N2, Sum
N1 = 1
N2 = 2
GOSUB 100;                *|Add 2 Numbers|
GOSUB 200;                *|Display Result|
GOTO 9999;                *|End Program|
=====
*****
100 * Add 2 Numbers *
*****
* Parameters: N1, N2
* Returns: Sum
Sum = N1 + N2
RETURN
*-----*
*****
* Display Result *
*****
* Parameters: N1, N2, Sum
* Returns: Nothing. Displays the parameters.
```

```

CRT N1 : " + " : N2 : " = " : Sum
RETURN
=====
*****
9999 * End Program *
*****
END

```

All the variables in a PICK/BASIC program are global, therefore there is no formal method of parameter passing to internal subroutines, as there is in Pascal. The global nature of the variables can create severe problems of side effects if one does not keep strict control over which routines use and change which variables. Therefore each subroutine should begin with an approximation of the Pascal form, to show what parameters a subroutine uses, and what it returns, as shown in the last example.

It is preferable that there be only one entry point into a subroutine, and only one exit point. Any variations from structured programming should be documented. (Such variations should occur much less frequently in PICK/BASIC than in IBM PC BASIC.) The convention used in this book is that each subroutine with a separate RETURN is shown separated by a line of dashes. (Major portions of a program are each separated by a line of equal signs.) If one routine is allowed to drop into another, the line of dashes does not appear, and the subroutine title bears the notation {Drop or Call} to indicate that the routine can be entered in two different ways. If there is more than one exit from the routine, the exits are numbered and a notation for each exit is placed against the right margin of the page, in order to stand apart from the code:

```

      * Drops into next routine
      *****
100  * Subroutine * {Drop or Call}
      *****
      IF ... THEN RETURN;                      *** Exit #1 ***
      ...
      ...
      RETURN;                                  *** Exit #2 ***

```

GOSUB...RETURN subroutines can be nested. Thus you can branch to a subroutine, from which another subroutine is called. Each return label number is placed on a stack that has a Last-In-First-Out structure. Each RETURN command returns to the statement following the most recent GOSUB. Thus you can climb down a ladder of six GOSUB statements, and climb back up the ladder with six RETURN commands. If more GOSUBs than RETURNs are executed, for example, if the program ends after six GOSUBs but only three RETURNs, there is no error. However this procedure is not recommended, except for early termination caused by a fatal error condition.

RETURN TO Label.Number

For example:

```
IF Error.Flag THEN RETURN TO 9000;    *|Error Routines|
```

RETURN TO Label.Number is an alternative form of subroutine exit. From the subroutine, you could RETURN TO one point under certain conditions or RETURN TO another point under other conditions.

You also could erase the most recent returning label number from the stack with a dummy RETURN TO Label.Number, where the label number specified was simply the next line in the subroutine. In that case the next RETURN would return, not to the most recent GOSUB, but to the one previous.

The use of RETURN TO is undesirable for the same reasons as GOTO. However, it might be used to handle error conditions when the subroutine would not be able to proceed further.

SUBROUTINES: EXTERNAL

PICK/BASIC

CALL, SUBROUTINE

For example:

```
CALL          External.Subroutine.Name(Param.1, Param.2, ...)
SUBROUTINE    External.Subroutine.Name(Param.1, Param.2, ...)
```

CALL is used to call an external function or procedure. For example:

* SHOW.ADDITION: Main Program that calls a subroutine.

```
N1 = 1
N2 = 2
CALL ADD(Sum, N1, N2)
CRT N1: " + " : N2 : " = " : Sum
END
```

```
SUBROUTINE ADD(Result, Num.1, Num.2)
```

* Subroutine to add 2 numbers and pass back Result.

```
Result = Num.1 + Num.2
RETURN
END
```

The first line of the subroutine must begin with the word "SUBROUTINE", followed by the name of the subroutine and any parameters in parentheses. Any initial comments used to document the subroutine must, therefore, start on the second line. The external subroutine also must contain a RETURN statement, to return to the main program, and an END statement. Within the external subroutine, you can use GOSUB and RETURN for internal subroutines, as in any program. When a RETURN without a corresponding GOSUB is encountered, control will pass back to the main program. If the external subroutine reaches the END statement, control does not return to the main program. The subroutine is compiled with the BASIC statement, then cataloged with the CATALOG statement, like a full program.

When passing parameters, it is not necessary to use the same variable names in the main program and the subroutine. However, the order of the variables must correspond. The changed values of the parameters are passed back to the calling program.

The CALL function adds a good deal of overhead to a program, and so should be used sparingly. CALL should be employed for utility routines that are used by several different programs, and that are used infrequently by those programs. In addition, because versions of the Pick System before open architecture restrict the size of an item to 32K, and program code must fit into a single item, CALL can be employed to spread the program over several items.

COMMON

Use COMMON to share data between programs and external subroutines, without passing specific parameters:

```
* Main Program
COMMON U, V, X$, Y$, Z$

SUBROUTINE Sub.Name
COMMON A, B, C$, D$, E$
```

As with passed parameters, the names of the variables can be different. Their values are shared according to their order. When used with CHAIN or ENTER, the (I) option must be included with the RUN command to prevent reinitialization of the variables.

Although the COMMON statement in the external subroutine does not have to contain all the variables included in the COMMON statement of the main program, it is better structured programming practice that they be the same. Unless there is a compelling reason otherwise, use the same variable names. You can keep a single copy of the COMMON statement and insert it into the individual modules when they are to be compiled.

ENTER

For example:

```
ENTER Program.Name
```

The ENTER statement does not return control to the main program. Both the main program and the program called with ENTER must be cataloged. You cannot pass parameters with ENTER, but COMMON can be used. ENTER is faster than the following CHAIN statement.

The ENTER statement also can refer to a program indirectly by placing the program name in a variable and using the "ENTER @" form:

```
Program.Name$ = "Program.Name"
ENTER @Program.Name$
```

CHAIN

For example:

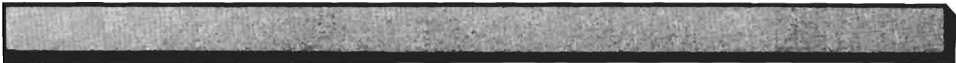
```
CHAIN "Command"
```

CHAIN can be used to execute any TCL level command from a PICK/BASIC program:

```
CHAIN "SORT File.Data"  
CHAIN "RUN Inventory.Program"
```

Unlike EXECUTE, CHAIN will not return control back to the program using it. CHAIN can be used to run another program, but the COMMON variables will not be preserved. Input could still be passed to a program or PROC with the DATA statement. The DATA statement must always precede the call to the program or PROC that uses the data.

System



SYSTEM: ACCESS SYSTEM INFORMATION

These commands are used to obtain information stored by the system. Such information can include the type of terminal in use or the number of disk drives, as well as information such as the current time and date.

(For system information relating to screen characteristics, see the section: "Screen." For printer characteristics, see the section: "Printer." For files and items, see the section: "Files, Items.")

IBM PC DOS Command Level

VER

Use VER to show the version of IBM PC DOS being used. The system will display the following, with the version number currently in use:

IBM Personal Computer DOS Version 3.10

IBM PC BASIC

PEEK(Memory.Address)

To obtain system information not available through standard functions such as DATE\$ and TIME\$ you can PEEK into a specific memory address.

Microsoft BASIC (Radio Shack 100)

FRE(0)

For example:

Memory.Remaining = FRE(0)

If the program is adding data to a file, the file must be closed and reopened each time that the amount of memory remaining is accessed to read the figure accurately. Otherwise it will be changed only in jumps of 256 bytes, with the result that the figure for memory will be too high by as much as 255.

FRE("")

For example:

Memory.String = FRE("")

The FRE("") function will show the amount of space left for storing string variables. The function also will reorganize the string space, eliminating the gaps caused by deleted or changed variables.

The amount of string space can be set at the beginning of the program with the CLEAR statement:

CLEAR String.Space

Otherwise a default CLEAR of 256 bytes will be performed.

PEEK(Memory.Address)

A PEEK at memory locations 1 and 2 will produce different results on the Radio Shack Model 100, 102, and 200:

Memory	100	102	200
1	51	167	171
2	125	27	152

The following routine will ascertain if the computer in use is a Radio Shack 100 or 102:

```
'* Test for Radio Shack 100 or 102
Model = PEEK(1)
IF Model = 51 THEN PRINT "Radio Shack Model 100"
IF Model = 167 THEN PRINT "Radio Shack Model 102"
```

PICK/BASIC

SYSTEM()

For example:

System.Code = SYSTEM(Function.Number)

The Function.Number used to determine exactly what kind of call is being made to the SYSTEM() function can be a literal, constant, or variable. The following values for Function.Number will return the information indicated:

- 0 Error code for a previous Basic tape operation:
 - 1: Tape unit not attached
 - 2: EOF (End of File) read from tape unit
 - 3: Attempted to write null string
 - 11: Attempted to write variable longer than tape record length
- 1 Printer or terminal condition:
 - 1: Output is going to printer, with PRINTER ON or (P) option
 - 0: Output is going to terminal
- 2 Page width for printer or terminal as defined by TERM command
- 3 Page length for printer or terminal " " " " "
- 4, 5 and 6 are used only with HEADING and FOOTING:
- 4 Number of lines remaining on current page
- 5 Current page number
- 6 Current line number
- 7 Terminal type (IBM PC = "I")
- 8 Tape record length
- 9 CPU millisecond count, from time logged onto current Account
- 10 Stack condition (if STON has been executed in a Proc):
 - 1: Active
 - 0: Inactive
- 11 LIST function condition:
 - 1: Active
 - 0: Inactive
- 12 System time in milliseconds
- 13 System RQM: seconds terminal "sleeps" with one RQM or SLEEP statement
- 14 Number of bytes awaiting input in input buffer
- 15 Verb options in effect. For example: option: (P), returns: "P"
- 16 Number of levels of nested EXECUTE statements
- 17 ERRMSG code for most recent TCL - Operating System command level
 - error, whether carried out before the program began, or within the program with an EXECUTE statement
- 100 System version, on the IBM PC-XT and PC-AT. On the PC-XT, the following string was returned: "PICK's R83 PC/XT Ver 2.0 07 May 1986"

If the Function.Number is set to a value that does not represent an available function, the value will be treated as zero, for the tape error function. Note that SYSTEM(0) indicates tape errors only, it does not signal other errors in a BASIC program.

After an error occurs to which the SYSTEM function responds, the function remains set at the error value. It does not clear. Thus if a TCL statement attempts to use a file that does not exist, SYSTEM(17) returns the value "10" for the file error. SYSTEM(17) remains set at "10" after other TCL verbs are executed, until another error occurs.

User Exits

For example:

```
ICONV(Parameter$, User.Exit$)
OCONV(Parameter$, User.Exit$)
```

A user exit is the entry point, in hexadecimal, of an Assembly Language routine. User exits can be employed either in PICK/BASIC, through ICONV or OCONV statements, or from a PROC. All of the examples shown are from PICK/BASIC.

User Exit: U50BB

For example:

```
Port.and.Account$ = OCONV(0, "U50BB")
Port.Num = FIELD(Port.and.Account$, " ", 1)
Account$ = FIELD(Port.and.Account$, " ", 2)
```

The user exit returns the same information as the WHO command at Pick TCL, the current port number and account name. The main terminal will be port number 0 (zero). A port number also is referred to as a line number or channel number.

User Exit: U3060

For example:

```
Password.Hex = OCONV>Password$, "U3060")
```

Use this command to convert a string to a sequence of 8 hexadecimal characters. This is the method used by the system to create encrypted passwords.

User Exit: U0079

For example:

```
Where.Info$ = OCONV(Options$, "U0079")
```

Same output and options as the WHERE statement at TCL (see below for WHERE).

User Exit: U3079

For example:

```
PCB$ = ICONV(FID$, "U3079")
FID$ = OCONV(PCB$, "U3079")
```

As an ICONV statement, U3079 returns the primary control block when given the Frame ID. As an OCONV statement, U3079 returns the frame ID when given the primary control block.

Pick TCL

LISTACC

For example:

```
LISTACC {Account.Name#Port.Number} {LPTR}
```

LISTACC is used to display information for each account logged onto the system, including the date and time logged on, the number of processing units utilized, and the number of pages printed. These figures can be used for charging for computer usage.

If no account name is specified, the accounting history for all the accounts will be shown. To specify an account, you must include both the account name and the port number, separated by a hash-mark: "#". For example, to display information for the New Zealand Account:

```
LISTACC New.Zealand#0
```

To send this listing to the printer:

```
LISTACC New.Zealand#0 LPTR
```

The port numbers begin with 0, not 1. To keep an accounting history of usage by an account, the account definition item for that account in the system dictionary must have a "U" in attribute 9. The accounting history is kept in the ACC file in the SYSPROG account.

LISTACC is a PROC in the PROCLIB file.

LISTCONN

For example:

```
LISTCONN {Dict.Filename} {LPTR}  
LISTCONN {Dict.Filename} {NOPAGE}
```

LISTCONN will display or print a list of all the connecting symbols such as AND, OR, <, >, etc. defined in a dictionary file. If no dictionary filename is included, the MD of the account is used as the default dictionary, thus displaying all the connecting symbols in the account.

LISTCONN is a PROC stored in the PROCLIB file.

LISTDICT

For example:

```
LISTDICT {Dict.Filename} {Modifiers} { (Options) }
```

To display a list of all the attribute and synonym definitions in a dictionary file, use LISTDICT. The MD is treated as the default dictionary filename.

The LISTDICT utility places no limits on the modifiers or the options in parentheses. It merely passes them on to the SORT verb to process. You can use any modifiers or options that an Access verb will accept. For example, to send the information to the printer:

```
LISTDICT Dict.Filename (P)
```

or:

```
LISTDICT Dict.Filename LPTR
```

LISTDICT is a PROC stored in the PROCLIB file.

LISTPROCS

For example:

```
LISTPROCS {Dict.Filename} {LPTR}  
LISTPROCS {Dict.Filename} {NOPAGE}
```

To produce a listing of all the PROCs defined in a dictionary file, use LISTPROCS. If no dictionary filename is included, the MD of the account is used as the default dictionary, thus displaying the names of all the PROCs in the account. You cannot use an option in parentheses, such as “(P)” for printer.

Because the LISTPROCS utility accesses only a dictionary file, it expects that PROC library files will be created with only a dictionary level, without a data level. (For file structure, see the beginning of the section: “Files, Items.”)

LISTPROCS is a PROC in the PROCLIB file.

LISTUSERS or LISTU

Use these commands to display information about the terminals currently logged onto the system. The report shows the port number to which each is attached, the beginning of the process control block, the name of the account, the time and date when logged on, and the location. The output of the LISTUSERS utility cannot be sent to the printer.

LISTU is a PROC in the PROCLIB file.

LISTVERBS

For example:

```
LISTVERBS {Dict.Filename} {LPTR}  
LISTVERBS {Dict.Filename} {NOPAGE}
```

Use LISTVERBS to display a list of all the verbs (commands that are in binary code rather than PROCs) available for use in the current account, or otherwise defined in a dictionary file. If no dictionary filename is included, the MD of the account is used as the default dictionary. You cannot use an account name as a parameter to see verbs in other accounts. Options in parentheses cannot be used, thus you must specify “LPTR” rather than “(P)” to send the information to the printer.

LISTVERBS is a PROC, stored in the PROCLIB file.

SORT

The TCL Access search commands can be used directly on the dictionary files to retrieve information when utilities such as LISTVERBS are not adequate or display more information than is desired. For example, to see both verbs and PROCs at the same time, in alphabetical order, limiting the selection to those that begin with the word “SET,” such as “SET-DATE,” “SET-TIME,” and so on:

```
SORT MD = "SET]" WITH *A1 = "P]"
```

Note that the right bracket is used for truncation, that “*A1” is a default designation for the first attribute in any item searched, and that the first attribute for either a verb or a PROC begins with “P”. The search statement will reveal that the “SET-DATE” and “SET-TIME” verbs are placed automatically in the SYSPROG account, but not in other

accounts, because they are not included in the "NEWAC" file that stores verbs for placement in new accounts.

TERM

For example:

TERM {Parameters}

Without parameters, the TERM command will display the current screen and printer setup. With parameters, the command will change them. (See the "Access System Information" portions of the sections: "Screen" and "Printer.")

WHAT

For example:

WHAT {LPTR}

To display information about the current computer setup, use WHAT. The information includes the amount of core memory installed, the number of ports, the amount of overflow frames available (indicating how much space is available on the disk, at 500 bytes of usable space per frame), the ports currently logged on, and the current status of the printer.

WHAT is a verb, not a PROC.

WHICH

For example:

WHICH

WHICH is used to display the date and version of the Pick Operating System, such as:

PICK's R83 PC/XT Ver 2.0 07 May 1986

WHICH is a program in the SYSPROG-BP file.

WHO

For example:

WHO {Port number{-Port number} }

WHO "Account.Name"

WHO is used to display the account name for a particular Port number, such as:

0 SYSPROG

One or more port numbers can be specified. If no number is included, the account name for the user's terminal is shown. Port numbers start at 0, not 1. There must be no spaces between the port numbers. If an account name is specified, it must be placed in quote marks. For example:

WHO
WHO 0
WHO 0-3
WHO "SYSPROG"

SYSTEM: ACCESS SYSTEM INFORMATION: DATE

Use the DATE command to get today's date from the system.

The routines shown set the string variable Date.Today\$ to today's date in the form MM/DD/YYYY in each language.

IBM PC BASIC

The year is returned in two digits in IBM PC BASIC, and "19" is inserted using the string handling functions. (For string insertion, see: "Strings: Replace, Insert.")

DATE\$

For example:

```
Date.Today$ = DATE$      '*Returned as: "MM/DD/YY"  
'* Insert "19" in front of the year. Note positions: 12345678  
Date.Today$ = LEFT$(Date.Today$, 6) + "19" + RIGHT$(Date.Today$, 2)
```

To set the date, assign a string in the format "MM/DD/YY" to the system variable DATE\$. For July 12, 1987:

```
DATE$ = "07/12/87"
```

Microsoft BASIC (Radio Shack 100)

On the Radio Shack 100, the year is updated automatically, but a small internal idiosyncrasy occasionally updates the system date by a year when communication is performed through the serial port or modem. On the Radio Shack Model 102, the year must be updated manually, thereby avoiding the previous model's unwanted updates.

DAY\$

For example:

```
Today$ = DAY$
```

The DAY\$ function returns the day of the week as a three-letter string, consisting of the first three letters of the alphabetical form of that day of the week. The first letter is capitalized, the others are lowercase. Thus if the day is Sunday, DAY\$ will return "Sun".

To set the day of the week, assign a three-letter string to DAY\$. It does not matter if upper- or lowercase letters are used in the assignment:

DAY\$ = "THU"
DAY\$ = "Thu"
DAY\$ = "thu"

IBM PC DOS Command Level

DATE

At IBM PC DOS command level, after giving the command:

DATE

the system responds, for example, for July 12, 1987:

Current date is Sun 7-12-1987
Enter new date (mm-dd-yy):

The user can enter the date in the format "MM-DD-YYYY" or "MM/DD/YYYY" or can cut the year to two digits, in which case the twentieth century is assumed. To accept the date already shown, simply press ENTER without typing anything.

PICK/BASIC

DATE(), OCONV

For example:

```
Date.System = DATE( );    * Internal format: Days since 12/31/1967  
Date.Today$ = OCONV(Date.System, "D/")
```

Pick stores a date internally as the number of days before or after December 31, 1967, with a negative number for days before and a positive number for days after. Output conversion codes are used to convert from Pick's internal format for external display of the date in a variety of formats. In the last example, "D/" is an output conversion code that causes the date to be displayed in the format "MM/DD/YYYY". For input, an input conversion code of "D" should be used to inform the system that a date has been entered. (For the use of conversion codes for dates, see the section: "Conversions and Correlatives in Pick: Date.")

Pick TCL

SET-DATE

For example:

```
SET-DATE 07/12/1987  
SET-DATE 12 JUL 1987  
SET-DATE JULY 12, 1987
```


If no automatic system clock is used, or if a different date is desired, the operating system command SET-DATE is used. The command would accept the date July 12, 1987 in any of the formats shown, and in others besides.

(For the other date formats that would be accepted, see the section: "Conversions and Correlatives in Pick: Date." This command might be available only in the SYSPROG account. To copy it into other accounts, see the section: "Files, Items: Copy, Backup.")

SET-DATE-EUR, SET-DATE-STD

For example:

```
SET-DATE-EUR      {DD/MM/YYYY}
SET-DATE-STD      {MM/DD/YYYY}
```

The examples in this book assume the use of the American style of displaying a date numerically in the format "MM/DD/YYYY." In the European style, the day comes before the month, in the format "DD/MM/YYYY." The command SET-DATE-EUR for "Set Date European" sets the system to the European style, while the command SET-DATE-STD for "Set Date Standard" rather chauvinistically designates the American style as standard usage. The American style is the default usage.

(These commands might be available only in the SYSPROG account. To copy them into other accounts, see the section: "Files, Items: Copy, Backup.")

Pick's standard display mode is to show the date in the format:

DD MMM YYYY

with the month as a three-letter abbreviation. Thus for July 12, 1987:

12 JUL 1987

The alphabetical characters format avoids incompatibilities in the numerical usage, but introduces language difficulties.

None of the differences between cultures in display of the date affect Pick's internal storage of the date as the number of days in relation to December 31, 1967. No changes need be made to the internal data when a change of output format is required.

SYSTEM: ACCESS TO INFORMATION: TIME

TIME commands are used to get the current time from the system.

(For details on methods of measuring the passage of time and timing the speed of a program, see the section: "Size, Speed of Program: Test Speed.")

The routines shown set the string variable Time.Now\$ to today's date in the form HH:MM in each language, with a 24-hour clock. When the date function also returns seconds, they are truncated from the display using the string handling functions. (See the section: "Strings: Extract.")

IBM PC BASIC

TIME\$

For example:

```
Time.Now$ = TIME$      '* Returned as: "HH:MM:SS"  
 '* Cut out the seconds. Note positions: 12345678  
Time.Now$ = LEFT$(Time.Now$, 5)
```

To set the time, assign a string in the format "HH:MM:SS" to the system variable **TIME\$**. For 1:30 P.M.:

```
TIME$ = "13:30:00"
```

The **TIMER** statements will cause program interrupts:

```
ON TIMER (Number.of.Seconds) GOSUB Line.Number  
Number.of.Seconds = 100  
ON TIMER(Number.of.Seconds) GOSUB 100
```

The passage of the designated number of seconds will interrupt the program and cause it to branch to the line number specified. (The form of the statement is the same as the **ON...GOSUB** form, shown in the section: "Conditions: IF, CASE, ON...GOSUB.")

TIMER ON, TIMER OFF, TIMER STOP

For example:

