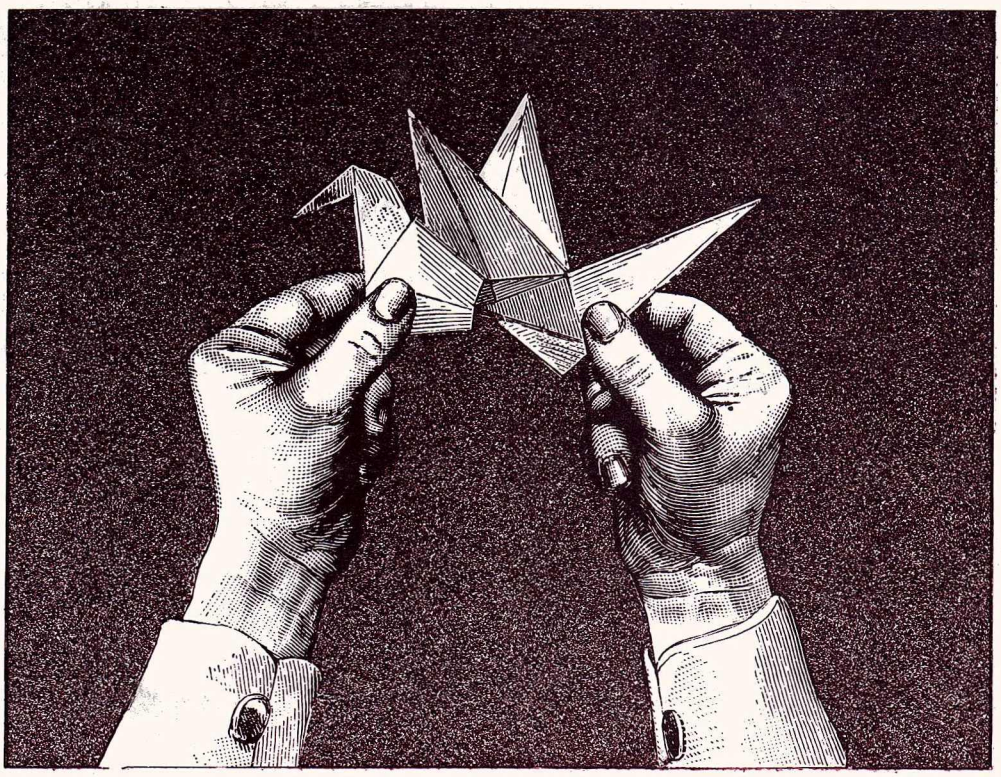
The Pick Series



Pick BASIC

A REFERENCE GUIDE

O'Reilly & Associates, Inc.

The Pick Series

Pick BASIC

A REFERENCE GUIDE

By Linda Mui

O'Reilly & Associates, Inc.

First Edition, Mar. 1990
ISBN 0-937175-42-0

The Pick Series

COMPLETE, ACCESSIBLE GUIDES TO PICK

The Pick Series offers user-oriented documentation—helping new users learn about Pick quickly and helping experienced users find accurate information easily. The Pick Series is a complete documentation set for all users. It tackles the Pick system at a level of depth not found elsewhere. Books in the series are based on a mature implementation of the Pick operating system (R83) with notes on SMA standards and specific differences among major Pick implementations.

TECHNICAL EDITOR

W. Clifton Oliver, CCP

SERIES EDITOR

Dale Dougherty

The Pick Series

The goal of the Pick Series is to provide Pick documentation that is user-oriented: to help new users learn about Pick quickly and to help experienced users find accurate information easily. These books are written in a conversational tone, with lots of examples, as if an experienced user were by the reader's side. The Pick Series offers complete documentation of the Pick operating system (R83) with notes on SMA standards and specific differences among major Pick implementations.

Books in the series include:

- *Pick ACCESS: A Guide to the SMA/RETRIEVAL Language*
  (ISBN 0-937175-41-2, $29.95)
- *A Guide to the Pick System*
  (ISBN 0-937175-43-9, $34.95)
- *Pick BASIC: A Reference Guide*
  (ISBN 0-937175-42-0, $39.95)
- *Master Dictionary Reference Guide: User Account Verbs*
  (Available 5/90, ISBN 0-937175-44-7, $39.95)
- *System Administration: A Guide to Managing the Pick/SMA Operating System*
  (Available 6/90, ISBN 0-937175-45-5, $34.95)
- *SYSPROG Reference Guide: SYSPROG Account Verbs*
  (Available 8/90, ISBN 0-937175-46-3, $29.95)
- *PROC: A Guide to the PROC Processor*
  (Available 9/90, ISBN 0-937175-47-1, $21.95)

O'Reilly & Associates, Inc.

632 Petaluma Avenue • Sebastopol, CA 95472 • 800-338-6887
in CA 800-533-6887 • local/overseas +1-707-829-0515

C O N T E N T S

List of Chapters

CONTENTS

PREFACE

About This Book

Application programming on the Pick system uses the Pick BASIC* programming language. Designed specifically for the Pick system, Pick BASIC opens a world of possibilities for Pick. With Pick BASIC in your grasp, you will come to understand why Pick is designed the way it is and how you can adapt it for your own applications.

When confronted with Pick BASIC, most users' first reaction is dismay at the prospect of programming in the BASIC language. The good news is that although Pick BASIC is based on Dartmouth BASIC, its scope goes far beyond BASIC as most people know it. Like Dartmouth BASIC, Pick BASIC is easy to learn and simple to implement; however, it is far more flexible and powerful than Dartmouth BASIC.

Pick BASIC: A Reference Guide is the third book in O'Reilly & Associates's user-oriented series of complete, accessible guides to the Pick system (the first two books in the series are *Pick ACCESS: A Guide to the SMA/RETRIEVAL Language*, and *A Guide to the Pick System*). The Pick Series offers a complete Pick documentation set for both new and experienced users. It is based on a mature implementation of the Pick R83 operating system, follows the SMA standards, and notes specific differences among major Pick implementations. Forthcoming books in the series will include:

- *System Administration: A Guide to Managing the Pick/SMA Operating System.*
- *Master Dictionary Reference Guide: User Account Verbs.*

---

* Also known as DATA/BASIC or INFO/BASIC.

- *SYSPROG Reference Guide: SYSPROG Account Verbs.*
- *PROC: A Guide to the PROC Processor.*

The Development of Pick BASIC

The BASIC language was developed at Dartmouth College in 1963 as a teaching aid for beginning programmers. The BASIC acronym itself stands for **B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode. It was designed to be easy to learn so that students could quickly master it, even students who had previously been intimidated by computers. In many ways BASIC can be thought of as a primer for programmers.

Consequently, practically every programmer today knows at least a little bit of BASIC. This is one of the beauties of Pick BASIC: because of its similarity to BASIC, it is instantly familiar to almost every programmer. Users will soon discover, however, that where Dartmouth BASIC ends, Pick BASIC is just beginning.

Pick BASIC Enhancements

Some of Pick BASIC's enhancements of Dartmouth BASIC are listed here, to give the reader a taste of what Pick BASIC can accomplish.

Program Format

- *Statement Labels.* In Dartmouth BASIC, numeric statement labels are mandatory for each line of source code. In Pick BASIC, statements are executed in the order in which they appear. Statement labels are not mandatory for each line. Furthermore, on many Pick systems statement labels need not be numeric: alphabetic and alphanumeric labels are also supported, with the provision that the label end with a colon (:). A particularly useful feature is that there is no limit to the length of a statement label, as long as the file item storing the source code does not exceed the maximum item size supported by your implementation.

- *Multiple Statements.* Pick BASIC allows several statements to be written on the same line of source code, as long as they are separated by a semicolon (;).
- *Variable Names.* Variables can have any name of any length, as long as the file item does not exceed the maximum item size supported by your implementation.
- *Fixed-Point Arithmetic.* Computations are done with fixed-point arithmetic, with 19-digit precision and up to at least six decimal digits.

New Features

- *Dynamic Arrays and String Handling.* Since string manipulation is what the Pick system is all about, string functions are key to the structure of Pick BASIC.

  File items are read as dynamic array strings in a Pick BASIC program, with each "line" of text separated by an attribute mark. String functions range from locating a substring or specifying a range of characters, to the powerful dynamic array functions for extracting, replacing, deleting, or inserting a specified field in the array.
- *Dimensioned Arrays.* In Pick BASIC, dynamic arrays can be converted into dimensioned arrays, and vice versa. Programmers therefore have the freedom to choose the data form that is most efficient for their applications.
- *Item and Execution Locking.* Item locks and execution locks in Pick BASIC can prevent multiple users from accessing the same data or executing the same subroutine at the same time.*
- *External Subroutines.* External subroutines can be executed in Pick BASIC with the CALL statement. In addition, any TCL verb can be executed with the EXECUTE statement, and its output and error messages can be captured for use in the program.

---

* Some Pick implementations use the older group locks instead of item locks. Since many Pick systems have already switched to item locks, and since it is likely that more manufacturers will switch in the near future, we have treated item locks in this book as the emerging standard.

- *Screen Manipulation.* The @ function in Pick BASIC provides a wide range of terminal control sequences. Full formatted screen programs can be produced with these sequences.
- *Tape I/O.* Pick BASIC provides statements for directly reading and writing magnetic tapes.

SMA Standards

All books in the Pick Series are fully compatible with the SMA standards. *Pick BASIC: A Reference Guide* includes all statements and functions listed in the *SMA/BASIC Language Specification* published in April, 1986. The SMA specification includes all of the statements and functions commonly available on all Pick systems; it does not, however, include many of the additional statements and functions that are currently supported by many Pick and Pick-related systems.

We have tried to be more inclusive. This book reflects the diversity of what is currently available on different Pick systems; it covers many statements and functions that are not included in the SMA standards but that are supported by a number of major Pick manufacturers. We have not documented every single statement and function available on every system, however. UniVerse BASIC, for example, has many special functions for working with dynamic arrays that are not found on other systems; since these are not generally available, we have not covered them in this book.

All statements and functions not included in the SMA standards are noted, as are many of the differences between the Pick and the Pick-related implementations (such as Prime INFORMATION and uniVerse).

Assumptions

The aim of this book is to teach the Pick BASIC programming language to a user unfamiliar with it. We assume that the user is familiar with some programming concepts and techniques and with the structure of the Pick system; however, an ambitious reader need not be an experienced programmer in order to learn Pick BASIC from this book.

How to Use This Manual

We recommend that you use the following order to read this book:

- Chapter 1, "Creating Pick BASIC Programs," describes how programs are created. It goes into more detail than the average beginner requires, so you may want to skim it for the general concepts at first and return to it for details as necessary.
- Chapter 2, "Format, Data, and Expressions," describes program format, data types, and the syntax for expressions. It should be read carefully.
- Chapter 3, "Introduction to Statements and Functions," provides a tour of the Pick BASIC language. Statements and functions are covered by topic. Read this chapter very carefully: it packs a lot of information into a few pages, so each topic should be fully digested before continuing with the next.

  We recommend that readers stop periodically, perhaps after every subsection of Chapter 3, to try out what has just been covered. Chapter 5 can be used for reference at this stage. By experimenting as you go, you can feel confident that you have fully mastered a topic before continuing with another.

After studying Chapters 1 through 3, you should be ready to start writing programs, using Chapter 5 as a reference.

When you are ready to learn about better ways to debug your programs, refer to Chapter 4, "Using the Pick BASIC Debugger." There are two parts to Chapter 4, a tutorial and a reference. Since there are only a handful of debugger commands, you can study and experiment with each one as you proceed.

Appendix A may be useful if you want some inspiration for working out your own applications. It contains several sample applications which you can study or copy for your own use.

Refer to the other three appendixes as necessary. Appendix B contains error messages which you may encounter while you are creating programs. Appendix C lists the ASCII codes, which are often necessary for using the CHAR or SEQ functions. Appendix D lists all statements, functions, and operators that are available on selected Pick and Pick-related systems.

Summary of Contents

Pick BASIC: A Reference Guide is divided into five chapters and three appendixes.

Chapter 1, "Creating a Pick BASIC Program," describes the syntax and use of the TCL commands necessary for creating Pick BASIC programs.

Chapter 2, "Format, Data, and Expressions," covers the format of Pick BASIC programs, data types, operators, and the syntax for valid expressions.

Chapter 3, "Overview of Statements and Functions," is a brief tour of the Pick BASIC language. The statements and functions, organized by topic, are each briefly described to familiarize the reader with the language and with how the statements and functions are designed to interact.

Chapter 4, "Using the Pick BASIC Debugger," is a tutorial and reference guide to the commands available to the Pick BASIC interactive debugger. The chapter begins with a quick reference to all debugger commands.

Chapter 5, "Statement and Function Reference," is a complete reference guide to every statement and function in Pick BASIC, arranged alphabetically.

Appendix A, "Pick BASIC Program Examples," contains several sample applications written in Pick BASIC.

Appendix B, "Error Messages," is a list of error messages generated by the COMPILE and RUN verbs, and by the Pick BASIC debugger.

Appendix C, "List of ASCII Codes," is a list of ASCII character, decimal, and hexadecimal codes.

Appendix D, "Pick BASIC Statements, Functions, and Operators," is a list of all statements, functions, and operators that are available on selected Pick and Pick-related systems.

Conventions

The following conventions are used for indicating the syntax of statements and functions in *Pick BASIC*.

Convention	Usage
BOLD CAPS	Anything shown in large bold characters must be typed exactly as shown.
italics	Anything shown in italics is variable information for which the user provides a specific value.
[ ]	Anything shown enclosed in square brackets is optional, unless stated otherwise. The square brackets themselves are not typed unless they are printed in bold.
{ }	One or more syntax elements may be enclosed in curly braces. *One* of the elements within the braces *must* be typed. The braces themselves are not part of the syntax and are not typed.
|	A vertical bar separating two or more elements indicates that any one of the elements can be typed.
< >	Bold angle brackets are part of the syntax, and must be typed unless indicated otherwise.
( )	Bold parentheses are part of the syntax. Both parentheses must be typed unless indicated otherwise.

In examples we use the following conventions:

Convention	Usage
BOLD	Anything the user types as input is shown in bold characters
PLAIN	Any output displayed by the system (prompts, responses to user input, etc.) is shown in plain characters
<RETURN>	Indicates that the RETURN key must be pressed
<CTRL-*char*>	Indicates that a control character is to be typed. To enter a control character, simultaneously hold down the CONTROL (CTRL) key and press the specified character.

All punctuation marks included in syntax lines (e.g., commas, parentheses, angle brackets, underscores, hyphens) are required in the syntax unless otherwise indicated.

In the following syntax example:

```
OPEN [ dict, ] filename TO filevar { THEN
   statements
END [ ELSE
   statements
END ] | ELSE
   statements
END }
```

the keywords OPEN and TO must be specified. Either the THEN or ELSE clause *must* be specified, but both are not required. The user must supply appropriate values for *filename*, *filevar*, and *statements*. The *dict* expression is optional, but if it is included, it must be separated from *filename* by a comma.

When variable syntax elements are two or more words long, we use hyphens instead of blank spaces to separate the words in order to show that only one element is required. For example, in the statement:

```
EXECUTE command-expr [CAPTURING cvar]
      [RETURNING rvar]
```

the word *command-expr* indicates a single element.

Please note that although the conventions for Pick BASIC syntax lines are similar to the conventions used in other books in the Pick Series, they are not absolutely identical to them.

☞ This symbol indicates an important note or caution.

Acknowledgements

We would like to thank everyone at Applied Digital Data Systems, Inc., for reviewing this material in its earlier life, and for the use of the ADDS Mentor 6000 system on which all of the examples in this book were generated. Special thanks to Robin White, Dave Yulke, Mike Hannigan, Joe Ferraro, Linda Krencik, Linda Lutz, and many technical support people, all of whom contributed to the book in both large and small ways.

Thanks are also due to the members of the O'Reilly & Associates staff who worked on this book: Walter Gallant who edited it, Michael Sierra who

typeset the book, and Sue Willing who steered it through the printing process. Edie Freedman designed the covers.

We have made every effort to verify all the information in this book. Any errors that remain are our own.

How to Contact Us

To help us provide you with the best possible documentation, please write and tell us about any mistakes you find in this book or about how you think it might be improved.

Our U.S. mailing addresses are:

Ordering	**Editorial (Walter Gallant)**
O'Reilly & Associates, Inc. 632 Petaluma Avenue Sebastopol, CA 95472	O'Reilly & Associates, Inc. 90 Sherman Street Cambridge, MA 02140
(800) 338-6887 (800) 533-6887 (in CA) (707) 829-0515 (local/overseas)	(617) 354-5800
FAX: (707) 829-0104	

CHAPTER 1

Creating Pick BASIC Programs

Writing a Pick BASIC program is relatively simple. You edit the program source code and then compile the program. If the program compiles successfully, you can then execute it. If the program runs correctly, you can then catalog it; otherwise you can debug it (optionally using the interactive debugger) and repeat the sequence.

A Sample Program

As an example, let's create a simple program. Our program is called ADDNUMS; it returns the sum of two numbers.

Before we can write the program, we need a program file. The source code for a Pick BASIC program is entered as an item in the program file. By convention, the program file for an account is often called BP (for **B**asic **P**rograms), but you are free to name the file whatever you like.

The file is first created with the CREATE-FILE command. Then the File Definition item in the Master Dictionary is changed to make the file a program file (we'll explain this in more detail later).

```
>CREATE-FILE BP 1 3

[417] FILE 'BP' CREATED; BASE = 10999, MODULO = 1.

[417] FILE 'BP' CREATED; BASE = 11000, MODULO = 3.
```

```
>ED MD BP
TOP
.<RETURN>
001 D
.R/D/DC
001 DC
.FI
'BP' FILED.

>
```

The CREATE-FILE command in the preceding example creates a file, with a modulo of 1 for the file dictionary and a modulo of 3 for the data file. Next, the D-pointer in the Master Dictionary is edited to change the file from a D-type to a DC-type for storing programs. The new source code can now be placed in item ADDNUMS in the program file BP.

For writing the program, we use the Editor:

```
>ED BP ADDNUMS
NEW ITEM
TOP
.I
001+PRINT "ENTER ONE NUMBER":
002+INPUT NUM1
003+PRINT "ENTER ANOTHER NUMBER":
004+INPUT NUM2
005+SUM = NUM1 + NUM2
006+PRINT "THE SUM OF ":NUM1 : " AND ":NUM2:
                                                    " IS ":SUM
007+STOP
008+END
009+ <RETURN>
TOP
.FI
'ADDNUMS' FILED.

>
```

We then compile the program with the COMPILE command. By compiling the program, we translate the program's source code into object code.

```
>COMPILE BP ADDNUMS
********
 SUCCESSFUL COMPILE!  1 FRAMES USED.

>
```

The program compiles without error. Once the program is compiled, execute it with the RUN command.

```
>RUN BP ADDNUMS
ENTER ONE NUMBER?4
ENTER ANOTHER NUMBER?9
THE SUM OF 4 AND 9 IS 13

>
```

The program seems to run successfully on the first attempt. We can now catalog it so that we can use it as if it were a TCL command:

```
>CATALOG BP ADDNUMS
[244] 'ADDNUMS' CATALOGED!

>ADDNUMS
ENTER ONE NUMBER?5
ENTER ANOTHER NUMBER?3
THE SUM OF 5 AND 3 IS 8

>
```

And that's all there is to creating a Pick BASIC program. More complex programs seldom compile the first time, and once they compile they don't always run without error. The process, however, remains unchanged: edit, compile, and run the program until you get it right.

Creating the Program File

Pick BASIC programs need to be stored in a special file, called a *pointer-file*. Pointer-files that contain programs always have two components, a file dictionary and a data file (like other files on the Pick system). The difference is that, instead of normal dictionary items, the dictionary of a pointer-file contains items that point to compiled object code. In addition, the File Definition item in the Master Dictionary must contain a Definition Code of "DC" in line 1, rather than the usual "D" code.*

On Prime INFORMATION and uniVerse systems, programs are stored in nonhashed files (Type 1 files). INFORMATION stores the object code as an item in the program file along with the source code. Object code items have the same name as the corresponding source code items but with the addition of the suffix .IRUN. UniVerse, on the other hand, stores object code items in a separate file. The file containing object code items has the

---

* This is no longer true for all systems. For example, on Ultimate systems the Master Dictionary pointer to a program file need not be a DC-type.

same name as the source code file but with the addition of the suffix .O . For example, if the program file containing source code is called BP, the file containing object code will be called BP.O.

Like any other file, the program file is created with the CREATE-FILE command. But once you've created the file, you must change the "D" code in line 1 of the data file D-pointer to "DC." You can do this with the Editor.

For example, to create a program file called BP with a modulo of 3 for the file dictionary and a modulo of 5 for the data file, first enter:

```
>CREATE-FILE BP 3 5
```

Then use the Editor to change line 1 of the D-pointer from "D" to "DC":

```
>ED MD BP
TOP
.<RETURN>
001 D
.R/D/DC
001 DC
.FI
'BP' FILED.

>
```

CREATE-FILE creates a file to be used for Pick BASIC programs by creating a File Definition item in the Master Dictionary. The File Definition item is initially created with a definition code of "D" in Attribute 1. Next, the Editor is used to replace the "D" on line 1 with "DC". Without the DC definition code, programs in the BP file will not compile.*

Editing and Listing the Source Code

The *source code* of the program is what the programmer writes and edits. As shown in the example discussed earlier, the source code is stored as an item in the program file.

The only restriction to the name of the program is that it not be the same as the name of the program file itself (e.g., item BP in file BP). Such a

---

* Some Pick systems have a special verb that allows you to create DC-type files automatically. For example, you can create pointer-files on the ADDS Mentor system with the CREATE-BFILE command. On these systems it is not necessary to edit the data file D-pointer.

program will not compile: if it did, the D-pointer in the file dictionary that points to the data file (i.e., the file containing source code) would be overwritten when the program was compiled, and the source code of all programs in the file would be lost.

Source code can be created and edited with the Pick Editor or with any full screen editor supported by your system. See Chapter 7 of *A Guide to the Pick System* for information about the Pick Editor.

Many systems have special verbs that allow you to list program source code in an easy-to-read format. ADDS Mentor's BLIST command and Prime INFORMATION's and uniVerse's FORMAT and FANCY.FORMAT commands are examples. These commands typically print the source code with indentations for THEN and ELSE clauses, for their corresponding END statements, and for the text between program loops. For example, when the following program is listed with BLIST:

```
001 LOOP
002 PRINT "Q FOR QUIT, C FOR CREATE A NEW ENTRY, "
003 PRINT "E FOR EDIT AN ENTRY, D FOR DELETE AN ENTRY"
004 PRINT
005 PRINT "ENTER A CHARACTER (Q,C,E OR D)":
006 INPUT ANSWER,1_
007 ON INDEX("QCED",ANSWER,1) GOSUB 100,200,300,400
008 REPEAT
009 STOP
010 SUBROUTINES
011 100 *** SUBROUTINE FOR CREATING A NEW ENTRY ***
012 PRINT "--EOJ"
013 STOP
014 RETURN
015 200 *** SUBROUTINE FOR CREATING A NEW ENTRY ***
016 PRINT "CREATING A NEW ENTRY ..."
017 RETURN
018 300 *** SUBROUTINE FOR EDITING AN ENTRY ***
019 PRINT "EDITING AN ENTRY ..."
020 RETURN
```

it is displayed like this on the screen:

```
01 MAR 1990        BASIC PROGRAM NAME: MENU              PAGE    1

0001        LOOP
0002           PRINT "Q FOR QUIT, C FOR CREATE A NEW ENTRY, "
0003           PRINT "E FOR EDIT An ENTRY, D FOR DELETE AN ENTRY"
0004           PRINT
0005           PRINT "ENTER A CHARACTER (Q,C,E OR D)":
0006           INPUT ANSWER,1_
0007           ON INDEX("QCED",ANSWER,1) GOSUB 100,200,300,400
0008        REPEAT
0009        STOP
0010        SUBROUTINES:
0011 100    *** SUBROUTINE FOR QUITTING ***
0012        PRINT "--EOJ"
0013        STOP
0014        RETURN
0015 200    *** SUBROUTINE FOR CREATING A NEW ENTRY ***
0016        PRINT "CREATING A NEW ENTRY ..."
0017        RETURN
0018 300    *** SUBROUTINE FOR EDITING AN ENTRY ***
0019        PRINT "EDITING AN ENTRY ..."
0020        RETURN
```

Compiling the Program

Source code must be compiled before the program can be executed. The source code can be written and edited by the programmer, but it cannot be directly interpreted by the Pick system until it is translated into object code.