```
TIMER ON  
TIMER OFF  
TIMER STOP
```

The **TIMER ON** statement begins the seconds count down. **TIMER OFF** cancels it. **TIMER STOP** also cancels it, but the result is remembered and acted upon with a subsequent **TIMER ON** statement.

Microsoft BASIC (Radio Shack 100)

The **TIMER** statements are not available in Microsoft BASIC on the Radio Shack 100.

ON TIME\$ = Interrupt\$ GOSUB Line.Number

For example:

```
Interrupt$ = "12:00:00"  
ON TIME$ = Interrupt$ GOSUB 100
```

The arrival of the designated time will interrupt the program and cause it to branch to the designated line number. The line number must be a literal. The time designated for an interruption can be a string variable or a string enclosed in quotes.

TIME\$ ON, TIME\$ OFF, TIME\$ STOP

For example:

```
TIME$ ON
TIME$ OFF
TIME$ STOP
```

The time interrupt first must be enabled with the **TIME\$ ON** statement. It can be disabled with the **TIME\$ OFF** statement. **TIME\$ STOP** also disables the interrupt, but stores the information that the designated time arrived, so that a subsequent **TIME\$ ON** command will cause branching to the designated line number.

IBM PC DOS Command Level

TIME

After giving the command:

```
TIME
```

the system responds, in this example for exactly 1:30 P.M.:

```
Current time is 13:30:00.00
Enter new time:
```

The user does not need to include the seconds or hundredths of seconds. However, it is necessary to use the 24-hour time format.

PICK/BASIC

TIME()

For example:

```
Time.System = TIME( );    * Internal format: Seconds since midnight.
Time.Now$ = OCONV(Time.System, "MT");    * "HH:MM"
```

Pick stores a time internally as the number of seconds since midnight. Output conversion codes are used to convert from Pick's internal format for external display of the time in a variety of formats. In this example, "MT" is an output conversion code that causes the time to be displayed in the format "HH:MM", based on a 24-hour clock. For input, the same standard conversion code of "MT" should be used to inform the system that a date has been entered.

SYSTEM(12)

For example:

```
Seconds.Since.Begin = SYSTEM(12) / 1000
```

The **SYSTEM** function will return the time elapsed in milliseconds since the system was turned on. This is very useful for measuring the time taken by an operation:

```
Time.Begin = SYSTEM(12)
* Operation statements here
Time.End = SYSTEM(12)
Time.Elapsed = (Time.End - Time.Begin) / 1000
CRT "Time elapsed = " : Time.Elapsed : " seconds"
```

The display would resemble:

Time elapsed = 34.78 seconds

Pick TCL

TIME

For example:

TIME

The command **TIME**, at the operating system level, causes Pick to display the date and the time:

13:30:00 12 JULY 1987

SET-TIME

For example:

SET-TIME 13:30

If no automatic system clock is used, or if a different time is desired, the **SET-TIME** command is used at the operating system level. The **SET-TIME** command will accept all or part of the HH:MM:SS designation:

```
SET-TIME 13:00:00
SET-TIME 13
```

A 24-hour clock format is used. **SET-TIME** will ignore a P.M. designation. Thus an attempt to set 1 P.M. will actually set the time to 1 A.M., but with no warning that the time is now incorrect. The correct designation for 1 P.M. is 13:00.

This command might be available only in the SYSPROG account. To copy it into other accounts, see the section: "Files, Items: Copy, Backup."

SYSTEM: BOOTING, START-UP

Pick

When the Pick system boots up, the normal logon message is:

LOGON PLEASE:

The user will then type in the name of the account that he intends to use. The format of the logon message easily can be changed by editing the item with the item ID "LOGON" in the SYSTEM dictionary file in the SYSPROG account.

If a password has been indicated in the account definition item of the account requested, the user will next be prompted for a password:

PASSWORD?

To log off of the Pick system, at the system prompt (either ">" before the open architecture release or ":" with open architecture) the user enters the command "OFF":

> OFF

COLDSTART

The logon process is generated by a PROC named COLDSTART in the SYSPROG-PL file of the SYSPROG account. (A PROC is a stored procedure, similar to an IBM PC DOS Batch file. An account is similar to a directory on an IBM PC DOS disk. However, the Pick PROC and account are much more powerful than their DOS counterparts.)

The COLDSTART PROC will execute automatically and can be used to set the date and time and to set the appropriate terminal type. The COLDSTART PROC also can ask for a password, to identify the user in order to transfer the user to an account set up specially for him. The following is an example of a COLDSTART PROC. The last statement in the PROC transfers control to another PROC. The second PROC transfers the user to the history account:

COLDSTART PROC

```
PQ
C COLDSTART Proc in the SYSPROG-PL file in the SYSPROG Account.
C Starts automatically when the system is booted.
C Pick Proc
C Pick IBM PC-XT R83 2.0
C Filename under IBM PC DOS: ColdStrt.prc
C *** (C) Copyright 1986 Pick Systems ***
C
HINIT-CURSOR CURSOR *
PH
HTERM-TYPE
PH
HSET-SYM PSYM
PH
C
```

```

C Prompt for time and date only if the system uses
C software clock
C
HCHECK-CLOCK
PH
IF E = 310 G 10
O
O Time and date: +
HTIME
P
G 30
10 C
O
O Please enter the time +
RI
IP
IF #A G 10
HSET-TIME
A
P
IF E = 298 G 10
20 C
O
O Please enter the date +
RI
IP
IF #A G 20
HSET-DATE
A1,11
A
P
IF E = 298 G 20
30 C
HSTARTPTR 0,0,0,P0
PH
C O
C O Verifying system modes, Please wait...
C O VERIFY-SYSTEM ... Has been disabled
C HVERIFY-SYSTEM
C PH
C
C *****
C * Custom Cold Start from this Point Forward *
C *****
C David L. Clark 08/22/1988
C
C Set Printer Line Length to 59 for HP Laser Jet
C Otherwise, throws an extra page for each page printed
HTERM ,,,,,,59
PH
C
C Install PC-KEYS from Modular Software to use cursor keys on Line zero
C Must install from SYSPROG Account
HLOGTO SYSPROG
P
HRUN PC-KEYS INSTALL
P
C
C O This is the end of the COLDSTART Proc

```

C Now go to History Account
[LOGTO.HISTORY]
PH

The LOGTO.HISTORY PROC

PQ
C This Proc logs to the History Account
HLOGTO History
P

In the COLDSTART PROC, INIT-CURSOR, TERM-TYPE, SET-SYM, and VERIFY-SYSTEM are all Pick TCL verbs.

VERIFY-SYSTEM { (0) }

For example:

VERIFY-SYSTEM

The VERIFY-SYSTEM command will verify the integrity of the Pick Operating System by means of checksums. If an error is reported, the system should be reloaded. When loading a new version of the system or an assembly language program such as the word processor JET, the "O" option (letter "O" for "Over-Write," not the number zero) should be used to over-write or reset the checksums.

Because the VERIFY-SYSTEM process takes approximately a minute, it has been "commented out" of the COLDSTART PROC by placing "C", the indicator of a PROC comment or remark, before the VERIFY-SYSTEM command.

Whenever an account is entered, whether through a PROC or by the user entering the command:

LOGTO Account.Name

at the terminal, the processor looks in the MD of the account for a PROC with the same name as the account. If such a PROC exists, it will be executed automatically. This PROC can be employed to present the user with a menu of the options available or to begin execution of a PICK/BASIC applications program.

Creating a separate account for each user has important advantages for security. Each account can have its own retrieval and update codes, stored in the account definition item. These codes determine which files in other accounts the user will be allowed to see and to change. The account also indicates a system security level for the user, from a low level of SYS0 to a high of SYS2. The highest level has access to all Pick commands, the lower levels do not.

TERM-TYPE

TERM-TYPE is a program in the BP (BASIC programs) file of the SYSPROG account. The TERM-TYPE program could be run as part of the Logon PROC of each account. Modify the NEWAC (new account) file in the SYSPROG account to create a Q-

pointer (file synonym) to BP in SYSPROG as SYSPROG.BP. The NEWAC file contains the items that are automatically copied to a new account when it is created.

```
Item ID: SYSPROG.BP
01 Q
02 SYSPROG      {Original Account name}
03 BP           {Original filename}
```

The account logon would contain:

```
Item ID: Account.Name
01 PQ
02 C Account Logon Proc to set screen characteristics
03 HTERM-TYPE
04 P
```

As an alternative, you could place the routine to run TERM-TYPE at the end of the COLDSTART PROC in the file SYSPROG-PL in the SYSPROG account, as in the COLDSTART example. This PROC executes automatically when entering the Pick System. You also could run the SET-TERM program to set the characteristics of the other ports. (For more information on setting terminal parameters, see the section: "Screen: Assign Terminals.")

SYSTEM: BOOTING, START-UP: SUPPRESS SYSTEM MESSAGES

Pick

Pick displays a variety of system messages each time that the system is booted or the user changes from one account to another. The messages come from three sources: the system monitor or controller, PROCs, and items in the ERRMSG (for error messages) files.

The plethora of messages displayed by Pick can be considered useful by the experienced programmer, but the novice user, whose only interest is in utilizing a specific application program, will probably find the messages intimidating. Those messages displayed directly by the system monitor cannot be suppressed. The other two sources, PROCs and the ERRMSG items, must be tracked down one at a time and neutralized.

Before changing any PROC or ERRMSG item, always first take the precaution of storing a backup copy with a standard prefix such as "OLD" followed by a period and then the original name.

For example, to store a backup copy of ERRMSG item 335, at Pick TCL type:

```
COPY ERRMSG 335
```

Pick will prompt:

TO:

Fill in the answer:

TO: OLD.335

To verify that a PROC or an ERRMSG item is the source of a message, insert a line to be displayed identifying the source. For example, the following SYSPROG PROC will run automatically each time that the user logs onto the SYSPROG account. It is in the account's MD, and the item ID of the item is the same as the name of the account: SYSPROG. The second line has been added to the item, to identify the source of the messages that will appear on the screen. To display a message from a PROC, insert a line beginning with the letter "O," which is the PROC command for output, followed by the message.

SYSPROG PROC, First Version

```
PQ
O This is the SYSPROG Proc in the MD file in the SYSPROG Account
HTERM-TYPE
P
HSET-SYM PSYM
P
O
O Documentation addenda may be reviewed +
O by typing 'ADDENDA' at TCL.
H?
P
```

To avoid the display of messages from a PROC, simply comment out the lines that begin with "O" for Output or "D" for Display, or which execute a command that will display a message. In the SYSPROG PROC example, the last two lines, containing "H?" and "P" execute the WHO command, which displays the port number and account. All of the lines that cause a display have been commented out by placing a "C" and a space at the start of the line. The letter "C" for comment is PROC's equivalent of "REM" for remark in IBM PC BASIC or PICK/BASIC.

SYSPROG PROC, Second Version, with Messages Suppressed

```
PQ
C O This is the SYSPROG Proc in the MD file in the SYSPROG Account
HTERM-TYPE
P
HSET-SYM PSYM
P
C O
C O Documentation addenda may be reviewed +
C O by typing 'ADDENDA' at TCL.
C H?
C P
```

Just as PROCs have a format different from that of PICK/BASIC programs, so unfortunately do ERRMSG items have a format different from PROCs. The following

ERRMSG items with the item IDs of 335, 336, and 340 cause message displays at the time of system or account logging on or off.

The first two lines have been inserted into item 335 to identify the source of the screen messages. To display a message from an ERRMSG item, insert a line beginning with the letter "H", followed by the message, to place the message into the output buffer. Insert a second line after it containing only the letter "L", which causes the output buffer to be displayed, followed by a carriage return and line feed.

ERRMSG 335

```
H This is Item 335 in the ERRMSG file in the SYSPROG Account
L
S(8)
H< < < ----- THE PICK SYSTEM ----- > > >
L
S(8)
H< < < -----
T
H ----- PC-XT Ver 2.0-----
D
H ----- > > >
L
S(8)
H< < < ----- (C) Copyright 1986 Pick Systems ----- > > >
L
```

ERRMSG 336

```
H< LOGGED OFF AT
T
S(33)
HON
D
S(60)
H>
L
L
L
```

ERRMSG 340

```
L
H< CONNECT TIME =
A
H MINS.;
H CPU =
A
H UNITS; LPTR PAGES =
A
S(60)
H>
```

You cannot comment out ERRMSG items as you can with PROCs or BASIC programs. No comment or remark statement is available. Instead, you should store a backup copy of the item as beginning with the prefix "OLD.", and then reduce the item to null by deleting all of the lines, followed by storing the empty item. Do not try to delete the item, or messages might appear, such as:

ERRMSG [335]

indicating that the system is calling out for a nonexistent ERRMSG item.

Although reduction to null will work for ERRMSG items 335 and 336, that procedure is not sufficient for ERRMSG item 340. The item contains three lines with the command "A", which move three parameters passed from the system into the output buffer. Something must be done with the three parameters, or they will be displayed without any explanation, causing additional confusion to the user.

The "X" command will cause the ERRMSG processor to skip a parameter. Therefore, after deleting all the lines from the item, insert one line containing only the command "X" for each line in the original item that contained the command "A". Thus item 340 would contain only three lines, each consisting of "X".

ERRMSG File, Item 340, with Messages Suppressed

X
X
X

When tracking down system messages, look in the following locations:

Account: SYSPROG
File: ERRMSG
Items: 335, 336, 340, LOGON
File: MD
Item: SYSPROG
File: SYSPROG-PL
Item: COLDSTART
File: SYSTEM
Item: LOGON
Account: All others
File: MD
Item: Item ID same as Account name

SYSTEM: BOOTING, START-UP: TERMINAL LOGON

Pick

Pick is a multiuser system, allowing even an inexpensive IBM PC/XT to accommodate three users at the same time. A PC/AT can accommodate 10 users, desktop machines

based on the 386 chip can support 17 or 33 users, and larger machines will support several hundred.

To log on with a remote terminal, connect the serial port of the terminal with the serial port of a computer that is running the Pick Operating System. Turn on the terminal and press the ENTER key. A Pick logon message should appear, either the message supplied by Pick Systems:

LOGON PLEASE:

or a special logon message created by the programmer, such as:

Please enter Account Name:

The logon message is contained in the SYSPROG account, in the ERRMSG file, in an item named LOGON. The message can be changed by changing this item.

To log off from Pick, on the remote terminal enter:

OFF

The logon message will reappear. You can enter a different account, or turn off the terminal.

SYSTEM: BOOTING, START-UP: TERMINAL LOGON WITH A STAND-ALONE COMPUTER

Stand-alone computers also can be employed as Pick terminals through the use of communication or terminal emulation programs. The following example demonstrates the use of the portable Radio Shack 100, which comes equipped with a telecommunications program named TELCOM, for communication either through the serial port or through the computer's built-in modem.

To connect the Radio Shack 100 to Pick, with Pick running on the larger machine, perform the following operations on the Radio Shack device:

Enter TELCOM

Look at the communications parameters displayed on the upper left portion of the screen. The parameters "88N1E" correspond to Pick and have the following meaning: 8 = 9600 baud, 8 = 8 Data Bits, N = No Parity, 1 = 1 Stop Bit, E = Enable Xon/Xoff.

If the parameters are not set to "88N1E press F3 for STAT, type in the correct parameters, and press ENTER. Press F3 for STAT again, followed by ENTER, to see the new parameters that have been applied.

When the parameters are correct, press F4 for TERM, to enter terminal mode. Press ENTER to get Pick's attention. The Pick logon message will appear.

After connecting with Pick, note the function key labels at the bottom of the screen:

F1 = Prev, F2 = Down, F3 = Up, F4 = Full, F8 = Bye

F1 will display the previous screen shown. F2 is used to download a file to the Radio Shack 100, F3 to upload a file to another computer. F4 is a toggle switch between full and half duplex. Leave it set at full. F8 will be used to disconnect the devices.

Not shown on the label line is the echo function of the F5 key, which will echo output on the screen to a printer connected to the parallel printer port of the Radio Shack 100. If no printer is connected, the computer will hang. The 100 cannot communicate through the modem and the serial port simultaneously, because the same internal mechanism is used for both. Thus you cannot echo output to a serial port printer.

To make the label line disappear to have use of the full eight lines of the screen, press the LABEL key. Pressing LABEL again will cause the label line to reappear.

The following program named "PkTerm" (for "Pick Terminal") will make the connection through TELCOM automatically. In addition, the program contains pokes that will add a Line Feed to each Carriage Return when sending a file from the Radio Shack 100 and allow the use of the F6 and F7 keys to display a directory of the files and the amount of memory remaining on the 100. The pokes are not restored automatically when the program ends, because the program never returns to BASIC.

PkTerm Program, Documentation Version

Pick Terminal Program: Dictionary of Constants and Global Variables

The short version of the name is used in the executable program, the long version in the documentation:

A\$	= Account\$	:	Name of Pick Account to logon to. The program uses the Account name "SYSPROG". Change the variable to the Account name desired.
CR\$	= Carriage.Return\$		
ES\$	= Escape\$		
K\$	= Key\$	:	Keystroke entered by user
L\$	= CRLF\$	:	Carriage Return — Line Feed
U\$	= Undo.Video.Reverse\$		
V\$	= Video.Reverse\$		
ZO\$	= Cursor.On\$	:	Turn on blinking cursor

```

1  'PkTerm.do: Pick Terminal  David L. Clark  04/30/1988
2  '* Use Radio Shack 100 as a Pick terminal.
3  '* Microsoft Basic (Radio Shack 100) program. Documentation version.
4  '* Radio Shack 100
5  '* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10 *****
   '* Assign Constants *
   *****
CLEAR                                     '* Clear the Paste Buffer
:   DEFINT A-Z
:   Carriage.Return$ = CHR$(13)
:   CRLF$             = Carriage.Return$ + CHR$(10)
:   Escape$          = CHR$(27)

```

```

: Undo.Video.Reverse$ = Escape$ + "q"
: Video.Reverse$      = Escape$ + "p"
: Cursor.On$          = Escape$ + "p"
'* End line
*****
20 '* Display Instructions *
*****
CLS
: PRINT                                                    CRLF$;
    "To use the Model 100 as a live terminal";           CRLF$;
    "start the Pick Operating System on";                 CRLF$;
    "the larger computer and connect the";               CRLF$;
    "Serial Ports of the two machines.";                 CRLF$;
                                                    CRLF$; Video.Reverse$;
    "Press ENTER to Continue ESCAPE to Quit"; Undo.Video.Reverse$;
    ":"; Cursor.On$;
'* End line
*****
30 '* Continue or Quit *
*****
Key$ = INKEY$
: IF Key$ = "" THEN
    GOTO 30
    '* Loop back
ELSE
    IF Key$ = Escape$ THEN
        GOTO 9999
        '*|End Program|
    '* End
'* End
'* End line
*****
40 '* Set Communication Parameters *
*****
'* Set parameters by opening and closing a file to the Serial Port
OPEN "COM:88N1E" FOR OUTPUT AS 1
: CLOSE
'* End line
*****
50 '* Add Line Feed to Carriage Return *
*****
'* Poke to add a Line Feed to each Carriage Return when transferring.
'* To restore: POKE 63066, 0
POKE 63066, 1
'* End line
*****
60 '* Reset Function Keys *
*****
'* Note: The command "POKE 63056, 64" used to disable BREAK and
'* PAUSE and reset the function keys when using the Model 100 on
'* a stand-alone basis has no effect when using the Model 100
'* thru TELCOM as a terminal.
*****
'* Set F6 to Show Files *
*****
'* F6 will display directory of files on the Model 100.
'* To restore: POKE 64268, 247 : POKE 64269, 127
POKE 64268, 58
: POKE 64269, 31
*****
'* Set F7 to Show Memory *

```

```

      '*****
      '* F7 will show remaining memory available on the Model 100.
      '* To restore: POKE 64270, 247 : POKE 64271, 127
: POKE 64270, 172
: POKE 64271, 126
'* End line
*****
70 '* Load Keyboard Buffer to Make Contact *
*****
'* Poke into the keyboard buffer the ASCII values which will be
'* send automatically to Pick when contact is made. The length
'* may not exceed 32 characters. The message, including Carriage
'* Returns, is poked into alternate bytes.
'* Two versions are shown. Comment out the version not to be used.
'* The first includes an Account name, to logon to an Account
'* automatically.
'* The second sends only a Carriage Return, to get Pick's
'* attention. The Pick logon prompt will appear, and the user
'* must supply the Account name.
*****
'* Version to Supply Account Name *
*****
'* Change value in Account$ from "SYSPROG" to the name desired.
'* The first Carriage Return is to get Pick's attention
Account$ = "SYSPROG"
: Account$ = Carriage.Return$ + Account$ + Carriage.Return$
: Length = LEN(Account$)
: FOR I = 1 TO Length
: POKE 65449 + 2 * I, ASC(MID$(Account$, I, 1))
: POKE 65450 + 2 * I, 0
: NEXT
: POKE 65450, Length + 1
'* End line
*****
75 '* Version to Get Pick's Attention Only *
*****
'* Send only a Carriage Return = ASCII 13.
'* This version is shown commented out.
'* POKE 65450, 1
: POKE 65451, 13
'* End line
*****
90 '* Make Connection to Pick *
*****
'* The machine language Call exits to TELCOM, directly into TERM
'* (for Terminal) mode. The communication parameters must therefore
'* already be set. To exit to TELCOM and reset the parameters,
'* rather than going straight to TERM, use CALL 20806. Then
'* press F3 for STAT to set the new parameters.
'* The program does not return to Basic after quitting TELCOM.
CLS
: CALL 21589
'* End line
*****
9999 '* End Program *
*****
MENU

```

Executable Version of PkTerm Program

```
1 'PkTerm.ba David L. Clark 04/30/1988
5 'Copyright 1988 David L. Clark -- TAB BOOKS Inc.
10 CLEAR:DEFINT A-Z:CR$=CHR$(13):L$=CR$+CHR$(10):ES$=CHR$(27):U$=ES$+"q" :V$=ES$+
20 CLS:PRINTL$"To use the Model 100 as a live terminal"L$"start the Pick Operati
30 K$=INKEY$:IFK$=""GOTO30ELSEIFK$=ES$GOTO9999
40 OPEN"COM:88N1E"FOROUTPUTAS1:CLOSE
50 POKE63066,1
60 POKE64268,58:POKE64269,31:POKE64270,172:POKE64271,126
70 A$="SYSPROG":A$=CR$+A$+CR$:L=LEN(A$):FORI=1TOL :POKE65449+2*I,ASC(MID$(A$,I,1
75 '*POKE65450,1:POKE65451,13
90 CLS:CALL 21589
9999 MENU
```

After a session with the Radio Shack 100 as a terminal, enter the command OFF to leave Pick, just as with any terminal. Then take the following actions to quit TELCOM:

- 1) Press LABEL to see the label line, if it is not showing already.
- 2) Press F8 for "Bye" to quit.
- 3) The prompt "Disconnect?" will appear. Respond by entering "Y" for Yes.
- 4) Press F8 again to return to Radio Shack's Menu.

(For more information on changing the values returned by keys, see the section: "Keyboard: Change Keys.")

SYSTEM: CALL SYSTEM COMMANDS FROM A PROGRAM

IBM PC BASIC

In interpreted IBM PC BASIC nearly all of the same commands can be entered interactively at the terminal, within the BASIC environment, as can be activated through a program. For example, the command "FILES" will display a list of the filenames, whether you type in the command or place it in a program.

SHELL, EXIT

For example:

```
SHELL {Command$}
EXIT
```

The SHELL command, issued without a parameter, will pass control from BASIC to DOS by loading an extra copy of COMMAND.COM, which is the main IBM PC DOS command file. When the command EXIT is issued, control will return to BASIC.

The SHELL command can include the name of a command or of an IBM PC DOS program. For example:

```
SHELL Command$
SHELL Program.Name$
```


The command will be executed or the program entered, and then control will return automatically to BASIC, without a need for the EXIT statement. However, if SHELL invokes a batch file, the last command in the batch file must be EXIT. Programs that remain resident in memory should not be used with SHELL.

SHELL is not available on simpler forms of Microsoft BASIC.

PICK/BASIC

EXECUTE

For example:

```
EXECUTE Command$ {RETURNING Error.Code$} {CAPTURING Output$}
```

The command to EXECUTE can be a TCL statement, the name of a PROC, or the name of a cataloged PICK/BASIC program.

The command can be placed in a variable, or it can be given as a literal surrounded with quotes:

```
EXECUTE "LISTPROCS"
```

or:

```
Command$ = "LISTPROCS"  
EXECUTE Command$
```

Some parts of a command might require their own surrounding quote marks. For example, a user might enter the following Access search statement at TCL level:

```
SSELECT History.Data WITH Subjects = "Harbors"
```

The word "Harbors" is the object to search for, and this object must be placed in quote marks. To execute the same command from a PICK/BASIC program with the EXECUTE statement, the entire command must be surrounded with Pick's alternate quote marks. Pick allows: ', ", or \ (right single quote, double quote, or backslash) for use as quote marks. The EXECUTE statement would be:

```
EXECUTE \SSELECT History.Data WITH Subjects = "Harbors" \
```

If the object for searching has been placed in a variable, the surrounding quote marks must be included in the variable. When the object is a variable, backslashes should be used as surrounding quotes, to allow for the possibility that double or single quotes might already be present in the variable:

```
EXECUTE "SSELECT History.Data WITH Subjects = \" : Object$ : \" \"
```

Each separate element of a complex command, such as a search statement, can be placed in a separate variable, and the whole concatenated into the final command to execute:

```

Verb$           = " SSELECT "
File.Data.Name$ = "History.Data"
Link$           = " WITH "
Attrib.Name$    = "Subjects"
Operator$       = " = "
Object$         = "Harbors"
Command$ = Verb$ : File.Data.Name$ : Link$ : Attrib.Name$ : Operator$
Command$ = Command$ : "\" : Object$ : "\"
EXECUTE Command$ RETURNING Error.Code$ CAPTURING Output$

```

Note the need to provide spaces to separate the elements in the final command.

The optional RETURNING clause will assign any error message code to the variable Error.Code\$. (Error codes are used for general system messages, not just errors.) Each error code will be separated by a blank. The function SYSTEM(17) will also return the error message codes:

```

EXECUTE Command$
Error.Code$ = SYSTEM(17)

```

The error codes are the item IDs of items in the ERRMSG file. The individual codes can be extracted with the DCOUNT and FIELD functions. (See the section: "Strings: Extract.")

The optional CAPTURING clause places any output from the command, such as the number of items selected, in the variable Output\$. Carriage Return—Line Feeds will become attribute marks, and clear screen commands will be eliminated. If you do not want the output from the command to appear on the screen, the CAPTURING clause should be used to suppress the screen display.

Select lists can be created, saved, and used from a BASIC program with the EXECUTE process. For example:

```

EXECUTE Command$ CAPTURING Output$
EXECUTE \SAVE-LIST Select.List\
EXECUTE \GET-LIST Select.List\

```

(For examples of the use of EXECUTE to search a file, see the section: "Files, Items: Input: Search with Pick Access Processor.")

EXECUTE can be used to make recursive calls to the same program. Thus within the program Factor there can be an EXECUTE statement to run Factor. The function SYSTEM(16) returns the number of nested EXECUTE levels.

CHAIN

For example:

```
CHAIN Command$
```

Because the EXECUTE process requires its own workspace, the first EXECUTE command can be delayed by up to 30 seconds. The CHAIN command will not be delayed.

However the CHAIN statement does not return to the BASIC program. It would be best to use CHAIN for those commands that would not return to the program anyway, even with the EXECUTE statement:

```
CHAIN "OFF"  
CHAIN \LOGTO \ : Account.Name$
```

DATA

For example:

```
DATA "Response to prompt"
```

The DATA command will pass a parameter from a PICK/BASIC program to a subsequent process that prompts for user input. The passed parameter is placed in a system buffer, and substitutes for the expected interactive input.

The following TEST.DATA program invokes the Pick TCL COPY command, which prompts for the name of the item to which the copy should be made. The TEST.DATA program contains two portions. In the first part, the item "TEST.DATA" is copied to an item "TEST.DATA.COPY." In the second part, the ESCAPE character is sent rather than the item ID, with the result that "TEST.DATA" is copied to the screen.

TEST.DATA Program

```
* TEST.DATA David L. Clark 07/06/1987  
* Test the DATA statement.  
* Pick Basic program.  
* Pick IBM PC-XT R83 2.0  
* Filename under IBM PC DOS: TestData.bp  
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.  
  EQUATE Escape$ TO CHAR(27)  
  * Make an extra copy of the TEST.DATA Item as TEST.DATA.COPY  
  * Load the buffer with the name to copy to  
  DATA "TEST.DATA.COPY"  
  * Execute the command  
  EXECUTE "COPY TEST.BP TEST.DATA"  
  * Copy TEST.DATA to the screen instead  
  * Load the buffer with ESCAPE rather than an Item name  
  DATA Escape$  
  * Execute the command  
  EXECUTE "COPY TEST.BP TEST.DATA"  
  CRT "Returned to program at end"  
  END
```

(For further examples of DATA see the section: "System: Partition Change from Pick to IBM PC DOS.")

PROCREAD, PROCWRITE

A Pick PROC is a stored procedure that can be used as a keyboard macro, to call up another system processor such as the Editor, or to carry out a series of operations. Any Pick terminal command can be placed in a PROC. A PROC can be called from PICK/BASIC with the EXECUTE or the CHAIN statements. Alternately, a PROC can call a PICK/BASIC program, which will return control to the PROC upon termination.

PROCREAD will read information from a buffer left by a PROC, PROCWRITE will write information to a buffer for later use by a PROC. In a PROC buffer, a space is used as the delimiter between elements. A backslash is interpreted as a null, and can be used to separate values or as a place holder.

Use PROCREAD to determine if a program has been called by a PROC. Note the required ELSE clause:

```
PROCREAD Buffer$ ELSE Buffer$ = ""; Proc.Call.Flag = False
```

Because elements in the PROC buffer are separated by spaces, use the TRIM function to eliminate any accidental redundant spaces:

```
Buffer$ = TRIM(Buffer$)
```

Use the FIELD function to pull out individual elements between the space delimiters:

```
Buffer.Element.3$ = FIELD(Buffer$, " ", 3)
```

Following is a PROC named "TEST.PROC.BUFFER" and a PICK/BASIC program named "TEST.PROC.BUFFER.BP". The PROC will place a series of parameters in the PROC input buffer. The user can type up to 20 parameters after the PROC name. If no parameters are supplied, the PROC will supply its own: "First," "Second," and "Third." In this manner parameters can be created either from user input or from text in a PROC.

The PROC will invoke the program. The program will read the parameters from the buffer and display them. The program will then add the string "Program#" to each of the parameters, place them back in the PROC input buffer, and return control to the calling PROC. The PROC will display the changed parameters from the buffer. Thus the examples demonstrate the passing of parameters in both directions, from PROC to program and from program to PROC.

TEST.PROC.BUFFER.PROC PROC:

```
C TEST.PROC.BUFFER.PROC
C Test PROCREAD, PROCWRITE. David L. Clark 07/06/1987
C Filename under IBM PC DOS: TProcBuf.prc
RO
C If parameters are not typed on the command line, then use text in Proc
IF A2 GO 200
100 C No parameters typed, so clear buffers, send text from Proc
RI
S1
IHFirst
S2
IHSecond
S3
IHThird
200 C Run program
HRUN TEST.BP TEST.PROC.BUFFER.BP
P
C Display parameters read back from program
```

```

SP
S1
O
OParameters read into Proc from Program:
300 C Loop to read each parameter in the buffer
IF #A GO 999
D
F
GO 300
999 C End Proc
X

```

TEST.PROC.BUFFER Program

```

* TEST.PROC.BUFFER David L. Clark 07/06/1987
* Test PROCREAD, PROCWRITE.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TProcBuf.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Begin by starting TEST.PROC.BUFFER. To test sending text from a Proc to a
* program, add no parameters. To test sending parameters from the command
* line, add up to 20 parameters, for example:
* TEST.PROC.BUFFER 1 2 3
* The parameters will be sent from the Proc TEST.PROC.BUFFER to the
* program TEST.PROC.BUFFER.BP. The program will display the parameters
* that it has received. It will also concatenate the word "Program#" to
* each parameter, and in that form send them back to TEST.PROC.BUFFER,
* which will display the parameters in their changed form.
*****
* Assign Parameters, Constants *
*****
EQU Proc.Name$ TO "TEST.PROC.BUFFER"
EQU Space$ TO " "
DIM Buffer.Array$(20)
*****
* Read from the Buffer *
*****
PROCREAD Buffer$ THEN
    Total.Parameters = DCOUNT(Buffer$, Space$)
    * If the Proc name is the 1st parameter, start at the 2nd
    IF FIELD(Buffer$, Space$, 1) = Proc.Name$ THEN
        J = 2
        Total.Parameters = Total.Parameters - 1
    END ELSE
        J = 1
    END
    FOR I = 1 TO Total.Parameters
        Buffer.Array$(I) = FIELD(Buffer$, Space$, J)
        J = J + 1
    NEXT I
    CRT "Parameters read into Program from Proc:"
    FOR I = 1 TO Total.Parameters
        CRT Buffer.Array$(I)
    NEXT I
END ELSE
    CRT "Proc Buffer missing"
    STOP
END; * Procread Buffer$

```

```

*****
* Write to the Buffer *
*****
* Add "Program#" to show that the program has written the elements
Buffer$ = "";           * Empty current contents
FOR I = 1 TO Total.Parameters
    Buffer.Array$(I) = "Program#" : Buffer.Array$(I)
NEXT I
* Separate each element with a space
FOR I = 1 TO Total.Parameters
    Buffer$ = Buffer$ : Buffer.Array$(I) : Space$
NEXT I
PROCWRITE Buffer$
*****
* End Program *
*****
END

```

User Exits

For example:

```

CRT ICONV(Parameter$, User.Exit$)
CRT OCONV(Parameter$, User.Exit$)

```

User exits are hexadecimal entry points into Assembly Language routines that can be employed either from PICK/BASIC, through ICONV or OCONV statements, or from a PROC. The need for user exits has been mostly eliminated now that the EXECUTE statement is available.

SYSTEM: DIRECT ENTRY OF COMMANDS

IBM PC BASIC

One of the great conveniences of an interpreted language is that most commands can be tested directly, without the need to put them into a program first.

To test whether a function will result in a function call error if given a null or negative parameter, enter BASIC from IBM PC DOS command level with the BASICA command and then type the statements to be tested. For example, to find out how the SPACE\$ function will react to a negative parameter:

```
PRINT SPACE$( - 10)
```

The conclusion: SPACE\$ will not accept the negative number.

To exit BASIC and return to IBM PC DOS command level, give the command:

```
SYSTEM
```

(On the Radio Shack 100, exit with the command MENU.)

SYSTEM: PARTITION CHANGE FROM IBM PC DOS TO PICK

IBM PC DOS

To switch to the Pick partition on the hard disk, the user starts the IBM PC DOS program FDISK.COM by typing FDISK at IBM PC DOS command level. The FDISK program will present two menus of options. The first menu displays the following:

1. Create DOS partition
2. Change Active Partition
3. Delete DOS partition
4. Display Partition Data

The user should select option 2, to change the active partition.

The second menu then displays the number of partitions, labeling each as "DOS" or "non-DOS," and asks which partition is to be made active. Assuming that Pick has been placed in the second partition, the correct answer would be "2."

FDISK then somewhat cryptically prompts:

Press Esc to return to FDISK Options

The correct response is to press ESCAPE twice, which will return the machine to IBM PC DOS command level. Then reboot the system by pressing CTRL-ALT-DEL. Upon starting up again, the computer will find its way to the Pick partition, which has now been designated as the active partition.

It will be much easier for the user to invoke a single command to carry out the whole operation automatically. The following example uses the IBM PC DOS batch file Pick.bat to call two small programs created with the DOS debugger. The first program, FDiskKey.com, will give the FDISK procedure the proper keystrokes. The second program, Reboot.com, will reboot the computer.

Pick.bat Batch File

```
ECHO OFF
CLS
REM Pick.bat: Switch to Pick partition. David L. Clark 09/04/1988
REM FDisk.com, FDiskKey.by, Reboot.com must be in root directory
C:
CD \
FDisk < FDiskKey.by
Reboot
```

Pick.bat first utilizes the FDiskKey.by file. The IBM PC DOS redirection symbol "<" sends the appropriate keystroke bytes stored in FDiskKey.by to the FDisk program. The FDiskKey.by file will simply hold the following keystrokes:

"2"
Carriage Return

"2"
Carriage Return
Escape
Escape

To create FDiskKey.by, at IBM PC DOS command level first call up the DOS debugger with the command:

DEBUG

In response to the DEBUG hyphen prompt, enter the following (after the hyphen):

-F100 L6 "2" 0D "2" 0D 1B 1B

The number "2" in quotes asks FDisk for the second choice on its menu. To take a choice other than the second, if Pick were in a third partition, change the second number "2" to "3." (Make sure that it is the second number that is changed, not the first, where a number "3" would delete the partition.) As always, pay attention to the difference between the letter "O" and the number zero "0," and between lower case "l" and the number one "1."

The command "F100" instructs DEBUG to fill memory from offset 100, which is DEBUG's default starting point for files. The next command, "L6," instructs DEBUG that 6 bytes will be loaded. The bytes will be the following 6 keystrokes, which will be sent to FDisk. The first keystroke for FDisk is the choice for number "2" on the FDisk menu. The next keystroke is a Carriage Return. The Carriage Return is ASCII 13 in decimal or 0D in hexadecimal. Another choice "2" is then taken, again followed by a Carriage Return. The sequence ends with two escape characters, which are ASCII 27 in decimal or 1B in hexadecimal.

After entering this sequence, in response to the next DEBUG hyphen prompt enter:

-R CX

The "R" command will display the contents of the "CX" register, and prompt for a desired change. The CX register will hold the number of bytes for the FDiskKey.com file. DEBUG will respond:

CX 0000

and on the next line present a colon prompt:

:

At the colon enter the number "6":

:6

when the hyphen prompt reappears, enter

-N FDISKKEY.BYT

The “N” command is to name a file. It is followed by a space and the name of the file, with extension.

At the next hyphen prompt, enter:

-W

The “W” command will write the FDiskKey.byt file to disk. DEBUG will display:

Writing 0006 bytes

To make sure that everything has been typed correctly, at the hyphen prompt enter the “L” command to load:

-L

Then enter the “D” command to display the 6 keystrokes at offset 100:

-D 100 L6

After the address, the result should be:

32 0D 32 0D 1B 1B

Note that the “2” in quotes has been changed to the ASCII hexadecimal value of the character, “32.” If the character “3” was entered for the second number, the result now would be the hexadecimal “33.” Make sure that “33” does not appear for the first number in the sequence.

To quit DEBUG, at the next hyphen prompt enter the “Q” for quit command:

Q

Pick.bat ends by passing control to Reboot.com, which reboots the system, performing the equivalent of pressing CTRL-ALT-DEL.

Reboot.asm contains the following code, which could be entered with a word processor and then assembled with the Microsoft macro assembler into Reboot.com:

```
MOV  AX,40
MOV  DS,AX
MOV  AX,1234
MOV  [0072],AX
JMP  F000:E05B
```

If no assembler is available, the following steps will employ DEBUG's assembler for the purpose. Enter DEBUG again with the DEBUG command:

DEBUG

Invoke the assembler with the "A" command and the offset 100:

A 100

Hexadecimal numbers will appear, which vary between machines. Enter the five lines of code for Reboot.com shown above.

After entering the code, press ENTER again. The hyphen prompt will reappear. Enter the command:

-U 100 L10

The "U" command will unassemble 10 bytes starting at offset 100. Make sure the display matches the code for Reboot.com. If it does not match, start again with the "A 100" command.

When the code has been entered correctly, at the hyphen prompt, enter the "R" command to replace the contents of register CX:

-R CX

DEBUG will respond:

CX 0000

and on the next line present a colon prompt:

:

At the colon enter the number "10":

:10

When the hyphen prompt reappears, enter the "N" command to name the file:

-N REBOOT.COM

Use the "W" command to write the file to disk:

-W

DEBUG will display:

Writing 0010 bytes

Quit DEBUG with the "Q" command:

-Q

The process can be activated by entering the command "PICK" or "pick" or "Pick" at IBM PC DOS command level.

(The process was checked with IBM PC DOS versions 2.0, 3.1, and 3.3. Beware of any future changes in the FDISK command menu, which would cause a keystroke to have a different result. Consult the IBM PC DOS manual for the FDISK command and compare the options to those described here.)

SYSTEM: PARTITION CHANGE FROM PICK TO IBM PC DOS

Pick

The switch is much easier in Pick, because of the ability of both PICK/BASIC and Pick PROCs to place data in the stack for use as input by a following process. (See the section: "System: Call System Commands from a Program.") In each case that follows, the input which the FDISK program requires will be placed in the stack.

The FDISK command will abort if any terminals are attached to the IBM PC/XT, PC/AT or 386 computer. The "ECHO OFF" command will not turn off the display of system messages by the FDISK program. "REBOOT" is a Pick command (contained in a PROC supplied by Pick Systems), it is not a special program that must be typed in.

PICK/BASIC

The following PICK/BASIC program will switch to the IBM PC DOS partition. The program should be named "DOS" and cataloged with the CATALOG verb, thereby allowing the user to switch over to IBM PC DOS simply by typing the word DOS.

DOS Program

```
* DOS David L. Clark 04/11/1987
* Switch to IBM PC DOS partition
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: Dos.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
  EQU Escape.Key$ TO CHAR(251)
  * Load input expected by the FDISK command, then execute
  DATA "2"
  DATA "1"
  DATA Escape.Key$
  DATA Escape.Key$
  EXECUTE "FDISK"
  * Reboot the system
  CHAIN "REBOOT"
  END
```

Pick TCL

The following DOS PROC will switch from Pick to the IBM PC DOS partition. The item containing the PROC could be named “DOS” and placed in the MD, thereby allowing the user to switch over to IBM PC DOS simply by typing the word “DOS”.

DOS PROC

```
PQ
C DOS Switch to IBM PC DOS partition. David L. Clark 04/04/1987
C Pick Proc.
C Pick IBM PC-XT R83 2.0
C Filename under IBM PC DOS: Dos.prc
STON
H2<
H1<
H[<
H[<
STOFF
HFDISK
PH
HREBOOT
P
```

The symbol that appears to be a left bracket after “H” is entered by pressing the ESCAPE key. The “<” adds a Carriage Return—Line Feed to the input.

Transfer



TRANSFER: FROM IBM PC DOS TO PICK

Pick TCL

COPYDOS

For example:

COPYDOS Drive: { \ Directory1 \ Directory2 \ } Filename.Extension { (Options) }

The COPYDOS command is given from the Pick partition, at Pick TCL level. The complete IBM PC DOS path of drive, directory levels, filename, and file extension must be used. The Pick System will respond:

TO:(

which should be filled in as:

TO:(Filename Item.ID

For example, to retrieve a file from the DOS partition, of the C drive, in the Pick directory, with the name TESTDATE.BP, and place it in the Pick partition, in the current account, in the file TEST.BP, under the name TEST.DATE, overwriting any item that already has that name:

```
COPYDOS C:\PICK\TESTDATE.BP (O)
TO:(TEST.BP TEST.DATE
```

The IBM PC DOS file must end with an end of file marker, also known as CTRL-Z or ASCII 26. The marker is the standard ending for an IBM PC DOS plain ASCII file, however, many word processors will not include it automatically. The character can usually be created, from within a DOS word processor, by holding down the ALT key, typing “26” on the numeric keypad, and then releasing the ALT key.

Transfer Database Records from IBM PC DOS to Pick

A PICK/BASIC program to bring database records from IBM PC DOS could loop to take a predetermined number of attributes. Each group of attributes would be written to a file as a single item, taking attribute 1 as the item ID, and storing the rest in the item under that item ID.

A faster and simpler procedure, however, would be to use single character translation, as provided for in the options of the COPYDOS command, to change the DOS delimiters to Pick delimiters. The following example shows the transfer of database records from R:Base in IBM PC DOS to Pick.

First, in R:Base, show the parameters with the command SHOW:

```
SHOW
```

Note the definitions for QUOTES, COMMA, and NULL. When data is unloaded from R:Base in ASCII format, the data elements will be delimited in the same manner as in a BASIC data file. Each field within a record will be separated by a comma. String fields also will be surrounded by quote marks. Every record will terminate with a Carriage Return—Line Feed.

The easiest conversion for Pick will be single character replacement, using the R:Base delimiters. Commas and Carriage Returns will be changed to attribute marks, while Line Feeds and quote marks will be deleted. For this operation to work correctly, none of the R:Base delimiters also can appear in the text. Change the delimiters with the SET command to characters that normally would not be used. Also change the null character from “-0-” to a null.

To set new file parameters in R:Base:

```
SET QUOTES = <
SET COMMA = ^
SET NULL < <
```

Note that there is no “=” in the SET NULL statement. The quote mark has been changed to “<”, the less than character, and the comma has been changed to “^”, a caret. If any subsequent statement requires quote marks, such as the SET NULL command, the “<” must be used. If a search command is necessary, use “LT” for “less than” and “le” for “less than or equal.” To make certain of the assignments, give the SHOW command again:

SHOW

Now unload the data into a file from R:Base:

```
OUTPUT Filename.dat
UNLOAD DATA FOR DataName AS ASCII +
SORTED BY FieldName1 FieldName2 +
WHERE FieldName3 CONTAINS <L.A. History< +
AND FieldName4 EXISTS AND FieldName5 FAILS
OUTPUT SCREEN
EXIT
```

In R:Base a command can be continued to the next line by placing a “+” at the end of the current line. R:Base uses “WHERE” while Pick uses “WITH.” The SORTED BY clause must appear before the WHERE clause. Neither are required.

R:Base employs the CONTAINS operator in place of the brackets used by Pick. The EXISTS operator confirms that a field is nonnull, the FAILS operator confirms that it is null. Because quotes have been redefined as the “<” character, this character is used when quote marks are required in a command.

In Pick, at TCL level, use COPYDOS to bring over the file:

```
COPYDOS C:\RBASE\FILENAME.DAT (MT)
TO:(FILE.DATA ID.
```

The “M” option indicates that the file is to be divided into multiple items. The “T” option indicates that the user intends to requires a translation of certain characters.

The “ID.” specification will cause Pick to assign each item an item ID beginning with “ID.” followed by a sequential number. If no item ID root is specified in the command, the filename will be used as the starting point for the item ID.

Pick asks for the length of fixed-length records, or for the number of field delimiters:

ENTER LENGTH OF RECORD (OR Dn):

Respond to the prompt with “D” and the number of fields that will become Pick attributes. For example:

ENTER LENGTH OF RECORD (OR Dn) : D22

Pick then asks for the first character that is to be replaced:

REPLACE:

Respond with a two-digit hexadecimal number. As soon as two digits are typed, the next prompt “WITH:” will appear. The following sequence will convert the file for Pick’s use:

```

REPLACE: 5E WITH: FE      {"^" with Attribute Mark}
REPLACE: 3C WITH: 3C      {"<" is deleted}
REPLACE: 0D WITH: FE      {Carriage Return with Attribute Mark}
REPLACE: 0A WITH: 0A      {Line Feed is deleted}
REPLACE:

```

Press ENTER without typing anything when done with characters to replace. Pick then asks the user to approve the replacement list:

```
OKAY (Y/N) : Y
```

When the replacement list has been approved, Pick shows the progress of the file transfer:

```

READING DIRECTORY
  RBASE
  FILENAME.DAT
WRITING ITEM
  ID.0
  ID.1
  ID.2
END OF FILE

```

The following TRIM.FILE program will be used to trim excess spaces from the attributes, because R:Base has situated each numeric field in a fixed number of spaces.

The program also will trim any empty items. R:Base ended each record with a Carriage Return—Line Feed, and ended the file with the end of file mark ASCII 26, which Pick requires for transfer of an IBM PC DOS file. However, the appearance of the ASCII 26 immediately after the last Carriage Return—Line Feed will cause Pick to create an empty additional item. The TRIM.FILE program will delete the item.

TRIM.FILE Program

```

* TRIM.FILE  David L. Clark  08/28/1988
* Trim excess spaces and empty Items from a file.
* Pick Basic program.
* Pick IBM PC-XT R83 2.0
* Filename under IBM PC DOS: TrimFile.bp
* Copyright 1988 David L. Clark -- TAB BOOKS Inc.
* Use after transfer of a database file from IBM PC DOS.
*****
* Assign Constants, Parameters *
*****
PROMPT ""
EQU Attrib.Mark$ TO CHAR(254)
EQU False TO 0
EQU True TO 1
EQU Value.Mark$ TO CHAR(253)
*****

```



```

*****
* Get Input from User *
*****
CRT
CRT "Name of file to trim: " :
INPUT Filename$
IF Filename$ = "" THEN GOTO 9999; *|End Program|
OPEN Filename$ TO File$ ELSE
    CRT "No file found: " : Filename$
    GOTO 9999; *|End Program|
END
* Giving a specific number of Attribs can avoid spending
* unnecessary time trimming a large final text Attrib.
* If actual number of Attribs is less, null Attribs will be created.
CRT "Enter number of Attributes or press ENTER for ALL: " :
INPUT Response$
IF Response$ = "" THEN Response$ = "ALL"; * Default
IF NUM(Response$) THEN
    All.Attribs.Flag = False
    Total.Attribs = Response$ + 0
END ELSE
    All.Attribs.Flag = True
END
CRT "1. Trim Attributes only 2. Trim Multi-Values" :
CRT " Pick a Number [1]: " :
INPUT Response$
Attrib.Only.Flag = (Response$ # "2")
CRT "1. Trim all Items 2. Selected Items only" :
CRT " Pick a Number [1]: " :
INPUT Response$
IF Response$ = "" THEN Response$ = "1"; * Default
IF NUM(Response$) THEN
    Num.Choice = Response$ + 0
END ELSE
    GOTO 9999; *|End Program|
END
Command$ = \SELECT \ : Filename$
* Show command output rather than capturing
IF Num.Choice = 1 THEN; * All Items
    EXECUTE Command$
END ELSE; * Selected Items
    *****
    * Get Selection Criteria from User *
    *****
    CRT
    CRT "Enter Item IDs or selection condition: " :
    INPUT Condition$
    Command$ = Command$ : \ \ : Condition$
    EXECUTE Command$
END
*****
* Trim File *
*****
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
    READNEXT ID$ THEN
        READ Item$ FROM File$, ID$ THEN
            IF Item$ # "" THEN
                IF All.Attribs.Flag THEN

```

```

        Total.Attribs = DCOUNT(Item$, Attrib.Mark$)
    END
    IF Attrib.Only.Flag THEN
        FOR I = 1 TO Total.Attribs
            Item$<I> = TRIM(Item$<I>)
        NEXT I
    END ELSE
        FOR I = 1 TO Total.Attribs
            Attrib$ = Item$<I>
            Total.Values = DCOUNT(Attrib$, Value.Mark$)
            FOR J = 1 TO Total.Values
                Attrib$<1, J> = TRIM(Attrib$<1, J>)
            NEXT J
            Item$<I> = Attrib$
        NEXT I
    END; * If only trimming Attribs
    WRITE Item$ ON File$, ID$
END ELSE
    *****
    * Delete Empty Item *
    *****
    DELETE File$, ID$
    END; * If Item$ not empty
END ELSE
    CRT "No such Item: " : ID$
    END; * Read
END ELSE
    Quit.Flag = True
    END; * Readnext
REPEAT
*****
*****
9999 * End Program *
*****
END

```

TRANSFER: FROM PICK TO IBM PC DOS

IBM PC DOS Command Level

COPYPICK

The COPYPICK command is given from the IBM PC DOS partition, at DOS command level. Upon entering the command, prompts will appear on the screen for the following information. The answers shown will take the program source item TEST.DATE from the TEST.BP file in the TEST account in the Pick partition into the file TEST-DATE.BP in the current directory in the IBM PC DOS partition:

```

Enter PICK account name: TEST
Enter PICK filename (or dict, dataname): TEST.BP
Enter PICK item name or <cr> for all: TEST.DATE
Enter DOS receive filename: TESTDATE.BP

```

To exit from COPYPICK, give a null response to the next account name, filename, and item name prompts.

TRANSFER: THROUGH A SERIAL PORT

IBM PC BASIC

ON COM GOSUB Line.Number

For example:

```
ON COM(Serial.Port.Number) GOSUB 100          '* IBM PC
ON COM GOSUB 100                               '* Radio Shack 100
```

A communication received through the serial port will interrupt the program and cause it to branch to the designated line number. The line number must be a literal, not a variable.

The IBM PC version of the command requires including the number of the serial port in parentheses, because the PC supports two serial ports. The number must be a literal "1" or "2," rather than a variable or expression. Because the Radio Shack 100 only supports one serial port, no number is needed.

COM ON, COM OFF, COM STOP

IBM PC:

```
COM(Serial.Port.Number) ON
COM(Serial.Port.Number) OFF
COM(Serial.Port.Number) STOP
```

Radio Shack 100:

```
COM ON
COM OFF
COM STOP
```

The communications interrupt must first be enabled with the COM ON statement. It can be disabled with the COM OFF statement. COM STOP also disables the interrupts, but stores the information that a serial port communication was received, so that a subsequent COM ON command will cause branching to the designated line number.

Transfer between IBM PC and Radio Shack 100

The following instructions provide a method of transferring a file between the IBM PC and the Radio Shack 100 using the built-in Telcom program on the 100 and the IBM PC DOS Operating System commands on the IBM PC. If there are any transmission errors, decrease the speed used. Remember that the two machines must use the same speed.

Connect the serial ports of the two machines with a cable and a null modem. The serial port can be labeled as "RS-232C".

Transfer File from Radio Shack 100 to IBM PC

On the IBM PC.

At the IBM PC prompt enter:

```
MODE COM1:48,E,7,1
```

The command will set serial port #1 for a transmission speed of 4800 baud, with even parity, 7 data-bits, and 1 stop-bit. If a printer or modem is using serial port #1, type COM2, rather than COM1. The parameters must be reset each time that the computer is turned on.

On the Radio Shack 100.

The file to send must have two Carriage Returns at the end of the portion of the file that is to be transmitted. If two Carriage Returns in a row occur before the end of the file, only that portion will be sent.

Enter the 100's Telcom program.

Press F3 for "Status."

Type 77E1E for a transmission speed of 4800 baud, with a word length of 7 data-bits, even parity, 1 stop-bit, and the Xon/Xoff feature enabled. The Radio Shack 100 parameters remain until changed.

On the IBM PC

The command has the form:

```
COPY COM1 Drive:Path \ Filename.Extension
```

Supply the actual names for the drive, path, filename, and extension. For example:

```
COPY COM1 C:\ Word \ Story.doc
```

The command will copy the file that is received from serial port #1 to the "Word" directory on the "C" drive. The file will be assigned the name "Story.doc" when it arrives on the IBM PC, no matter what its name was originally on the 100.

On the Radio Shack 100, within the Telcom program

Press F4 for "Terminal." Press F3 for "Upload."

At the prompt "File to Upload?" type the name that the file actually has on the Radio Shack 100. Type a filename, period, and extension, then press ENTER. The extension will be ".do" because only document files can be transferred in this way. (To transfer a BASIC program, first save it within BASIC in ASCII format with the statement: SAVE "Filename", A. The quote marks are required around the filename.)

At the prompt "Width?" type 80.

For example:

```
File to Upload? Story.do  
Width? 80
```

When transmission of the file has been completed, the IBM PC will display the message "1 File copied."

If the transmission does not terminate, tap the ENTER key on the Radio Shack 100 twice.

If the IBM PC reports a read error, type R for retry, and repeat the Radio Shack 100 sequence.

Transfer File from IBM PC to Radio Shack 100

On the IBM PC

At IBM PC DOS command level enter:

```
MODE COM1:60,E,7,1
```

The command will set serial port #1 for a transmission speed of 600 baud, with even parity, 7 data-bits and 1 stop-bit. The speed is much slower than from the Radio Shack 100 to the IBM PC, because the transfer is now taking place from the more powerful to the less powerful machine. If a printer or modem is using serial port #1, type COM2, rather than COM1.

On the Radio Shack 100

Enter the Telcom program.

Press F3 for "Status."

Type 47E1D for a transmission speed of 600 baud, with a word length of 7 data-bits, even parity, 1 stop-bit and the Xon/Xoff feature disabled. Press F4 for "Terminal."

Press F2 for "Download."

At the prompt "File to Download?", type a filename, period, and extension, then press ENTER. Choose the name that the file is to have on the Radio Shack 100. Remember that on the 100, a filename can have only six characters, compared to eight characters on the IBM. Therefore you might have to shorten the original filename. The extension will be ".DO" because only document files can be transferred through the modem. (BASIC programs must be sent as document files, in ASCII format, and then loaded into BASIC.)

For example:

```
File to Download: Story.do
```

After entering the filename, the "Down" indicator for "Download" at the bottom of the Radio Shack 100 screen will change to reverse video.

On the IBM PC

The command has the form:

```
COPY Drive:Path \ Filename.Extension COM1
```

Supply the actual names for the drive, path, filename, and extension. For example:

```
COPY C:\ Word \ Story.doc COM1
```

The command will copy the file "Story.doc" from the "Word" directory on the "C" drive to serial port #1.

When it has completed transmission of the file, the IBM PC will display the message "1 File copied."

On the Radio Shack 100.

The file being transmitted will appear on the Radio Shack 100 screen. When the IBM PC indicates that the file has been sent, to terminate the download on the 100 press F2 for "Download" again. The "Down" indicator will change back to normal video.

To exit Telcom at the end of the session:

Press F8 to end communication.

At the prompt "Disconnect?" type Y for "Yes," then press ENTER, then press F8 again.

TRANSFER: TO AND FROM TAPE

Pick TCL

T-ATT, T-REW, T-DET

All of the Pick commands for diskettes are the same as the commands for tape. Each session using diskettes or tape should begin with the commands to attach and rewind a tape:

T-ATT
T-REW

The session should end with the command to detach the tape:

T-DET

Many Pick TCL commands can be used for tape or diskette operations by adding the modifier "TAPE" or "(T)." Consult the options available with each individual command. For copying files, items, or an entire database to diskette or tape, see the section: "Files, Items: Copy, Backup."

PICK/BASIC

READT, WRITET, SYSTEM(0)

The READT and WRITET commands will read from and write to a tape or diskette in the same manner as the READ and WRITE commands will read from and write to the currently installed system. When reading or writing with tape or diskette, use the SYSTEM function with a zero parameter to determine the cause of a failure to read or write:

```
READT Item.from.Tape$ ELSE  
  BEGIN CASE
```

```

CASE SYSTEM(0) = 1
  CRT "Please attach tape recorder"
CASE SYSTEM(0) = 2
  CRT "End of file"
END CASE
END; * Read Tape

```

```

WRITET Item.to.Tape$ ELSE
  BEGIN CASE
    CASE SYSTEM(0) = 1
      CRT "Please attach tape recorder"
    CASE SYSTEM(0) = 3
      CRT "The Item is empty"
    CASE SYSTEM(0) = 11
      CRT "The Item is longer than the tape record length"
  END CASE
END; * Read Tape

```

(For details on the SYSTEM function, see the section: "System: Access to System Information.")

TRANSFER: TO SCREEN OR PRINTER

In some cases the screen or a printer can be treated as a file to which a transfer can be made. You cannot make a transfer from the screen or printer. The device is "opened" as a file, text is written to it, and then the file (device) is closed.

IBM PC BASIC

IBM PC BASIC assumes a printer carriage width of 80 characters. Therefore BASIC automatically adds a Carriage Return—Line Feed after printing 80 characters with the LPRINT statement. This limitation can be avoided by treating the printer as a file, as shown below, thereby allowing condensed print or a wider carriage return of 132 characters.

However, each print statement will be at least three characters longer. Rather than writing "LPRINT X\$" the statement will be "LPRINT #1, X\$".

LPT1:, LPT2:, LPT3:

The line printer (parallel printer port) is designated as "LPT1:", "LPT2:" or "LPT3:".

```

OPEN "LPT1:" AS #1      '* Printer
PRINT #1, "This text will be printed"
CLOSE #1

```

Output to a serial printer would occur through the communications ports "COM1:" or "COM2:". (See the section: "Transfer: Through a Serial Port.")

Microsoft BASIC (Radio Shack 100)

LPT:

The line printer (parallel printer port) is designated as "LPT:".

```
OPEN "LPT:" FOR OUTPUT AS #1      '* Printer
PRINT #1, "This text will be printed"
CLOSE #1
```

Output to a serial printer would occur through the communications port "COM:". (See the section: Transfer: Through a Serial Port.)

LCD:

The liquid crystal display is designated as "LCD:".

```
OPEN "LCD:" FOR OUTPUT AS #1      '* Screen
PRINT #1, "This text will be displayed on the screen"
CLOSE #1
```


Variables



VARIABLES: INITIAL VALUES

Is it legal to test a variable before any value has been assigned to it?

IBM PC BASIC

Yes. An unassigned variable will be considered to have a null value.

PICK/BASIC

No. Either a value must be assigned, or the CLEAR statement must be used.

CLEAR

CLEAR sets all the variables equal to null. The CLEAR statement should be avoided because the programmer will receive no warning message for a misspelled variable name.

VARIABLES: NAMES, LIMITS

All languages forbid the use of reserved words such as AND and OR as names.

IBM PC BASIC

A name can be up to 40 characters in length. It must begin with a letter, and can include numbers.

IBM PC BASIC automatically converts all letters to uppercase, unless they are part of a string in quotation marks, a DATA statement, or a comment. Therefore there will be no difference between the variables ED\$ and Ed\$. The interpreter will change Ed\$ to ED\$.

The name can end with one of the following type indicators:

- ☐ \$ = String, limit of 255 characters
- ☐ % = Integer
- ☐ ! = Single precision - 6 digits
- ☐ # = Double precision - 14 digits

If there is no type symbol, single precision is assumed, unless the letter with which the name begins has been defined. The statements:

```
DEFINT A-W  
DEFSTR X-Z
```

would define all variables beginning with the letters A through W as integers and X through Z as string variables. Type symbols on variable names override any prior type definitions.

Type endings will make a different variable out of the same name, for a type other than the default type. Thus if the program began with the statement "DEFINT A-W", making integer the default type for a variable beginning with the letter "N", then NU, NU\$, NU! and NU# would be different variables. However NU and NU% would be the same.

The same name also can be used for both a single and an array variable: thus NU and NU() would be different, as would NU\$, NU\$(), NU!, NU!(), and so on.

A variable name on the IBM PC also can include periods. A name beginning with FN is assumed to be a user-defined function.

Microsoft BASIC (Radio Shack 100)

In Microsoft BASIC on the Radio Shack 100, and in simpler versions of BASIC generally, the period is not allowed in variable names, and only the first two characters are significant. Thus the interpreter would see NU, NUMBER, NUMBERCHARS, and NUMBERWORDS as the same variable.

Because only the first two characters matter, be careful that those characters do not duplicate a reserved word. You could not use the name ORDERS, because the interpreter would see the reserved word OR. A syntax error would result when the program was run.

PICK/BASIC

A name can be up to 64 characters in length. It must begin with a letter, and then can include numbers, periods, and dollar signs. Both upper- and lowercase letters can be used, and they will be regarded as different.

The same characters cannot be used for both a single and an array variable, thus to have both Num.Chars and Num.Chars() would be illegal.

A string can have a maximum length of 32,267 characters.

A program can hold a maximum of 3200 variables. Each element of a dimensioned array is considered a separate variable, whereas a dynamic array is considered a single variable separated by delimiters.

VARIABLES: SCOPE

In the IBM PC BASIC and PICK/BASIC all variables are global, with the same value throughout the program. You must be aware of unintended side effects, which occur when one routine changes the value of a variable, but the action has not been taken into account by other routines.

Appendix

Programs and Instructions or Pick Release 2.2 and 3.0

FOLLOWING ARE PROGRAMS AND INSTRUCTIONS FOR SPECIFIC ITEMS COVERED IN THIS book that are different for Pick Release 2.2 and 3.0 than for other versions of Pick. They are presented here in the same order found in the main portion of the book.

Foreign Language Characters for Pick 2.2, 3.0

Pick Release 1983 2.2 and 3.0 do not provide quite as much flexibility for foreign language characters. ASCII 250 is defined as the lead character for the degree symbol, 251 for the circumflex, 253 for the accent acute, 252 for the accent grave, and 254 for the umlaut. Pressing a key that has been defined to return a lead character followed by a key for a letter that uses the lead character's accent symbol will return a letter with the appropriate accent. For example, ASCII 251 followed by "o" will return "o" with the circumflex. There is no method however, for using the lead characters to produce a Spanish "n" with a tilde, a French "c" with a cedilla, or other combinations that previously could be defined. (Note that the Pick manual incorrectly states that 252 will return the accent accute and 253 will return the accent grave.)

BUILD.KEYS.FILE Program for Pick 2.2, 3.0

A different version of the Build Keys File program must be used for Pick R83 Versions 2.2 and 3.0, under which 128 key positions are defined, in order of the IBM scan codes returned. The Pick keyboard definitions still appear in the KEYBOARDS file in the SYSPROG Account, but the definitions distinguish between keyboards of 84 keys, as on the original IBM PC and PCXT, and keyboards of 101 keys, for the PCAT and PS/2. The

standard USA keyboards are represented by the Items "USA-84" and "USA-101". The program will place the keyboard definitions in a file named KEYBOARD.CHOICES-IBM.PC.R83.30. The Item will retain the original Item ID it had in the KEYBOARDS file.