On most Pick systems, the compiler translates source code into object code and places a pointer to the object code in the file dictionary. The compiler can therefore be thought of as a translator from your language (or more accurately, the language of Pick BASIC) into the machine's language.

Table 1-1 lists the structure and contents of items that point to BASIC object code.

As we mentioned earlier, Prime INFORMATION and uniVerse handle object code differently. Instead of creating object code pointers, the object code is stored as an item in a file. INFORMATION object code is stored in the same file as the source code, and uses two general conventions for naming the item containing the object code. Both conventions use the same

Table 1-1. Format of Object Code Pointers.

Attribute	Example	Description
0	MENU	**Item ID**: Same as the item ID of the source code.
1	CC	**Definition Code**: "CC" indicates an object code pointer.
2	8504	**Base Frame ID**: Indicates the starting location of the object code.
3	1	**Number of Frames**: Indicates the number of frames occupied by the object code.
4		Not used.
5	15:28:10 06 FEB 1990	**Time and Date Stamp**: Indicates when the program was compiled.

name as the source code item, but with the addition of either a suffix (.IRUN) or a prefix ($).

UniVerse object code, on the other hand, is stored in a different file from the source code. The file containing the object code uses the same name as the source code file but with .O added to the filename as a suffix. The object code item ID is identical to the source code item ID.

Two commands can be used to compile a program, BASIC and COMPILE.* These two commands are synonymous.

BASIC *filename program-list* [(*options*)]
COMPILE *filename program-list* [(*options*)]

filename is the name of the Pick BASIC program file, and *program-list* contains the item IDs of the programs to be compiled. An asterisk (*) specifies all programs in the file.

---

* The COMPILE command may not be available on all systems. On Prime INFORMATION systems there is a COMPILE.DICT verb that should not be confused with the COMPILE synonym for BASIC. COMPILE.DICT is used to compile I-descriptor code in Prime INFORMATION dictionary items.

For example, to compile the program ADDNUMS in the file BP, enter:

```
>BASIC BP ADDNUMS
```

or:

```
>COMPILE BP ADDNUMS
```

If the compile is successful, the user sees something similar to:

```
>BASIC BP ADDNUMS
*********
 SUCCESSFUL COMPILE! 1 FRAMES USED.

>
```

The asterisks each represent a source line successfully compiled into object code.* If an error occurs in compilation, the error code is printed with a message. See Appendix B for a list of error messages generated by the COMPILE command.

If the File Definition item in the Master Dictionary does not have DC in line 1, the program will not compile, and the following message is displayed:

```
FORMAT ERROR IN SOURCE FILE DEFINITION
```

Use the Editor to change the Definition Code in line 1 to DC.

Options to the BASIC and COMPILE Commands

There are a number of options to the BASIC and COMPILE commands that enhance their functioning. There are options for program debugging, for compacting object code once the program has been fully debugged, for listing source code as it is compiled, and for sending the compiled output to the printer.

Options for Debugging a Program

The A, M, and X options each supply information that the programmer may find helpful in debugging a program.

---

* On uniVerse and Prime INFORMATION systems, each * equals ten lines of source code successfully compiled.

Listing the Opcodes (the A Option). The A option lists the pseudo-assembler code after the program has compiled. After compilation, press the RETURN key to see the first page of opcodes. As an example, compile the example program ADDNUMS with the A option:

```
>COMPILE BP ADDNUMS (A)
********
```

and press the RETURN key after compilation is complete.

```
ADDNUMS                VERSION  2.5  17:09:22  01 MAR 1990    PAGE  1


001  09          LOADS         "ENTER ONE NUMBER"
001  5A          PRINTCAT
001  01          EOL
002  B9          GET...ELSE
002  03          LOADA         NUM1
002  BA          INPUT
002  01          EOL
003  09          LOADS         "ENTER ANOTHER NUMBER"
003  5A          PRINTCAT
003  01          EOL
004  B9          GET...ELSE
004  03          LOADA         NUM2
004  BA          INPUT
004  01          EOL
005  16          LOAD          NUM1
005  16          LOAD          NUM2
005  2B          ADD
005  18          STOREA        SUM
005  01          EOL
006  16          LOAD          SUM
006  09          LOADS         " IS "
```

Press the RETURN key again to read a second screenful.

The source code line numbers are listed in the leftmost column, followed by hexadecimal numbers representing the Pick BASIC opcodes. The hexadecimal numbers are followed by the associated opcode. Although no statement labels appear in this particular example, they are also displayed as they appear.

You can use the listing generated by the A option to determine what opcodes have been generated by the program. Knowing how to interpret them is another matter and is beyond the scope of this book.

Creating the Cross-Reference (the X Option). The X option to COMPILE creates a cross-reference of all variables and labels used in the program and places it in the BSYM file in the current account. For example:

```
>COMPILE BP ADDNUMS (X)
********
 SUCCESSFUL COMPILE!  1 FRAMES USED.

>
```

The BSYM file now has three items with item IDs SUM, NUM1, and NUM2. Each item has a single attribute, with value marks separating the line numbers at which the variable was accessed. For example, item SUM contains in Attribute 1:

```
005*]006
```

From this we can tell that the SUM variable is accessed in lines 5 and 6 of the program. The asterisk after "005" signifies that the variable is assigned a value on line 5 of the program. Similarly, item NUM1 of the file BSYM contains:

```
002*]005]006
```

The data section of the BSYM file is initialized each time the X option to COMPILE is used, regardless of whether the same program is compiled or what program file the program resides in.

If the BSYM file does not already exist, an error message results and the cross-reference is not created.

Listing the Variable Map (the M Option). The M option to COMPILE lists a map of variables and statement labels after the program has compiled. For example:

```
>COMPILE BP ADDNUMS (M)
********
 050 SUM       030 NUM1     040 NUM2
FRAMES LINES
 01  001-008

 SUCCESSFUL COMPILE!  1 FRAMES USED.

>
```

Variable names are preceded by their location in the user's workspace. In the example above, the variables NUM1, NUM2, and SUM have locations of 30, 40, and 50 respectively. In addition, the range of lines in each frame of source code is printed. In the example, only one frame is used, and lines 1 to line 8 of the source code are contained in frame 1.

Options for Cataloging

When a programmer has determined that no more changes will be made to the program, the C or S options can be used to compact the object code or suppress the symbol table.

Suppressing the EOLs (the C Option). As seen in the example of output from COMPILE with the A option, the last opcode on each line is EOL, for End-Of-Line. When a program is fully debugged, a programmer might choose to compile it with the C option to suppress the EOL opcodes, since they are necessary only for debugging purposes. Thus the programmer can reduce the size of the object code by one byte for each line of source code. In the eight-line program ADDNUMS, only eight bytes would be saved, but the C option could make a significant difference for a larger program.

The EOL opcodes, however, are used to count lines for error messages and for the interactive debugger. By using the C option, further debugging of the program becomes exceedingly difficult. If errors do occur in a program compiled with the C option, all errors will report on line 1, and the interactive debugger will be largely useless.

Suppressing Messages and the Symbol Table (the S Option). The compiler creates a symbol table along with the object code. The symbol table, along with any messages generated by the compiler, can be suppressed with the S option to COMPILE.

The symbol table is necessary for the Pick BASIC interactive debugger. Without the symbol table, the debugger is largely inoperational. A programmer might choose to suppress the symbol table, however, when the program is fully debugged and ready for use.

See Chapter 4 for more information about the Pick BASIC debugger.

Listing the Source (the L Option)

The L option allows the programmer to scan the source code as each line is compiled. This option might be used with the P option to obtain a printed listing of the source code. Error messages are printed with the source line at which they are detected, so the programmer can study the program source code adjacent to the errors that it produces.

As an alternative, the E option prints only the lines with errors.

Printing Compiler Output (the P Option)

The P option sends all compiler output to the printer. This option is particularly useful with the A, M, and L options.

Running the Program

The RUN command executes a compiled Pick BASIC program. Its syntax is:

RUN *filename program* [(*options*)]

where *filename* is the name of the Pick BASIC program file, and *program* is the item ID of the program that you want to run. (Multiple programs cannot be listed on the syntax line for RUN.)

For example, to execute the compiled program MENU in file BP, enter:

```
>RUN BP MENU
```

On uniVerse and Prime INFORMATION systems, the RUN command uses BP as the default filename if no filename is specified on the command line. If MENU were stored in the BP file on these systems, you would be able to execute it by entering the following command:

```
>RUN MENU
```

Debugging Options (D, E, A, and S)

The Pick BASIC interactive debugger can be entered during execution by pressing the BREAK key. Alternatively, the program can be run with the D option. The D option forces the program to enter the debugger before executing line 1.

The debugger is also entered when a fatal run-time error is encountered. If RUN is used with the E option, nonfatal errors will invoke the debugger as well. See Appendix B for a list of error messages generated by the RUN command.

The A option to RUN prevents a fatal error from invoking the debugger by forcing an abort of the program instead. However, the BREAK key can still be used if the A option is specified.

With the S option, error messages generated by the RUN command are suppressed.

See Chapter 4 for a more detailed description of the Pick BASIC debugger.

Printing Output (the P and N Options)

The P option can be used to send output generated by PRINT, HEADING, and FOOTING statements to the printer instead of the screen. The P option is equivalent to placing a PRINTER ON statement at the beginning of the program.

If a HEADING statement has been specified in a program, the program waits for a carriage return by the user after each page of output to the terminal. The RUN command's N option suppresses this feature. This is equivalent to using the N option in the HEADING statement in the source code.

Inhibiting Initialization (the I Option)

The I option inhibits the initialization of the data area when a new program is called with the CHAIN statement. When the EXECUTE statement was incorporated into Pick BASIC, however, the CHAIN statement became largely obsolete. The I option is included for compatibility with older code, but its use is not recommended, since problems will arise if the workspace area is at all corrupted.

Cataloging the Program

The CATALOG command creates a Verb Definition item in the Master Dictionary of the user's account by creating a direct pointer to the object code. After cataloging, a program can be executed directly from TCL, without using the RUN command. Once cataloged, the program can be recompiled without having to be recataloged.

The syntax of CATALOG is:

CATALOG *filename program-list*

where *filename* is the name of the Pick BASIC program file, and *program-list* is a list of the item IDs of the programs to be cataloged. An asterisk can be used to specify all programs in the file.

Programs do not have to be cataloged, but subroutines must be cataloged before they can be accessed by a program.

For example, to catalog the program ADDNUMS in the file BP, enter:

```
>CATALOG BP ADDNUMS
[244] 'ADDNUMS' CATALOGED!

>
```

The program can then be accessed as if it were a command:

```
>ADDNUMS
ENTER ONE NUMBER?3
ENTER ANOTHER NUMBER?4
THE SUM OF 3 AND 4 IS 7

>
```

If the program fails to be cataloged, it is because the object code cannot be found, or because there already exists an item in the Master Dictionary with the same name as the program.

The DECATALOG command deletes the Verb Definition item from the user's account.*

DECATALOG *filename program-list*

filename is the name of the Pick BASIC program file, and *program-list* is a list of the item IDs of the programs to be decataloged. An asterisk can be used to specify all programs in the file.

For example:

```
>DECATALOG BP ADDNUMS
[242] 'ADDNUMS' DECATALOGED.

>
```

The DECATALOG command not only deletes the Verb Definition item, it also deletes the pointer to the object code in the file dictionary. The program must be recompiled after it has been decataloged before it can be executed again with RUN.

---

* Prime INFORMATION uses another verb, DELETE.CATALOG, to remove BASIC programs from the catalog space. UniVerse supports both DECATALOG and DELETE.CATALOG.

CHAPTER 2

Format, Data, and Expressions

This chapter provides an overview of the essential components of the Pick BASIC language. It describes program format, types of data, and how to store and access data within a Pick BASIC program.

Program Format

A Pick BASIC program is a sequence of statements directing the computer to perform a series of tasks in a specified order. A statement consists of keywords, constants, variables, and expressions.

Keywords are special words recognized by the Pick BASIC compiler. *Constants* are values that do not change during the execution of a program, and *variables* are values that can change. *Expressions* can be any constant, variable, or combination of constants and variables with logical, arithmetic, or other operators that produce a resulting value.

A line of Pick BASIC source code corresponds to a single attribute in the source code item. The line can begin with a statement label, but statement labels are not mandatory for all lines in Pick BASIC. More than one Pick BASIC statement can be placed on the same line, as long as each statement is separated from the next by a semicolon (;).

The format for a Pick BASIC source line is therefore:

[*label*] *statement* [; *statement* [; *statement* ...]]

For example:

```
100 PRINT "HELLO, WORLD" ; PRINT "GOODBYE, WORLD"
```

In the example, the statement label is "100". It is followed by two PRINT statements separated by a semicolon. In each statement, the word PRINT is a keyword and is followed by a constant string value ("HELLO WORLD" or "GOODBYE WORLD"), delimited by quotation marks.

On most Pick systems, a statement cannot be broken onto more than one line unless it contains a comma in its syntax (for example, in the EQUATE, COMMON, or DIM statements), in which case it can be broken into multiple lines after each comma. On Prime INFORMATION and uniVerse systems, on the other hand, it is possible to break statements over more than one physical line.

Types of Statement

Pick BASIC statements can be broken into the following general categories:

- **Input/Output control:**

  Input statements control where the computer can expect data, and output statements control where the data is displayed or stored. The input or output device can be a terminal, a printer, a tape, or a file item. Input statements include INPUT, READ, and READT. Output statements include PRINT, WRITE, and WRITET.

- **Program control (sequence of execution):**

  In general, Pick BASIC statements are executed in the order in which they appear, and program control statements are used to alter that sequence. Program control statements include IF, CASE, LOOP, FOR, GOTO, GOSUB, CALL, and EXECUTE.

- **Assignment of variables and constants:**

  Assignment statements assign values to variables and give names to constant values. Values can be directly assigned to variables with the assignment statement (=), and constants can be assigned with the EQUATE statement.

Statement Labels

A statement label is a unique identifier that identifies a particular program line. It consists of a string of numeric or alphanumeric characters at the beginning of a source line.* Source lines do not require statement labels. If the program is directed to a statement label with the GOTO, GOSUB, or INPUTTRAP statements, however, the label must exist somewhere in the program; if it doesn't, the program will not compile.

A numeric statement label can be any constant number (decimals are allowed). On systems that support alphanumeric labels, a numeric statement labels can end with a colon (:), but the colon is not mandatory. An alphanumeric statement label must begin with a letter and be followed by any combination of letters, numbers, periods, or dollar signs. An alphanumeric statement label *must* be followed by a colon, or it is not recognized as a label.

Writing Readable Code

A Pick BASIC program should be relatively easy to read both for the programmer and for those who maintain the program. The readability of a program can be greatly enhanced by:

- Spaces to indent sections of code.
- Blank comment lines to group sections of code together.
- Meaningful variable and subroutine names.
- Comments or remarks to document the program.

When spaces or blank lines appear in a program, they are generally ignored. Therefore, you can use spaces and blank lines freely in order to improve the appearance and readability of a program. You can use spaces to indent sections of code and make the program structure more apparent. Blank comment lines can also be used to set apart a subroutine or any other significant part of the program.

You should make a habit of assigning meaningful names to variables and constants. It is much easier to keep track of what the variable signifies if variable names are intelligible—for example, an array containing customer

---

* A few systems support only numeric statement labels. The SMA standards do not yet include nonnumeric characters in labels.

names and addresses is easier to identify if it is called "CUSTOMERS" rather than "X".

Using Remarks

Program documentation is accomplished by including comments in the Pick BASIC program that explain or identify various parts of the program. Comments are part of the source code only (the original program), and as such they are not executable. They do not substantially affect the size of the object code.

Comments must begin with one of the following symbols:

```
*                    !                    REM
```

To place a comment on the same physical line as another statement, the first statement must first be ended with a semicolon (;), as in the following example:

```
IF SUM < 0 THEN
   LOSS = SUM    ; * CORRECTLY FORMATTED COMMENT
END ELSE PROFIT = SUM
```

Comments cannot be placed between multiple statements on one physical line. In the second line of the following example:

```
IF SUM < 0 THEN
   LOSS = SUM  ; * THE REST OF THIS LINE IS IGNORED ; END ELSE
```

all the text following the * symbol, including "END ELSE", is treated as part of the comment and is not executed.

Comments can, however, be placed in the middle of a statement that occupies more than one physical line, as in the following example:

```
IF SUM < 0 THEN
      LOSS = SUM
      * THIS COMMENT IS ON A LINE OF ITS OWN
END ELSE PROFIT = SUM
```

Remarks in the Object Code

A special form of comment, not available on all systems, can be used to embed a comment directly into the object code. A statement beginning with "$*" places the following text into the object code created when the program is compiled. For example:

```
$* "OPERATING SYSTEM VERSION 2.5"
```

Comments in the object code are particularly useful for including the version number of the program, or for entering copyright information.

Constants, Variables, and Data Types

Assigning and Using Constants

Constants are values that remain unchanged throughout program execution. Constants can be used directly, or they can be assigned to a name with the assignment (=) or EQUATE statement.

For assigning a constant, the EQUATE statement is preferable since there is nothing to stop a constant assigned with "=" from being changed later in the program's execution. A constant assigned with EQUATE, on the other hand, can never be changed: if EQUATE is used, you can be sure that a constant will remain a constant.

The EQUATE statement is also more efficient, since the value of a constant assigned with EQUATE does not need to be fetched from the variable each time it is used at run time. A constant assigned with "=", on the other hand, needs to be reassigned each time the program is executed.

For example, in the following statement:

```
PRINT "HELLO, WORLD"
```

the string "HELLO, WORLD" is a constant string value used directly in the PRINT statement. Alternatively, the program might have read:

```
EQUATE GREETING TO "HELLO, WORLD"
PRINT GREETING
```

By assigning the constant string "HELLO WORLD" to the name GREETING, it can be accessed by that name any time later in the program.

Assigning and Using Variables

Variables are symbolic names that represent stored data values and can change in value during program execution. The value can be explicitly assigned by the programmer, can be read as input, or can be the result of operations performed by the program during execution.

At the start of program execution, all variables are set to an unassigned state. Any attempt to use a variable in the unassigned state produces an error message, and a value of 0 is assumed.

Names for both variables and constants must begin with an initial alphabetic character. They can also include one or more digits, letters, periods, or dollar signs. (Note that hyphens and underscores are *not* valid in a variable name.) Uppercase and lowercase are interpreted differently. A variable name can be any length.

Data Typing in Pick BASIC

In many other programming languages, such as Pascal and PL/I, a distinction is made among types of data. In these languages, all constants, variables, and their data types (integer, real, string, character, etc.) have to be declared at the beginning of the program so that the compiler will know how to store the data. Furthermore, the size of the variable often has to be declared so that the compiler will know how much space to set aside.

In Pick BASIC, on the other hand, no data typing is made by the compiler: all data typing is made at run time, by context. A variable can therefore alternate between numeric and string values within the program. The only thing to be careful of is that when string values are assigned in the program text, they must be delimited by single quotes ('), double quotes ("), or backslashes (\ ). Otherwise, they are assumed to be variable names.

There is, of course, a difference between the way a numeric value and a string value can be treated: it is unreasonable to expect a program to take the square root of the string "CARL". In such a situation, however, a fatal error will not occur—when a string value is applied to a numeric function, a value of 0 is assumed, a warning message is printed, and the program continues from there. String operations, on the other hand, can be executed on numeric values without conflict.

The advantage of not having to declare the type of data is obvious: less work for the programmer and more flexibility for the program. The disadvantage is that errors which one might expect the compiler to detect are not caught. For example, if a variable name is misspelled, the compiler will simply assume that it is a new variable, and the program will successfully compile without an error or warning. Similarly, if a string variable containing "CARL" were accidentally used in the SQRT function, the programmer would not find out until the program was executed.

Building Expressions

Simple Assignment

The assignment statement is used in Pick BASIC to assign values to variables. There are several forms of the assignment statement, but its most commonly used form is:

variable = expr

where *variable* is the name of the variable and *expr* is a valid expression.* An expression is a value that is evaluated at the time of execution and can be anything from a simple constant to a complex sequence of variables, operators, and functions. For a simple example, to assign the constant number 4 to the variable NUMBER, enter:

```
NUMBER = 4
```

Similarly, to assign the constant string "FRED" to the variable NAME, enter:

```
NAME = "FRED"
```

Using Operators and Functions

In the preceding examples, simple constants are assigned. However, any valid expression can be used instead. A simple expression might be a variable name combined with an operator and a constant.

Operators perform mathematical, string, and logical operations on two values. *Operands* are the values on which specified operations are performed. For example, to assign NUMBER plus 1 to the variable NUMBER2 , enter:

```
NUMBER2 = NUMBER + 1
```

In this example, "+" is the operator, and "NUMBER" and "1" are the operands. NUMBER is interpreted as a variable name, and 1 is interpreted as a numeric constant. If NUMBER contains 4, then after the above statement, NUMBER2 will contain 5.

---

* This syntax also permits the use of the optional keyword LET, i.e., LET *variable = expr*.

Another simple expression might involve an intrinsic function. *Functions* perform mathematical, string, and logical operations on a value passed within parentheses. For example, to assign the square root of NUMBER to the variable ROOT, enter:

```
ROOT = SQRT( NUMBER )
```

In this example, "SQRT" is the function and "NUMBER" is the value passed to it. If NUMBER contains 4, then after the above statement, ROOT will contain 2.

Multiple operators and functions can be combined in an expression to evaluate to a single value. For example, to assign NUMBER3 to be 1 plus the square root of NUMBER, enter:

```
NUMBER3 = SQRT( NUMBER ) + 1
```

After the above statement, NUMBER3 will contain 3. Note that this is different from:

```
NUMBER3 = SQRT( NUMBER + 1 )
```

which will return into NUMBER3 the square root of 5, or 2.236.

Valid expressions can therefore be as simple as a constant or a single variable name, or they can consist of multiple operations to be evaluated at run time.