```
* BUILD.KEYS.FILE.IBM.PC.R83.30 David L. Clark 07/20/1989
* Original program BUILD.KEYS.FILE.IBM.PC.XT.R83 for IBM PC-XT R83 2.0
* adapted for Pick R83 2.2 and 3.0
* Create the file KEYBOARD.CHOICES.IBM.PC.R83.30 of alternative keyboards.
* Pick Basic program.
* Pick IBM PC R83 3.0
* Filename under IBM PC DOS: BdKyFiPC.bps
* Copyright 1989 David L. Clark -- TAB BOOKS Inc.
* Reformat Items in the KEYBOARDS file supplied by Pick Systems for the
* IBM PC into Items in the KEYBOARD.CHOICES.IBM.PC.R83.30 file used by
* the CHANGE.KEYS.IBM.PC.R83.30 external subroutine.
* Pick R83 2.2 and 3.0 define 4 sets of 128 key positions each
* by IBM scan codes. The counting for the 128 positions is zero
* through 127. Each set contains a blank (hex FF) at positions 0, 89-127.
* The program stores the keyboard in the form expected by User Exit
* "U80BA", used to change the keyboard in the external subroutine
* CHANGE.KEYS.IBM.PC.R83.30. User Exit "U80BA" expects each set of
* Shifted, Unshifted, Control, and Alt keys separately, each with the
* following elements:
*   The number of the set, with the sets numbered 0 - 4.
*   A Value Mark.
*   A set of 128 scan codes in hex, 2 characters each
* For example, for the 1st set, of Shifted Keys:
*   Error.Num = ICONV("0" : Value.Mark$ : Key.Set.Hex(1), "U80BA")
* The function keys are included in each 128 key set. The function keys can
* be redefined as part of the keyboard redefinition for each 128 key set
* with User Exit "U80BA" or redefined as 4 separate 12-key sets with User
* Exit "UA0BA". The function key redefinitions follow the same pattern as
* the keyboard redefinitions. Each set is preceded by a set number, with
* numbering starting at zero, a value mark, and 12 scan codes in hex, 2
* characters each.
* With R83 2.2 and 3.0 Pick also offers a Lead Character for the function
* keys. When a function key is pressed, the key will return first the
* Lead Character, and then the character definition for the particular
* key. The default Lead Character is ASCII 2. The redefinition of the
* Lead Character is treated as a 5th function key set (numbering 0-4). To
* eliminate the Lead Character, redefine it as blank (hex FF):
*   Error.Num = ICONV("4" : Value.Mark$ : "FF", UA0BA")
* The program will store a hex value for the Function key Lead Character as
* a 5th key set, with a default character of ASCII 2.
*****
* Assign Constants, Parameters *
*****
*****
* Set Constants, Parameters Valid for All Models *
*****
PROMPT ""
EQU Attrib.Mark$ TO CHAR(254)
EQU False TO 0
File.Keys.Pick.Name$ = "KEYBOARDS"; * File supplied by Pick Systems
EQU True TO 1
EQU Value.Mark$ TO CHAR(253)
```

```

*****
* Set Parameters by PC Model, Version *
*****
* Change these for model other than IBM PC R83 2.2 or 3.0
* File holding alternative keyboards
File.Keys.Name$ = "KEYBOARD.CHOICES.IBM.PC.R83.30"
*****
* Parameters for Number of Key Positions *
*****
* There are 4 sets of hex key values, for: Shifted, Unshift, Ctl,
*   Alt, plus a 5th set for the Function Key Lead Character.
* Function keys are part of the keyboard definitions. They are not
*   in a separate Attribute, as for Pick R83 2.0.
EQU Total.Keybd.Sets TO 4;      * Not including Func Lead Char
EQU Total.Key.Sets TO 5;       * Including Func Lead Char
DIM Key.Set$(5)
* Define the expected length of the keyboard string, not including
*   the Function Key Lead Character:
* For each set there is a string of 128 2-character hex numbers
*   stored by Pick as the definition of the keys in that set.
* Scan.Codes.Keybd.Set: The 128 positions of 0-127 which Pick
*   stores and reads. Positions zero and 89-127 are blank "FF".
EQU Scan.Codes.Keybd.Set TO 128
* Convert from key position to character position for 2-char codes.
Scan.Codes.Keybd.Set.Length = Scan.Codes.Keybd.Set * 2
Scan.Codes.Keybd.Total = Total.Keybd.Sets * Scan.Codes.Keybd.Set
* Convert from key position to character position for 2-char codes.
Scan.Codes.Keybd.Total.Length = Scan.Codes.Keybd.Total * 2
*=====
*****
* Open Files *
*****
OPEN File.Keys.Pick.Name$ TO File.Keys.Pick$ ELSE
  CRT "Missing file: " : File.Keys.Pick.Name$
  GOTO 9999;          *|End Program|
END
OPEN File.Keys.Name$ TO File.Keys$ ELSE
  CRT "Missing file: " : File.Keys.Name$
  GOTO 9999;          *|End Program|
END
*****
* Get Pick Keyboards, Put in Choices File *
*****
EXECUTE \SSELECT \ : File.Keys.Pick.Name$ CAPTURING Capture.Output$
Quit.Flag = False
LOOP UNTIL Quit.Flag DO
  READNEXT Keyboard.ID$ THEN
    * Check if same ID already exists in Choices file
    READV Dummy$ FROM File.Keys$, Keyboard.ID$, 1 THEN
      CRT
      CRT "Keyboard Item " : Keyboard.ID$ : " already exists.":
      CRT "   Over-write old Item?"
      CRT "1. Yes   2. No       Pick a Number [1]: " :
      INPUT Num.Choice
      IF Num.Choice # "2" THEN GOSUB 100;   *|Store Keyboard|
    END ELSE
      * Give user a choice, rather than automatically storing
      *   all keyboards.
      CRT

```

```

        CRT "Do you wish to store Keyboard Item " : Keyboard.ID$
        CRT "1. Yes    2. No      Pick a Number [1]: " :
        INPUT Num.Choice
        IF Num.Choice # "2" THEN GOSUB 100;    *|Store Keyboard|
    END
END ELSE
    Quit.Flag = True
END
REPEAT
GOTO 9999;                                *|End Program|
*-----*
*****
100  * Store Keyboard *
*****
    * Parameters: Keyboard.ID$, File.Keys.Pick$, File.Keys$
    * Reformats and stores keyboard from File.Keys.Pick$ into File.Keys$
    READ Keyboard.Pick$ FROM File.Keys.Pick$, Keyboard.ID$ ELSE
        CRT "No keyboard in Item: " : Keyboard.ID$
        RETURN;                                *** Exit #1 ***
    END
    * Reformat Item into 1 string
    Total.Attribs = DCOUNT(Keyboard.Pick$, Attrib.Mark$)
    Keys$ = ""
    FOR I = 1 TO Total.Attribs
        Attrib$ = TRIM(Keyboard.Pick$<I>)
        IF Attrib$[1, 1] # "" THEN
            Keys$ = Keys$ : FIELD(Attrib$, " ", 1)
        END
    NEXT I
    * Check that the total length is correct
    Keys.Length = LEN(Keys$)
    IF Keys.Length # Scan.Codes.Keybd.Total.Length THEN
        CRT "Keyboard Item: " : Keyboard.ID$ : " is incorrect"
        CRT "String length expected: " : Scan.Codes.Keybd.Total.Length
        CRT "String length found:    " : Keys.Length
        RETURN;                                *** Exit #2 ***
    END
    * Divide long string into 4 keyboard sets for Shifted, Unshifted,
    *   Control, Alt
    FOR I = 1 TO Total.Keybd.Sets
        Key.Set.Start = 1 + (I - 1) * Scan.Codes.Keybd.Set.Length
        Key.Set$(I) = Keys$[Key.Set.Start, Scan.Codes.Keybd.Set.Length]
        * Store in form expected by User Exit "U80BA". See note at start.
        Key.Set$(I) = I - 1 : Value.Mark$ : Key.Set$(I)
    NEXT I
    * Add Func Key Lead Character, in form expected by User Exit "UA0BA".
    * Default value from Pick is ASCII 2
    Key.Set$(Total.Key.Sets) = Total.Key.Sets - 1 : Value.Mark$ : "02"
    MATWRITE Key.Set$ ON File.Keys$, Keyboard.ID$
    RETURN;                                * Normal Exit *** Exit #3 ***
    *=====*
*****
9999  * End Program *
*****
END

```

CHANGE.KEYS.FILE Program for Pick 2.2, 3.0

A different version of the program for changing the IBM PC Keyboard for Pick R83 version 2.2 and 3.0 follows.

```
* CHANGE.KEYS.FILE.IBM.PC.R83.30 David L. Clark 07/20/1989
* Change keyboard in KEYBOARD.CHOICES.IBM.PC.R83.30 file
* Pick Basic program.
* Pick IBM PC R83 3.0
* Filename under IBM PC DOS: ChKyFiPC.bp3
* Copyright 1989 David L. Clark -- TAB BOOKS Inc.
* The program stores the keyboard in the form expected by User Exits
* "U80BA" and "UA0BA", used to change the keyboard in the external
* subroutine CHANGE.KEYS.IBM.PC.R83.30.
* For explanations, see notes for program BUILD.KEYS.FILE.IBM.PC.R83.30.
*****
* Assign Constants, Parameters, Initial Values *
*****
* Set Constants, Parameters Valid for All PC Models *
*****
PROMPT ""
EQU Blank.Hex TO "FF";           * For undefined key. Stored in hex.
Clear.Screen$ = @(-1)
Erase.Screen.from.Pos$ = @(-3)
EQU False TO 0
EQU Keyboards.Max TO 20;         * Maximum number of different keyboards
DIM Keyboard.Name$(20)
Row.Bottom = SYSTEM(3)
EQU True TO 1
EQU Value.Mark$ TO CHAR(253)
*****
* Set Parameters by PC Model, Version *
*****
* Change these for model other than IBM PC R83 2.2 or 3.0
* File holding alternative keyboards
File.Keys.Name$ = "KEYBOARD.CHOICES.IBM.PC.R83.30"
*****
* Parameters for Number of Key Positions *
*****
* There are 4 sets of hex key values, for: Shifted, Unshift, Ctl,
* Alt, plus a 5th set for the function key Lead Character.
* Function keys are part of the keyboard definitions. They are not
* in a separate Attribute, as for Pick R83 2.0.
EQU Total.Keybd.Sets TO 4;       * Not including Func Lead Char
EQU Total.Key.Sets TO 5;         * Including Func Lead Char
DIM Key.Set$(5)
* For each of the 1st 4 sets (not including Function Key Lead Char)
* there is a string of 128 2-character hex numbers stored
* by Pick as the definition of the keys in that set.
* The 128 positions include many blanks.
* For each set there are thus 3 different totals:
* Keys.Defined.Set: The 78 main and func keys, spread thru key
* positions 1-88, which this program offers for redefinition.
EQU Keys.Defined.Set TO 78
* Titles for Key Sets
DIM Title$(5)
```



```

Title$(1) = "SHIFTED KEYS + NUMERIC KEYPAD"
Title$(2) = "UNSHIFTED KEYS + CURSOR KEYS"
Title$(3) = "CONTROL KEYS"
Title$(4) = "ALT KEYS"
Title$(5) = "FUNCTION KEY LEAD CHARACTER"
*****
* Setup Arrays to Hold Key Values *
*****
* Key$( )      = Name of the key.
* Key.Ascii( ) = ASCII decimal code for character returned by key.
* Key.Pos( )   = Starting position of key definition in Keyboard$.
*              = Equals position on scan code list x 2 - 1.
* Key.Scrn$( ) = Column, Row position on screen for display.
* Dimension by Keys.Defined.Set, for keys offered for redefinition
DIM Key$(78), Key.Pos(78), Key.Scrn$(78)
* Dimension by Total.Keybd.Sets, Keys.Defined.Set
DIM Key.Ascii(4, 78)
* Set key names, position in string from KEYBOARDS, screen position
* Shifted keys are first in the string
* 78 keys / 4 cols = 20 rows
* For Pick R83 3.0:
*   Print Screen key is not defined. Just an asterisk.
*   Break key is not defined.
Key$(1)  = "A";      Key.Pos(1)  = 59;   Key.Scrn$(1) = @(0, 2)
Key$(2)  = "B";      Key.Pos(2)  = 95;   Key.Scrn$(2) = @(0, 3)
Key$(3)  = "C";      Key.Pos(3)  = 91;   Key.Scrn$(3) = @(0, 4)
Key$(4)  = "D";      Key.Pos(4)  = 63;   Key.Scrn$(4) = @(0, 5)
Key$(5)  = "E";      Key.Pos(5)  = 35;   Key.Scrn$(5) = @(0, 6)
Key$(6)  = "F";      Key.Pos(6)  = 65;   Key.Scrn$(6) = @(0, 7)
Key$(7)  = "G";      Key.Pos(7)  = 67;   Key.Scrn$(7) = @(0, 8)
Key$(8)  = "H";      Key.Pos(8)  = 69;   Key.Scrn$(8) = @(0, 9)
Key$(9)  = "I";      Key.Pos(9)  = 45;   Key.Scrn$(9) = @(0, 10)
Key$(10) = "J";      Key.Pos(10) = 71;   Key.Scrn$(10) = @(0, 11)
Key$(11) = "K";      Key.Pos(11) = 73;   Key.Scrn$(11) = @(0, 12)
Key$(12) = "L";      Key.Pos(12) = 75;   Key.Scrn$(12) = @(0, 13)
Key$(13) = "M";      Key.Pos(13) = 99;   Key.Scrn$(13) = @(0, 14)
Key$(14) = "N";      Key.Pos(14) = 97;   Key.Scrn$(14) = @(0, 15)
Key$(15) = "O";      Key.Pos(15) = 47;   Key.Scrn$(15) = @(0, 16)
Key$(16) = "P";      Key.Pos(16) = 49;   Key.Scrn$(16) = @(0, 17)
Key$(17) = "Q";      Key.Pos(17) = 31;   Key.Scrn$(17) = @(0, 18)
Key$(18) = "R";      Key.Pos(18) = 37;   Key.Scrn$(18) = @(0, 19)
Key$(19) = "S";      Key.Pos(19) = 61;   Key.Scrn$(19) = @(0, 20)
Key$(20) = "T";      Key.Pos(20) = 39;   Key.Scrn$(20) = @(0, 21)
Key$(21) = "U";      Key.Pos(21) = 43;   Key.Scrn$(21) = @(19, 2)
Key$(22) = "V";      Key.Pos(22) = 93;   Key.Scrn$(22) = @(19, 3)
Key$(23) = "W";      Key.Pos(23) = 33;   Key.Scrn$(23) = @(19, 4)
Key$(24) = "X";      Key.Pos(24) = 89;   Key.Scrn$(24) = @(19, 5)
Key$(25) = "Y";      Key.Pos(25) = 41;   Key.Scrn$(25) = @(19, 6)
Key$(26) = "Z";      Key.Pos(26) = 87;   Key.Scrn$(26) = @(19, 7)
Key$(27) = "1";      Key.Pos(27) = 3;    Key.Scrn$(27) = @(19, 8)
Key$(28) = "2";      Key.Pos(28) = 5;    Key.Scrn$(28) = @(19, 9)
Key$(29) = "3";      Key.Pos(29) = 7;    Key.Scrn$(29) = @(19, 10)
Key$(30) = "4";      Key.Pos(30) = 9;    Key.Scrn$(30) = @(19, 11)
Key$(31) = "5";      Key.Pos(31) = 11;   Key.Scrn$(31) = @(19, 12)
Key$(32) = "6";      Key.Pos(32) = 13;   Key.Scrn$(32) = @(19, 13)
Key$(33) = "7";      Key.Pos(33) = 15;   Key.Scrn$(33) = @(19, 14)
Key$(34) = "8";      Key.Pos(34) = 17;   Key.Scrn$(34) = @(19, 15)
Key$(35) = "9";      Key.Pos(35) = 19;   Key.Scrn$(35) = @(19, 16)
Key$(36) = "0";      Key.Pos(36) = 21;   Key.Scrn$(36) = @(19, 17)

```

```

Key$(37) = "-";      Key.Pos(37) = 23;   Key.Scrn$(37) = @(19, 18)
Key$(38) = "=";      Key.Pos(38) = 25;   Key.Scrn$(38) = @(19, 19)
Key$(39) = "[";      Key.Pos(39) = 51;   Key.Scrn$(39) = @(19, 20)
Key$(40) = "]";      Key.Pos(40) = 53;   Key.Scrn$(40) = @(19, 21)
Key$(41) = ";";      Key.Pos(41) = 77;   Key.Scrn$(41) = @(39, 2)
Key$(42) = "\"";     Key.Pos(42) = 79;   Key.Scrn$(42) = @(39, 3)
Key$(43) = "`";      Key.Pos(43) = 81;   Key.Scrn$(43) = @(39, 4)
Key$(44) = "\\";     Key.Pos(44) = 85;   Key.Scrn$(44) = @(39, 5)
Key$(45) = ", ";     Key.Pos(45) = 101;  Key.Scrn$(45) = @(39, 6)
Key$(46) = ".";      Key.Pos(46) = 103;  Key.Scrn$(46) = @(39, 7)
Key$(47) = "/";      Key.Pos(47) = 105;  Key.Scrn$(47) = @(39, 8)
Key$(48) = "BKSP";   Key.Pos(48) = 27;   Key.Scrn$(48) = @(39, 9)
Key$(49) = "DEL";    Key.Pos(49) = 165;  Key.Scrn$(49) = @(39, 10)
Key$(50) = "END";    Key.Pos(50) = 157;  Key.Scrn$(50) = @(39, 11)
Key$(51) = "ENTER";  Key.Pos(51) = 55;   Key.Scrn$(51) = @(39, 12)
Key$(52) = "ESC";    Key.Pos(52) = 1;    Key.Scrn$(52) = @(39, 13)
Key$(53) = "HOME";   Key.Pos(53) = 141;  Key.Scrn$(53) = @(39, 14)
Key$(54) = "INS";    Key.Pos(54) = 163;  Key.Scrn$(54) = @(39, 15)
Key$(55) = "PGDN";   Key.Pos(55) = 161;  Key.Scrn$(55) = @(39, 16)
Key$(56) = "PGUP";   Key.Pos(56) = 145;  Key.Scrn$(56) = @(39, 17)
Key$(57) = "SPACE";  Key.Pos(57) = 113;  Key.Scrn$(57) = @(39, 18)
Key$(58) = "TAB";    Key.Pos(58) = 29;   Key.Scrn$(58) = @(39, 19)
Key$(59) = "DOWN";   Key.Pos(59) = 159;  Key.Scrn$(59) = @(59, 2)
Key$(60) = "LEFT";   Key.Pos(60) = 149;  Key.Scrn$(60) = @(59, 3)
Key$(61) = "RIGHT";  Key.Pos(61) = 153;  Key.Scrn$(61) = @(59, 4)
Key$(62) = "UP";     Key.Pos(62) = 143;  Key.Scrn$(62) = @(59, 5)
Key$(63) = "KPD *";  Key.Pos(63) = 109;  Key.Scrn$(63) = @(59, 6)
Key$(64) = "KPD +";  Key.Pos(64) = 155;  Key.Scrn$(64) = @(59, 7)
Key$(65) = "KPD -";  Key.Pos(65) = 147;  Key.Scrn$(65) = @(59, 8)
Key$(66) = "KPD 5";  Key.Pos(66) = 151;  Key.Scrn$(66) = @(59, 9)
Key$(67) = "F1";     Key.Pos(67) = 117;  Key.Scrn$(67) = @(59, 10)
Key$(68) = "F2";     Key.Pos(68) = 119;  Key.Scrn$(68) = @(59, 11)
Key$(69) = "F3";     Key.Pos(69) = 121;  Key.Scrn$(69) = @(59, 12)
Key$(70) = "F4";     Key.Pos(70) = 123;  Key.Scrn$(70) = @(59, 13)
Key$(71) = "F5";     Key.Pos(71) = 125;  Key.Scrn$(71) = @(59, 14)
Key$(72) = "F6";     Key.Pos(72) = 127;  Key.Scrn$(72) = @(59, 15)
Key$(73) = "F7";     Key.Pos(73) = 129;  Key.Scrn$(73) = @(59, 16)
Key$(74) = "F8";     Key.Pos(74) = 131;  Key.Scrn$(74) = @(59, 17)
Key$(75) = "F9";     Key.Pos(75) = 133;  Key.Scrn$(75) = @(59, 18)
Key$(76) = "F10";    Key.Pos(76) = 135;  Key.Scrn$(76) = @(59, 19)
Key$(77) = "F11";    Key.Pos(77) = 173;  Key.Scrn$(77) = @(59, 20)
Key$(78) = "F12";    Key.Pos(78) = 175;  Key.Scrn$(78) = @(59, 21)

```

```
OPEN File.Keys.Name$ TO File.Keys$ ELSE
```

```
GOTO 9999;                                *|End Program|
```

END

```
CRT Clear.Screen$ : @ (20) : "CHOOSE KEYBOARD FOR BEGINNING VALUES" :
```

```
FOR I = 1 TO Keyboards.Max UNTIL Quit.Flag
```

```
CRT @(30, I + 1) : I : ". " : Keyboard.Name$(I) :
```

```
Quit.Flag = True
```

```

NEXT I
CRT @(20, Row.Bottom) : "Pick a Number or (Q)uit [1]: " :
INPUT Response$ :
IF NOT(NUM(Response$)) THEN GOTO 9999; *|End Program|
IF Response$ = "" THEN Num.Choice = 1 ELSE Num.Choice = Response$ + 0
Keyboard.ID$ = Keyboard.Name$(Num.Choice)
MATREAD Key.Set$ FROM File.Keys$, Keyboard.ID$ ELSE
  CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(20) :
  CRT "No keyboard found in Item: " : Keyboard.ID$ :
  GOTO 9999; *|End Program|
END
CRT @(0, Row.Bottom - 1) : Erase.Screen.from.Pos$ :
CRT "Reading keyboard " : Keyboard.ID$ : " in file ":File.Keys.Name$
CRT "Please wait ..." :
* Load arrays from strings by position
*****
* Extract Key.Ascii() from Keys.Set$() *
*****
FOR I = 1 TO Total.Keybd.Sets; * Not including Func Lead Char
  * Remove Set number, Value.Mark$ expected by User Exit "U80BA".
  * Each keyboard set starts with blank: "FF". So start string
  * at char 3 rather than char 1 within the set.
  Key.Set$(I) = Key.Set$(I)<1, 2>[3, 9999]
  * Place in Key.Ascii() the values for the keys which this
  * program offers for redefinition or which it passes on.
  * Convert from hex to decimal.
  FOR J = 1 TO Keys.Defined.Set
    Key.Ascii(I, J) = OCONV(Key.Set$(I)[Key.Pos(J), 2], "MCXD")
  NEXT J
NEXT I
NEXT I
* For Function Key Lead Character
* Remove Set number, Value.Mark$ expected by func User Exit "UA0BA".
* Convert from hex to decimal.
Key.Func.Lead.Ascii = OCONV(Key.Set$(Total.Key.Sets)<1, 2>, "MCXD")
*****
* Set New Key Values, Not Including Func Lead Char *
*****
Quit.Flag = False
New.Set = 1
LOOP UNTIL Quit.Flag DO
  Set = New.Set
  CRT Clear.Screen$ : @(27) : Title$(Set) :
  FOR J = 1 TO Keys.Defined.Set
    CRT Key.Scrn$(J) : J : ". " : Key$(J) : " = ":Key.Ascii(Set,J):
  NEXT J
  LOOP; * While Set remains the same
  CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(15) :
  CRT "Pick a Number, (N)ext, (P)revious, or (Q)uit [N]: " :
  INPUT Response$ :
  BEGIN CASE
    CASE Response$ = "" OR Response$ = "N"; * Next, Default
      New.Set = Set + 1
      Quit.Flag = (New.Set > Total.Keybd.Sets)
    CASE Response$ = "P" AND Set > 1; * Previous
      New.Set = Set - 1
    CASE Response$ = "Q"; * Quit
      Quit.Flag = True
    CASE True; * Not a valid response. Do nothing.
  END CASE

```

```

WHILE Set = New.Set AND NOT(Quit.Flag) DO
  IF NUM(Response$) THEN
    N = Response$ + 0
    IF N >= 1 AND N <= Keys.Defined.Set THEN
      CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ :@(15):
      CRT Key$(N) : " = " : Key.Ascii(Set, N) :
      CRT "      Enter new ASCII number: " :
      INPUT Response$ :
      * Default choice to keep same ASCII number
      IF Response$ # "" THEN
        IF NUM(Response$) THEN
          Ascii = Response$ + 0;      * Convert to number
          IF Ascii > -1 AND Ascii < 256 THEN
            Key.Ascii(Set, N) = Ascii
            * Spaces on end in case old number longer
            CRT Key.Scrn$(N) : N : ". " :
            CRT Key$(N) : " = " : Ascii : "      " :
          END; * If valid range for ASCII
        END; * If a number entered for ASCII
      END; * If Response$ not null
    END; * If within range of keys defined per set
  END; * If a number entered
  REPEAT; * While Set remains the same
  REPEAT; * Until Quit.Flag
  *****
  * Set New Key Value for Func Lead Char *
  *****
  CRT Clear.Screen$ : @(20) : Title$(Total.Key.Sets) :
  CRT @(10,10) : "Function Key Lead Character = " :Key.Func.Lead.Ascii:
  CRT @(10,12) : "Use of ASCII 255 eliminates the Lead Character"      :
  Quit.Flag = False
  LOOP UNTIL Quit.Flag DO
    CRT @(10,14) : "Enter new ASCII number, " :
    CRT "      or press ENTER to accept: " :
    CRT "      Erase.Screen.from.Pos$ :
    INPUT Response$ :
    * Default choice to keep same ASCII number
    IF Response$ = "" THEN
      Quit.Flag = True
    END ELSE
      IF NUM(Response$) THEN
        Ascii = Response$ + 0;      * Convert to number
        IF Ascii > -1 AND Ascii < 256 THEN
          Key.Func.Lead.Ascii = Ascii
          Quit.Flag = True
        END; * If valid range for ASCII
      END; * If a number entered for ASCII
    END; * If Response$ null for default
  REPEAT
  *****
  * Store New Keyboard *
  *****
  CRT @(0, Row.Bottom) : Erase.Screen.from.Pos$ : @(5) :
  CRT "Name to store keyboard or (Q)uit [" : Keyboard.ID$ : "]: " :
  INPUT Response$ :
  IF Response$ = "Q" OR Response$ = "q" THEN GOTO 9999; *|End Program|
  IF Response$ # "" THEN Keyboard.ID$ = Response$
  CRT @(0, Row.Bottom - 1) : Erase.Screen.from.Pos$ :
  CRT "Storing keyboard " : Keyboard.ID$ : " in file " :File.Keys.Name$

```

```

CRT "Please wait ..." :
*****
* Load New Key.Set$() from Keys.Ascii() *
*****
* Load Key.Set$() from Key.Ascii() by position
* The definitions for positions 0 and 89 thru 127 are blank: "FF".
* For explanations, see: |Extract Key.Ascii() from Key.Set$()$|
* Only need to change the values which the program offers for
* redefinition. The other values remain in Key.Set$() and are
* carried through to the newly-defined keyboard.
FOR I = 1 TO Total.Keybd.Sets;      * Not including Func Lead Char
  FOR J = 1 TO Keys.Defined.Set
    P = Key.Pos(J)
    Ascii.Hex = OCONV(Key.Ascii(I, J), "MCDX") "R(%2)"
    Key.Set$(I)=Key.Set$(I)[1,P-1]:Ascii.Hex:Key.Set$(I)[P+2,999]
  NEXT J
  * Restore Set number, Value.Mark$ expected by User Exit "U80BA".
  * In R83 3.0 each set starts with blank: "FF". Restore blank.
  Key.Set$(I) = I - 1 : Value.Mark$ : Blank.Hex : Key.Set$(I)
NEXT I
* For Function Key Lead Character
Ascii.Hex = OCONV(Key.Func.Lead.Ascii, "MCDX") "R(%2)"
* Restore Set number, Value.Mark$ expected by User Exit "UA0BA".
Key.Set$(Total.Key.Sets) = Total.Key.Sets-1 : Value.Mark$ : Ascii.Hex
* Write to file
MATWRITE Key.Set$ ON File.Keys$, Keyboard.ID$
=====
*****
9999 * End Program *
*****
END

```

CHANGE.KEYS.IBM.PC Subroutine for Pick 2.2, 3.0

A version of the subroutine to apply a new IBM PC keyboard definition for Pick R83 versions 2.2 and 3.0 follows.