Numeric Expressions

Numeric data is represented as a sequence of digits (0 through 9) with an optional decimal point. A leading plus (+) or minus (–) sign might be used, but commas are not allowed. Any data containing any characters other than numbers and a single decimal point is interpreted as a string. Some examples of numeric values are:

```
–34
42368.99
+3.1416
```

Prime INFORMATION and uniVerse systems also support floating point numbers. The format is similar to the fixed point form shown above, but with the addition of an "E" followed by the power of ten exponent, which can be positive or negative. Some examples of floating point values are:

```
2.3E3
–6.4E38
–1858E–4
```

Numeric data can contain up to 19 digits, including a maximum of at least 6 decimal positions. See the PRECISION statement for information on how to set the maximum number of fractional digits.

Arithmetic Operators

Arithmetic operations range from the simplest calculations (such as COST = COST + 5) to complex expressions combining trigonometric and logarithmic functions. In general, when several arithmetic operations are used in one expression, they follow accepted mathematical guidelines to precedence. The arithmetic operators available to Pick BASIC, in order of precedence, are shown in Table 2-1.

Table 2-1. Arithmetic Operators.

Operator	Operation	Sample Expression	Precedence
^	Exponential	COST^2	1
+	Unary plus	+COST	1
–	Unary minus	–COST	1
*	Multiplication	COST * EXPENSES	2
/	Division	COST / EXPENSES	2
+	Addition	COST + EXPENSES	3
–	Subtraction	COST – EXPENSES	3

In cases where operators are used which are equivalent in precedence (such as * and /), the order of evaluation follows left to right.

Parentheses in Expressions

The order of evaluation can be changed by using parentheses. Operations on expressions enclosed in parentheses are performed before the others. In the following example:

```
(14 * 8) + 12 / 2 + 2
```

the expression is evaluated as 112+6+2 or 120. On the other hand, the following arithmetic expression:

```
14 * (8 + 12) / (2 + 2)
```

is evaluated as 14*20/4 or 70. In arithmetic expressions parentheses must be placed correctly in order to obtain the desired result.

Character Strings in Arithmetic Expressions

If a character string variable that evaluates to a number is used within an arithmetic expression, the character string is treated as a numeric variable. That is, the numeric character string is converted to its equivalent internal number and then evaluated numerically within the arithmetic expression. For example:

```
55 + "22"
```

is evaluated as 77.

If a character string variable that does not evaluate to a number is used within an arithmetic expression, a warning message is displayed and the string is treated as zero. For example, the following expression:

```
55 + "TWENTY TWO"
```

is evaluated as 55, and a message such as the following is displayed, warning that the data is nonnumeric:

```
[B16] LINE 16 NON-NUMERIC DATA WHEN NUMERIC REQUIRED;
    ZERO USED!
```

Intrinsic Mathematical Functions

An intrinsic function is a built-in Pick BASIC function to be used on numeric operands. Table 2-2 lists the mathematical functions available in Pick BASIC.

Table 2-2. Intrinsic Mathematical Functions.

Function	Description
ABS( )	Returns the absolute value of a given arithmetic expression.
COS( )	Returns the cosine value of the angle given in the expression.
EXP( )	Returns an exponential value that raises the base number *e* (2.7183) to the value of the expression.
INT( )	Truncates the decimal portion of a given arithmetic expression and returns the integer value.
LN( )	Generates the natural logarithm (log base *e*) of the given expression.
MOD( )	Divides an expression by another, and returns the remainder value only.
PWR( )	Raises the value of an expression to the power denoted by a second expression.

Function	Description
REM( )	Divides an expression by another, and returns the remainder value only.
RND( )	Generates a random number within the range of 0 and the value of the expression minus 1.
SIN( )	Returns the sine value of the angle given in the expression.
SQRT( )	Computes the square root of any positive numeric expression.
TAN( )	Returns the tangent value of the angle given in the expression.

See Chapter 5 for full information on the syntax and behavior of these functions.

String Expressions

String data consists of a sequence of ASCII characters. They can represent either numeric or nonnumeric information, and are limited in length to the maximum item size supported by your system.

Character string constants consist of a sequence of ASCII characters enclosed in single quotes ('), double quotes ("), or backslashes (\ ). Some examples of character string constants are:

```
"EMILY DANIELS"
'$42,368.99'
'NUMBER OF EMPLOYEES'
"34 CAIRO LANE"
\"FRED'S PLACE" ISN'T OPEN\
```

The beginning and terminating delimiters must match. In other words, if you begin a string with a single quotation mark, you must use a single quotation mark to terminate the string. If one of the delimiters is used within the character string, a different delimiter must be used to begin and terminate the string. For example, using single quotes to enclose the following string is incorrect:

```
'IT'S A LOVELY DAY.'
```

Instead, the string should be delimited with double quotes (or backslashes), as follows:

```
"IT'S A LOVELY DAY."
```

Two adjacent identical delimiters specify a null, or empty, string.

Any ASCII character can be used in character string data except the ASCII character 10 (carriage return), which is used to separate the physical lines of a program.

The CAT String Operator

String expressions can be concatenated, or linked, by using the concatenation operator (: or CAT) as follows:

```
NAME = FIRST : LAST
```

or

```
NAME = LAST CAT ", " CAT FIRST
```

If, for instance, the current value of FIRST is "JANE" and the current value of LAST is "GREY", the preceding string expressions have the values:

```
" JANE GREY "
" GREY, JANE "
```

Multiple concatenation operations are performed from left to right. Expressions in parentheses are evaluated before other operations are performed.

All operands in concatenated expressions are considered to be string values regardless of whether they are string or numeric expressions. However, the priority of arithmetic operators is higher than the concatenation operator. If both types of operator appear in the same expression, arithmetic operations are performed first. For example:

```
"JANE IS" : "2" + "2" : "3" : "YEARS OLD."
```

has the value:

```
"JANE IS 43 YEARS OLD."
```

Logical Data (Booleans)

All data has a logical (or *Boolean*) value, which is to say that it can be evaluated as true or false. If the data contains only numeric values and the numeric value is zero (0), it is false; any other numeric value is true. If the data contains character string values other than the null string (""), it is true; the null string is false.*

---

* This holds true for most Pick systems. On some Pick systems, however, negative numbers also evaluate to false.

Logical values are used for testing conditionals. If a statement reads:

```
IF FOUND THEN ...
```

the variable FOUND is tested to see if it is null or zero. If it is neither, then the condition is determined to be true, and the statements following the THEN clause are executed.

Relational Operators

Relational operators are used to compare both numeric and character string data. The result of the comparison, either true (1) or false (0), can be used for conditional statements. The relational operators are listed in Table 2-3.

Table 2-3. Relational Operators.

Operator	Relation	Example
EQ or =	Equality	X = Y
NE or #	Inequality	X # Y
> < or < >	Inequality*	X < > Y
LT or <	Less than	X < Y
GT or >	Greater than	X > Y
LE or <=	Less than or equal to	X <= Y
GE or >=	Greater than or equal to	X >= Y

When arithmetic and relational operators are both used in a single expression, the arithmetic operation is always performed first.

The same relational operators can be applied to both numeric and string data, but the operations will be calculated differently according to the type of data in the operands. Relational numeric comparisons are calculated as expected, by comparing the literal value of the operands. String comparisons, however, are made by comparing the ASCII values of single characters from each string.

In string comparisons, characters are compared from left to right, and the first string to yield a higher numeric ASCII code equivalent is considered to be greater. If all of the ASCII codes are the same, the strings are considered equal. If the two strings have different lengths but the shorter string is otherwise identical to the beginning of the longer string, the longer string is considered greater than the shorter string. Note that leading and trailing

---

* The operators > < and < > are not included in the SMA standard.

blank spaces are significant, since the space character has an ASCII value of 032.

If both string values can be converted to numeric, then the comparison is always made numerically. If only one operand is numeric, the comparison is made as if both were string values.

The following string comparisons are true and return a value of 1:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"FILENAME" < "NAMEFILE"
"CL  " > "CL"
"kg" > "KG"
B$ < "9/12/78" where B$ = "8/12/78"
```

Logical Operators

Logical operators perform Boolean operation tests on logical expressions. They have the lowest precedence among all operators: they are evaluated after all other operators have been evaluated. Table 2-4 lists the two forms of logical operation in Pick BASIC.

Table 2-4. Logical Operators.

Operator	Syntax	Definition
AND	*x* AND *y*	True (evaluates to 1) if both *x* and *y* are true.
&	*x* & *y*	True (evaluates to 1) if both *x* and *y* are true.
OR	*x* OR *y*	True (evaluates to 1) if either *x* or *y* is true.
!	*x* ! *y*	True (evaluates to 1) if either *x* or *y* is true.

The NOT function can be used to invert a logical value.

For example, in the following statement:

```
IF FOUND AND QUIT = "Y" THEN . . .
```

the variable FOUND is tested to see if it is null or zero, and is evaluated as true if it is neither. Then the relational expression QUIT = "Y" is evaluated. If both FOUND and QUIT = "Y" are evaluated as true, then the condition as a whole is evaluated as true and the statements following the THEN clause are executed.

The MATCH Operator

The pattern matching operator, MATCH or MATCHES, can be used to compare a string expression to a pattern specification and return a value of 1 if they match. The syntax for a MATCH operation is:

expr **MATCH[ES]** *pattern-expr*

The pattern is a general description of the format of the string and can be specified as a constant or as an expression. The pattern specification codes and their definitions are listed in Table 2-5.

Table 2-5. Pattern Matching Codes.

Pattern	Definition
*n*N	*n* numeric characters.
*n*A	*n* alphabetic characters.
*n*X	*n* characters of numeric or alphabetic type.
string	any literal string.

n must be a whole number. If *n* is 0, the relation is true only if all the characters match the specified type. (Note that the null string ("") matches 0N, 0A, and 0X.) For example,

```
ZIP.CODE MATCH "0N"
```

will be true only if all the characters in the string ZIP.CODE are digits.

Patterns can be combined in any sequence. For example,

```
IF LICENSE MATCHES "3N"-"3A" THEN ...
```

confirms that a license number entered consists of 3 digits, a dash, and 3 alphabetic letters.

Logical Functions

Logical functions are functions that return a value of 0 or 1. Table 2-6 lists the logical functions available in Pick BASIC.

Table 2-6. Logical Functions.

Function	Description
ALPHA( )	Tests the given expression for an alphabetical value.
NOT( )	Returns the logical inverse of a given expression.
NUM( )	Tests the given expression for a numeric value.

See Chapter 5 for a full description of the syntax and behavior of these functions.

Advanced Data Types

Thus far we have discussed simple numeric and string data only. There are other types of data in Pick BASIC, however, which are assigned with special syntax.

Array Variables

An array variable is a variable that represents more than one data value. There are two types of array: dynamic and dimensioned.

Dynamic Arrays

A dynamic array is a mapping of the structure of file items to string data. Any string, however, can be considered a dynamic array.

A dynamic array is a string containing substrings that are separated by special delimiter characters. At the highest level, these elements are called *attributes*, and are separated by *attribute marks* (CTRL-^). Each attribute can contain *values* separated by *value marks* (CTRL-]). Each value can contain *subvalues* separated by *subvalue marks* (CTRL-\). Thus, an example of a dynamic array is as follows:

```
PETER THOMPSON]333-8989\232-8665^JOE FRIDAY]872-
    1789\865-0096
```

In this dynamic array string, there are two attributes:

```
PETER THOMPSON]333-8989\232-8665
JOE FRIDAY]872-1789\865-0096
```

there are four values:

```
PETER THOMPSON
333-8989\232-8665
JOE FRIDAY
872-1789\865-0096
```

and there are four subvalues:

```
333-8989
232-8665
872-1789
865-0096
```

The primary use of dynamic arrays is to store data that is either read from or written to a file item. Each line in a file item corresponds to a separate attribute. However, Pick BASIC includes facilities for manipulating dynamic array elements that make dynamic arrays a powerful data type for processing information independently of file items.

Dynamic arrays are called "arrays" because they can be referenced by array functions using three subscripts, and they are called "dynamic" because elements can be added or deleted without having to recompile the program. The maximum size of a dynamic array is as large as the maximum item size permitted on your system. Null attributes, values, and subvalues are represented by two consecutive attribute marks, value marks, or subvalue marks, respectively.

See Chapter 3 for more information on processing dynamic arrays.

Dimensioned Arrays

Dimensioned arrays (also called *standard arrays*) are one- or two- dimensioned structures. Each value in a standard array is called an element of the array.

A one-dimensioned array (also called a *vector*) has its elements arranged in sequence. An element of a vector is specified by the variable name, followed by the *index* of the element enclosed in parentheses. The index of the first element is (1).

A two-dimensioned array (also called a *matrix*) has the elements of the first row arranged sequentially in memory, followed by the elements of the second row, and so on. An element of a matrix is specified by the variable name, followed by two indexes enclosed in parentheses, representing the row and column position of the element. The indexes of the first element are (1,1).

The indexes used to specify the elements of a matrix that has four columns and three rows are illustrated by the following:

COST:

	Column 1	Column 2	Column 3	Column 4
Row 1	COST (1,1)	COST (1,2)	COST (1,3)	COST (1,4)
Row 2	COST (2,1)	COST (2,2)	COST (2,3)	COST (2,4)
Row 3	COST (3,1)	COST (3,2)	COST (3,3)	COST (3,4)

Note that vectors, or one-dimensioned arrays, are treated as matrixes with a second dimension of 1. COST(3) and COST(3,1) are equivalent specifications and can be used interchangeably.

Indexes can be written as constants or as expressions.

Before a dimensioned array can be used in a Pick BASIC program, a DIM or COMMON statement must be used to declare the maximum number of elements it will store throughout the program. See the DIM and COMMON reference pages in Chapter 5 for more information.

File Variables

A file variable is created by a form of the OPEN statement. Once opened, a file variable is used in I/O statements to access the file. See Chapter 3 for more information on assigning and using file variables.

Select-List Variables

A select-list is a set of item IDs or attributes created by the SELECT statement or by TCL select-list generators, to be used in a READNEXT statement. There are three ways to generate a select-list:

- TCL list generators such as SELECT, SSELECT, and FORM-LIST,* can be used outside the Pick BASIC program or called with the EXECUTE statement.
- The Pick BASIC SELECT statement can be used on a file variable.

---

* Or, on some systems, QSELECT.

- The Pick BASIC SELECT statement can be used on a string variable which is stored as a dynamic array.

See Chapter 3 for more information on select-list variables.

CHAPTER 3

Overview of Statements and Functions

This chapter is designed to give a brief topical overview of the statements and functions in Pick BASIC. For full descriptions of the statements and functions covered in this chapter, see Chapter 5, "Statement and Function Reference." The reader should refer to the reference chapter whenever further information or elaboration is needed on a topic. The topics of this chapter are covered in the following order:

- Assignment statements.
- Intrinsic functions.
- Internal and external program control.
- Sending output to the screen and printer.
- Terminal input.
- Dynamic array processing.
- Generalized string processing.
- Dimensioned arrays.
- Reading and updating file items.
- Reading and writing tapes.
- Execution locks.
- Compiler directives.
- Miscellaneous statements and functions.

Assignment Statements

The simplest assignment statement in Pick BASIC is of the form *var = expr* (or LET *var = expr*). A full list of operators is given in Chapter 2. For example, the variable NUMBER can be assigned the value 7 with:

```
NUMBER = 7
```

Any valid expression can be used in an assignment statement. For example, NUMBER2 can be assigned the value of NUMBER plus 2 with:

```
NUMBER2 = NUMBER + 2
```

Some versions of Pick BASIC also support a special form of assignment called *operator assignment.* For example, 2 can be added to the value of NUMBER with:

```
NUMBER += 2
```

as a shorthand for:

```
NUMBER = NUMBER + 2
```

Accepted operators* for operator assignment are:

+	addition
–	subtraction
:	concatenation

Initializing Variables (CLEAR)

The CLEAR statement acts to initialize all variables to the value 0. It cannot be used to initialize a single variable, however, and initializing all variables to 0 may result in errors due to unassigned variables remaining undiscovered.

Assigning Constants (EQUATE)

The EQUATE statement is used to make a variable functionally equivalent to another or to assign a constant. It cannot be used to assign a variable, since values assigned with an EQUATE statement cannot be reassigned during the program.

---

* ADDS Mentor additionally supports * (multiplication) and / (division).

The EQUATE statement assigns values at compile time. No operators or functions can be incorporated into an EQUATE statement, with the exception of the CHAR function. Thus, the EQUATE statement can be used to supply a meaningful name for a special character in a program or for an element of a dimensioned array: for example, to equate AM to an attribute mark:

```
EQUATE AM TO CHAR(254)
```

or to equate QTY to element 4 of the dimensioned array INVENTORY:

```
EQUATE QTY TO INVENTORY(4)
```

Intrinsic Functions

Numeric Functions

In addition to the standard arithmetic operators (+, –, *, /), Pick BASIC provides several functions for evaluating numeric calculations.

ABS(*expr*)	returns the absolute value of a given expression. The absolute value of a number is its positive value, or the difference between itself and zero.
INT(*expr*)	gives the integer value of an expression. It truncates the decimal portion of a number and returns the result.
MOD(*expr1*,*expr2*)	returns the remainder value when the first expression is divided by the second.
REM(*expr1*,*expr2*)	returns the remainder value when the first expression is divided by the second.
SQRT(*expr*)	returns the square root of a positive expression.
RND(*expr*)	returns a random number between 0 and the given expression minus 1.
PWR(*expr1*,*expr2*)	returns the first value to the power of the second.

In addition, the following trigonometric functions are available in Pick BASIC:

SIN(*expr*)	returns the sine of the angle.
COS(*expr*)	returns the cosine of the angle.
TAN(*expr*)	returns the tangent of the angle (SIN/COS).
LN(*expr*)	returns the natural logarithm (log base *e*) of the expression.
EXP(*expr*)	returns *e* to the power of the expression (the inverse of LN).

The accuracy of each numeric function is dependent on the decimal precision used by the program, i.e., the number of decimal places to which numeric values are calculated. On most Pick systems, all numeric values are calculated to four decimal places by default.* To reassign this value, use the PRECISION statement.

Logical Functions (NOT, NUM, ALPHA)

A logical function, or *Boolean* function, is one which returns either 0 or 1. A return value of 0 is taken to mean "false," and a return value of 1 is taken to mean "true." On most Pick systems, all positive and negative integers evaluate to "true," and zero or null evaluate to "false."† Logical functions are most useful in conditional statements (IF, CASE), or in the exit for loops.

In addition to the relational operators (=, #, >, >=, <, <=, MATCH), the following intrinsic logical functions are supported in Pick BASIC:

- The NOT function returns the logical inverse of a given expression. That is, if the expression evaluates to 0 or the null string (""), the NOT function returns 1; if the expression evaluates to anything other than 0 or the null string, the NOT function returns 0.
- The NUM function returns 1 if the given expression is numeric, or 0 if it is nonnumeric. (Note that the NUM function might return 0 for

---

* The SMA standard supports a maximum precision of 6; Ultimate and ADDS Mentor support 9.

† On some systems, negative integers evaluate to "false."

a clearly numeric value if it contains more decimal places than the current precision.)

- The ALPHA function* returns 1 if the given expression is alphabetic, or 0 if it is nonalphabetic.

For example, suppose a program expects a positive number in Attribute 3 of a file item. The source code might read:

```
PRICE = RECORD<3>
IF NOT(NUM(PRICE)) THEN
   PRINT "ERROR — NON-NUMERIC DATA IN ATTRIBUTE 3."
   STOP
END ELSE
   IF PRICE < 0 THEN
      PRINT "ERROR — NEGATIVE DATA IN ATTRIBUTE 3."
      STOP
   END
END
```

Using the Boolean operators (AND, OR), the same code might read:

```
PRICE = RECORD<3>
IF NOT(NUM(PRICE)) OR PRICE < 0 THEN
   PRINT "ERROR IN ATTRIBUTE 3 — ":
   PRINT "POSITIVE NUMBER EXPECTED."
   STOP
END
```

Internal Program Control

Statements in a Pick BASIC program are executed in the order in which they appear in the source code. Program control statements are used to alter that sequence. This section discusses the internal program control constructs that do not involve other programs, external subroutines, or TCL verbs. More advanced program control statements are discussed later in this chapter.

The IF Conditional

The IF construct is used to execute a statement (or series of statements) if a condition has a logical value of true, and (optionally) a different set of statements if the condition has a logical value of false.

---

* Not included in the SMA standards.

The THEN and ELSE Clauses

Either a THEN or ELSE clause, or both, must be supplied with IF. The syntax of THEN and ELSE clauses is important to understand, because they are used not only in IF statements but also in numerous file I/O and tape I/O statements.

THEN and ELSE clauses can be written on the same statement line like this:

```
IF NET >= 0 THEN PRINT "PROFIT IS ": ELSE PRINT "LOSS IS ":
                                                PRINT ABS(NET)
```

(In the preceding and following examples we have not been able to fit the statement lines on one printed line due to their length.) If there are multiple result statements for either the THEN or the ELSE clause, they can be separated by semicolons (;). For example:

```
IF NET >= 0 THEN FLAG = 1; PRINT "PROFIT IS ": ELSE FLAG = 0;
                                    PRINT "LOSS  IS ": PRINT ABS(NET)
```

The IF statement becomes difficult to read, however, if the THEN and ELSE clauses are written on a single line. It is preferable to write it on several lines, even if the conditional statement is very short.

When splitting an IF statement onto several lines, the THEN or ELSE keyword must end a program line, with the result statements beginning on the next. At the end of the result statements, an END statement must be used to group them together. Thus, if the condition tested by IF is true, all statements between the THEN clause and the corresponding END statement are executed; otherwise, all statements between the ELSE clause and the corresponding END statement are executed.

So the lines of the program shown earlier that prints out profit or loss on a transaction might read:

```
IF NET >= 0 THEN
   FLAG = 1
   PRINT "PROFIT IS " :
END ELSE
   FLAG = 0
   PRINT "LOSS IS " :
END
PRINT ABS(NET)
```

NULL Statements

NULL statements are often included in THEN or ELSE clauses as placeholders. A NULL statement performs no action; however, you can

use NULL statements to make the logic of a conditional somewhat clearer: for example, to test if a value is numeric, you might write:

```
IF NUM(PRICE) THEN
   NULL
END ELSE
   PRINT "ERROR: NON-NUMERIC PRICE.  STOP"
   STOP
END
```

There are certainly ways of doing this without using a NULL statement (by using the NOT function in the condition, or by omitting the THEN clause entirely). However, you may prefer to use NULL statements to make conditionals easier to read.

CASE Constructs

The CASE construct acts to perform multiple IF conditionals. It tests several conditions until one returns a value of true. It then executes the associated set of statements.

A CASE construct must begin with a BEGIN CASE statement and end with an END CASE statement. In between, each CASE statement tests a single condition, and, if true, the statements between the current CASE and the next CASE are executed. The program then jumps to the statement after the END CASE statement, ignoring all remaining CASE statements in that group. For example, the above profit-or-loss example might read:

```
BEGIN CASE
CASE NET > 0
   PRINT "PROFIT IS " : NET
CASE NET < 0
   PRINT "LOSS IS " : ABS(NET)
CASE NET = 0
   PRINT "NO PROFIT OR LOSS ON THIS TRANSACTION."
END CASE
```

Loops (LOOP, FOR)

Program loops are constructs that repeat the same sequence of statements while a condition holds true or until a condition is met.

The LOOP Construct

The LOOP statement is the general-purpose looping construct in Pick BASIC. It has (optionally) two sets of statements, the first of which is executed before testing the condition clause, and the second of which is executed only if the condition is verified. The condition is written as either a WHILE or an UNTIL clause. If the WHILE clause is used, the loop will continue as long the condition remains true; if the UNTIL clause is used, the loop will continue as long as the condition is *not* true.

The following example will continue to prompt for a number until a numeric value is entered:

```
LOOP
   PRINT "ENTER A NUMBER":
   INPUT NUMBER
UNTIL NUM(NUMBER) DO
   PRINT "NUMERIC INPUT EXPECTED!"
REPEAT
```

The UNTIL clause can be replaced with a WHILE by negating the condition:

```
WHILE NOT( NUM( NUMBER ) ) DO
```

FOR Loops

In its simplest form a FOR loop performs a set of statements while incrementing a number by 1. When the number reaches or surpasses a specified maximum, the FOR loop exits. For example, a program printing out the first ten perfect squares might read:

```
FOR I = 1 TO 10
   PRINT I * I
NEXT I
```

In Pick BASIC, the FOR loop has been enhanced in two ways: an increment other than 1 can be specified with the STEP clause, and the WHILE and UNTIL clauses of the LOOP statement have been incorporated into it.

Stopping a Program (STOP, ABORT, END)

Two statements cause a program to stop executing immediately: the STOP statement and the ABORT statement. The difference between them is that if the current program is called by a proc or another program, the STOP statement returns to the calling proc or program, but an ABORT statement

returns directly to TCL. In general, STOP statements are used for a normal or nonfatal termination of a program, and ABORT statements are used for abnormal termination.

STOP and ABORT are often used in ELSE clauses to file I/O and tape I/O statements when the program becomes pointless if the statement fails. STOP statements are also used between the main part of a program and its internal subroutines. When a program is written with internal subroutines at the end, a STOP statement is necessary to ensure that the subroutines are not directly executed at the end of the program.

The END Statement

Beyond its function for delimiting THEN or ELSE statements, the END statement can also be used (optionally) to designate the end of compilation. When the compiler reaches an END statement that does not correspond to a THEN, ELSE, or LOCKED clause, all compilation stops: any statements or subroutines that come after the END statement in the source code are ignored.

Internal Subroutines (GOSUB, RETURN)

An internal subroutine is a discrete sequence of statements starting with a statement label and ending with a RETURN statement. In the source code, subroutines are placed after the main part of the program, and precautions are generally taken to ensure that they are never executed directly. Internal subroutines are executed by GOSUB statements, which point to the statement label.

For example, a subroutine labelled "8000" prints a report of the session's transactions. The subroutine might be called by the following code:

```
PRINT "PRINTING A REPORT ..."
GOSUB 8000
DISPLAY "REPORT PRINTED."
```

The subroutine itself might read:

```
8000 *PRINT A REPORT
   PRINTER ON
   PRINT "NUMBER OF TRANSACTIONS" , NO.OF.TRANS
      .
      .
      .
   PRINTER CLOSE
   PRINTER OFF
RETURN
```

When GOSUB is executed, program control is transferred to the statement label "8000" and continues until the RETURN statement is encountered. Program control then continues with the statement following the GOSUB statement, and the message "REPORT PRINTED" is displayed on the screen.

The GOTO Statement

The GOTO statement is often grouped together with GOSUB, because they share the same syntax and perform similar functions. However, the GOTO statement serves only to transfer program execution to the statement label and never returns to the statement following the GOTO statement unless another GOTO is used.

External Program Control

This section discusses the program control constructs that call external subroutines and that execute TCL commands.

External Subroutines (CALL)

The CALL statement transfers execution to an external subroutine. An external subroutine is a sequence of statements that performs a discrete function, compiled separately from the calling program. The subroutine must be cataloged in the account before it can be called.

The first statement of the subroutine must be the SUBROUTINE statement, and the last statement executed must be the RETURN statement. The SUBROUTINE statement can take several parameters that correspond to the parameters on the CALL statement that calls the subroutine. The *n*th

parameter on the SUBROUTINE statement and the *n*th parameter on the CALL statement become equivalent.

For example, suppose a simple subroutine named ADDEMUP is called with the following source lines:

```
PRINT "ENTER A NUMBER":
INPUT NUMBER1
PRINT "ENTER ANOTHER NUMBER":
INPUT NUMBER2
CALL ADDEMUP(NUMBER1, NUMBER2, NUMBER3)
PRINT NUMBER1 : " PLUS " : NUMBER2 : " IS " : NUMBER3
```

and the subroutine ADDEMUP reads:

```
SUBROUTINE ADDEMUP(A,B,C)
   C = A + B
RETURN
```

The value of NUMBER1 is passed to the variable A in the subroutine, the value of NUMBER2 is passed to B, and the value of NUMBER3 is passed to C. At the conclusion of the subroutine, the parameters are returned with their new values (if any). Thus, ADDEMUP serves to place the sum of the first two numbers in the variable NUMBER3.

Passing Parameters (COMMON)

The alternative to passing parameters with the CALL and SUBROUTINE lines is the COMMON area, by which several programs can share the same variables.

The COMMON statement permits multiple programs and subroutines to use the same variables by accessing them according to the sequence in which they are stored. Each program using the COMMON area must include a COMMON statement, and the variables will be considered equivalent according to their positions. Thus, in the simple example of a subroutine shown earlier in this section, the main program might have read:

```
COMMON NUMBER1, NUMBER2, NUMBER3
PRINT "ENTER A NUMBER":
INPUT NUMBER1
PRINT "ENTER ANOTHER NUMBER":
INPUT NUMBER2
CALL ADDEMUP
PRINT NUMBER1 : " PLUS " : NUMBER2 : " IS " : NUMBER3
```

and the subroutine ADDEMUP:

```
SUBROUTINE ADDEMUP(A,B,C)
   COMMON A, B, C
   C = A + B
RETURN
```

The variable NUMBER1 in the main program and the variable A in the subroutine are considered equivalent because of their positions in the COMMON statement. The same is true of NUMBER2 and B, and of NUMBER3 and C.

Executing a TCL Command (EXECUTE, DATA)

Both the EXECUTE statement and the CHAIN statement can be used for executing a TCL command. EXECUTE is by far the more powerful of the two statements, and is to be preferred to CHAIN.

The EXECUTE statement executes any TCL command and returns to the current program. In addition, the RETURNING clause can be used to determine error messages which may have resulted, and the CAPTURING clause can be used to capture the terminal output generated by the command.

The CHAIN statement will execute the command but will not return to the calling program.

The DATA Statement

The DATA statement places data in the secondary output buffer, or *data stack*. If the data stack is not empty, any subsequent requests for input will accept the response directly from the data stack, and the user will not be given the opportunity to respond.

The data stack is helpful for executing TCL commands that request information that the program can supply. For example, if a programmer wishes to copy a file item before altering it in a program, the COPY verb might be used (rather than writing a new item with the WRITE statement). The COPY verb requests the operator to supply the new item ID, so the DATA statement could be used to store the new item ID on the data stack before using EXECUTE to run the COPY verb.

Before EXECUTE returns to the calling program, it checks the data stack for input. If the data stack is not empty, its contents are sent to TCL. Any data left in the stack is cleared upon exiting from EXECUTE.

Using EXECUTE with Select-Lists

Although the Pick BASIC language has a SELECT statement for creating a select-list, the EXECUTE statement can be used to run one of the ACCESS select-list generators (e.g., SELECT, SSELECT, FORM-LIST, QSELECT). The ACCESS select-list generators are often preferable to the SELECT statement because they can include selection and sort expressions. Selection and sort expressions cannot be supplied with the Pick BASIC SELECT statement.

On most systems the active select-list is accessible to the next EXECUTE statement. For example, the next EXECUTE might use SAVE-LIST to save the list in the POINTER-FILE.*

See the section "Reading and Updating File Items" later in this chapter for more information about select-lists.

Executing Another Pick BASIC Program (ENTER)

To execute another Pick BASIC program, the EXECUTE statement is recommended. There is, however, an ENTER statement which acts only to execute another Pick BASIC program (the program must be cataloged) and then exit without returning to the calling program.

Sending Output to the Screen and Printer

Output Devices (PRINT, CRT, DISPLAY)

There are two standard output devices available to a Pick BASIC program: the terminal screen and the printer. The CRT and DISPLAY statements† are identical: both send output only to the terminal screen. The PRINT statement sends its output either to the terminal screen or to the printer, depending on which has been selected as the "output device." The syntax of

---

* On some implementations the select-list is unavailable to subsequent EXECUTE statements; on these systems, the DATA statement can be used to stack a SAVE-LIST command to be executed before returning to the program.

† The SMA standards only include CRT.

all three statements is similar, except that the PRINT statement accepts the ON keyword for multiple print units.

File items and attached tape devices can also be considered output devices in a broader sense. For information on file and tape I/O, refer to the sections "Reading and Update File Items" and "Reading and Writing Tapes" later in this chapter.

Sending Output to the Printer (PRINTER ON/PRINTER OFF)

The PRINT statement by default sends output to the screen. There are two ways, however, to force the PRINT statement to send output to the printer: by the P option to the RUN command, or by the PRINTER ON statement. The PRINTER ON statement signifies that the output from all subsequent PRINT statements will be sent to a spooler print file. When the program finishes executing, the print file is spooled to the printer. (See *A Guide to the Pick System* for a detailed explanation of print files and the Pick spooler.)

The PRINTER OFF statement returns to the default condition: the output from all PRINT statements after a PRINTER OFF statement will be sent to the terminal screen again.

To print output before the program finishes executing, use the PRINTER CLOSE statement to send everything in the print file directly to the printer.

Print Units

When output is being sent to a printer, the ON keyword to PRINT becomes significant. Generally, all printer output is sent to BASIC print unit 0. However, if several reports are being generated simultaneously, the ON keyword can be used to place output in several different print units. These print units are integral to Pick BASIC. When the print jobs held in various print units are queued to the spooler for printing, they may be assigned spooler entry numbers that are different from the print unit numbers assigned by the BASIC source code.

For example, suppose a program generates two reports, one displaying the names of all customers who are two months late on their bills, and the other displaying the names of all customers who have birthdays approaching. The program goes through each customer's record in sequence. If bills have not been paid, the customer's name and address are sent to print unit 0, and the customer is billed; if the customer has a birthday coming up, the name and

address are sent to print unit 1, and the customer is sent a birthday card. At the end of the program, two complete and distinct lists are printed out.

Formatting and Positioning Output

Normally output will be printed at the current position and will force a carriage return and linefeed at the end of output. The print expression, however, may include features to tabulate output, to suppress the carriage return and linefeed, and (in the case of screen output) to place output at any coordinate, clear the screen, clear the line, or access any of several terminal capabilities.

In addition, data can be formatted directly using a *format expression.* Format expressions in Pick BASIC are used to convert stored data into a readable format without changing the data itself.

Tabulation and Carriage Return Suppression

A comma in the print expression will force a tab to be printed at that position. A trailing colon (:) specifies that the automatic carriage return and linefeed be suppressed in output.

Formatted Screens (@)

The @ function provides direct control of a terminal screen. When the @ function is used in a print expression, it generates a command sequence that is sent to the terminal screen, and the screen responds accordingly. In particular, the @ function can be used to move the cursor to any coordinate position on the screen. It can also be used to clear the screen, to clear to the end of the line, or to place the text in blinking mode. A full list of the features for the @ function is included in the reference page for @.

Using the @ function, a formatted screen can be generated. Programs can use the @ function to clear the screen and show a menu by sending menu options to different positions on the screen. The programmer might choose to turn the echo feature off to prevent user input from appearing on the screen.

For formatted screens, the INPUT @ statement can take input from any position on the screen. In addition, the INPUT @ statement can include an

expression to convert the format of the input. See the section "Terminal Input" later in this chapter for more information on INPUT @.

For example, to print a menu on the screen, the source code might read:

```
PRINT @(-1) :
PRINT @(8,3) : "CHOOSE ONE: " :
PRINT @(16,6) : @(-13) : "E" : @(-14) : "DIT AN ENTRY" :
PRINT @(16,8) : @(-13) : "N" : @(-14) : "EW ENTRY" :
PRINT @(16,10) : @(-13) : "D" : @(-14) : "ELETE AN ENTRY" :
PRINT @(16,12) : @(-13) : "Q " : @(-14) : "UIT" :
ECHO OFF
INPUT @(1,23) : ANSWER,1
```

The code in the preceding example does the following:

- The first line of code clears the screen.
- The second line prints "CHOOSE ONE" at column 8, row 3.
- The third through sixth lines print the menu options at specific positions, with the first character in reverse video mode. Thus, the first character stands out on the screen.
- The seventh line turns off the echo.
- The eighth line places the cursor at the bottom of the screen and accepts a single character as a response.

Formatting Data

Numbers are often stored in a special *internal format*. Storing data in an internal format makes calculation easier, but stored data can be difficult to read. *Format expressions* convert numbers into a format that is easier to read. In addition, the ICONV function converts string data that is being input into a specified internal format, and the OCONV function converts data strings stored in internal format into a specified output format.

For example, if you were to store the dollar amount "$14,912.15" with the dollar sign and comma, no calculations on that number would be possible—dollar signs and commas cannot be stored as part of a numeric string. Also, if interest is being calculated on this dollar amount, it would be more accurate if more than two decimal places were being kept.

Suppose the given dollar amount represents the current balance of a bank account. The bank keeps this figure to five digits of precision to ensure that any calculations are accurate—the actual figure stored might be "1491214987". When this amount is printed in a monthly statement, the data needs to be converted into a readable form. The program which

generates the monthly statements will therefore use a format expression when it prints the data. The format expression descales the number, rounds it to two decimal places, inserts commas where necessary, and places a dollar sign in front of the number. If the variable BALANCE contains "1491214987" and the program contains the lines:

```
PRECISION 4
.
.
.
PRINT BALANCE "29,$"
```

the output will be displayed or printed as:

```
$14,912.15
```

Output format is specified by adding a format expression after the data, as shown in the example. In the source code, the "2" signifies that output should be rounded to two decimal places. The "9" is a *descaling code*, which determines where to place the decimal point—in this case, 5 digits from the right. (Since the default precision of 4 decimal places was specified earlier in the program, a descaling code of "9" is specified in the format expression. The actual number used to descale the data is calculated by subtracting the decimal precision (4) from the descaling code (9) in the format expression, which gives a result of 5.) The "," specifies that a comma be printed at every thousands place, and the "$" places a dollar sign in front of the expression. There are many other codes available for formatting data. For a full list and explanation of these codes, see Chapter 5.

Headings and Footings

The HEADING statement can be used to specify that a heading be printed at the top of each page. It also has the facility to set up page parameters for use by FOOTING and PAGE.

If the output is being sent to the screen, output will stop after each page of text. If a FOOTING statement is specified, a footing will be supplied at the end of the page, and the program will wait for a carriage return before continuing with output.

The PAGE statement can be used to force a new page at any point in the program.

Note that HEADING, FOOTING, and PAGE only affect the same output device that PRINT does. If multiple print units are being used to print several print files simultaneously, HEADING, FOOTING, and PAGE will affect only print unit 0 (the default).

Terminal Input

The INPUT Statement

In the simplest form of terminal input, the user can be prompted to enter a value for the variable ANSWER with the statement:

```
INPUT ANSWER
```

This statement prints the prompt character on the screen and waits for terminal input at that position. The user can type a response and press the RETURN key for the response to be accepted. (The prompt character can be reassigned with the PROMPT statement.)

Variations on INPUT

There are several variations to the INPUT statement. For example, suppose that the answer the user is prompted for can be only "Y" for "yes" or "N" for "no." The INPUT statement can specify that only one character is expected, with the statement:

```
INPUT ANSWER,1
```

The maximum number of characters that will be accepted as input is one. When the user types one character, the program assumes that the input is complete and continues execution immediately, without waiting for a carriage return. Any positive integer can be used in an INPUT statement as the maximum length of input, up to the size of the input buffer.

To force the program to wait for a carriage return before accepting the response, an underscore (_) can be placed at the end of the INPUT statement. When you use the underscore, the program will send a "beep" to the terminal if the user tries to type more than the maximum number of characters, but it will wait for a carriage return before accepting the input. Thus the user is given a chance to verify the response before continuing with the program.

In addition, the INPUT statement can be used to print data in the field to be written. The field might contain a default answer, to be accepted (by pressing the RETURN key) or to be reassigned (by backspacing, typing the new answer, and pressing the RETURN key). The field might also be filled by a "fill character," showing (for example) five asterisks when a five-digit zip code is requested.

Masked Input Statements (INPUT @)

The INPUT @ statement combines two valuable enhancements to the INPUT statement: it accepts screen coordinates for the input string, and it allows input and output formatting to be applied directly to the input.

The INPUT @ statement takes screen coordinates as arguments. When INPUT @ is executed, the prompt character is printed at the position specified by the coordinates. If the variable to be input already has a value, that value is displayed just after the prompt, with the cursor positioned at the first character of the displayed value. The user can then either accept the displayed value by pressing the RETURN key, or enter new data at the prompt. Once the response has been supplied, the INPUT @ statement verifies it against the format expression (if there is one). When the input data is verified, it is converted to internal format for storage. If the response is not verified, an error message is printed at the last line of the screen and the user is prompted again.

There are a number of different kinds of format verification available. These are discussed in the section "Formatting and Positioning Output" earlier in this chapter, and also in Chapter 5 under the heading "Format Expressions."

Consider the case of a screen-formatted program which prompts for a date and then converts it into internal format. Using the standard INPUT statement, the program would have to use a loop to place the cursor at the right position, prompt for the input, test for every way a date can be written ("JUNE 4 1990", "4 JUN 1990", "6/4/90", "06/04/1990", "6-4-90", etc.), and then convert it to internal format. With the INPUT @ statement, on the other hand, the programmer can simply write:

```
INPUT @(14,10) BIRTHDATE "D"
```

See the section "Sending Output to the Screen and Printer" earlier in this chapter for more information on formatting data.

INPUTTRAP, INPUTNULL and INPUTERR

Several statements were designed to be used concurrently with INPUT @.

The INPUTTRAP and INPUTNULL statements were designed to provide escapes from the internal loop of INPUT @. The INPUTNULL statement specifies a character which will be interpreted as the null string by INPUT @. The INPUTTRAP statement allows the programmer to specify characters which, if supplied as answers to INPUT @, will branch to another statement label. Using INPUTTRAP, the operator can be told, in a

menu program for example, that entering ESC at any prompt will exit the program, and entering CTRL-Z will return to the main menu.

The INPUTERR statement prints a message on the last line of the screen. This message will be cleared when correct input is taken by a succeeding INPUT @ statement. The INPUTERR statement can be used to print a message about what sort of input is expected, or it can be used in a loop with INPUT @ if a response requires further testing. For example, if the programmer requires a date within the next year, the code might read:

```
TODAY=DATE( )
VALID = 0
INPUTERR "PLEASE ENTER A DATE WITHIN THE NEXT YEAR"
LOOP
   INPUT @(14,10) RES.DATE "D"
   BEGIN CASE
   CASE RES.DATE < TODAY
      INPUTERR "INVALID INPUT.  PLEASE ENTER A FUTURE DATE."
   CASE RES.DATE > TODAY + 365
      INPUTERR "PLEASE ENTER A DATE WITHIN THE NEXT YEAR."
   CASE 1
      VALID = 1
   END CASE
UNTIL VALID DO REPEAT
```

See the section "Sending Output to the Screen and Printer" earlier in this chapter for more information both on data formatting and on formatted screens.

INPUT and the Data Stack

If the data stack is not empty, its contents will be supplied to any terminal input statement, and the user will not be prompted. The data stack is assigned with the DATA statement.

See the section "External Program Control" earlier in this chapter for more information about the data stack.

Dynamic Array Processing

All data on Pick systems is stored as a string. File items are separated by segment marks, and lines in file items are separated by attribute marks. For processing data in file items, therefore, Pick BASIC supplies several powerful string functions. Using these functions, fields in a file item can be distinguished and processed separately.

There are two categories of string function: those that require a delimiter to be specified, and those that assume the dynamic array delimiters.

File Items and Dynamic Arrays

Dynamic arrays are powerful data structures in Pick BASIC, since they can be used to represent the contents of a file item. A dynamic array is simply a string variable with attribute marks, value marks, and subvalue marks taken to be field delimiters.

When a file item is read into a string variable by a READ or READU statement, the fields are separated by attribute marks (CTRL-^, or CHAR(254)), and subfields are separated by value marks and subvalue marks (CTRL-] and CTRL-\, or CHAR(253) and CHAR(252)). The string variable is thus in the form of a dynamic array and can be manipulated by the dynamic array functions.

Note that when these delimiters are sent to the screen by a PRINT or CRT statement, they do not appear as you might expect them to: Pick BASIC subtracts 127 from the ASCII value of a high-order character on output. Thus CHAR(254) appears on output as ~, CHAR(253) appears as }, and CHAR(252) appears as | .

See the section "Reading and Updating File Items" later in this chapter for more information on reading file items into a Pick BASIC program.

Dynamic Array Functions

To examine or alter the contents of a particular attribute, value, or subvalue of a dynamic array, Pick BASIC provides the EXTRACT, REPLACE, INSERT, and DELETE functions.

The EXTRACT function returns the contents of the specified attribute, value, or subvalue. For example, if Attribute 6 of the dynamic array CUST contains the customer's zip code, a variable ZIP can be assigned with:

```
ZIP = EXTRACT(CUST,6)
```

The REPLACE function replaces the contents with new data. For example, if a customer had a new zip code NEW.ZIP, the program can replace the old zip code in Attribute 6 of the array CUST with:

```
CUST = REPLACE(CUST,6; NEW.ZIP)
```

The INSERT function inserts data as an attribute, a value, or a subvalue at the given position. For example, if Attribute 6 of CUST does not exist, it can be assigned to the customer's zip code ZIP with:

```
CUST = INSERT(CUST,6; ZIP)
```

The difference between REPLACE and INSERT is that REPLACE overwrites any data already in the given position, whereas INSERT simply moves the data to the next position. That is, if Attribute 6 already exists in the previous example, the newly inserted data becomes Attribute 6, the old Attribute 6 becomes Attribute 7, and so on.

The DELETE function deletes the specified attribute, value, or subvalue. For example, Attribute 7 of CUST can be deleted with:

```
CUST = DELETE(CUST,7)
```

The DELETE function does not perform the same function as using REPLACE with the null string. By deleting Attribute 7, Attribute 8 becomes Attribute 7, Attribute 9 becomes Attribute 8, and so on. On the other hand, by replacing Attribute 7 with the null string, Attribute 7 becomes null and all other attributes remain unchanged.

The LOCATE Statement

The LOCATE statement provides an extremely powerful way to manipulate dynamic arrays. The LOCATE statement searches for a particular attribute, value, or subvalue within a dynamic array string (or subset thereof). If the data has been sorted in ascending or descending order, the order can be specified in the LOCATE statement. THEN and ELSE clauses are accepted by LOCATE to specify action if the string is or is not found.

If the string is found, the LOCATE statement sets a specified variable to the position where the data was found, and the statements of the THEN clause, if included, are executed. If the string is not found where expected, the variable is set to the current position plus one, and the statements in the ELSE clause are executed. In the ELSE clause, the variable can be used with an INSERT function to place the data in the proper position.

For example, if a dynamic array LIST contains names in alphabetical order separated by attribute marks, a new name NAME can be inserted with:

```
LOCATE(NAME, LIST; POSITION; 'AL') THEN
   PRINT NAME : " ALREADY LISTED."
END ELSE
   NAMELIST = INSERT(LIST , POSITION;  NAME)
END
```

Alternate Forms for Dynamic Array Processors

Many Pick implementations now support alternate forms for each of the dynamic array processing functions. These forms uses angle brackets in referencing a dynamic array field, thus simulating the syntax for referencing dimensioned arrays. Since the angle brackets tend to be more "intuitive," they are generally preferred over the older syntax forms.

The preceding example lines might have read:

Old	New
ZIP = EXTRACT(CUST,6)	ZIP = CUST<6>
CUST = REPLACE(CUST,6; NEW.ZIP)	CUST<6> = ZIP
CUST = INSERT(CUST,6; ZIP)	INS ZIP BEFORE CUST<6>
CUST = DELETE(CUST,7)	DEL CUST<7>
LOCATE(NAME, LIST; POSITION; 'AL')	LOCATE NAME IN LIST BY AL

Counting Delimiters and Substrings

The COUNT function returns the number of times a specified substring appears in a string. The DCOUNT function returns the number of fields separated by a given delimiter.

DCOUNT can be very useful for processing dynamic arrays as well as other strings. For example, the number of attributes in a string ADDRESSES can be determined with:

```
NO.OF.ATTRS = DCOUNT(ADDRESSES, CHAR(254))
```

Generalized String Processing

The EXTRACT, REPLACE, INSERT, and DELETE statements are very powerful for referencing and adapting dynamic arrays in Pick BASIC, but they depend on the standard delimiters being used within the array. If a string with different delimiters between its fields needs processing, the programmer is forced to use the more generalized string processing functions.

Substring Extraction

A substring is specified by a starting character position and a substring length, separated by commas and enclosed in square brackets. The general syntax for a portion of a string is:

string **[***start,length* **]**

where *start* is the starting column position and *length* is the length of the substring. The brackets here are part of the syntax and must be typed.

Substring Assignment

Some Pick systems allow you to combine substring extraction syntax with the assignment statement. The syntax is as follows:

var **[** *start,length* **]** = *string*

Using this syntax, substrings can be extracted and characters can be replaced like the fields of a dynamic array or the elements of a dimensioned array. For example, if the string NAME contains "SHAW, GEORGE BERNARD", the variable FIRSTNAME can be assigned "GEORGE" with:

```
FIRSTNAME = NAME [7,6]
```

By using column positions, therefore, the EXTRACT function can be simulated for string variables. To simulate the REPLACE, INSERT, and DELETE functions, portions of a string can be assigned values directly. To substitute the string "RALPH" for "GEORGE", for example, the code might read:

```
NAME[7,6] = "RALPH"
```

The only thing which is not obvious in manipulating substrings is how to determine the column position and length of the substring. For this purpose, the FIELD, COL1, COL2, LEN, and INDEX functions are a vital part of string processing.

The FIELD Function

The FIELD function accepts any character as a delimiter and returns the specified field, thus acting as a generalized EXTRACT. For example, if the string NAME contains "SHAW, GEORGE BERNARD", then the last name "SHAW" can be placed in the variable SURNAME with:

```
SURNAME = FIELD(NAME, "," ,1)
```

Notice that the second comma is enclosed in quotes to indicate that it is the second of three expressions and not an expression delimiter.

The COL1, COL2, and LEN Functions

The COL1 and COL2 functions respectively return the column positions immediately before and immediately after the last FIELD function. The LEN function returns the number of characters in a string.

For example, if the string NAME contained "SHAW, GEORGE BERNARD", the first name "GEORGE" could be deleted with:

```
FIRSTNAME = FIELD(NAME, " " ,2)
NAME [COL1( ), COL2( ) ] = " "
```

The string NAME now contains "SHAW, BERNARD".

The INDEX Function

The INDEX function returns the column at which a particular substring can be found in a string. This value can be used in a substring assignment statement to replace or delete the substring.

For example, if the variable NAME contained the string "SHAW, GEORGE BERNARD", the word "BERNARD" could be stripped out with:

```
COLUMN = INDEX(NAME, " ",2)
NAME = NAME [ 1, COLUMN –1 ]
```

Trimming Spaces

The TRIM function removes all extra spaces from a string. It trims all multiple spaces and all spaces at the beginning and at the end of a string. Some Pick implementations also support the TRIMF and TRIMB functions, which respectively trim leading blanks and trailing blanks from a string but leave all other blank spaces intact.

Dimensioned Arrays

A dimensioned array is a one- or two-dimensioned structure for data. Elements of the array can be thought of as cells rather than fields.

The most significant difference between dimensioned and dynamic arrays is that the size of a dimensioned array must be assigned at compilation, whereas the size of a dynamic array can vary at run-time according to need. Each element in a dimensioned array has a direct pointer; each field in a dynamic array, on the other hand, requires the entire string to be searched through from the start.

For example, suppose a dynamic array contains a thousand attributes. Attribute 999 actually refers to the data between the 998th and 999th attribute marks; therefore, to access attribute 999, the processor has to search through the string until 999 attribute marks are found. If several fields towards the end of the string need to be accessed this way, the run-time of the program can be drastically increased.

If, however, a vector (a one-dimensional array) were used instead of a dynamic array, the processor has to access only a single direct pointer to element number 999 in order to retrieve the data. Dimensioned arrays, therefore, provide a shortcut to the field.

See Chapter 2 for a more complete description of the structure of dimensioned arrays.

Assigning Dimensioned Array Variables (DIM)

Before a dimensioned array can be used, its dimensions need to be declared. The DIM statement declares a dimensioned array. Since the DIM statement is interpreted by the compiler,* neither variables nor expressions can be used in a DIM statement.

If the array is to be allocated space in the common area, the COMMON statement can also be used to declare a dimensioned array. Arrays that have been declared with the COMMON statement, however, should not be declared again with a DIM statement.

---

* Except on Prime INFORMATION and uniVerse systems, where DIM statements are executed at run time.

MATREAD and MATWRITE

The MATREAD and MATWRITE statements read file items directly into dimensioned arrays and write dimensioned arrays back into file items. See the section "Reading and Updating File Items" later in this chapter for more information on reading and writing file items.

The MAT Statement

The MAT statement assigns all elements of a dimensioned array to a single value or to the values of another array. For example, to assign all elements of an array ARRAY to "6", the code might read:

```
MAT ARRAY = 6
```

This use of the MAT statement is a shorthand for:

```
FOR I = 1 TO MAXELTS
   ARRAY(I) = 6
NEXT I
```

where MAXELTS represents the maximum dimensions of the array.

To assign the elements of ARRAY to the elements of ARRAY2, the code might read:

```
MAT ARRAY = MAT ARRAY2
```

which is a shorthand for:

```
FOR I = 1 TO MAXELTS
   ARRAY(I) = ARRAY2(I)
NEXT I
```

Reading and Updating File Items

File Variables (OPEN)

Before an item in a file can be accessed, the file must be assigned a symbolic name, called a *file variable*. The file variable is necessary to provide a pointer to the file. This pointer is used by the program each time the file is accessed. The OPEN statement assigns a file variable to a file, so that the program can read, write, select, or delete items in the file. All

subsequent access of the file must use the file variable rather than the filename to reference the file.

If a file is opened and no file variable is specified, the program uses the default file variable. Any subsequent file access statements that do not use a file variable to reference the file will also use the default file variable. Only one file at a time can be assigned to the default file variable.

Reading and Writing a File Item

Once a file is opened, any item can be directly accessed. The READ statement assigns the string value of a file item to a dynamic array variable. The fields of the array can then be accessed by the dynamic array processing functions EXTRACT, REPLACE, INSERT, DELETE, and LOCATE. The WRITE statement writes a new or updated dynamic array string into a file item.

Pick BASIC provides several variations to READ and WRITE. The READV and WRITEV statements read and write only a single attribute of an item, as a shortcut for programs which are concerned only with a single attribute. In addition, the MATREAD and MATWRITE statements read and write items as dimensioned arrays, with each attribute corresponding to an element of the array.

The file-reading statements are each equipped with THEN and ELSE clauses. If the item is not found, the ELSE statements are executed; if it is found, the THEN statements are executed. See the section "Internal Program Control" earlier in this chapter for more information on the syntax of THEN and ELSE clauses.

Item Locks (READU, WRITEU, RELEASE, etc.)

Each of the statements for reading a file item have corresponding statements that place a lock on the item as it is read. These statements are the READU, READVU, and MATREADU statements. (The "U" suffix stands for "Update," declaring that the item might be changed and rewritten.) The item lock is lifted either when the item is released with a RELEASE statement, or when it is written with a WRITE, WRITEV, or MATWRITE statement, or when the program is terminated. Until the lock is lifted, no other users will be able to access the same item with a READU, READVU, or MATREADU statement.

Item locks only affect other READU, READVU, and MATREADU statements. While an item is locked, programs can access the item with a normal READ, READV, or MATREAD statement, or they can even write it with any of the file writing statements. The success of an item lock is dependent on its being respected by all other programs that access the same file.

If an item is to be written but the programmer does not want the lock removed, the WRITEU, WRITEVU, and MATWRITEU statements should be used in place of WRITE, WRITEV or MATWRITE. These statements will write the item but retain the item lock for subsequent update. The "U" suffix again stands for "Update," declaring that further updates might occur.

The LOCKED Clause

The item-locking statements READU, READVU, and MATREADU are each equipped with an optional LOCKED clause. Normally when a program attempts to read and lock an item which is already locked, the program waits for the item to be released before continuing with execution. If the LOCKED clause is included, however, the program simply executes the LOCKED statements and continues with execution immediately. The LOCKED statements follow the syntax of THEN and ELSE clauses in Pick BASIC.

The LOCKED clause helps to avoid the situation called the "deadly embrace." A deadly embrace occurs when two users both lock items, and before releasing their locks, each user then tries to read and lock the other item. Without the LOCKED clause, both users will be indefinitely stuck, since neither is free to unlock its item. If the LOCKED clause is used, however, the deadly embrace cannot occur.

Select-Lists (SELECT, READNEXT)

Select-list variables can be created with the Pick BASIC SELECT statement, or by using the EXECUTE statement to call one of the ACCESS select-list generators. The SELECT statement does not accept the selection or sort expressions accepted by the ACCESS verbs; however, the SELECT statement does allow a select-list to be created from the attributes of a dynamic array string. See the section "External Program Control" earlier in this chapter for more information on using EXECUTE for generating select-lists.

A select-list can also be created by executing one of the ACCESS select-list generators and then immediately running the program. If the program is designed this way, the SYSTEM(11) function* is recommended to test if there is an active select-list.

Once the select-list is created, it can be read with the READNEXT statement. READNEXT reads the next item ID in the select-list. After selecting a file, the READNEXT statement is generally used in a loop to perform a procedure on all selected items.

Deleting File Items (DELETE, CLEARFILE)

The DELETE statement deletes a specific file item from an open file. It should not be confused with the DELETE function (or the DEL statement available on some implementations), both of which delete a field from a dynamic array.

The CLEARFILE statement deletes all items in the data file.

Reading and Writing Tapes

Pick BASIC includes several statements for tape processing. For reading and writing strings on tape, there are the READT and WRITET statements. As expected, the READT statement reads the next record off the attached tape device, and the WRITET statement writes a record onto the tape.

There are also statements to simulate the T-WEOF and T-REW commands. The WEOF statement writes an End-Of-File mark at the current position of the tape, and the REWIND statement rewinds the tape to the beginning.

Each of the tape I/O statements includes THEN and ELSE clauses to specify action according to whether the tape statement was successful. The ELSE clause is often used to produce a meaningful error message by calling the SYSTEM(0) function. The SYSTEM(0) function returns a number that indicates whether the latest tape I/O statement worked, and if it didn't, what the problem was. See the SYSTEM function for more information.

---

* The number of the list-testing function may be different on your implementation. Check your system documentation to see if your system's SYSTEM function includes a list-testing option.

Execution Locks

The LOCK statement sets an *execution lock*, which establishes that only one process can use a program or subroutine until the lock is removed. The maximum number of locks differs on different Pick implementations: earlier PICK Systems releases allowed up to 48, whereas 64 are now supported. Other systems support up to 128. ADDS Mentor supports up to 256. On some implementations these locks can be accessed only through Pick BASIC.

Execution locks should not be confused with item locks, since they use a very different mechanism.

Execution locks are set with the LOCK statement by specifying a lock number. That lock number is determined only by local convention. All the system does is establish the maximum number of "slots," and keep track of whether the slot is taken or not. It is up to the application programs to take advantage of this structure.

For example, suppose a particular subroutine tends to slow down the system each time it is used. If a LOCK statement is used at the beginning of the subroutine, only one user will be able to execute the subroutine at any given time. The lock number should be unique.

Execution locks are released at the termination of the program, or at the encounter of an UNLOCK statement. The UNLOCK statement can be used to release a specific lock number or to release all locks set by the current program.

The THEN and ELSE Clauses to LOCK

LOCK is supplied with optional THEN and ELSE clauses that have the same effect as the LOCKED clause for READU. Normally, when a LOCK statement is used on a lock number which is already locked, the program will wait for the lock to be lifted before continuing with execution. However, if the THEN or ELSE clause is included, the program will simply execute the ELSE clause, if present, and continue with execution immediately.

The THEN and ELSE clauses help to avoid the "deadly embrace." A deadly embrace is when, for instance, one user sets execution lock 1, and another user sets execution lock 2. The first user then attempts to access the procedure controlled by lock 2, and the second user attempts to access the procedure controlled by lock 1. Each user is now stuck, since the program

will wait indefinitely for the lock to be released, and neither is free to release the lock that has already been set by their process. If the THEN or ELSE clause is used, however, the deadly embrace cannot occur.

To display a list of all execution locks, use the WHAT verb.

Compiler Directives

This section discusses four compiler directive statements. These statements are not supported by all systems* and are not included in the SMA standards. Many of these statements begin with a dollar sign ($).

Comments in the Object Code

The $* statement places a comment directly in the object code of a program when it is compiled. It is most useful for entering version numbers or copyright information before software is distributed.

Reading In External Source Code

There are a number of statements, different on different Pick systems, that tell the compiler to read source code from another file item. Among these statements are INCLUDE, $INCLUDE, $INSERT, and $CHAIN.†

INCLUDE, $INCLUDE, and $INSERT are similar statements. All of these statements result in the program being compiled as if the external source code were written at the point where the INCLUDE statement has been entered. Compilation then continues at the line after the INCLUDE statement.

The INCLUDE statements are useful for any code that is used by several different programs. An example of such code might be a file item containing COMMON statements.

The $CHAIN statement is different from the INCLUDE statements in that the compilation does not return to the original program. The $CHAIN statement is not intended for code which might be shared by several

---

* Among those that *do* support them are Ultimate and ADDS Mentor.

† Only INCLUDE is listed in the SMA standards.

programs, but for programs which may have source code longer than 32K. The $CHAIN statement allows several different file items containing source code to be chained together.

The only restriction to INCLUDE, $INCLUDE, $INSERT, and $CHAIN is that the number of bytes in the resulting object code cannot exceed 32K.

Miscellaneous Statements and Functions

The SLEEP and RQM statements suspend program execution for a number of seconds, or until a specified time of day.

The ECHO ON/ECHO OFF statements turn the echo feature for the attached process on or off.

The REM, *, and ! statements allow comment lines to be placed in the source code. This statement allows the programmer to document code and make it more accessible to future modification.

The PROCREAD and PROCWRITE statements allow the program to read to and write from the primary input buffer of the calling proc.

Conversion Codes (ICONV, OCONV)

The ICONV function translates a string from external to internal format, according to the ACCESS conversion codes. The conversion codes supported are those for dates, time, hexadecimal, and table translation. The OCONV function translates back from internal to external format. See *Pick ACCESS: A Guide to the SMA/RETRIEVAL Language* for more information about conversion codes.

The SYSTEM Function

The SYSTEM function returns significant information about the system. The codes vary from system to system. The SYSTEM function provides such information as:

- The command-line options used to the RUN command.
- The error code for a failed tape I/O statement, or the tape record length.

- Whether the program was called by a proc, whether there is an active select-list, whether the data stack is empty, or whether the program is cataloged.
- Whether output is being sent to the printer, the number of lines left on the current page, the current page number, and current open spooler files.
- The operator's terminal type, or the number of lines or columns on the operator's terminal.
- The user's account number, process number, or line number.

Entering the Debugger

If the BREAK key is pressed during program execution, the user is placed in the Pick BASIC debugger. This feature can be disabled and reinstated with the BREAK statement. The BREAK statement does not simply toggle the break feature, it also increments and decrements the Break Inhibit Counter.

Alternative ways of entering the debugger are to use the D option to the RUN command or to place a DEBUG statement directly in the source code.

Chapter 4 has a full description of debugger commands and how to use them. See Chapter 4 for more information on using the Pick BASIC debugger.

CHAPTER 4

Using the Pick BASIC Debugger

Debugger Commands: Quick Reference

B ..... Set breakpoint condition.

D ..... Display breakpoint and trace tables.

DE ..... Escape to system debugger.

DEBUG ..... Escape to system debugger.

E ..... Set or disable execution step.

END ..... End program and return to TCL.

G ..... Continue execution until next breakpoint, fatal error or step.

K ..... Delete breakpoint condition.

LP ..... Toggle output to the printer.

N ..... Ignore specified number of breakpoints.

OFF ..... Log off.

P.....Toggle printing of program output.

PC .....Close print file and spool print job to the printer.

R ..... Pop return stack.

S.....Display subroutine stack.

T.....Set trace variable or toggle trace table off and on.

U .....Remove trace variable from table.

V ..... Verify the object code.

Z.....Identify source code.

$..... Print current line number.

? ..... Print current line number.

/.....Print and change the value of a simple or array variable.

/*.....Print entire symbol table (all variables).

[.....Set or remove a string window.

Fixing a Bug

A bug in a program is an error in the program's logic which either prevents or impedes the program's performance. If a program doesn't work perfectly, it is said to have a bug, and you must either fix it or find a way around it.

Most new programmers debug a program by running it, reading the error message and its associated line number, and then examining the source code at the specified line. In many cases, this is enough: short programs that return messages like "RETURN EXECUTED WITH NO GOSUB" are fairly easy to fix.

You can also place PRINT statements at key points of execution that report what's happening as the program executes. You can place a PRINT statement within a conditional to determine whether a condition has proven true or not, or to display the value of variables. You can also include a PRINT statement in a program loop to report how many times the loop is being executed.

These debugging methods are fine as far as they go; however, they tend to be tedious to implement, and you have to recompile the program after each attempt. When you use the Pick BASIC interactive debugger, the debugging process becomes simpler and tidier.

Your first experience with the Pick BASIC debugger is probably the result of an accident, either through a fatal error or because you pressed the BREAK key by mistake. At first, the only thing you want to know about the debugger is how to get out of it (if you got into the debugger through a fatal error, enter "END" at the debugger prompt; if you got into it because you accidentally pressed the BREAK key, enter "G"). With a little patience, however, you can learn to make the debugger do the dirty work of fixing a program.

A Sample Program

To demonstrate how the debugger can be used, take the example of an internal office program called BIRTHDAY. The BIRTHDAY program asks users for their birthdays and then tells them how many days they have until their next birthday. The program works fine for the programmer:

```
>RUN BP BIRTHDAY
```

The screen is cleared and the following prompt appears:

```
ENTER YOUR BIRTHDAY: MM/DD/YY
```

The underline represents the position of the cursor. You enter a date:

```
ENTER YOUR BIRTHDAY: 6/4/65
```

The program converts the date and then prints out how many days until the programmer's next birthday and how many years old they will be.

```
ENTER YOUR BIRTHDAY: 04 JUN 1965
ON 06/04/1991, IN 319 DAYS, YOU WILL BE 26 YEARS OLD.

>
```

You are satisfied by this performance, since the tricky part of the program is to make sure the it doesn't report a birthday that has already passed (e.g., "IN –46 DAYS, YOU WILL BE 25"). Since the test above was performed in July 1990 and the date entered was in June, the tricky part seems to have been solved.

Later the same day, however, it was reported that the program did not work. In particular, when supplied a September date, the program still jumped ahead a year:

```
ENTER YOUR BIRTHDAY: 3 SEP 1961
ON 09/03/1991, IN 410 DAYS, YOU WILL BE 30 YEARS OLD.
```

You quickly examine the source code but can't find an error. However, using the debugger, perhaps we can discover something you missed.

Printing Source Code

First, we run the program with the D option. The D option forces the program to enter the debugger before executing line 1.

```
>RUN BP BIRTHDAY (D)
```

Before the program begins execution, the debugger is invoked and "E1" is printed, signifying that it stopped before executing line 1. The asterisk (*) is the debugger prompt.

```
*E1
*
```

The first thing we want to do is examine the source code. Before we can access the source code, however, we need to identify it with the Z command.

```
*Z<RETURN> FILE/PROG NAME?BP BIRTHDAY
```

The source code is now available for listing by the debugger with the L command. Since it is a short program, L can be used with the * (asterisk)

option, specifying that the entire source code item should be printed. (In actual practice it's usually a good idea to have a printed program listing next to you when you are debugging a program.)

```
*L*
001   PROMPT " "
002   DIM BIRTHDAY(3), TODAY(3)
003   EQUATE TRUE TO 1,
004       FALSE TO 0,
005       BIRTH.MONTH TO BIRTHDAY(1),
006       BIRTH.DATE TO BIRTHDAY(2),
007       BIRTH.YEAR TO BIRTHDAY(3),
008       THIS.MONTH TO TODAY(1),
009       THIS.DATE TO TODAY(2),
010       THIS.YEAR TO TODAY(3)
011   INCREMENT = FALSE
012   PRINT @(-1) : "ENTER YOUR BIRTHDAY:":
013   INPUT @(20,0) BIRTHDAY.INT "D"
014   TODAY.INT = DATE()
015   TODAY.EXT = OCONV(TODAY.INT,"D/")
016   MATPARSE TODAY FROM TODAY.EXT,"/"
017   BIRTHDAY.EXT = OCONV(BIRTHDAY.INT,"D/")
018   MATPARSE BIRTHDAY FROM BIRTHDAY.EXT,"/"
019   IF THIS.MONTH > BIRTH.MONTH THEN
020       INCREMENT = TRUE
021   END
022   IF (THIS.MONTH = BIRTH.MONTH OR THIS.DATE >
                                          BIRTH.DATE) THEN
023       INCREMENT = TRUE
024   END
025   IF INCREMENT THEN
026       THIS.YEAR += 1
027   END
028   AGE = THIS.YEAR - BIRTH.YEAR
029   NEXT.BIRTHDAY = BIRTH.MONTH : "/" : BIRTH.DATE : "/" :
                                                  THIS.YEAR
030   NEXT.BIRTHDAY.INT = ICONV(NEXT.BIRTHDAY,"D")
031   DAYS.TO.BIRTHDAY = NEXT.BIRTHDAY.INT - TODAY.INT
032   IF DAYS.TO.BIRTHDAY = 0 THEN
033       PRINT
034       PRINT "HAPPY BIRTHDAY!"
035       PRINT "TODAY, " : TODAY.EXT : ", YOU ARE " : AGE : "
                                                  YEARS OLD"
036   END ELSE
037       PRINT
038       PRINT "ON ":NEXT.BIRTHDAY:", IN ": DAYS.TO.BIRTHDAY
                                                  :" DAYS,":
039       PRINT " YOU WILL BE ":AGE:" YEARS OLD."
040   END
```

```
041 STOP
042 END

*
```

At first glance, it is obvious that the problem lies in the variable THIS.YEAR: THIS.YEAR is being incremented when it should not be. Before we start editing the program, however, we can use the debugger to confirm our suspicions.

Using Breakpoints and Trace Variables

To examine the value of THIS.YEAR at the end of the program, we need to set up a *breakpoint* so that we can examine variables before the program ends. A breakpoint condition is a condition that invokes the debugger whenever it is true.

The B command is used to assign a breakpoint condition. We choose to break when the THIS.YEAR variable is equal to 1991.