```

SUBROUTINE CHANGE.KEYS.IBM.PC.R83.30.SUB(Keybd.to.Apply$, Error.Flag)
* Apply keys from KEYBOARD.CHOICES.IBM.PC.R83.30 David L. Clark 07/20/1989
* Pick Basic external subroutine.
* Pick IBM PC R83 2.2 and 3.0
* Filename under IBM PC DOS: ChKeysPC.bs3
* Copyright 1989 David L. Clark -- TAB BOOKS Inc.
* Read the Keybd.to.Apply$ Item from KEYBOARD.CHOICES.IBM.PC.R83.30 file
* and apply with User Exits "U80BA" and "UA0BA" to change current
* keyboard and Function Key Lead Character.
* For explanations, see notes for program BUILD.KEYS.FILE.IBM.PC.R83.30.
* If no keyboard is applied, the Error.Flag returned to the caller as True
* and an error message is displayed.
*****
* Assign Constants, Parameters *
*****
PROMPT ""
* Error.Flag is received as False
EQU False TO 0
Port.Num$ = FIELD(OCONV(0, "U50BB"), " ", 1)
Port.Num = Port.Num$ + 0;      * Convert to number

```

```

EQU True TO 1
EQU Value.Mark$ TO CHAR(253)
*****
* Set Parameters by PC Model, Version *
*****
* Change these for model other than IBM PC R83 2.2 or 3.0
* File holding alternative keyboards
File.Keys.Name$ = "KEYBOARD.CHOICES.IBM.PC.R83.30"
Keys.User.Exit$ = "U80BA"
Funcs.User.Exit$ = "UA0BA"
*****
* Parameters for Number of Key Positions *
*****
* There are 4 sets of hex key values, for: Shifted, Unshift, Ctl,
* Alt, plus a 5th set for the Function Key Lead Character.
* Function keys are part of the keyboard definitions. They are not
* in a separate Attribute, as for Pick R83 2.0.
EQU Total.Keybd.Sets TO 4;      * Not including Func Lead Char
EQU Total.Key.Sets TO 5;      * Including Func Lead Char
DIM Key.Set$(5)
=====
*****
* Check for Port Zero *
*****
IF Port.Num # 0 THEN
    Error.Flag = True
    CRT "This program must be run from an IBM PC on Port zero"
    CRT "Press ENTER: " :
    INPUT Response$
    GOTO 9999;                  *|End Subroutine|
END
*****
* Open Files *
*****
OPEN File.Keys.Name$ TO File.Keys$ ELSE
    Error.Flag = True
    CRT "Missing file: " : File.Keys.Name$
    CRT "Press ENTER: " :
    INPUT Response$
    GOTO 9999;                  *|End Subroutine|
END
*****
* Check Passed Parameter *
*****
IF Keybd.to.Apply$ = "" THEN
    Error.Flag = True
    CRT "Keyboard definition to apply not designated"
    CRT "Press ENTER: " :
    INPUT Response$
    GOTO 9999;                  *|End Subroutine|
END
*****
* Read Keyboard *
*****
MATREAD Key.Set$ FROM File.Keys$, Keybd.to.Apply$ ELSE
    Error.Flag = True
    CRT "No keyboard found in Item: " : Keybd.to.Apply$
    CRT "Press ENTER: " :
    INPUT Response$

```