```
*BTHIS.YEAR=1991<RETURN> +
```

The breakpoint condition says to transfer into the debugger when the variable THIS.YEAR is equal to 1991. The plus sign (+) is printed after pressing the RETURN key to signify that the breakpoint was accepted into the breakpoint table.

We enter THIS.YEAR as a *trace variable*. A trace variable is a variable that is printed whenever a breakpoint is encountered. We enter it into the trace table with the T command:

```
*TTHIS.YEAR<RETURN> +
```

We add to the trace table the variables THIS.MONTH, BIRTH.MONTH, THIS.DATE, and BIRTH.DATE. We suspect that the problem may be that these variables are not being assigned correctly.

```
*TTHIS.MONTH<RETURN> +
*TBIRTH.MONTH<RETURN> +
*TTHIS.DATE<RETURN> +
*TBIRTH.DATE<RETURN> +
```

To display the breakpoint and trace tables, we use the D command:

```
*D
T1 THIS.YEAR
T2 THIS.MONTH
T3 BIRTH.MONTH
T4 THIS.DATE
T5 BIRTH.DATE
```

```
T6
B1 THIS.YEAR=1991
B2
B3
B4

*
```

The G command continues execution of the program.

```
*G<RETURN>
```

The screen clears and the prompt is printed. We type "9/3/61" and press the RETURN key:

```
ENTER YOUR BIRTHDAY: 9/3/61
```

The program then halts when the breakpoint is reached.

```
ENTER YOUR BIRTHDAY: 3 SEP 1961
*B1 28     END
 THIS.YEAR 1991
 THIS.MONTH 07
 BIRTH.MONTH 09
 THIS.DATE 20
 BIRTH.DATE 03
*
```

As expected, the condition is true after line 27 has been executed. The message "B1 28" means that item 1 on the breakpoint table caused the break, and the line about to be executed is line 28. (The actual text of that line is displayed in half-intensity on the terminal screen.) The current values of the trace variables are printed.

Displaying and Changing a Variable

The error in the program becomes increasingly obvious as we continue in the debugger. Since the THIS.MONTH, BIRTH.MONTH, THIS.DATE, and BIRTH.DATE contain the correct data, the problem is in the way they are being compared. In order for THIS.MONTH to be incremented, the INCREMENT variable must be true. We confirm this by using the / command to print out the current value of INCREMENT:

```
*/INCREMENT
```

When we press the RETURN key, the value of INCREMENT is displayed, and we are given the opportunity to change its value.

```
*/INCREMENT<RETURN> 1=_
```

We type "0" as the new value for INCREMENT and press the RETURN key.

```
*/INCREMENT<RETURN> 1=0<RETURN>

*
```

We also reset the value of THIS.YEAR to 1990.

```
*/THIS.YEAR<RETURN> 1991=1990<RETURN>

*
```

Using Execution Steps

To find out which comparison is failing, we want to step through the crucial lines of the program this time around, to see what exactly is happening. We want to begin the program executing again after line 19 with an *execution step* of 1. An execution step is a number of lines that should be executed before returning to the debugger. The E command should be used to specify the execution step:

```
*E1
```

We also add the variable INCREMENT to the trace table.

```
*TINCREMENT<RETURN>+
```

The G command continues execution after line 19. One line executes and the program returns to the debugger. The line about to be executed will be printed on the screen, along with any trace variables.

```
*G19<RETURN>
*E20     IF THIS.MONTH > BIRTH.MONTH THEN
 THIS.YEAR 1990
 THIS.MONTH 07
 BIRTH.MONTH 09
 THIS.DATE 20
 BIRTH.DATE 03
 INCREMENT 0
*
```

We step through 3 more times, until we find that the INCREMENT variable has been changed.

```
*G
*E25     END
 THIS.YEAR 1990
 THIS.MONTH 07
 BIRTH.MONTH 09
 THIS.DATE 20
 BIRTH.DATE 03
 INCREMENT 1
*
```

By stepping through the program, we see that the INCREMENT variable is changed immediately before line 25. We list lines 23 through 25 with the L command:

```
*L23-25
023   IF (THIS.MONTH = BIRTH.MONTH OR THIS.DATE >
                                          BIRTH.DATE) THEN
024       INCREMENT = TRUE
025   END

*
```

The problem is in the conditional for the IF statement. It becomes obvious that, as usual, the bug in the program is a simple logical error: the "OR" in line 23 should be an "AND". With that simple edit, the program should run correctly.

We exit the debugger with the END command, edit the source item, and recompile. The output now reads:

```
ENTER YOUR BIRTHDAY: 3 SEP 1961
ON 09/03/1990, IN 45 DAYS, YOU WILL BE 28 YEARS OLD.

>
```

Assigning New Values for Testing

It appears that the bug is gone. We are not confident, however, that the program will still behave correctly at the end of the year. Using the debugger, we can change the value of the variable TODAY.INT in the program and see whether the program still works.

Run the program again with the D option, use the Z command to supply the file name and program name, and set a breakpoint to stop executing before line 16. (The "$" symbol on the breakpoint table represents the current line number.) Then continue execution with the G command.

```
>RUN BP BIRTHDAY (D)

*E1
*Z<RETURN>  FILE/PROG NAME?BP BIRTHDAY<RETURN>
*B$=16<RETURN> +
*G<RETURN>
```

You are prompted for a birthday. After typing a date, press the RETURN key. The program halts before executing line 16.

```
ENTER YOUR BIRTHDAY:  03 SEP 1961
*B1 16 TODAY.EXT = OCONV(TODAY.INT,"D/")
*
```

Before line 16 is executed, reassign the value of the TODAY.INT variable with the / (slash) command, then enter the G command to continue execution.

```
*/TODAY.INT<RETURN>  7672=8401<RETURN>

*G<RETURN>
```

(8401 is the internal value of December 31, 1990.)

The program runs successfully.

```
ON 09/03/1991, IN 246 DAYS, YOU WILL BE 29 YEARS OLD.

>
```

Now that you have a taste of what the debugger can do and why you would use it, we can go over the specifics of its operation.

Entering the Debugger

There are several ways to enter the debugger:

- Pressing the BREAK key during program execution invokes the Pick BASIC debugger. (This feature can be turned off within a program with the BREAK statement.)
- The D option to the RUN command causes the program to enter the debugger before starting execution.
- On systems that support the DEBUG statement, when DEBUG is encountered during program execution, the program enters the debugger at that point.

Fatal errors also invoke the debugger, with or without the user's consent. (Nonfatal errors invoke the debugger only if the RUN command is used with the E option.)

When control passes to the debugger for any of the above reasons, the current line number (preceded by I for "Interrupt," E for "Execution step," or B for "Breakpoint") is printed, and the "*" prompt is displayed.

Once in the debugger, you can print and change variable values, set breakpoint conditions or execution steps, and continue execution with the G

command. When a breakpoint condition or execution step is reached, the debugger is instantly re-entered.

The Symbol Table

Variables within a program are each assigned a symbol, to be referenced by the interactive debugger. When a program or subroutine is compiled, a symbol table is generated with the object code. The debugger accesses the symbol table through the object code pointer in the file dictionary. If the program calls an external subroutine, a complete symbol table can be accessed by the debugger for the external subroutine.

The S option to COMPILE prevents the symbol table from being generated, but it should be used only when a program is fully operational. Without the symbol table, the debugger's function is greatly impeded.

Exiting the Debugger

Other than returning to program execution with G, the debugger can be exited using the following commands:

END	The END command exits program, debugger, and calling proc or program (if any), and returns directly to TCL.
OFF	The OFF command exits both program and debugger, and logs you off the system directly.
DE[BUG]	The DEBUG or DE command transfers execution to the system debugger, and the "!" prompt is displayed. You can return to the Pick BASIC debugger ("*" prompt) with the G command to the system debugger.

Displaying and Changing a Variable (/)

One of the most valuable things you can learn about a failed program is what happens to the variables at different points in the program. By examining variable values, you can determine which variables are being

assigned incorrectly and thus find out which statements are not being executed properly.

Displaying All Variables

The /* command displays all variables in the symbol table. All variables are reported, including file variables, select-list variables, and dimensioned-array variables. For example:

```
*/*
FILEVAR
PROGRAM=TESTIT
FILE.ARRAY(1,1)=JOEY
FILE.ARRAY(1,2)=FRED FLINTS
   .
   .
   .
```

By using the /* command, you can scan the values for all variables at once. However, the /* command doesn't let you change any values, and if there are many variables or some extremely long string variables, the values may scroll past the screen too quickly to be read.

Displaying and Changing Simple Variables

You can print the values of simple variables (and optionally reassign them) with the / command.

/var [;*length*]

var	the variable name.
length	(not available on all systems). *length* limits the number of characters to be displayed. When *length* is specified, it overrides any string window specified with a [ ] command.

A single variable can be displayed with the / (slash) command and the variable name. You are shown the current value and prompted with an equals sign to change the value at will. Whatever you type before pressing the RETURN key becomes the new value for the variable.

For example, to display the current value of the variable STRING, you might type:

```
*/STRING
```

The debugger responds with the current value of STRING and an equals sign. If STRING contains the word "HELLO", you see:

```
*/STRING<RETURN> HELLO=_
```

The underscore represents your cursor position.

You can then enter a value for STRING and press the RETURN key. To leave the value unchanged, press the RETURN key without reassigning the variable.

On systems that support the *length* argument, you can specify the number of characters to be printed by ending the command with *;length*. For example, suppose the string RECORD is exceedingly long, containing over 2000 characters. If you try to print the output of RECORD in the debugger, the entire screen fills up and the beginning of the string is scrolled off the screen. Since you are interested only in the beginning of the string, limit the number of characters to be printed out by specifying the *;length* parameter.

```
*/RECORD;70
```

This limits output to the first 70 characters of the variable.

Displaying and Changing Dimensioned Arrays

You can also print the values of a single element or of all elements of a dimensioned array (and optionally reassign them) with the / command.

/array [(*row* [,*column*])]

array	the name of the dimensioned array.
row	the row number of the array element. If *row* is omitted, all elements of the array are printed.
column	the column number of the array element. If *array* is two-dimensional, *column* must be supplied if *row* is supplied.

The individual elements of a dimensioned array can be treated like simple variables by referencing them with parentheses. For example, to display the current value of element 2,3 of array NAME.ARRAY, you might type:

```
*/NAME.ARRAY(2,3)
```

The debugger responds with the current value of NAME.ARRAY(2,3) and an equals sign. If element 2,3 contains "HERB", you would see:

```
*/NAME.ARRAY(2,3)<RETURN> HERB=_
```

The underscore represents your cursor position.

You then have the option of filling in a value for NAME.ARRAY(2,3) or leaving it unchanged by pressing the RETURN key.

Alternatively, you can display and change all elements of a dimensioned array by omitting the element reference. For example, to display the current values of all elements of array NAME.ARRAY, you can type:

```
*/NAME.ARRAY
```

You are prompted with the value of each element of the array as if each had been specified individually. For example:

```
*/NAME.ARRAY<RETURN> NAME.ARRAY(1,1)="JOEY"=
NAME.ARRAY(1,2)="FRED FLINTS"=_
   .
   .
   .
```

String Windows ([ ])

The value of some string variables might be too long to be printed on a single screenful. For these strings, use the [ ] command to specify a subset of each string to be printed.

[*start,length* **]**

start	the starting column of the substring.
length	the length of the substring. If *length* is 0, turn off string windowing.

If *start* and *length* are both omitted, string windowing is turned off.

For example, suppose the string RECORD contains over a hundred addresses separated by attribute marks, totaling over 2000 characters. When you try to print the output of RECORD in the debugger, the entire screen

fills up and the beginning of the string is scrolled off the screen. Since you are interested only in characters towards the middle of the string, limit the number of characters to be printed out with:

```
*[800,400]
*/RECORD
```

This limits output to 400 characters.

Accessing Source Code

Lines of source code can be printed with the L command. However, before L can be used, the Z command is necessary to identify the source code. For that reason the Z command should be one of the first commands used when entering the debugger.

Identifying Source Code (Z)

The Z command is necessary for the debugger to locate the source code.

After entering Z and pressing the RETURN key, the prompt "FILE/PROGNAME?" will appear. Supply the filename and program name separated by a space (not a slash, as the prompt would suggest). For example, if the source code is in item ID TESTIT in file BP, enter:

```
*Z<RETURN> FILE/PROGNAME BP TESTIT
```

If the debugger prompt (*) returns, the command was accepted. If the command is illegal for any reason (misspelling, extraneous spaces, etc.), the message "NO SOURCE" is printed. If you press the RETURN key without entering a file and program name, prompting continues until you supply a filename and program name, or until you press the BREAK key.

The Z command not only enables source code listing but also permits the current source line to be printed at half-intensity each time the debugger is re-entered, or when the ? or $ commands are used.

If the program calls an external subroutine, the Z command can be used again to point to the source code of that subroutine. However, it does not have to be reinvoked each time execution transfers between the program and the subroutine.

Displaying Source Code (L, $, ?)

Once the source code has been identified with the Z command, you can use the L command to display the source.

L [*n* [–*m*] | *]

n	shows line *n* in the source code.
n–m	shows the specified range of lines in the source code.
*	shows all lines of the source code.

For example, to print out lines 59 through 61:

```
*L59-61
059       INPUT NAME:
060       IF NAME = "" THEN
061           GOTO EXIT

*
```

If the Z command is not used before the L command, the message "NO SOURCE" is printed.

In addition to L, the $ and ? commands print out the current line number and the corresponding source line.

The $ and ? commands are functionally identical. For example, the current line can be shown with:

```
*$<RETURN> CUST.ENTRY L 59    INPUT NAME:  OBJECT VERIFIES
```

In the example, "CUST.ENTRY" is the name of the program, "L 59" refers to line number 59, and "INPUT NAME:" is the statement on that line of code.

The source line ("INPUT NAME:") is printed at half-intensity. If the Z command is not used before the $ command, the source code is omitted.

Breakpoints and Tracing

A *breakpoint* is a condition that invokes the debugger whenever it is true. A *trace variable* is a variable that is defined to be printed automatically when a breakpoint is encountered. The Pick BASIC debugger can support up to four breakpoints and up to six trace variables at a time. Each external

subroutine to the program has its own breakpoint and trace table, independent of the one created for the program.

Establishing a Breakpoint (B)

Use the B command to define a breakpoint in a program.

B*var op value* [**&***var op value*]

var the variable name to be tested. Alternatively, *var* can be the symbol $, specifying that the line number should be tested.

op one of the following operators:

= equals

not equals

> greater than

< less than

value the value to test the variable by. Can be a numeric value, a string, or another variable in the program. If the value is a string, it must be enclosed in single or double quotes. A backslash is not accepted as a delimiter.

& logical connector for two conditions.

Although spaces have been supplied above for clarity, spaces are not accepted in the syntax for the B command. If the command is accepted, a plus sign (+) is printed. If the breakpoint table is already full with its maximum of four breakpoints, the message "TBL FULL" is printed.

For example, to enter the debugger whenever the variable COUNT is greater than 10 and the variable FOUND has a logical value of false (0):

```
*BCOUNT>10&FOUND=0
```

Line numbers can be tested as well as variables. To specify that a line number is being tested, use a dollar sign in place of a variable name. For example,

```
*B$>75&$<95
```

causes the program to re-enter the debugger whenever the program is executing a line between 75 and 95 (exclusively). Conditions comparing line numbers can be combined with conditions comparing variables. For example,

```
*B$>75&FOUND=0
```

After a breakpoint condition is established with B, the program can continue execution with the G command or with a linefeed (CTRL-J). When a breakpoint condition is encountered, the debugger is re-entered. The letter "B" with the breakpoint number and the line number are printed, along with any trace variables. Trace variables are discussed in a later section of this chapter.

Deleting a Breakpoint (K)

A breakpoint can be deleted from the breakpoint table with the K command.

K [*n*]

n delete breakpoint *n*. If *n* is omitted, delete all breakpoint conditions. *n* is determined by its position on the breakpoint table.

If the command is accepted, a minus sign (–) is printed.

Defining a Trace Variable (T)

The T command defines a trace variable. It also turns the trace table on and off.

T [*var*]

var trace the variable *var*. If *var* is omitted, toggle the trace table on or off.

If the trace variable is accepted, a plus sign (+) appears. If the trace table is already full with its maximum of six variables, the message "TBL FULL" is printed.

If the T command is used without any arguments, it toggles the trace table on and off. When T turns the trace table on, the word "ON" is printed; if it turns it off, the word "OFF" is printed. When the trace table is turned off, trace variables are not printed when a breakpoint is reached.

For example, to print the value of the variable COUNT every time a breakpoint is reached, enter:

```
*TCOUNT
```

Deleting a Trace Variable (U)

You can delete a variable from the trace table with the U command.

U [*var*]

var delete variable *var* from the trace table. If *var* is omitted, all variables are deleted.

If the command is accepted, a minus sign (–) is printed.

Displaying Breakpoints and Trace Variables (D)

You can display the breakpoint and trace tables with the D command. For example:

```
*D
T1 COUNT
T2 CUST.ARRAY(5)
T3
T4
T5
T6
B1 COUNT>10&FOUND=0
B2 $>75&FOUND=0
B3
B4
```

In the example, two trace variables and two breakpoints have been established.

Execution Control

Once you've entered the debugger, you often need to start the program executing again, to see what actually happens. You can "control" the program by establishing either breakpoints or execution steps before continuing execution, to specify that the debugger should be reinvoked when a condition becomes true or when a number of statements have been executed.

Continue Execution (G)

The G command continues program execution.

G [*line*]

line the line number to continue execution at. If omitted, the current line number is assumed.

Once the G command is used, the program continues execution until the next breakpoint or, if an execution step has been specified, until the specified number of lines have been executed. See the next section for more information on execution steps.

A linefeed (CTRL-J) is a synonym for the G command with no arguments—that is, it continues execution at the current line number. The linefeed has the advantage that it does not need to be followed by the RETURN key in order to be interpreted.

The G command should *not* be used if the operator has entered the debugger because of a fatal run-time error. Continuing execution after a fatal error may result in corrupted data structures. In such an instance, the operator should exit the debugger immediately with the END or the OFF command.

Setting an Execution Step (E)

The E command establishes or removes an execution step. An execution step is a number of lines to be executed before automatically reinvoking the debugger:

E [*lines*]

lines return control to the programmer every specified number of *lines*. If *lines* is omitted or 0, turn off the previous E command.

After enabling E, you can return to the program with the G command. When the specified number of lines is executed, the program returns to the debugger with the current line number preceded by "E". By using an execution step of 1, the program can be stepped through: you can examine every source line before it is executed, thus tracing the action as it occurs.

While an execution step is in effect with E, breakpoints are disabled. The program will not stop at breakpoints until E has been disabled again.

Execution steps are global; that is, if the program enters a subroutine, the step remains unchanged.

Ignoring Breakpoints (N)

The N command specifies that the next *n* breakpoints should be ignored.

N [*n*]

n bypass *n* breakpoints before returning control to the programmer. If *n* is omitted, bypass 1 breakpoint.

Although breakpoints are ignored by using the N command, the trace table is still printed at each bypassed breakpoint if it is enabled. By using the N command, you can monitor the sequence of execution and the values of trace variables at each breakpoint without re-entering the debugger. You can disable the trace table with the T command.

The N command is global; that is, if the program enters a subroutine, the N command remains in effect.

Printing Output

Toggling Program Output (P)

The P command toggles printing of output from the program. If P is toggled OFF, only output from the debugger is printed; the output from the program execution is not shown. When output is disabled with the P command, the word OFF is printed; when it is re-enabled, the word ON is printed. The default setting is ON.

Toggling Line Printing (LP)

The LP command toggles printing of all output on the line printer. If LP is toggled ON, output is directed to the printer. When line printing is enabled

with the LP command, the word "ON" is printed; when it is disabled again, the word "OFF" is printed. The default setting is "OFF."

The LP command is an equivalent to the PRINTER ON and PRINTER OFF statements in Pick BASIC.

Close the Printer (PC)

The PC command spools the debugger printer output to the printer. The PC command in the debugger is an equivalent of the PRINTER CLOSE statement in Pick BASIC. All debugger output held for the printer is sent to be printed immediately, rather than after the program is completed.

Return Stack

When the program transfers to an internal subroutine, the return address is pushed onto the return stack. The return address is the line the program transfers to at the end of the subroutine—that is, the line containing the statement immediately following the GOSUB that called the subroutine. If the subroutine calls another subroutine, the second address is pushed on top of the first address on the return stack, and so on for each embedded subroutine. When a RETURN statement is encountered, the top value on the stack is popped and taken as the return address for that subroutine.

Displaying the Return Stack (S)

The S command generates a list of line numbers on the return stack. The debugger may respond in one of the following ways:

= *line# source-line* The current subroutine will exit to line *line#*. When a RETURN statement is encountered, execution will transfer to line *line#*. (The corresponding source line is printed in half-intensity.)

= *line1# source-line*[= *line2# source-line...*]
Multiple subroutines are in effect. The current (top-level) subroutine will exit to line *line1#*. The next level subroutine will exit to line

	line2#, and so on. (Corresponding source lines are printed in half-intensity.)
STK EMP	Nothing in stack.
ILSTK	Illegal stack. This usually signifies that an external subroutine is being executed. (External subroutines are not monitored by the return stack.)

Popping the Return Stack (R)

The R command pops the return stack. The program returns from the current subroutine as if a RETURN statement had been encountered at that point.

CHAPTER 5

Statement and Function Reference

This chapter is the reference section for Pick BASIC. Each statement or function entry includes a description of its purpose, an explanation of its syntax, and at least one example of its use. All statements and functions listed in the *SMA/BASIC Language Specification* published in April, 1986, are included. Footnotes are used to indicate which statements and functions are not included in the SMA standards.

Readers who are unfamiliar with Pick BASIC should read Chapters 2 and 3 before referring to this chapter.

! : Enter a remark in the source code.*

The ! statement begins a comment line in the program.

! anything

anything any text can be placed after a ! statement.

The ! statement can be used to begin a comment line in the source code. It is functionally identical to the REM and * statements. Comment lines should be used to thoroughly document source code.

---

* Not included in the SMA standards.

!

Comment lines can be inserted into the object code with the $* statement. See the $* statement for more information.

Examples

A program might be documented as follows:

```
!
!       Get attribute definitions from DICT of inventory file
!
        OPEN 'DICT', 'INV' TO INV.DICT ELSE PRINT 'CANT OPEN
                                                "DICT INV"'; STOP
        READV DESC.AMC FROM INV.DICT,2 ELSE
           PRINT 'CANT READ "DESC" ATTR'; STOP
        END
        READV QOH.AMC FROM 'QOH',2 ELSE
           PRINT 'CANT READ "QOH" ATTR'; STOP
        END
!
!       Open data portion of inventory file
!
        OPEN '', 'INV' TO INV.FILE ELSE PRINT 'CANNOT OPEN
                                                    "INV"'; STOP
!
!       Prompt for part number
!
100     PRINT
        PRINT 'PART-NUMBER ':
        INPUT PN
        IF PN = '' THEN PRINT '--DONE--'; STOP
        READ INV.ITEM FROM INV.FILE,PN ELSE
           PRINT 'CANT FIND THAT PART'; GOTO 100
        END
        DESC = INV.ITEM<DESC.AMC>
        QOH = INV.ITEM<QOH.AMC>
!
!       Print description and quantity-on-hand
!
        PRINT 'DESCRIPTION -': DESC
        PRINT 'QTY-ON-HAND -': QOH
        PRINT
        GOTO 100
END
```

$* : Place a comment into the object code.*

The $* statement allows a comment to be embedded directly into the program's object code at compilation.

$* *text*

text the comment text, enclosed within string delimiters (single quotes, double quotes, or backslashes).

The $* statement directs the compiler to write the quoted text directly into the object code of the program. These comments are generally used to place copyright information or version numbers into object code before it is distributed.

Examples

To place the string "HELLO" directly into the object code of a program, the code might read:

```
$* "HELLO"
```

In the following application, the $* statement is used to place a copyright into an adventure game:

```
$* " DRAGONS -- VERSION 1.1 "
$* " COPYRIGHT 1990 N. WEST, NBM INC. "
EQUATE TRUE TO 1, FALSE TO 0,
   BEL TO CHAR(7)
   .
   .
   .
```

* Not included in the SMA standards.

$CHAIN: Transfer to another file item for source code.*

The $CHAIN statement allows source code to be read in from another file item.

$CHAIN [*filename*] *item-ID*

filename the name of the file containing the item. If *filename* is omitted, the current file is assumed.

item-ID the item ID of the item containing the source code

The $CHAIN statement directs the compiler to read source code from the specified item and compile it as if it were a part of the current item. The $CHAIN statement differs from the INCLUDE, $INCLUDE, and $INSERT statements in that the compiler does *not* return to the current item after compiling the source code in the remote item; any statements appearing after the $CHAIN statement are ignored.

The $CHAIN statement was originally designed for programs with source code larger than 32K. By using $CHAIN statements, a very long source program can be broken up and stored in several smaller items which are $CHAINed together, as long as the compiled object code does not exceed the maximum item size supported by your system.

Examples

To transfer compilation to source code in item ID PROG2 in file BP, the code might read:

```
$CHAIN BP PROG2
```

In the following application, a very long program is broken up into three different source file items, each one but the last ending with a $CHAIN statement that calls the next.

The source file item CUST.ENTRY contains the following:

```
EQUATE TRUE TO 1, FALSE TO 0,
   AM TO CHAR (254), VM TO CHAR (253),
   .
   .
   .
```

* Not included in the SMA standards.

```
PRINT "PRINT REPORTS?"
$CHAIN CUST.ENTRY2
```

At the end of CUST.ENTRY, source file item CUST.ENTRY2 is called with a $CHAIN statement. CUST.ENTRY2 contains:

```
INPUT ANSWER
   .
   .
   .
FOUND = FALSE
FOR I = 1 TO NO.OF.ELTS UNTIL FOUND
   IF CUST.ARRAY (I) < 1, 3 > = NAME THEN
$CHAIN CUST.ENTRY3
```

CUST.ENTRY2, in turn, calls CUST.ENTRY3, which reads:

```
      FOUND = TRUE
   END
NEXT I
   .
   .
   .
```

Note in the example that a FOR loop and an IF construct are broken over two file items.

$INCLUDE: Read in source code from another file item.*

The $INCLUDE statement allows source code to be read in from another file item.

$INCLUDE [*filename*] *item-ID*

filename the name of the file containing the item. If *filename* is omitted, the current file is assumed.

item-ID the item ID of the item containing the source code.

The $INCLUDE and $INSERT statements both direct the compiler to read in source code from the specified file item and compile it as if it were written in the current item. The $INSERT statement is identical to the $INCLUDE statement.

* The SMA standards form of this command is INCLUDE.

The $INCLUDE statement differs from the $CHAIN statement in that the compiler returns to the main item and continues compiling with the statement following the $INCLUDE.

The $INCLUDE statement is particularly useful for reading in items containing COMMON and EQUATE statements, or any statements which a programmer might want to be consistent among several different programs. Be careful, however, of naming conflicts among different file items.

$INCLUDE statements can be nested; that is, a program can $INCLUDE a file item which $INCLUDEs another file item. However, the total object code when compiled cannot exceed the maximum item size supported by your system.

If the source code read in through an $INCLUDE statement generates a run-time error message, the error message will display only the number of the line which contains the $INCLUDE statement. The line numbers from the external file item are not kept in the object code.

Examples

To read in the source code written in item ID SETUP in file BP, the code might read:

```
$INCLUDE BP SETUP
```

In the following application, the $INCLUDE statement is used at the beginning of a program to read in common variables, equated constants, and the part of the program which opens the file.

```
$INCLUDE STARTUP
PRINT "ENTER THE CUSTOMER ID : ":
INPUT ID
MATREAD PHONE.ARRAY FROM CUSTFILE, ID ELSE
   PRINT "CANNOT READ RECORD!"
   STOP
END
```

The file item STARTUP contains the text:

```
COMMON PHONE.ARRAY(10), PHONEREC
EQUATE TRUE TO 1, FALSE TO 0, AM TO CHAR(254)
PROMPT " "
OPEN "CUSTOMERS" TO CUSTFILE ELSE
   ABORT 201, "CUSTOMERS"
END
```

$INSERT: Read in source code from another file item.*

The $INSERT statement allows source code to be read in from another file item.

$INSERT [*filename*] *item-ID*

filename the name of the file containing the item. If *filename* is omitted, the current file is assumed.

item-ID the item ID of the item containing the source code.

The $INSERT and $INCLUDE statements both direct the compiler to read in source code from the specified file item and compile it as if it were written directly in the current item. The $INCLUDE statement is identical to the $INSERT statement.

The $INSERT statement differs from the $CHAIN statement in that the compiler returns to the main item and continues compiling with the statement following the $INSERT.

The $INSERT statement is particularly useful for reading in items containing COMMON and EQUATE statements, or any statements which a programmer might want to be consistent among several different programs. Be careful, however, of naming conflicts among different file items.

$INSERT statements can be nested; that is, a program can $INSERT a file item which $INSERTs another file item. However, the total object code when compiled cannot exceed the maximum item size supported by your system.

If the source code read in through an $INSERT statement generates a run-time error message, the error message will display only the number of the line which contains the $INSERT statement. The line numbers from the external file item are not kept in the object code.

Examples

To read in the source code written in the item SETUP in file BP, the code might read:

```
$INSERT BP SETUP
```

---

* Not included in the SMA standards.

In the following application, the $INSERT statement is used at the beginning of a program to read in common variables, equated constants, and the part of the program which opens the file.

```
$INSERT STARTUP
PRINT "ENTER THE CUSTOMER ID:":
INPUT ID
MATREAD PHONE.ARRAY FROM CUSTFILE, ID ELSE
   PRINT "CANNOT READ RECORD!"
   STOP
END
```

The file item STARTUP contains the text:

```
COMMON PHONE.ARRAY(10), PHONEREC
EQUATE TRUE TO 1, FALSE TO 0, AM TO CHAR(254)
PROMPT " "
OPEN "CUSTOMERS" TO CUSTFILE ELSE
   ABORT 201, "CUSTOMERS"
END
```

* : Enter a remark in the source code.*

The * statement begins a comment line in the program.

** anything*

anything any text can be placed after a * statement.

The * statement can be used to begin a comment line in the source code. It is functionally identical to the REM and ! statements. Comment lines should be used to thoroughly document source code.

Comment lines can be inserted into the object code with the $* statement. See the $* statement for more information.

---

* Not included in the SMA standards.

Examples

A program might be documented as follows:

```
*********************************************
******* THIS IS A PROGRAM FOR ...
*********************************************

***** MAIN*********************************
LOOP UNTIL ...
   GOSUB ROUTINE
   .
   .
   .
REPEAT
STOP

****SUBROUTINES**********************
ROUTINE:
```

= : Assign a value to a variable.

The assignment statement assigns a value to a variable.

var = *expr*

var	the variable whose value is to be assigned.
expr	an expression evaluating to the value assigned to *var*.

The direct assignment statement (=) assigns the value of *expr* to the variable *var*. Some Pick implementations also support some or all of the following special forms of the assignment statement. These have the general form *var op= expr*, which is equivalent to *var* = *var op expr*. They are not included in the SMA standards.

var = *expr*	*var* takes the current value of *expr*.
var += *expr*	*var* becomes *var* plus the current value of *expr*.
var –= *expr*	*var* becomes *var* minus the current value of *expr*.
var *= *expr*	*var* becomes *var* multiplied by the current value of *expr*.
var /= *expr*	*var* becomes *var* divided by the current value of *expr*.

=

var := *expr* — *var* becomes *var* concatenated with the current value of *expr*.

Examples

If the SURNAME were to be appended to the variable NAME, the code might read:

```
NAME : = SURNAME
```

In the following application, assignment statements are used to assign to the variable PROFIT the value of COST minus PRICE, and then to subtract from PROFIT the value of the overhead allotted to that sale. An external subroutine CALC.OVERHEAD assigns the value OVERHEAD based on the value of PROFIT.

```
PRINT "ENTER COST OF ITEM: "
INPUT COST
PRINT "ENTER PRICE AT WHICH ITEM WAS SOLD: "
INPUT PRICE
PROFIT = COST – PRICE
CALL CALC.OVERHEAD (PROFIT,OVERHEAD)
PROFIT –= OVERHEAD
PRINT "WITH OVERHEAD TAKEN INTO ACCOUNT, THE PROFIT IS:"
PRINT PROFIT
```

@( ): Screen Control Function.

The @ function generates the screen format control sequences for a terminal.

@(*col*,*row*)
@(–*code*)

col — an expression to be taken as the column (*x*-coordinate) of the position desired.

row — an expression to be taken as the row (*y*-coordinate) of the position desired.

–code — an expression to be taken as a numeric code signifying a specific effect, such as clearing the screen. Codes are listed below.

All terminals have built-in internal command sequences which move the cursor to a particular position, clear the screen, place text in reverse video, etc. The @ function can be used to return the proper command sequence for performing many terminal control functions. When the @ function is used in a statement that produces terminal output (PRINT, CRT, DISPLAY, or INPUT), the command sequence is sent to the terminal and performs the appropriate action.

There are two forms of the @ function.

- The first form, @(*col*,*row*), returns the command sequence for moving to a specified column and row. Although both expressions should be within the ranges of your particular display screen (usually either 79 or 131 columns by 23 rows, with 0,0 as the upper left corner of the screen), this is not enforced. If *row* is omitted, the current row is assumed.
- The other form of the @ function, @(–*code*), uses special codes, each preceded by a minus sign. Table 5-1 lists some of the codes available on most Pick implementations. Only the first four codes are included in the SMA standards.

Table 5-1. Commonly Used @ Function Codes.

Code	Description
@(–1)	Clear screen and position cursor at "home" (upper left corner).
@(–2)	Position cursor at "home" (upper left corner). Same as @(0,0).
@(–3)	Clear from cursor position to end of the screen.
@(–4)	Clear from cursor position to end of current line.
@(–5)	Begin blinking field.
@(–6)	End blinking field.
@(–7)	Begin protected field. Data in this field cannot be overwritten.
@(–8)	End protected field.
@(–9)	Backspace one character.
@(–10)	Move cursor up one line.

Other codes may vary from system to system, although the codes listed in Table 5-2 are found on many systems.

Table 5-2. Some Additional @ Function Codes.

Code	Description
@(–11)	Begin protected field.
@(–12)	End protected field.
@(–13)	Begin reverse video mode.
@(–14)	End reverse video mode.
@(–15)	Begin underline field.
@(–16)	End underline field.

The @ function generates the command string for the terminal being used at run time, according to the current terminal type defined by the TERM verb. The most important thing to grasp about the @ function is that all it does is generate a string of control characters which trigger a unique response when they are sent to the screen.

The @ sign is used with the INPUT statement to prompt for input at a specified position on the screen. The INPUT @ statement, however, does not provide for any of the screen control codes listed in Tables 5-1 and 5-2, only for moving the cursor. If the formatting properties of the INPUT @ statement are not taken advantage of, the same effect might be achieved by preceding a standard INPUT statement with a PRINT statement which uses the @ function directly.

The CALL and ENTER statements also recognize the @ sign in their syntax lines, to signify that the name of the program to be called or entered is kept in a variable. This use of the @ symbol, however, should not be confused with its use in the @ function.

Examples

To clear the screen for a program, the code would read:

```
PRINT @(–1)
```

To print the words "QUIT?" at the bottom of the screen, the code would read:

```
PRINT @(0,22) : "QUIT?"
```

In the following application the @ function is used in a PRINT statement to clear the screen and prompt the user at position (30,10) on the screen. If the user supplies an invalid answer, an error message appears on the bottom

of the screen in blinking mode, and the previous invalid answer is erased. The operator is then prompted again until a valid answer is provided.

```
PRINT @(–1) :
CLEAR.ANSWER = @(30,10) : @(–4)
PRINT @(10,10) : " ENTER A NUMBER: " : @(30,10):
LOOP
   INPUT NUMBER :
   IF NUM(NUMBER) THEN
      VALID = 1
      GOSUB COMPUTE
   END ELSE
      VALID = 0
      PRINT @(0,22) : @(–5) : " NON-NUMERIC INPUT.":
      PRINT "ENTER A NUMBER" : @(–6) : @(–4):
      PRINT CLEAR.ANSWER :
   END
UNTIL VALID DO REPEAT
```

The following techniques are used in the application:

- All PRINT and INPUT statements are followed by a colon to suppress the automatic carriage return and line feed. This provides more control over the screen: in auto-scroll mode, a line feed on line 23 causes all lines above the cursor position to scroll up one line.
- The sequence to clear the previous answer is placed in a variable CLEAR.ANSWER, which is later sent to the screen with the PRINT statement.

[ ]= : Assign a substring.*

The substring assignment statement replaces a part of a string.

string **[***start,length* **]** = *expr*

string	the string variable to be changed.
[...]	the square brackets are part of the syntax and must be typed.
start	an expression evaluating to the starting character position.

* Not included in the SMA standards.

length an expression evaluating to the ending character position.

expr an expression evaluating to the replacement string.

The substring assignment statement allows any part of a string to have another value assigned to it. The behavior of the substring assignment is dependent on the values of *start* and *length*. The rules are as follows:

start >= 0 If *start* is nonnegative, it is taken as the starting character position of the string from left to right. If *start* evaluates to 0, the starting character position is 1. For example, if STRING is "HI THERE", then

```
STRING [2,2 ] = " OW "
```

produces "HOWTHERE".

If *start* evaluates to a number greater than the length of the string, then the replacement string is appended at the end.

start < 0 If *start* is negative, it is taken as the starting character position from right to left. For example, if STRING is "HI THERE", then

```
STRING [–2,2 ] = " OW "
```

produces "HI THEOW".

If the absolute value of *start* is greater than the length of the string, its behavior is the same as if *start* were 0.

length >= 0 If *length* is nonnegative, it is taken as the length of the string to be replaced. Note that this length need not correspond with the length of the replacement string.

If *length* evaluates to zero, then the replacement string should be inserted without replacing any characters in the string. (If *start* is negative, it is inserted to the left of that position, otherwise it is inserted to the right.)

length < 0 If *length* is negative, it is taken as the ending character position of the string portion to be replaced, counting from right to left. For example, if STRING is "HI THERE", then:

```
STRING [2, –2 ] = "OW"
```

produces "HOWE".

Similarly,

```
STRING [–2, –2 ] = "OW "
```

produces "HI THERWE".

If the positions specified by *start* and *length* overlap, characters are repeated in the resulting string. For example, if STRING is "HI THERE", then:

```
STRING[7,–7] = "OW"
```

produces:

```
"HI THEOW THERE"
```

Similarly,

```
STRING[7,–7] = "OW"
```

also produces:

```
"HI THEOW THERE"
```

With the statement:

```
DIGITS[ n , m ] = "XX"
```

and the variable DIGITS containing "0123456789", the behavior of the substring assignment statement can be summarized by the following table:

	$n > 0$	$n = 0$	$n < 0$
$m > 0$	Starting at position *n*, replace the next *m* characters. DIGITS[3,4]="XX" results in: DIGITS="12XX7890"	Same as $n = 1$: replace the first *n* characters. DIGITS[0,4]="XX" results in: DIGITS="XX567890"	Starting at the *n*th position from the end of the string, replace the next *m* characters. DIGITS[–6,4]="XX" results in: DIGITS="1234XX90"
$m = 0$	Insert the replacement string at position *n*, with no characters deleted. DIGITS[3,0]="XX" results in: DIGITS="12XX34567890"	Same as $n = 1$: insert the replacement string at the beginning of the original string, with no characters deleted. DIGITS[0,0]="XX" results in: DIGITS="XX1234567890"	Starting at the *n*th position from the end of the string, insert the replacement string with no characters deleted. DIGITS[–6,0]="XX" results in: DIGITS="12345XX67890"

[ ]=

	$n > 0$	$n = 0$	$n < 0$
$m < 0$	Replace all characters from the nth position from the beginning of the string up to the nth position from the end of the string. DIGITS[3,–4]="XX" results in: DIGITS="12XX890"	Same as n =1: replace all characters from the first position up to the mth position from the end of the string. DIGITS[0,–4]="XX" results in: DIGITS="XX890"	Starting at the nth position from the end of the string, replace all characters up to the mth position from the end of the string. DIGITS[–6,4]="XX" results in: DIGITS="1234XX890"

Example

In the following application, a full name in the string variable NAME is reduced to the first initial and last name. (EQUATE statements are used to differentiate a blank space from the null string for readability.)

```
EQUATE BLANK TO " ", NIL TO ""
NO. OF WORDS = DCOUNT (NAME, BLANK )
SURNAME = FIELD (NAME, BLANK, NO. OF WORDS)
POSITION = COL1( ) –1
NAME [2, POSITION] = BLANK
```

ABORT: Abort a program and return to TCL.

The ABORT statement terminates the current program and returns the user to the TCL prompt, regardless of the environment in which the program was executed.

ABORT [*errmsg* [,*parameter1*, *parameter2*, ...]]

errmsg an integer corresponding to an error message from the system message file (ERRMSG). The message will be output upon termination of the program.

parameter1, *parameter2*, ...
parameters to be passed to the error message.

The ABORT statement differs from the STOP statement in that a STOP statement returns control to the calling environment (often a proc), whereas ABORT terminates the calling environment (including procs and programs within the current EXECUTE level) as well as the Pick BASIC program.

In general, the ABORT statement is used for abnormal terminations of a program, and the STOP statement is used for normal terminations.

Example

The following example demonstrates how the ABORT statement can be used to terminate a program on failure to open a file.

```
OPEN 'CUSTOMERS' TO CUSTFILE ELSE
   ABORT 201, 'CUSTOMERS'
END
   .
   .
   .
```

If the file CUSTOMERS is not found, the user receives the message:

```
[201] 'CUSTOMERS' IS NOT A FILE NAME
```

and returns directly to the TCL prompt.

ABS( ): Return the absolute value of an expression.

The ABS function returns the absolute value of the given expression.

ABS(*expr* **)**

expr an expression evaluating to a numeric value.

The absolute value of any expression is defined as its *positive* value, that is, the difference between itself and 0. The absolute value of any positive expression is itself, and the absolute value of a negative expression is calculated by reversing the sign—that is, the absolute value of –6 is 6.

Examples

If the variable NUMBER contains "–1.732", then

```
ABS(NUMBER)
```

returns "1.732".

In the following application the ABS function is used to discover an error in bookkeeping. Note that in printing the discrepancy a format expression ("2,$") is used to make the data more readable.

```
DIFF = ABS(PRICE – COST – PROFIT)
IF DIFF THEN
   PRINT "ERROR OF  " : DIFF " 2,$ " : " .  PLEASE CHECK. "
END
```

If the PRICE has been established as 9.99, the COST is 4.25, and the estimated PROFIT has been set at 5.75, the resulting output will be:

```
ERROR OF  $0.01.  PLEASE CHECK.
```

ALPHA( ): Test for an all-alphabetic string.*

The logical function ALPHA evaluates an expression to determine if it is a string containing only alphabetic characters.

ALPHA(*expr* **)**

expr the string expression to be tested.

The ALPHA function determines whether the expression is an alphabetic or nonalphabetic string. If the expression contains the characters "A" through "Z" or "a" through "z" (ASCII 65 – 90, 97 – 122), it evaluates to true and a value of 1 is returned. If the expression contains any other character (such as numeric or special characters), it evaluates to false and a value of 0 is returned.

Examples

If the variable NAME contains "HENRY FRENKL", then:

```
ALPHA(NAME)
```

returns "1". However, if NAME contains "HENRY FRENKL 4th", the ALPHA function returns 0.

In the following application, airline reservations require the traveler's starting point and final destination. The travel agent must enter these with the three-letter code assigned to airports.

```
PROMPT " "
PRINT @(–1)
PRINT @(2,2) : "ENTER FLIGHT DATE: ":
INPUT @(30,2) DATE "D"
```

* Not included in the SMA standards.

```
PRINT @(2,4) : "ENTER FLIGHT NUMBER: ":
INPUT @(30,4) FLTNO,3 "0"
PRINT @(2,6) : "STARTING POINT: ":
VALID = 0
LOOP
   INPUT @(30,6) START,3
   IF NOT ( ALPHA( START )) THEN
      INPUTERR "PLEASE ENTER 3-LETTER AIRPORT CODE."
   END ELSE
      VALID = 1
   END
UNTIL VALID DO REPEAT
PRINT @(2,8) : "FINAL DESTINATION: ":
VALID = 0
LOOP
   INPUT @(30,8) DEST,3
   IF NOT( ALPHA( DEST )) THEN
      INPUTERR "PLEASE ENTER 3-LETTER AIRPORT CODE."
   END ELSE
      VALID = 1
   END
UNTIL VALID DO REPEAT
```

ASCII( ): Convert a string from EBCDIC to ASCII code.

The ASCII function converts a string in EBCDIC code into ASCII code.

ASCII(*expr***)**

expr an expression evaluating to the string to be converted.

The ASCII function converts each character of the given expression from its EBCDIC representation value to its ASCII representation value. It is the inverse to the EBCDIC function.

The ASCII function does not convert a character to its numeric ASCII value, or vice versa. Use the SEQ and CHAR functions for this.

Appendix C provides a full listing of ASCII codes.

Example

In the following application, data is read from a tape which was written in EBCDIC code. The ASCII function converts it into ASCII code.