```

        GOTO 9999;                                *|End Subroutine|
END
*****
* Apply Keyboard *
*****
* Each Key.Set$() contains set number, Value.Mark$ expected by User
* Exits "U80BA" and "UA0BA".
* Not including Function Key Lead Character
FOR Set = 1 TO Total.Keybd.Sets UNTIL Error.Flag
    Error.Num = ICONV(Key.Set$(Set), Keys.User.Exit$)
    Error.Num = Error.Num + 0
    Error.Flag = Error.Num > 0
NEXT Set
* For Function Key Lead Character
IF NOT(Error.Flag) THEN
    Set = Total.Key.Sets
    Error.Num = ICONV(Key.Set$(Set), Funcs.User.Exit$)
    Error.Num = Error.Num + 0
    Error.Flag = Error.Num > 0
END
IF Error.Flag THEN
    CRT "Error in keyboard redefinition: " : Keybd.to.Apply$
    CRT " in Set: " : Set
    BEGIN CASE
        CASE Error.Num = 1
            CRT "Error on index for set"
        CASE Error.Num = 2
            CRT "Error on string length for set"
    END CASE
    CRT "Press ENTER: " :
    INPUT Response$
END; * If Error.Flag
=====
*****
9999 * End Subroutine *
*****
RETURN
END

```

Index

" " string indicator, 7, 366

! comment designator, 5

! operator, 258, 259

! single precision indicator, 438

! variable type, 9

double precision indicator, 438

operator, 258, 259

variable type, 9

\$ string indicator, 438

\$ variable type, 9

% integer indicator, 438

& operator, 258, 259

' comment designator, 4

* comment designator, 5

() parameter marks, 22

() operator, 258

• comment designator, 5

• group extract delimiter, 56

• multiplication operator, 5, 16, 258

• Section Name, 8

• wildcard, 5, 100

•••Exit#•••, 9

•A9999, item size using, 89

••••• boundary lines, 8

+ addition operator, 16, 258

+ string concatenator, 368

- subtraction operator, 16, 258

-o- variable type, 9

/ delimiter, 63

/ division operator, 16, 258

: end-of-statement designator, 4, 7, 34

: operator, 258

: separator, 29

: string concatenator, 369

: end-of-statement designator, 5, 7

< > command indicator, 65

< > operator, 258

< operator, 258, 426, 427

<= operator, 258

= operator, 258, 426

= variable assignment statement, 330

=< operator, 259

=> operator, 258, 259

> operator, 258

> redirection symbol, 324

>= operator, 258

@(), 229, 289, 306, 307, 316, 330

[] operator, 258, 378

[] string extract, 370

] truncation, 393

] value mark, 65, 67, 77, 306

^ attribute mark, 65, 67, 77

^ operator, 258, 426

' string indicator, 366

{ } optional parameter, 8

section name, 8

\ integer division, 16

\ operator, 258

\ string indicator, 366

\ subvalue mark, 65, 67, 77

.Flag, 10

.Hex, 10

.Name\$, 10

.variable name, 13

.a. Division, 16

A

A correlatives, 52

ABORT end-of-program designator, 6

ABS, 253

accents, 200

access commands, 140

conversions and correlations and, 134

dependent and controlling relationships, 133

searching with, 354

access control, 163, 391

access display options, 140

access processor, 83, 131-144

ACCOUNT-RESTORE, 102

ACCOUNT-SAVE, 101

accounts, 84, 391, 393

accessing files in different, 161

backing up, 101

cleaning up, 352

deleting, 102

file copying between, 101

port definition items in, 297

port numbers, 394

restoring, 102, 104

saving, 101, 103

system information on, 391

updating, 85

acute accents, 200

Advanced BASIC, 271

ALPHA, 254, 368

alternative computer use, 360

AND, 258, 259

AND bitwise operator, 21

APPEND, 148

APPLY.TERMINAL.AND.

KEY.IDS, 235, 238-241

arithmetic, 16-17

A vs. F correlatives, 52

concatenation precedence and, 259

conversions and correlatives for, 45-52

operators for (see operators)

arrays (see also dynamic arrays, fixed arrays)

creation of, 351

dimensioning, 42

dynamic, locating, 59

fixed, dimension or declare, 42, 60

IBM PC BASIC, 348

longest string in, Boolean flag for, 23-24

PICK/BASIC, 359

ASC, 27

ascending sort, 130

AsciChar.bas program, 26

ASCII

character conversion of, 26

convert characters to, 27-28

keyboard and, 174-179

ASCII 252-255, delimiter use of, 65, 77

ASCII.TO.CHAR program, 27

asm extension, xi

assemblers, 422

assembly language routines, 418

Assign Cursor Off, 305

Assign Screen Constants for Pick Version, 304

Assign Screen Constants Not Supported by Pick, 306

Attrib. variable type, 9

attribute definition item, 83, 139-140

access display options, 140

break on, 141

file searching and, 133

justification of, 140

master dictionary storage of, 83-84

PROC use of, 83

attribute mark, 65, 67, 77

concatenation and, 69

attributes, 9, 352

break on, 141

data, 83

decimal ASCII numbers, 306

defining, 43

definition items (see attribute definition item)

deleting, 105

delimiters, 65

derived, 46, 48-51

dictionary list of, 392

fixed array placement of, 120

insert values, 69

null value setting, 67

null, insertion of, 70

reading, 116

replacing values in, 75

sorting, 74, 78

B

ba extension, xi, 146

BACKUP, 98, 99

bar extension, xi

bas extension, xi, 7, 10, 146

bases, conversions between, 252

BASIC, 1, 4

conditions in, 34

END end-of-program designator, 6

MENU end-of-program designator, 6

ON MDM GOSUB, 34

ON TIMES = Interrupt\$ GOSUB, 34

BASIC command, 274

Basic.bat batch file, 273

BASICA, 272, 418

BASICA /F, 149, 150

BASICA.COM, 271

bat extension, xi, 19

batch files, 18-19, 341, 419

IBM PC BASIC, 272-274

IBM PC DOS, 19

initials.bat, 160

Begin Line comment, 9

bin extension, 10

bitwise operators, 21

blinking, 293, 335

Boolean flags, 20-25, 29, 31, 37, 153, 243, 280, 348

choose string or null, 22

IBM PC BASIC, 20-21

IF...THEN replacement by, 22

longest string in array with, 23-24

null strings and, 22

numeric variables as, 20

odd or even number determination, IBM PC BASIC, 21

swap two numbers if first is larger, 24

testing, 342

booting, 400-404

COLDSTART, 401

LOGTO.HISTORY PROC, 402

Pick Terminal program, 409-412

re-, 421

suppress system messages at, 404-407

terminal log-on, stand alone computers, 408-412

terminal log-on, 407

boundary lines, program parts, 8

bp extension, xi

bpr extension, xi

bps extension, xii

Branch According to Key routine, 242, 243

branching, 35, 39, 242-243, 356

brd extension, xii

break key, 187

disabling, 169-171

user exit to disable, 171

BREAK Test.Flag, 171

BREAK-OFF, 171

BREAK-ON, 141, 171

bsd extension, xii

BUILD.KEYS.FILE.IBM.PC.
XT.R83 program, 202-204
BUILD.KEYS.FILE.WYSE.60.
AT, 214-217

C

CALL, 356, 385, 386
calls, 8
PROC, 416
recursive, 414
capitalization, xi, 8
CAPTURING, 92, 325, 414
carriage return end-of-statement
designator, 7
CASE statement, 7, 32, 38-39, 243
CAT, 96, 258, 369
CATALOG, 276
CHAIN, 6, 61, 149, 382, 386, 387,
414, 415
CHANGE.EDITING.KEYS.
FILE, 224-229
CHANGE.KEYS, 210-212, 235
CHANGE.KEYS.FILE.IBM.
PC.XT.R83 program, 205-210
external subroutine, 212-213
CHANGE.KEYS.FILE.WYSE.
60.AT, 217-220
changing keys, 172-241
APPLY.TERMINAL.AND.
KEY.IDS, 238-241
ASCII value returned by each
key, 174-179
BUILD.KEYS.FILE.IM.PC.
XT.R83 program, 202-204
BUILD.KEYS.FILE.WYSE.
60.AT, 214-217
CHANGE.EDITING.KEYS.
FILE, 224-229
CHANGE.KEYS, 210-212
CHANGE.KEYS.FILE.IBM.
PC.XT.R83, 205-210
CHANGE.KEYS.FILE.
WYSE.60.AT, 217-220
CHANGE.KEYS.IBM.PC.XT.R83,
external subroutine, 212
CHANGE.KEYS.WYSE.60.AT,
external subroutine, 220-222
choosing ASCII value for editing
function, 224-229
COMMON statement, 236
EQUATE statement, 236
foreign language characters,
200-202
IBM PC BASIC, 179-185
Install program\running,
avoidance of, 195
key and ASCII number for
editing function table, 179-185
keys\returning each ASCII
number, 179
Microsoft BASIC, 186-189
patch INPUT command, 192-194
PICK/BASIC, 189-241
reset IBM PC/XT keyboard to
Pick, 197-200, 202-213
SHOW.ASCII, 222-224
STORE.TERMINAL.AND.
KEY.IDS, 230-235
suppressing control character
echo, 191-192
symbols used by tables, 174
TEST.CALL.TERMINAL.
AND.KEY.IDS, 236-238
user exit in place of INPUT,
194-197
user exits for, 199
Wyse 60 terminals, 214-222
CHAR function, 26, 41
CHAR.TO.ASCII program, 28
characters, 26-28
convert ASCII to, 26
convert characters to ASCII,
27-28
CharAscii.bas program, 28
Check\Start\and\End Position
routine, 24
CHOOSE.TERMINAL, 298-300
CHRS, 26
circumflex, 200
CLEAR, 61, 150, 351, 382, 389

CLEAR-FILE, 106
CLEARFILE, 105
CLOSE, 148
CLS, 312
co extension, 146
COL1(), 371, 373, 377
COL2(), 371, 373, 377
COLDSTART PROC, 196, 401
COLOR, 327, 335, 336, 337
color display, 327
columns, 302
printing, 262
com extension, 146
COM OFF, 431
COM ON, 431
COM STOP, 431
COMMA, 426
commands, listing of, 393
comments, 8, 355
designators for, 4
IBM PC BASIC, 4
PICK/BASIC, 5
COMMON, 62, 236, 356, 386
COMPARE, 97
compiler options, 355
concatenation, 259, 345, 368-369
attribute marks and, 69
conditional clauses, 36
conditions, 29-40
BASIC, 34
IBM PC BASIC, 29-34
connecting symbols, list of, 392
constants, 41-42, 62, 65, 345, 357
control blocks, 393
control characters (see also screen
constants), 172, 290
delete, 374
echo suppression, 191-197
editing with, 33, 40, 63, 242-243
editing with, 242-243, 242
control keys, 172, 187
control variables, 33, 40
controlling relationships\access
commands and, 133
conversions and correlatives, 43-59
access commands and, 134
access statement\operation
sequence of, 44
comparison between A and F
correlatives, 52
date and time operator, 46
date comparison example, 47-51
dates, 52-56
derived attributes, 46
group extract, 56-57
ICONV, OCONV, 44
name operator, 45
pattern matching, 57
repetition operator, 46
substring operator, 46
text extract, 57
translate from file, 57-59
user exits, 59
COPY, 98, 98, 99, 159, 415, 433
COPY COM, 432
COPY-LIST, 134-135, 143
COPYDOS, 76, 425, 427
COPYPICK, 430
correlatives (see conversions and
correlatives)
COUNT, 79, 371, 372
CREATE-ACCOUNT, 84
CREATE-FILE, 152, 161
CREATE.TEST.FILE, 117-120
CRT, 194, 286, 287, 359, 360
CRT @, 168, 267, 300
CRT @ (), 287, 318, 360
CRT @ (-), 313, 320, 322, 329,
336, 337, 339
CSRLN, 320
CT, 97
cursor, 165
Assign Cursor Off, 305
escape code placement of, 316
keys for, 173
placement of, 314-320
Prompt Message routine, 315,
317, 319
return position, 320
turn off, 320

D

d0 extension, 146
DATA, 349, 387, 415
data files (see files)
data structures, 60-75
dynamic arrays, 64-75
dynamic arrays, count elements
in, 66
dynamic arrays, extract elements
from, 67-68
dynamic arrays, insert elements,
68-70
dynamic arrays, locating, 70
dynamic arrays, replacing, 74-75
enumerated type, 63
fixed arrays, 60, 63
set type, 63
date and time operator, 46
Date Input routine, 168
DATES, 395
DATE(), 396
dates
American vs. European display,
56, 397
B.C. dates, 56
conversions and correlatives for,
47-51, 52-56
system information, 394-397
TEST.DATE program, 55
DAYS, 395
dBASE, 3
DCOUNT, 66, 95, 96, 371, 372,
414
DEBUG, 420, 421, 422
DECATALOG, 276
decimal numbers (see numbers)
DEFINE-TERMINAL, 289, 298,
316
menu for, 301-303
DEFINT, 150, 438
DEFSTR, 150, 438
DELETE, 66, 105, 106, 143
DELETE-ACCOUNT, 102
DELETE-FILE, 106
DELETE-LIST, 134-135
delimiters, 63
attribute, 65
dynamic arrays, 65
group extract, 56
items, 65
splitting strings with, 344
system, 65, 77
total, 371
variables and, 439
dependent relationships\access
commands and, 133
derived attributes, 46, 48-51
derived values, 43
descending sort, 119
dictionary files (see master
dictionary)
DIM statement, 60, 62, 348
dimensioned array (see fixed
arrays)
DIR, 85, 341
DISABLE.BREAK PROC, 171
do extension, xii
DOS
editor, 76
file listing, 85-86
file names, 146
file structure in, 82
file synonyms, 160
Initials.bat batch file, 160-161
opening files, 149
partition change from Pick to,
423-424
proc for, 424
redirect screen output, 324-327
renaming files, 159
return to, 418
system date, 396
system information access, 388
system time, 399
transfer from Pick to, 430
double precision, 438
downloading, 433
DTR, 252
DUMP, 65, 92, 195
dumping, 92

dynamic arrays, 64-75
count elements in, 66, 372
delimiters, 65
extract elements from, 67-68
insert elements in, 68-70
insert string, 378
insertion, values separated by
semicolons, 69-70
listing selected items in, 73-74
locating, 59, 70
MATREAD and, 115-116
PICK/BASIC, 359
placing items in, 119
reading items in, 117
replace string, 378
replacing, 74-75
sorting attributes in, 74, 78
variables in, 439
writing to items, 156

E

ECHO, 287
ECHO OFF, 166, 173, 191, 249
patching INPUT command,
192-194
ECHO ON, 166
echo suppression, 191-197
ECHO Test.Flag, 166
echoing keyboard, 166-167
EDIT, 141, 147
EDIT-LIST, 134-135
editing, control characters for, 33,
40
EDITING.KEYS, 229
Editor, 65, 77-78
replacing with prestored
commands, 77
searching, TCL and
PICK/BASIC vs., 141
system delimiter handling, 78
EDLIN, 76
ELSE clause, 29, 35, 36, 70
error testing with, 80
ELSE GOTO, 31
ELSE IF clause, 32
ENABLE.BREAK PROC, 171
END end-of-program designator, 5,
6, 35, 149, 385
End Line comment, 9
end-of-program designators, 5-6
end-of-statement designators, 6-7
IF...THEN...ELSE and, 35
end-of-statement semicolon, 356
ENTER, 6, 386
ENTER @, 386
enumerated data structures, 63
EQ, 258
EQUATE, 41, 42, 62, 65, 66, 120,
236, 330, 358, 359, 360
EQV, 258
ERASE, 61, 348
ERL, 79, 80
ERR, 79, 80
ERRMSG, 80, 404, 414
336, 340, 406
ERROR, 80
error recovery, 33, 79-81
keyboard error, 167
message suppression, 404-407
testing files, 153
testing for errors, 80
escape codes (see also screen
constants), 290
cursor placement and, 316
escape key
interrupt with, 243-247
Test Escape program, 244
TEST.ESCAPE, 246-247
Example.bat batch file, 273
EXCHANGE, 159
exe extension, 146
EXECUTE, 6, 88, 92, 119, 142,
387, 413, 414
EXISTS, 427
EXIT, 412
extensions, xi-xii, 146
external subroutine, 345, 385-387
APPLY.TERMINAL.AND.
KEY.IDS, 235, 238-241
CHANGE.KEYS.WYSE.60.
AT, 220-222, 220

external subroutine (continued)

CHANGE.KEYS.IBM.PC.XT-
R83, 212
placement of, 356
EXTRACT, 66, 67
extraction, 369-374
dynamic arrays, 67-68
strings, 69, 96, 259

F

F correlatives, 52
FAILS, 427
fatal errors, 80
FDISK, 419, 420, 421
FIELD, 69, 371, 373, 377, 414
files
defining, 43
searching, 3, 4
variable length, 3, 4
File\$, 10
FILED, 372
files, 64, 65, 82-163
account deletion of, 102
account listing of (see also
accounts), 84, 102, 391
account restoration of, 102
backup, 98-105
closing, 148-153
comparing, 97
conversion from IBM PC BASIC
to Pick, 123-130
copying, 98-105
creating, 107, 152-153
definition item for, 84
deleting, 105-107
dictionary, 392
display information about, 96-97
DOS structure of, 82
downloading, 433
dumping, 104
end of file routine, 108
extensions for, 146
index, 135-138, 353
inputting characters to, 109-113
inputting, 62, 107-123
insert attribute values in items, 69
item attribute reading, 116
item count in, 91
item deletion from, 67
items in, 83
listing, 84-91, 134-135
loading, 104
locking, 74, 157
Microsoft BASIC, 351
modulo, 144-145
naming, 146, 151
open and test, 153
opening, 148-153, 352
order of access, 130
outputting, 69, 156-158
Pick levels in, 82
Pick structure of, 82
reading, 107, 113, 358
redirect output of, 157, 324-327
removing unnecessary, 352
renaming, 158-160
replacing attribute values of items
in, 75
restore from backup, 103
restoring, 98, 104, 145, 352
retrieving list of, 88-91
sampling procedure for, 100
saving, 103, 352
searching, 113, 131, 141
size of, 89-96, 351, 353
size reallocation, 145
sorted list of, 97
sorting, 84, 88
structure of, 341
synonyms, 160-163, 298, 392
system information on, 85-98
temporary synonym creation, 162
testing input, 110, 123-130
time marks for, 91
transferring, 108, 109, 425-436
unneeded, 341
updating account of, 85
video parameters, 303
FILES =, 149
Find Word to Right routine, 376

fixed arrays, 60

attribute placement in, 120
copying, 63
dimension or declare, 42, 60
dynamic arrays vs., 64
full, MATREAD and, 115-116
IBM PC BASIC, 348
PICK/BASIC, 359
placing items in, 118
set values of, 63
variables in, 439
writing items to, 157
FOOTING, 268, 321
footings, 268, 321
FOR...NEXT (see also loops,
repetition), 30, 36, 89, 244,
277-284, 342, 347, 348, 349
breaking, 279
testing index value, 281
FOR...UNTIL...NEXT, 282
FOR...WHILE...NEXT, 282
foreign language characters,
200-202
test program for, 202
format marks, 267-268, 321
fractions (see integer division)
fragmentation, 341
frame faults, 355
FRE(""), 389
FRE(0), 389
free-text programs, 4
function keys, 34, 187, 247-248
resetting, 199
function.number, values for, 390
functions
A vs. F correlatives, 52
repeating, 356

G

GE, 258
GET-LIST, 134-136
global variables, 9, 275, 344, 345,
409, 439
GOSUB, 150, 168, 348, 351, 356,
380, 382, 385
GOSUB...RETURN, 380, 382,
384
GOTO, 168, 348, 350, 356, 382
graphics characters, 292-293
display of, 307-312
grave accent, 200
group extract, conversions and
correlatives in, 56-57
GT, 258

H

HASH-TEST, 351
HEADING, 268, 321
headings, 268, 321
program, 8
sectional, 8
hexadecimal numbers (see /
numbers)

I

I/J, 9
IBM PC BASIC, 1-4
: end-of-statement designator for, 7
Advanced BASIC, 271
arrays in, 348
ASC convert characters to
ASCII, 27-28
Basic.bat batch file, 273
BASICA.COM, 271
batch files, 272-274
blinking, 335-336
Boolean flags in, 20-21
booting, 400
Branch According to Key
routine, 242
break key disabling, 170
breaking loops, 279
call system commands from
program, 412-413
CHR\$ convert ASCII to
characters in, 26
clear screen, 312
closing files, 148
color display, 327
column printing, 262
comment designators for, 4
comments in, 14
compiling programs, 271-274
concatenate strings, 368
concatenation, 345
conditions in, 29-34
constants in, 41, 345
control characters for editing,
242
cursor placement, 314-315
cursor return position, 320
cursor turn-off, 320
DIM, 60
direct command entry, 418
documentation different from
code, 14-15
dynamic arrays (see
PICK/BASIC, dynamic arrays)
END end-of-program designator,
5
end of file routine, 108
ending loops, 281
enumerated and set type data
structures, 63
ERASE, 61
error recovery in, 79-80
escape key to interrupt, 243-245
Example.bat batch file, 273
executing loops, 280
extract strings, 369-370
file backup, 98
file conversion to Pick from,
123-130
file copying, 98
file creation, 107
file deletion, 105
file listing, 86
file reading, 107
file restore, 98
file searching, 131
file size, 89-91
file time marks, 91
file transfer, 108, 109
Find Word to Right routine, 376
fixed arrays, set value of, 63
fixed arrays, dimension or
declare, 60
format of program in, 10
function keys, 247-248
graphics characters, 292-293
IF...THEN...ELSE, 29-31
indentation in, 14
inputting characters to files,
109-113
insert string, 377
integer division in, 16
intensity of video, 336
internal subroutines, 380-385
initializing variables, 437
item size, 91
key values, resetting, 179-185
keyboard input, 164-165
label printing, 262-264
length of string, 375
line input, 107, 164-165
line numbering in, 14, 348
loops, 348
menu choice selection, 250
number conversion, negative to
positive, 253
number conversion, number to
string, 253
number conversion, string to
number, 254
odd or even number
determination, 21
one keystroke data entry,
248-249
open and test files, 153-155
opening files, 148
operator precedence, 257-260
output formatting, 349
outputting files and items, 156
Pick a Number routine, 250
printer commands, 261-265
printer selection, 264
program documentation, 341
program size, 346
program structure, 341
programming tips, 346-350

Prompt Message routine, 315
PrtLabel.bsd program, 263-264
queueing for printer, 264
QuickBASIC, 271
Read Database Fields routine,
108
Read Plain Text routine, 109
remainder [from] division
(modulo), 17
renaming files, 158
repeat loops, 278-281
repeat loops, testing for, 279
repeating character string, 378
repetition, 341
repetition, syntax for, 277
replace string, 377
RESUME, 33
return to DOS, 418
reverse video prompt, 328
reverse video, 327, 337-338
Run.bat batch file, 273
running programs, 271-274
scope of variables, 439
screen characteristic information,
288
screen commands, 285-286
screen constant assignment (see
PICK/BASIC, screen constant)
separating elements, 350
serial port transfer, 431-434
ShowAscii.bas, 179, 185
space and strings, 350
speed test, 361-364
STOP 'end-of-program
designator, 6
string functions, 345
string indicator, 366
string of spaces, 378
string or null, Boolean flag to
determine, 23
string splitting by delimiter
routine, 344
subroutine placement, 348
subroutines, 345
substrating position, 376
SWAP command, 25
swap two numbers if first is
larger, 24
system date, 395
system information access, 388
system time, 397
Test and Open File.Input\$
routine, 154-155
Test Escape program, 244
test file input, 110-115
testing conditions, 342-343
THEN and GOTO, 350
transfer between computers,
431-433
transfer to screen or printer, 435
TSPEED program, 361-364
TstFile.bas program, 158
TSubInt.bas, 381
underline, 337
variable names, 13, 437
variables, 344
video characteristics setting,
327-328
ICONV, 44, 116, 141-144, 168,
356, 390, 391, 418
IF clause, 29
IF...GOTO, 32, 279, 342, 349
IF...THEN, 46, 48, 280, 342, 348
Boolean flags in place of, 22
longest string in array with, 24
swap two numbers if first is
larger, 25
IF...THEN...ELSE, 7, 70
IBM PC BASIC, 29-31
PICK/BASIC, 34-37, 39
IMP, 258
IN, 197
indentation, 8
INDEX, 40, 242, 377
index files, 353
items list, 73-74
index number, LOCATE for, 70
INIT-CURSOR, 403
Initials.bat batch file, 160-161
INKEY\$, 187, 243, 248

InMagic, 4
INPUT, 165, 191, 249
 patching, 192-194
 user exit instead of, 194-197
INPUT @, 167, 168
INPUT\$, 109-113, 154-156, 170, 187, 248
INPUTERR, 167
INPUTNULL, 167
INPUTTRAP, 167, 168
INSERT, 68, 73
insertion, 377
 dynamic arrays, 68
 null attributes, 70
INSTALL, 195
Install program, changing keys and, 195
INSTR, 33, 242, 376
INT and backslash integer division, 17
integer division, 16
intensity, 336
internal subroutine, 345, 380-385
 placement of, 356
 testing, 383
interrupt
 escape key for, 243-247
 Test Escape Program, 244
 TEST.ESCAPE, 246-247
ISTAT, 351
ITEM, 65
Item.\$, 10
Item.ID, 10
items, 9, 64, 65, 82-163
 account definition, 84
 ascending sort, 130
 attribute definition, 83
 attribute reading, 116
 backup, 98-105
 copying, 98-105
 data attribute, 83
 default attribute list, 139-140
 deleting, 67, 105, 107, 136
 delimiters, 65
 descending sort, 119
 display on terminal, 97-98, 131
 dynamic array placement of, 119
 file definition, 84
 file storage of, 83
 finding, 120
 fixed array placement of, 118
 index files, 353
 inputting, 107-123
 insert attribute values, 69
 item ID, 354
 listing, 73, 88, 117-120, 131, 135-136
 loading, 104
 locking, 74, 120-123, 157
 mixing implicit and explicit item ID lists, 136-139
 modulo, 144-145
 names, item ID, 147-148
 order of access, 130
 outputting, 69, 156-158
 printing, 131
 reading, 113, 117, 120-123
 reading into dynamic arrays, 117
 renaming, 159-160, 159
 replacing attribute values in, 75
 restore from backup, 103
 running count of, 141
 searching, 136, 142
 selecting list of, 100
 size of, 89-96, 353
 sorting, 131
 test individual fields in, 133
 writing dynamic arrays to, 156
 writing to fixed arrays, 157

J

justification, 140

K

KEY OFF, 248
KEY ON, 248
KEY STOP, 248
keyboard, 164-251
 access system information, 169
 break key disabling, 169-171

changing|keys|(see|changing
 keys)
control characters for editing, 172, 242-243
control keys, 172
cursor keys, 173
default prompt, 166
echoing, 166-167
escape key to interrupt, 243-247
foreign language characters, 200-202
function keys, 247-248
IBM|PC|XT|to|Pick|use, 197-200, 202-213
identification of, 173
input conversion, 168
inputting errors, 167
inputting list of variables, 165
line input, 164-169
menu choice selection, 249-251
one|keystroke|data|entry, 248-249
Pick standard USA, 198-200
row and column coordinates, 167
toggling, 172
KEYBOARD.CHOICES, 229
KILL, 105, 158

L

labels, 8, 42
 printing, 262-265
 PrtLbl.bsd program, 263-264
languages, 1-4
 comments in, designations for, 4
LCD:, 436
LEFT\$, 370
LEN, 375
lib extension, 146
LINE INPUT #, 107
LINE INPUT command, 164, 165, 170, 187
line input, 164-169
line numbering, 8, 14
line printing, 261
LIST, 136, 141, 157, 275
LIST-LABEL, 265
LISTACC, 391
LISTCONN, 392
LISTDICT, 392
LISTFILES, 88, 325, 341
listing, 355
 files, 84-91
 items, 73, 131, 136
LISTPROCS, 393
LISTU, 393
LISTUSERS, 393
LISTVERBS, 393
literals, 357, 358
LOAD, 61, 272, 382
local variables, 9, 275, 344, 345, 409
LOCATE, 64, 70, 73, 143, 286, 314, 320, 353
LOCK-FRAME, 195
LOCKED, 121
locking, 121, 157
 files, 74
 items, 74, 120-123
 recovery from, 122-123
log off, 401
log-on, 400
 Pick Terminal program, 409-412
 stand alone computers, 408-412
 terminal, 407
LOGON, 400
Logon PROC, 296
LOGTO.HISTORY PROC, 402
loop counters, 347
LOOP...UNTIL...DO...REPEAT, 282
LOOP...UNTIL...REPEAT, 7
LOOP...WHILE...DO...REPEAT, 282
LOOP...WHILE...REPEAT, 7
loops (see also FOR...NEXT, repetition), 37, 341-342
 breaking, 279, 282
 ending, 281, 282
 executing, 280, 282, 283
 IBM PC BASIC, 348

repeating, 29
 testing for, 279, 283
LPRINT, 261, 435
LPT1/3, 435
LPT:, 435
LT, 258

M

master dictionary, 352
 attribute definition item storage in, 83-84
 attributes and synonyms list for, 392
 calling procs from, 19
 connecting symbols listing for, 392
 file access from other account, 161
 listing files in, 88-89
 PROC listing, 393
 verb (command) listing, 393
MAT, 63
MATCH, 258, 367
MATCHES, 258, 367
matching (see pattern matching)
MATREAD, 62, 113, 116, 117, 358
MATREADU, 120-123
MATWRITE, 156
MATWRITEU, 157
MAXFILES, 61, 149, 150, 351, 382
memory, system information on, 394
MENU, 187, 418
MENU end-of-program designator, 6
menus
 choice selection, 249-251
 Pick a Number routine for, 249
MERGE, 61, 149, 382
messages
 suppression of, 404-407
 suppression of, ERRMSGs for, 406
 suppression of, |SYSPROG PROC, 405
Microsoft BASIC
 booting, 400
 break key disabling, 170-171
 clear screen, 312-313
 clear string spaces, 351
 concatenation, 345
 constants, 345
 cursor placement, 315-318
 cursor turn-off, 321
 file listing, 86-87
 file names, 146-147
 files in, 351
 key values, resetting, 186-189
 opening files, 149-150
 program documentation, 341
 program structure, 341
 programming tips for, 350-351
 Prompt Message routine, 317
 repetition, 341
 reverse video, 328, 338-339
 reverse video prompt, 329
 screen commands, 286
 screen constants, 294-295
 scroll lock, 327
 separating elements, 350
 ShwAsc program, 189-190
 Store Message routine, 294
 string functions, 345
 string|splitting|by|delimiter routine, 344
 subroutines, 345
 system date, 395
 system information access, 389
 system time, 398
 testing conditions, 342-343
 transfer to screen or printer, 435
 variable names, 438
 variables in, 344, 351
 video|characteristics settings, 328-329
MID\$, 369, 377
MLOAD, 196

MOD remainder from division, 17, 258
MODE COM, 433
modules, 351
modulo, 17, 144-145, 351-352
MT (Mask Time), 44
multiuser system, Pick as, 3

N

NAME, 149, 158
Name and Open Parameter Files routine, 151
name operator, 45
NAME...AS, 158
NE, 258, 259
nested statements, 8
NEW, 61, 382
NEXT statements, 30, 37, 279
NO ECHO statement, 19
NOT, 258
NOT bitwise operator, 21
notation, special, 7
NULL, 36, 70, 426
null attributes, insertion of, 70
null strings, 22
nulls, 48
 Boolean flags to determine, 22
NUM, 255, 368
numbers, 252-256
 conversion between bases, 252
 conversion of, string to number, 65
 convert negative to positive, 253
 convert|number|to string, 253-254
 convert|string|to|number, 254-256
 searching, 73
 sorting, 73
 test for valid, 255
 TEST.NULL.ZERO|program, 256
numeric variable, 20, 22

O

obj extension, 146
OCONV, 44, 116, 168, 356, 390, 391, 396, 418
odd or even number determination, IBM PC BASIC, 21
ON COM GOSUB, 33, 431
ON ERROR GOTO, 33, 79
ON KEY GOSUB, 33, 247
ON MDM GOSUB, 34
ON TIMES, 398
ON TIMES = Interrupt\$ GOSUB, 34
ON TIMER GOSUB, 34
ON...GOSUB, 32, 33, 39, 40, 168, 242, 243, 398
ON...GOTO, 32, 39, 168
OPEN, 36, 67, 148, 150, 158
Open Data File routine, 152
operators, 257-260, 367
 TEST.OPERATORS, 259-260
OR, 258, 259
OR bitwise operator, 21
OUTPUT, 148
overflow frames, 394

P

P, 287
PAGE, 268, 321
page numbering, 268, 321
paging, 268, 321
parallel ports (see screen, serial ports, terminals)
parameters
 optional, 8
 passing, 415
 parameters files, 303
 partitioning, 419-424
 change from DOS|to Pick, 419-423
 from Pick to DOS, 423-424
 Pick.bat batch file, 419
Pascal, 344
passing parameters, 415
passwords, 163, 391, 401
PASTE, 187

PATH, 160
 pattern matching, 367-368
 conversions and correlatives, 57
 PAUSE, 170
 PC-KEYS, 195, 197
 installation of, 196
 PCKEYS, 191
 PEEK, 388, 389
 Pick a Number routine, 249, 250
 Pick operating system, 3-4
 Pick standard USA keyboard, 198-200
 Pick TCL (see TCL)
 Pick Terminal program, 409-412
 Pick.bat batch file, 419
 PICK/BASIC, 1, 4
 ; end-of-statement designator, 7
 ABORT end-of-program designator, 6
 access order, 130
 accounting history file, 352
 active and inactive information, 353
 alternative computer use, 360
 APPLY.TERMINAL.AND.KEY.IDS, 238-241
 arrays in, 359
 Assign Cursor Off, 305
 Assign Screen Constants for Pick Version, 304
 Assign Screen Constants Not Supported by Pick, 306
 attributes, 352
 blinking, 293, 336
 Boolean flags in, 21-22
 booting, 400
 Branch According to Key routine, 243
 break key disabling, 171
 breaking loops, 282
 BUILD.KEYS.FILE.IBM.PC.XT.R83 program, 202-204
 BUILD.KEYS.FILE.WYSE.60.AT, 214-217
 call system commands from program, 413-418
 CASE statement, 38-39
 change printer settings, 266
 CHANGE.EDITING.KEYS.FILE, 224-229
 CHANGE.KEYS, 210-212
 CHANGE.KEYS.IBM.PC.XT.R83, external subroutine, 212-213
 CHANGE.KEYS.FILE.IBM.PC.XT.R83, 205-210
 CHANGE.KEYS.FILE.WYSE.60.AT, 217-220
 CHANGE.KEYS.WYSE.60.AT, external subroutine, 220-222
 CHAR convert ASCII to characters, 26
 choosing ASCII value for editing functions, 224-229
 cleaning disk, 352
 clear screen, 313-314
 codes for different machines, 291
 comments, 5, 355
 compare strings, match pattern, 367-368
 compiler options, 355
 compiling programs, 276
 concatenate strings, 369
 concatenation, 345
 conditions in, 34-40
 constants in, 41-42, 345, 357
 control characters for editing, 242-243
 CREATE.TEST.FILE, 117-120
 cursor placement, 318-320
 date routine input, 168
 default prompt, 166
 Delete Control Characters routine, 374
 DIM, 62
 dynamic arrays, 64-75
 echoing keyboard, 166-167
 editor for, 76
 END end-of-program designator, 6
 ending loops, 282
 enumerated and set type data structures, 63
 EQUATE statement, 41-42
 error recovery in, 80-81
 escape key to interrupt, 245-247
 executing loops, 282, 283
 external subroutines, 385-387
 extract strings, 370-374
 file deletion, 105
 file listing, 88-89
 file names, 147, 151
 file opening, 352
 file reading, 113, 358
 file searching, 113, 141
 file size and module, 351, 353
 file structure in, 82
 finding items, 120
 fixed arrays, set value, 63
 foreign language characters, 200-202
 format masks, 267-268, 321
 format of programs in, 10
 function key reset, 199
 function number, values for, 390
 functions for versions of Pick Table, 290-291
 graphics characters, 292-293
 headings and footings, 268, 321
 IF...THEN...ELSE, 34-37
 index files, 353
 initializing variables, 437
 input conversion, 168
 input error handling, 167-168
 insert string, 378
 integer division in, 17
 intensity of video, 336
 internal subroutines, 382-385
 item deletion, 105, 136
 item ID, 147-148, 354
 item listing, 117-120
 item locking, 120-123
 item reading, 113, 120-123
 item searching, 142
 item size, 89, 353
 key values, resetting, 189-241
 keyboard input, 165-169
 keyboard reset, 199
 length of string, 375
 literals, 357
 locking, 121
 master dictionary, 352
 menu choice selection, 250-251
 multiple statements on one line, 356-357
 Name and Open Parameter Files routine, 151
 number conversion, negative to positive, 253
 number conversion, number to string, 254
 one keystroke data entry, 249
 open and test files, 155-156
 Open Data File routine, 152
 opening files, 150-152
 operator precedence, 257-260
 outputting files and items, 156
 paging, 268, 321
 partition change from Dos to Pick, 419-423
 partition change from DOS to, 423-424
 patch INPUT command, 192-194
 Pick a Number routine, 250
 program deletion, 106
 program documentation, 341
 program size and frame faults, 355
 program structure, 341
 programming tips for, 351-360
 Prompt Message routine, 319
 protect screen, 322-324
 Recovery From Lock routine, 122-123
 redirect screen output, 325-327
 remainder from division (modulo), 17
 repeating characters string, 379
 repeating functions, avoidance of, 356
 repeating loops, 281
 repeating loops, testing for, 281-282
 repetition, 341
 repetition, syntax for, 277
 replace string, 378
 reset IBM PC/XT keyboard to Pick, 197-200, 202-213
 reverse video, 293, 329, 330, 339-340
 reverse video prompt, 330
 row and column coordinate prompt, 167
 running programs, 276
 scope of variables, 439
 screen characteristic information, 288
 screen commands, 286-287
 screen constant assignment, 289-294
 screen display, 359
 SCREEN.TO.FILE, 326-327
 Search Files, Input Items routine, 142
 searching with access commands, 354
 searching, TCL and editor vs., 141
 select lists, 355
 SEQ convert characters to ASCII, 28
 Set Name without Spaces, 373
 SET.TERMINAL.PARAMETERS, 307
 SHOW.ASCII program, 222-224
 SHOW.DAY.IN.FRENCH program, 63-64
 SHOW.SIZE program, 91-95
 SPACE and STR, 358
 special cursor placement functions, 320
 speed test, 364-365
 STORE.TERMINAL.AND.KEY.IDS, 230-235
 string functions, 345
 string indicator, 366
 string of spaces, 379
 string or null, Boolean flag to determine, 23
 string splitting by delimiter routine, 344
 subroutine placement, 356
 subroutines, 345
 substring position, 377
 suppressing control character echo, 191-192
 system information access, 389-391
 system date, 396
 system time, 399
 terminal parameter assignment, 303-312
 Test and Open File.Input\$ routine, 156
 test file input, 123-130
 test for valid number, 255
 TEST.CALL.TERMINAL.AND.KEY.IDS, 236-238
 TEST.DATA, 415
 TEST.ESCAPE, 246-247
 TEST.EXECUTE.FILE, 89-91
 TEST.FOR.NEXT, 281-282
 TEST.FORMAT.MASK, 268
 TEST.HEADING, 270
 TEST.INPUT program, 123-130
 TEST.INPUT.LONG, 168-169
 TEST.INSERT.NULL, 70-72
 TEST.LOCATE.SORT, 74
 TEST.LOOP, 283-284
 TEST.NULL.ZERO, 256
 TEST.OPERATORS, 259-260
 TEST.PRINT.FOREIGN.CHARS program, 202
 TEST.PROTECT, 322
 TEST.PROC.BUFFER.PROC, 416
 TEST.SELECT.ID, 138-139
 TEST.SELECT.LIST, 143-144
 TEST.SPEED, 362-365
 TEST.SUBROUTINE.INTER.NAL, 383
 TEST.VIDEO, 331-333
 TEST.VIDEO.FUNCTIONS, 333-335
 testing conditions, 342-343
 transfer R:Base to Pick, 426-428
 transfer to and from tape, 434
 TRIM.FILE, 428
 underline, 293, 337
 user exit in place of INPUT, 194-197
 Values Separated By Semicolons routine, 69-70
 variable names, 438
 variables, 344, 357
 video and graphic character description display, 307-312
 video characteristics setting, 329-335
 video functions test, 330, 333-335
 Wyse 60 terminals, changing keys on, 214-222
 POKE, 170, 171, 186
 ports (see also screen, terminals), 391, 393
 account name for, 394
 definition items for, 297
 system information on, 394
 pre precedence, operators, 257
 precision indicators, 438
 prestored commands, replacing with, 76
 prestored sequence, 77
 primary control block, 391
 PRINT, 165, 194, 285, 286, 287
 PRINT #, 156
 PRINT @, 286, 315, 317
 PRINT @ (), 287, 318
 PRINT @ (-), 313, 320, 322, 329, 336, 337, 339
 PRINT CHR\$(27) + , 312, 316, 321, 327, 328, 338
 PRINT-ERR, 81
 PRINT-SCREEN, 325
 printer, 261-270
 access system information and change settings, 266
 format masks, 267-268
 items, 131
 paging, headings, and footings, 268-270
 redirect screen output, 324-327
 status report, 394
 system information on, 394
 TEST.FORMAT.MASK, 268
 TEST.HEADING, 270
 transfers to, 435
 PRINTER CLOSE, 264
 PRINTER OFF, 264
 PRINTER ON, 264
 PROCREAD, 415, 416
 PROCs, 18-19, 341
 attribute definition item use of, 83
 listing of, 393
 master dictionary calling of, 19
 Pick TCL, 19
 program called by, 416
 TEST.PROC.BUFFER.PROC, 416
 PROCWRITE, 415, 416
 program code
 capitalization in, 8
 format of, 10-13
 IBM PC BASIC documentation difference from code, 14-15
 indentation in, 8
 quotes in, 7
 programs
 access command searching, 354
 accounting history file in, 352
 active and inactive information, 353
 alternative computer use, 360
 arrays in, 348
 attributes, 352
 calling system commands from, 412-418
 clear string spaces, 351

comments in, 355
 compiler options, 355
 constants in, 357
 deleting, 106
 design of, 341
 documentation, 341
 file reading, 358
 file size and module, 351
 files and, 351
 fixed and dynamic arrays in, 359
 headings for, 8
 IBM PC BASIC, 346-350
 index files, 353
 item ID, 354
 item size, 353
 line numbering in, 348
 literals in, 357
 loops in, 348
 master dictionary, 352
 multiple statements on one line, 356
 opening files, 352
 output formatting, 349
 parameter passing, 415
 PROC calling, 416
 recursive calls, 414
 repeating functions, avoidance of, 356
 repetition in, 341-342
 running and compiling, 271-276
 screen display, 359
 searching, 352
 select lists, 355
 separating elements, 350
 size and speed of, 341-365
 size of, 346, 355
 SPACE and STR, 358
 spaces and strings in, 350
 structure, 341
 subroutine placement, 348, 356
 TEST.DATA, 415
 TEST.PROC.BUFFER.PROC, 416
 TEST.SPEED, 364-365
 testing speed of, 361-365
 THEN and GOTO in, 350
 time tests of SSELECT, 354
 TSpeed program, 361-364
 user exits and, 418
 variables in, 346-348, 351, 357, 439
 PROMPT, 166, 249
 Prompt Message routine, 315, 317, 319
 Prompt Message routine, 315, 317, 319
 prompts, 401
 default, 166
 program for, 331
 reverse video, 328, 329, 330
 row and column coordinate placement, 167
 protect screen, 322-324
 PntLabel.bsd program, 263-264
 punctuation and special notation, 7

Q

Q-pointer, 101, 298
 creation of temporary, 162
 reflexive, 161
 password setting, 163
 QSELECT, 135-138, 143
 queueing, 264
 QuickBASIC, 271, 380
 quotation marks, 7, 366, 426

R

R:Base
 transfer to Pick, 426
 TRIM.FILE, 428
 radix numbers (see numbers)
 READ, 67, 113, 117, 141, 143, 349, 358
 Read Database Fields routine, 108
 Read Plain Text routine, 109
 READNEXT, 89, 117, 119, 130, 142, 143, 358
 READT, 434
 READU, 120-123
 READV, 107, 113, 116, 117, 358

READVU, 120-123
 reallocating files, 145
 rebooting, 421, 422
 Recover From Lock routine, 122-123
 recursive calls, 414
 redirection, 324-327
 reflexive Q-pointer, 161
 REFORMAT, 157
 REM comment designator, 4, 5
 REM remainder from division, 17
 remainder from division, 17
 remarks (see comments)
 RENAME, 159
 repeating loops, 29
 repetition (see also FOR...NEXT, repetition), 277-284
 programs and, 341-342
 TEST.FOR.NEXT, 281-282
 TEST.LOOP, 283-284
 repetition operator, 46
 REPLACE, 67, 74
 reserved words, 350, 437
 RESTORE, 98, 99
 RESUME, 33, 79
 RETURN, 39, 380, 382, 383, 384, 385
 RETURN TO, 384, 385
 RETURNING, 414
 reverse video, 293, 327, 328, 329, 330, 337-340
 prompt in, 328
 RiBase, 3
 RIGHTS, 370
 RTD, 252
 RUN, 61, 272, 276, 382, 386
 Run.bat batch file, 273

S

SAMPLING, 100
 SAVE, 103, 149, 272
 SAVE-LIST, 134-135
 scope of variables, 439
 screen, 285-340
 access system information with, 288-289
 Assign Screen Constants for Pick Version, 304
 Assign Screen Constants Not Supported by Pick, 306
 assign screen constants, 289-295
 blinking, 293, 335-336
 change screen parameters, 295
 CHOOSE TERMINAL, 298-300
 clear screen, 312-314
 column-only positioning, 302
 cursor placement, 314-320
 cursor return position, 320
 cursor turn-off, 320
 format masks, 321
 intensity of, 336
 line modification, 302
 Logon PROC to set terminal, 296
 modify terminal screen, 301
 paging, headings, footings, 321-322
 PICK/BASIC, 359
 port definition items in ACC file, 297
 protect, 322-324
 redirect output, 324-327
 reverse video, 293, 337-340
 SCREEN TO FILE, 326-327
 scroll lock, 327
 serial port default definition, 296
 SET TERMINAL
 PARAMETERS, 307
 Store Message routine, 294-295
 system cursor definitions utility, 301
 system information on, 394
 terminal assignment, 295-303
 terminal definition, creation of, 300-303
 terminal definition, cursor file access to, 298
 terminal parameter assignment, 303-312
 TEST.PROTECT, 322
 TEST.SCREEN.COL, 303

TEST.VIDEO, 331-333
 TEST.VIDEO.FUNCTIONS, 333-335
 transfers to, 435
 underline, 293, 337
 video and graphic character description display, 307-312
 video characteristics setting, 327-335
 screen constants
 Assign Cursor Off, 305
 Assign Screen Constants for Pick Version, 304
 Assign Screen Constants Not Supported by Pick, 306
 assignment of, 289-294
 codes for different machines, 291
 functions for versions of Pick table, 290-291
 SET TERMINAL
 PARAMETERS, 307
 Store Message routine, 294
 video and graphic character description display, 307-312
 SCREEN TO FILE, 325-327
 scroll locking, 165, 327
 Search Files and Input Item routine, 142
 searching, 352
 access commands for, 354
 conversions and correlatives and, 43
 fields, 3, 4
 files, 113, 141
 items, 136, 142
 matching values (LOCATE), 70
 numbers, justification of, 73
 section names, 8
 segment mark, 65
 SEL-RESTORE, 98, 103
 SELECT, 100, 117, 131-141, 354, 358
 SEQ, 28
 sequential testing, 38
 serial port (see also screen, terminals), 297
 default definition, 296
 Logon PROC to set terminal, 296
 transfers through, 33, 431-434
 SET, 393
 set data structures, 63
 Set Name Without Spaces routine, 373
 SET NULL, 426
 SET-DATE, 396
 SET-DATE-EUR, 397
 SET-DATE-STD, 397
 SET-FILE, 101, 162
 SET-FUNC, 199
 SET-KBRD, 199
 SET-SYM, 403
 SET-TERM, 266, 288, 289, 295
 SET-TIME, 400
 SET TERMINAL.PARAMETERS, 229, 307
 SHELL, 412
 SHOW, 426
 SHOW.ASCII program, 222-224
 SHOW.DAY.IN.FRENCH program, 63-64
 SHOW.SIZE program, 91-95
 ShowAscii.bas, 179, 185
 ShwAsc program, 189-190
 single precision, 438
 SORT, 84, 88, 97, 131-141, 157, 275, 393
 Sort with Use Exit routine, 59
 SORT LABEL, 265
 SORTED BY, 427
 sorting
 ascending, 130
 attributes in dynamic arrays, 74, 78
 descending, 119
 files, 84, 88, 97
 items, 131
 numbers, justification of, 73
 strings, justification of, 73
 SP-ASSIGN, 266
 SPACE, 358, 379

SPACES, 378
 spacing, xi, 8, 350, 358
 special notation, 7
 SREFORMAT, 157, 158
 SSELECT, 89, 100, 119, 131-141
 time test of, 354
 startup (see booting)
 statements, logical nested, 8
 STOP end-of-program designator, 6, 79
 Store Message routine, 294
 STORE TERMINAL.AND.KEY.IDS, 229-235
 stored procedures (see procs)
 STR, 358, 379
 STR\$, 253
 string functions, 345
 strings, 64, 65, 350, 366-279
 Boolean flags to determine number of characters in, 22
 comparing, 38, 367
 comparing, match patterns, 367
 concatenation, 368
 Delete Control Characters routine, 374
 delimiter or substring within, 371
 extract, 48, 69, 96, 259, 369-374
 find length of, 374-375
 Find Word to Right routine, 376
 indicators for, 366-367
 line input, 164-169
 longest, in arrays, Boolean flags for, 23-24
 number conversion of, 253-256
 operators for (see operators)
 replace or insert, 76, 377-378
 searching, 73
 Set |Name|Without|Spaces routine, 373
 sorting, 73
 spaces, repeated, 378-379
 splitting, delimiters for, 344
 substring position, 376-377
 variables in, 345, 439
 STRINGS command, 378
 SUBROUTINE, 385
 subroutines, 8, 62, 345, 380-387, 380
 exit points in, 9
 external, 212, 220-222, 238-241, 385-387
 internal, 380-385
 labeling, 42
 local variables in, 9
 placement of, 348, 356
 TEST.SUBROUTINE.INTERNAL, 383
 TSubInt.bas, 381
 substring operator, 46
 substrings
 position of, 376-377
 total, 371
 SubValue, 10
 search for, 68
 subvalue mark, 65, 67, 77
 subvalues, deleting, 105
 SWAP command, 25
 synonyms, dictionary list of, 392
 SYSFROG PROC, 405
 SYSTEM, 266, 272
 SYSTEM (), 80
 SYSTEM (0), 80
 system commands, 18
 system delimiter, 77
 system information, 388-424
 accessing, 388-395
 account names, 394
 accounts for, 391
 booting and startup, 400-412
 booting and startup, suppress system messages, 404
 call system commands from program, 412-418
 COLDSTART PROC, 401
 connecting symbols listing, 392
 date, 395-397
 direct command entry, 418
 LOGTO.HISTORY PROC, 402
 memory, 394
 partition change from Pick to

system information (continued)

DOS, 423-424
 Pick Terminal program, 409-412
 Pick version, 394
 printer setup, 394
 PROC list, 393
 rebooting, 421
 screen setup, 394
 SYSPROC PROC, 405
 terminal log-on, 407
 terminal log-on, stand-alone computers, 408-412
 terminals in use listing, 393
 TEST.DATA, 415
 TEST.PROC.BUFFER.PROC, 416
 time, 397-400
 user exits to access, 390
 users list, 393
 verb (command) listing, 393
 SYSTEM(), 288, 356, 389, 390
 SYSTEM(0), 434
 SYSTEM(12), 399
 SYSTEM(14), 245
 SYSTEM(17), 414

T

T-ATT, 434
 T-DET, 434
 T-DUMP, 104, 141
 T-LOAD, 104, 141
 T-REW, 434
 TAB(), 262, 286, 314
 TCL
 access control, 85
 account backup, 101
 account definition item creation in, 84
 account deletion, 102
 account restoration, 102
 account saving, 103
 booting, 400
 break key disabling, 171
 change printer settings, 266
 change screen parameters, 295
 CHOOSE.TERMINAL, 298-300
 column-only positioning, 302
 compiling programs, 274-276
 concatenation, 345
 constants, 345
 default attribute list, 139-140
 dumping, 92, 104
 error recovery in, 81
 file copying, 99
 file copying, different accounts, 101
 file creation, 152-153
 file deletion, 106
 file list copying, 134-135
 file list deleting, 134-135
 file list editing, 134-135
 file list retrieval, 134-135
 file listing, 134-135
 file restoring, 98
 file searching, 131-141
 file searching, attribute definition items and, 133
 file searching, controlling and dependent relationships, 133
 file searching, conversion and correlation and, 134
 file sorting, 84
 file synonyms, 161
 index files, 135-138
 item copying, 99, 100
 item deletion, 106
 item display, 131
 item listing, 131, 135
 item printing, 131
 item size, 89
 item sorting, 131
 label printing, 265
 line modification, 302
 loading files and items, 104
 mixing implicit and explicit item ID lists, 136-139
 modify terminal screen, 301
 number/conversion, between bases, 252
 outputting files and items, 157-158

partition change from Pick to DOS, 424
 port definition items in ACC file, 297
 printer commands, 265
 program documentation, 341
 program structure, 341
 programming tips for, 351-360
 renaming files, 159-160
 repetition, 341
 restore item or file from backup, 103
 running programs, 274-276
 sampling procedure, 100
 screen characteristic information, 288-289
 screen commands, 287
 searching, editor: and
 PICK/BASIC vs., 141
 serial port default definition, 296
 string functions, 345
 string splitting by delimiter routine, 344
 subroutines, 345
 system cursor definitions utility, 301
 system date, 396
 system time, 400
 temporary Q-pointer or file synonym creation, 162
 terminal assignment, 295-303
 terminal definition, cursor file access to, 298
 terminal definitions, creation of, 300-303, 300
 test individual fields in items, 133
 TEST.SCREEN.COL, 303
 testing conditions, 342-343
 transfer from DOS to Pick, 425-430
 transfer to and from tape, 434
 variables, 344
 wildcard copying, 100
 Telcom programs, 187, 408, 433, 434
 TERM, 266, 288, 289, 295, 325, 394
 TERM-TYPE, 296, 403
 terminal control language (TCL), 4
 TERMINAL.AND.KEY.IDS, 228, 229, 235
 TERMINAL.PARAMETERS, 229
 terminals
 Assign Cursor Off, 305
 Assign Screen Constants for Pick Versions, 304-305
 Assign Screen Constants Not Supported by Pick, 306
 assignment of, 295-303
 change screen parameters, 295
 CHOOSE.TERMINAL, 298-300
 column-only positioning, 302
 item display, 131
 line modification, 302
 log-on, 407
 log-on, stand alone computers, 408-412
 Logon PROC to set, 296
 modify screen, 301
 parameter assignment, 303-312
 Pick Terminal program, 409-412
 port definition items in ACC file, 297
 serial port default definition, 296
 SET.TERMINAL.PARAMETERS, 307
 system cursor definitions utility, 301
 system information on, 393
 terminal definition, creation of, 300-303
 terminal definition, cursor file access to, 298
 TEST.SCREEN.COL, 303
 video and graphic character description display, 307-312
 Wyse 60 (see also Wyse terminals), 214
 Test and Open File.Input\$ routine, 154-156

Test Escape program, 244
 TEST.CALL.TERMINAL.AND.KEY.IDS, 235-238
 TEST.DATA, 415
 TEST.DATE program, 55
 TEST.EXECUTE.FILE, 89-91
 TEST.FLAG, 9, 20, 30
 TEST.FOR.NEXT, 281-282
 TEST.FORMAT.MASK, 268
 TEST.HEADING, 270
 TEST.IF.THEN ELSE program, 37
 TEST.INPUT program, 123-130
 TEST.INPUT.LONG, 168-169
 TEST.INSERT.NULL, 70-72
 TEST.LOCATE.SORT, 74
 TEST.LOOP, 283-284
 TEST.NULL.ZERO, 256
 TEST.OPERATORS, 259-260
 TEST.PRINT.FOREIGN.CHARS, 202
 TEST.PROC.BUFFER.PROC, 416
 TEST.PROTECT, 322
 TEST.SCREEN.COL, 303
 TEST.SELECT.ID|program, 138-139
 TEST.SELECT.LIST, 143-144
 TEST.SPEED, 364-365
 TEST.SUBROUTINE.INTERNAL, 383
 TEST.VIDEO, 289, 331-333
 TEST.VIDEO.FUNCTIONS, 289, 333-335
 text[extract, conversions|and correlatives, 57
 THEN, 350
 THEN clause, 36
 THEN GOTO, 31
 thrashing, 357
 TIME\$ program, 31
 time, 34
 system, 397-400
 TIME command, 399, 400
 TIMES, 397
 TIMES OFF, 398
 TIMES ON, 398
 TIMES STOP, 398
 TIME(), 399
 TIMER OFF, 398
 TIMER ON, 398
 TIMER STOP, 398
 toggles, 172
 transfer, 425-436
 files, 108, 109
 from DOS to Pick, 425-430
 from Pick to DOS, 430
 Telcom programs, 431-432
 through serial port, 33, 431-434
 to and from tape, 434-435
 to screen or printer, 435-436
 TRIM.FILE, 428
 translations, conversions|and correlatives, 57-59
 TRIM, 69, 372, 373, 379
 TRIM.FILE, 428
 truncation, 393
 TSpeed program, 361-364
 TatFile.bas program, 158
 TSubInt.bas, 381
 TYPE, 160
 type indicators, 438

U

U/, V:, 9
 unlaut, 200
 unary + operator, 258
 unary - operator, 258
 unary = operator, 258
 underline, 293, 337
 UNTIL, 279, 342
 upper and lowercase, xi, 8
 usable space, 394
 user exits
 break key disabling, 171
 calling system commands from programs, 418
 changing keys with, 199
 control[character]echo suppression, 191-192

conversions and correlatives, 59
 replacing INPUT with, 194-197
 users, list of, 393

V

VAL, 254
 value mark, 65, 67, 77, 306
 values, 9, 10
 deleting, 105
 insert, 69
 matching, 70
 replacing, 75
 Values Separated By Semicolons routine, 69-70
 variable length fields, 3, 4
 variables, 344, 357, 437-439
 constants and, 41
 creation of, 351
 definition statements for, 346
 dictionary of, 409
 global, 9, 275, 344, 345, 409, 439
 IBM PC BASIC, 346-348
 index, 347
 initialization of, 347, 349, 437
 listed, inputting, 165
 local, 9, 275, 344, 345, 409
 Microsoft BASIC, 351
 names for, 9, 13, 346, 350, 437
 reserved words and, 437
 scope of, 439
 string, 345
 temporary, 347
 VER, 388
 verbs, listing of, 393
 VERIFY, 98, 99
 VERIFY-SYSTEM, 403
 versions, system information on, 394
 video characteristics
 blinking, 335-336
 color, 327
 intensity, 336
 prompt in reverse video, 328, 329
 reverse video prompt, 330
 reverse video, 327, 337-340
 setting, 327-335
 string length of, 374
 TEST.VIDEO prompt program, 331-333
 testing, 333-335
 underline, 337
 video[characters, display]of, 307-312
 video parameters file, 303

W

WEND, 281
 WHAT, 394
 WHERE, 427
 WHICH, 394
 WHILE, 279, 342
 WHILE..WEND, 281
 WHO, 197, 394
 wildcards, 100
 WRITE, 67, 156
 WRITET, 434
 WRITEU, 157
 WRITEV, 69, 75, 156, 157
 WRITEVU, 157
 Wyse terminal
 BUILD.KEYS.FILE.WYSE.6
 0.AT, 214-217
 CHANGE.KEYS.FILE.WYSE
 60.AT, 217-220
 CHANGE.KEYS.WYSE.60.AT,
 external subroutine, 220-222
 changing key values on, 214-222
 graphics characters, 293

X

X\$, 9
 XOR, 258

Y

Y\$, 9

Z

Z\$, 9

**A quick reference for
programmers integrating
the most widely used
programming language
and the most powerful
database management
system**

This is the only comparative guide available on the integration of IBM® PC BASIC® and the PICK® database system. Organized alphabetically by topic, this book allows programmers who are already familiar with one system to perform similar operations in the other quickly and easily. David Clark thoroughly explains compatible system operations, gives parallel examples in other languages, and provides sample programs and testing routines for duplicating applications between systems.

This book covers all conceivable cross-operations, including previously undocumented functions, with explicit warnings about underlying differences between systems that can put you on the road to error message purgatory. A sampling of topics includes:

- Techniques for making programs run shorter and faster
- Doing arithmetic operations
- Determining characters associated with ASCII numbers
- Performing tests for conditions (IF...THEN)
- Defining and using fixed arrays
- Recovering from error conditions
- Accessing and manipulating data between files
- Standardizing screen display and keyboard operations between systems
- Converting between numbers and strings
- Sending information to a printer
- Running and exiting a program
- Transferring and converting data between systems

Designed specifically to meet the needs of the multilingual, multisystem environment, this book offers you clear, readable instruction for optimizing the most powerful combination of data management features available today.

David L. Clark has been a PICK programming instructor for six years and is the author of the History Database computer program used for historical research, writing, and cataloging.



TAB Professional and Reference Books

Division of TAB BOOKS Inc.
Blue Ridge Summit, PA 17294-0850

\$34.95

ISBN 0-8306-3322-7

PRICES HIGHER IN CANADA

0989