```
READT STRING ELSE
   .
   .
   .
END
STRING = ASCII(STRING)
```

BREAK: Control access to the debugger.

The BREAK statement allows the Break Inhibit Counter of a program to be incremented or decremented, thus controlling access to the debugger.

BREAK [KEY] ON | OFF | *expr*

ON	decrement the Break Inhibit Counter by 1.
OFF	increment the Break Inhibit Counter by 1.
expr	an expression evaluating to true or false. If true, the BREAK key is enabled; if false, the BREAK key is disabled.

While a program is being executed, pressing the BREAK key will normally transfer control from the program to the debugger. The BREAK statement gives the programmer control over this feature.

The BREAK statement does not directly toggle the BREAK feature on and off, but increments and decrements the Break Inhibit Counter. The counter is usually set to 0, meaning that the BREAK feature is enabled. Each BREAK OFF statement increments the counter by 1, and each BREAK ON statement decrements the counter by 1. When the counter is set to any number other than 0, the BREAK feature is off. Thus if two BREAK OFF statements have been used in a program, the counter is set to 2, and two BREAK ON statements will be necessary to return the counter to 0 and re-enable the BREAK key.

Using a counter instead of directly turning the BREAK key on and off simplifies situations where a program calls another program or external subroutine. By using a counter, it is ensured that the status of the BREAK key in the calling program is maintained. This, of course, is dependent on

each BREAK OFF statement being paired with a BREAK ON statement before the end of the program or subroutine.

If the BREAK key is enabled (ON) and the BREAK key is pressed during the execution of a SLEEP or RQM statement, the debugger is entered. When "G" is pressed at the debugger prompt, the program resumes at the next statement after the SLEEP statement. If the BREAK key is disabled (OFF) and BREAK is pressed, it terminates the sleep without entering the debugger, and the program resumes at the next statement after SLEEP or RQM.

The debugger can also be entered through a run-time error or by encountering a DEBUG statement. See Chapter 4 for more information on the Pick BASIC debugger.

Examples

To turn the break feature off for a program, the code would read:

```
BREAK OFF
```

At the end of the program, the break feature can be reinstated with:

```
BREAK ON
```

In the following application, a quiz program gives 60 seconds for the user to answer a question. To answer the question before the time is up, the user is allowed to press the BREAK key. A BREAK OFF statement is used to turn off the debugging feature of the BREAK key during the sleep, so that pressing the BREAK key will interrupt the sleep but not enter the debugger. The INPUTIF statement is used after the sleep to examine the type-ahead and determine if a response was entered.

```
ITEM = RND(99)
READ QUESTION FROM QUESTFILE,ITEM ELSE
   PRINT "ERROR IN READING " : ITEM
   STOP
END
PRINT @(0,23) : "ENTER ANSWER, PRESS RETURN AND BREAK." :
PRINT QUESTION <1> :
BREAK OFF
SLEEP 60
BREAK ON
INPUTIF ANSWER THEN
   GOSUB EVAL
END ELSE
   PRINT @(-1) : "NOT ANSWERED IN TIME. -3 POINTS."
   POINTS - = 3
END
```

CALL: Call an external subroutine.

The CALL statement transfers control from a main program to a cataloged external subroutine.

CALL *name* [(*expr1*, *expr2*, *expr3*, ...)]
CALL *@var* [(*expr1*, *expr2*, *expr3*, ...)]

name	is the name of the subroutine to be called.
expr...	values to be passed to the cataloged subroutine. If one of the values is an array variable, it must be preceded by the MAT keyword and should be dimensioned in both the main program and the subroutine.
@var	*var* is a variable which has been assigned the item ID of the cataloged subroutine to be entered.

The CALL statement can be used to enter a cataloged external subroutine. An external subroutine is a subroutine that is compiled and cataloged separately from the programs that call it. When the final RETURN statement of the subroutine is encountered, program control is returned to the original program at the statement following the CALL statement.

The subroutine to which the CALL statement branches must be cataloged, and the first line of the subroutine must contain the SUBROUTINE statement.* Control will be returned to the main program when a RETURN is encountered in the subroutine which does not correspond to a previous GOSUB within the same external subroutine. If there is no RETURN statement, control will not return to the main program.

Each of the parameters listed in the CALL syntax line is passed into the corresponding variable list on the SUBROUTINE syntax line. Other than their positions on the CALL and SUBROUTINE syntax lines, there is no correspondence between variable names in the calling program and subroutine.

An alternative way of passing variables between programs and subroutines is by using COMMON statements in both program and subroutine. See the COMMON statement for more information.

---

* Some implementations allow comment lines before the SUBROUTINE statement.

Passing Arrays

When arrays are being passed from the main program to a subroutine, the array name must be preceded by the MAT keyword and there must be a one-to-one correspondence to the elements being passed. For example, to pass the 3 by 4 matrix MATRIX, enter:

```
CALL SUBR( MAT MATRIX )
```

The MATRIX array must be previously dimensioned in the program with the DIM statement.

In the subroutine SUBR, the corresponding dimensioned array must also be dimensioned. Note, however, that the corresponding arrays do not need to have the same dimensions, as long as they have the same number of elements.* The first two lines of the subroutine SUBR might read:

```
SUBROUTINE SUBR( MAT ARRAY1)
DIM ARRAY1(6,2)
```

If the 3 by 4 matrix MATRIX in the main program contains:

1	2	3	4
RED	BLUE	GREEN	YELLOW
A	B	C	D

then when it is passed to ARRAY1, the 6 by 2 matrix will contain:

1	2
3	4
RED	BLUE
GREEN	YELLOW
A	B
C	D

* This is not true on all systems. On Prime INFORMATION, for example, the dimensions of an array cannot be changed between the calling program and a subroutine.

Example

To call the subroutine ADDTHEM, passing variables A, B, and C, the calling line in the main program would read:

```
CALL ADDTHEM(A, B, C )
```

The first line of the source code for ADDTHEM might then read:

```
SUBROUTINE ADDTHEM(X, Y, Z )
```

Variable A is passed to variable X, B is passed to Y, and C is passed to Z. When the subroutine has finished, these values are passed in the opposite direction.

CASE: Perform conditional execution.

The CASE construct performs a conditional selection of a sequence of statements.

```
BEGIN CASE
      CASE expr
           statements
      CASE expr
           statements
                .
                .
                .
END CASE
```

expr	an expression to be evaluated for its logical value.
statement	statements to be executed if the previous *expr* had been tested to be logically true.

The CASE construct evaluates a series of conditions until one is true and executes a set of statements accordingly. The expressions in the CASE statements are evaluated sequentially for their logical value until a value of true is encountered. When an expression evaluates to true, the statements between the CASE statement and the next CASE statement are executed, and all subsequent CASE statements are skipped. Execution continues with the next statement following the END CASE statement.

If none of the expressions evaluate to true, no action is performed, and program execution continues with the statement after the END CASE statement.

The CASE statement can usually replace multiple nested IF constructs: it is much more readable and easier to use.

Example

To test a variable NUMBER for positive or negative value, the source code might read:

```
BEGIN CASE
CASE NUMBER > 0
   PRINT "POSITIVE"
CASE NUMBER < 0
   PRINT "NEGATIVE"
CASE 1
   PRINT "ZERO"
END CASE
```

Note that the third and last condition reads "CASE 1" instead of "CASE NUMBER = 0". In this situation the two conditions are equivalent since the last condition would only be tested if the first two failed. "CASE 1" is often used as the last condition of a CASE statement, as a catch-all condition.

CHAIN: End program and execute a TCL command.

The CHAIN statement terminates execution of a program and executes a TCL command.

CHAIN *command-expr*

command-expr any command to be passed to TCL.

Like the EXECUTE statement, the CHAIN statement executes a TCL command. The CHAIN statement differs from the EXECUTE statement, however, in that it does not support any of EXECUTE's features (such as capturing output or error messages), and it does not return to the program, but returns directly to the environment which called the program.

If the CHAIN statement is used to execute another program, parameters cannot be directly passed to the second program. However, if the I option

(which suppresses initialization of all values) is used with the RUN verb, the COMMON area can be used to pass parameters from one program to the next. See the COMMON statement for more information.

☞ SMA recommends that the I option not be used with the RUN verb. The I option is available to support old code (pre-EXECUTE statement). Any corruption of the workspace area will cause problems if you try to use the I option.

The data stack can be used to supply input which the TCL command might request. See the DATA statement for more information.

Examples

To end a program by running another program, CONCLUDE, the code might read:

```
CHAIN "RUN BP CONCLUDE"
```

CHAR( ): Return the ASCII character of a decimal value.

The CHAR function returns the character with the given ASCII decimal code.

CHAR(*expr* **)**

expr an expression evaluating to a numeric value.

The CHAR function converts a decimal value to its corresponding ASCII character. It is particularly useful for accessing characters like attribute marks (CHAR(254)) and error bells (CHAR(7)). The CHAR function is commonly used in EQUATE statements, as demonstrated in the example below.

The SEQ function is the inverse of the CHAR function, producing the ASCII value of a given character. See the SEQ function for more information.

See Appendix C for ASCII character codes.

Example

To send an error bell to the screen, the code might read:

```
CRT CHAR(7)
```

CLEAR: Initialize all variables to zero.

The CLEAR statement assigns a value of 0 to all variables throughout the program.

CLEAR

The CLEAR statement is generally used at the beginning of a program to set previously assigned and unassigned values of all variables to zero.* This procedure avoids run-time errors for unassigned variables. If you use the CLEAR statement later in the program, any values that have already been assigned to variables (including array variables) are lost.

The CLEAR statement cannot be used to initialize only selected variables. If it is used, *all* variables in the program are initialized to 0.

☞ The CLEAR statement is often used to prevent the run-time warning message which normally ensues when an unassigned variable is used. This practice, however, is not always desirable, since the unassigned variable message can be useful for detecting programming errors (such as misspelled variable names).

Prime INFORMATION and uniVerse have a CLEAR COMMON statement that assigns a value of zero to all variables previously named in the common area.

Example

In the following application the CLEAR statement is used at the beginning of the program to initialize variables. Thus, when the previously unused variable STOPNOW is used as the loop control, no error message ensues.

---

* On some Pick systems, the values of all variables are set to null rather than zero.

```
CLEAR
   .
   .
   .
LOOP UNTIL STOPNOW DO
   .
   .
   .
   PRINT "DO YOU WANT TO STOP (Y OR N)" :
   INPUT ANSWER,1
   IF ANSWER = "Y"  THEN STOPNOW = 1
REPEAT
```

CLEARFILE: Clear the data from a file.

The CLEARFILE statement deletes all items in a previously opened file. This statement does not delete the file itself, but it clears out the data from it completely.

CLEARFILE [*filevar*]

filevar the file variable to which the file had been opened. If *filevar* is not specified, the default file variable is used, which is the last file opened without an assigned file variable.

☞ Be cautious when using CLEARFILE. If the dictionary of a file is open to *filevar*, on some systems CLEARFILE deletes the entire contents of the dictionary, including the data file D-pointer!

Examples

To clear the data from a file opened to DATAFILE, the code might read:

```
CLEARFILE DATAFILE
```

In the following application, the file TRANS.LOGS contains logs for all transactions during the week. The program fragment shown clears all logs at the request of the operator.

```
OPEN " TRANS.LOGS " TO LOGFILE ELSE
   ABORT 201, " PRINTLOG "
END
PRINT " EMPTY ALL TRANSACTION LOGS (Y OR N) " :
```

```
INPUT ANSWER
IF ANSWER = " Y " THEN
   CLEARFILE LOGFILE
   PRINT " ALL TRANSACTION LOGS EMPTIED. "
END ELSE
   PRINT " TRANSACTION LOGS UNTOUCHED. "
END
```

COL1( ): Return preceding column position.

The COL1 function returns the column position of the character immediately preceding the substring returned by the most recent FIELD function.

```
COL1( )
```

After the execution of a FIELD function, the COL1 function returns the column position immediately preceding the selected substring. Although the COL1 function takes no arguments, the parentheses are required to identify it as a function.

If no FIELD function precedes the COL1 function, a value of zero is returned. If the delimiter expression of the FIELD function is null or if the string is not found, the COL1 function returns a zero value.

The COL1, COL2, and FIELD functions can be used to perform array processing for strings with delimiters other than the attribute mark (CHAR(254)), value mark (CHAR(253)), and subvalue mark (CHAR(252)).

Examples

To determine the column position before the third word in a string STRING, the code might read:

```
WORD = FIELD(STRING, " ", 3)
POS = COL1( )
```

If STRING contains "IT WAS TWENTY YEARS AGO TODAY", WORD will contain "TWENTY" and POS will contain "7". With this information, the string can be cut off after the second word with:

```
STRING = STRING[ 1, POS ]
```

and STRING will contain, " IT WAS ".

COL1( )

In the following application, the NAMES string contains a list of names separated by commas. To replace a name, the COL1 function is necessary to determine where the replacement should start.

```
CURR.NAME = FIELD( NAMES, ",", NAME.NBR)
IF NEW.NAME # "" THEN
   NAMES = NAMES [1,COL1()] : NEW.NAME : NAMES [COL2 (),9999]
END
```

COL2( ): Return following column position.

The COL2 function returns the column position of the character immediately after the substring returned by the most recent FIELD function.

COL2( )

After the execution of a FIELD function, the COL2 function returns the column position immediately following the selected substring. Although the COL2 function takes no arguments, the parentheses are required to identify it as a function.

If no FIELD function precedes the COL2 function, a value of zero is returned. If the delimiter expression of the FIELD function is null or if the string is not found, the COL2 function returns a zero value.

The COL2, COL1, and FIELD functions can be used to perform array processing for strings with delimiters other than the attribute mark (CHAR(254)), value mark (CHAR(253)), and subvalue mark (CHAR(252)).

Example

To determine the column position after the third word in a string STRING, the code might read:

```
WORD = FIELD(STRING, " ", 3)
POS = COL2( )
```

If STRING contains "IT WAS TWENTY YEARS AGO TODAY", WORD will contain "TWENTY" and POS will contain "14". With this information the string can be cut off after the third word with:

```
STRING = STRING[ 1, POS–1 ]
```

and STRING will contain, "IT WAS TWENTY".

COMMON: Assign space allocation sequence for variables.

The COMMON statement is used to determine the sequence in which specified variables are allocated space. It allows programs and subroutines to access the same variables.

COM[MON] *var1* [,*var2* , ...]

var... the names of the variables to be shared. *var* can be a simple variable, a file variable, or an array variable.

The COMMON statement provides a storage area for the specified variables which is accessible by other programs and by external subroutines. The variables can be defined using different names in separate programs and subroutines, but they must be defined in the same exact order. The COMMON statement must precede any use of the variables that it names during compilation.

By using a COMMON statement, the sequence in which they are allocated space is explicitly set, and other programs using the same COMMON area can access the same variables by position. By using COMMON, variables do not have to be supplied in CALL and SUBROUTINE statements. COMMON can also be used for programs that have been linked via the CHAIN statement, as long as the I option is used with the RUN verb to prevent reinitialization.

It is essential that the order in which variables are listed in COMMON statements be consistent between programs and subroutines. Once a COMMON statement is changed, all other subroutines and programs using the same COMMON area need to be recompiled with the same change in COMMON. For that reason it is suggested that if the COMMON area is used, the same variable names be used in programs and subroutines and that the COMMON statement be placed in a library, to be read via an INCLUDE, $INCLUDE, or $INSERT statement: this way, a change needs to be made only once, although all related programs and subroutines still need to be recompiled.

Arrays can be declared in a COMMON statement, with the same syntax as in a DIM statement. If an array is declared by a COMMON statement, it should not also be declared in a DIM statement, or an error will occur at compile time.

Example

If PROGRAM1 contains the line:

```
COMMON A, B, ADDRESSES(3,3)
```

and SUBR2 contains:

```
COMMON X, Y, MATRIX(3,3)
```

then, if PROGRAM1 calls SUBR2 with the line:

```
CALL SUBR2
```

variables A and X will be equivalent, and variables B and Y will be equivalent, and elements of the dimensioned arrays ADDRESSES and MATRIX will be equivalent.

CONVERT: Convert characters in a string.*

The CONVERT statement can be used to replace characters in a variable.

CONVERT *expr1* **TO** *expr2* **IN** *var*

expr1	the list of original characters to be converted.
expr2	the list of characters to replace original characters.
var	the variable to be converted.

The CONVERT statement replaces every occurrence of each of the specified characters with the corresponding replacement character. It treats each expression as a list of characters, not as a string: the first character in *expr1* is replaced with the first character in *expr2*, the second character in *expr1* is replaced with the second character in *expr2*, and so on.

Every time a character listed in *expr1* appears in the string, it is replaced by the corresponding replacement character, regardless of how many times it appears. If *expr1* contains more characters than *expr2*, the extra characters are deleted from the converted string. If the second expression contains more characters than the first, the extra characters in *expr2* are ignored. If a character is repeated in the first expression, only the first assignment is made and all subsequent assignments of that character are ignored.

---

* Not included in the SMA standards.

Examples

If the variable STRING contains "I LIKE IT", all "K"s can be changed to "V"s with:

```
CONVERT "K" TO "V" IN STRING
```

The resulting string will be "I LIVE IT". However,

```
CONVERT "LIKE" TO "LOVE" IN STRING
```

will produce the string, "O LOVE OT".

In the following application the CONVERT statement is used to turn a comma-separated list of names, NAMES, into a dynamic array, by converting each comma into an attribute mark.

```
EQUATE AM TO CHAR(254)
  .
  .
  .
CONVERT "," TO AM IN NAMES
```

As a comma-separated list, fields of the NAMES array can be deleted, inserted, or arranged only through a sequence of statements involving FIELD, COL1( ), etc. By converting commas to attribute marks, however, fields can be manipulated using the more powerful (and more intuitive) dynamic array functions.

Another common use of CONVERT is to delete extraneous characters from input. For example:

```
EQUATE NIL TO ""
  .
  .
  .
INPUT SOC.NBR
CONVERT "-., " TO NIL IN SOC.NBR
```

COS( ): Return the cosine of the expression.

The COS function returns the trigonometric cosine of the expression.

COS(*expr* **)**

The expression *expr* is treated as an angle expressed as a numeric value in degrees. Values outside the range of 0 to 360 degrees are interpreted as modulo 360.

Example

In the following application the COS function is used with a standard trigonometric formula to calculate the sine of an angle without using the SIN function.

```
SINE = SQRT(1 – COS(ANGLE) * COS(ANGLE) )
PRINT " THE SINE IS CALCULATED AS : " : SINE
```

COUNT( ): Count the number of occurrences of a substring.

The COUNT function determines how many times a character or string of characters occur within a specified string.

COUNT(*string,chars* **)**

string	an expression evaluating to the string to be searched.
chars	an expression evaluating to the substring to be searched for and counted.

The COUNT function returns the number of times a substring is repeated within a string. If the substring is null, the number of characters in the string is returned.

The COUNT function actually counts the number of starting points for the specified substring within the string. That is, for each character in the string, it determines whether an occurrence of the specified substring begins at that character. If it does, its return value will be incremented by one. This means that if there are overlapping occurrences of the substring within the string, COUNT will return as many occurrences as it can find, regardless of whether the starting character is a part of a previous occurrence.*

The DCOUNT function returns the number of fields separated by a given one-character delimiter. See the DCOUNT function for more information.

---

* Not true on all systems. On Prime INFORMATION and uniVerse systems, for instance, COUNT( ) counts only discrete instances of *chars*. No part of the matched substring is recounted toward another match.

Example

To assign the variable NUMS to the number of times the substring "ANA" occurs in the string "BANANA", enter:

```
NUMS = COUNT("BANANA","ANA")
```

This statement assigns a value of 2 to NUMS. Note that the two occurrences of "ANA" overlap.

CRT: Send data to the terminal display screen.

The CRT statement sends data to the terminal display screen. It is similar to the PRINT statement except that it writes only to the terminal. The DISPLAY statement is functionally similar to the CRT statement.

CRT *print-expr*

print-expr is a print expression, optionally combined with commas and colons to designate the format of the output. If *print-expr* is omitted, a blank line is output.

The CRT statement causes data to be output to the terminal screen, regardless of whether a PRINTER ON statement has been executed.

Formatted Output

Format expressions can be used to provide complex output formatting for variables. In the CRT, DISPLAY, and PRINT statements, however, commas and colons can be used to specify tab stops and suppress line feeds.

- Expressions separated by commas are printed at preset tab positions. Multiple commas can be used together to insert consecutive tabs between expressions. However, tab positions cannot be specified without being surrounded by expressions.
- Colons (:) encountered between expressions are interpreted normally as the string concatenation operator. If the last character of the CRT statement is a colon, however, the line feed and carriage return which usually follow the PRINT statement are suppressed. This is especially useful when an INPUT statement is to follow, or in formatted screen programs.

- The @ function can be used with the CRT statement to send the cursor to a specified location on the screen. In this way formatted screens can be generated within programs.

For examples of using commas and colons to format output, see the PRINT statement page.

Examples

To print the string "HELLO..." to the screen, the code might read:

```
CRT "HELLO..."
```

In the following application the CRT statement is used to send output to the terminal screen even if the PRINTER ON statement is active.

```
FOR I = 1 TO 10
   CRT "RESULT NUMBER " : I : " TO PRINTER (Y OR N)" :
   INPUT ANSWER
   IF ANSWER = "Y" THEN
      PRINTER ON
   END ELSE
      PRINTER OFF
   END
   PRINT "RESULT NUMBER " : I : " IS: " : RESULTS(I)
NEXT I
```

The program execution might look like this (with the user's input in bold):

```
RESULT NUMBER 1 TO PRINTER (Y OR N)?Y

ENTRY # 6
RESULT NUMBER 2 TO PRINTER (Y OR N)?N
RESULT NUMBER 2 IS: 98
RESULT NUMBER 3 TO PRINTER (Y OR N)?Y

ENTRY # 7
RESULT NUMBER 4 TO PRINTER (Y OR N)?
```

DATA: Store data in an input stack.

The DATA statement stores the specified data for use by subsequent input requests.

DATA *expr1* [*expr2*, ...]

expr an expression evaluating to the data to be stored.

The DATA statement places one or more values in a first-in-first-out (FIFO) input stack. These values will be used as responses to INPUT statements encountered later in the program, in the order in which they are placed in the stack.

The DATA statement is intended to be used when a CHAIN, ENTER, EXECUTE, or INPUT statement is used to execute another program or a proc, TCL, or ACCESS verb: the first value in the input stack will be used to supply any input which is prompted for. Thus a program which generally prompts the user for values can be called directly, and each of the values can be supplied without user intervention.

Example

The DATA statement can be used in a sequence to save a file item before it is changed in the program. Since the COPY verb prompts for an item name to copy to, the DATA statement is used to stack the response to the prompt.

```
DATA ID : ".OLD "
EXECUTE "COPY CUSTOMERS " : ID
   .
   .
   .
```

DATE( ): Return the date in internal format.

The DATE function returns the current date in internal format.

DATE( )

The DATE function returns the numeric value of the internal system date. The internal format for the date is the number of days since December 31, 1967, which is considered day 0. All dates after December 31, 1967, are positive numbers representing the number of days that have elapsed since then. All dates prior to day 0 are negative numbers representing the number of days prior to this date.

Data calculations are generally much easier when dates are stored in internal format. For example, 90 days from –940 is easier to calculate than 90 days from June 4, 1965. To convert an internal date to external ("human-readable") format, the OCONV function can be used with the "D"

specification. Similarly, the ICONV function can be used to convert from external to internal format.

The TIME function returns the current time in internal format, and the TIMEDATE function returns the current time and date in external format. See the TIME and TIMEDATE functions for more information.

Example

In the following application the last billing date for each customer is stored in Attribute 8 of the file item, in internal format. Thirty days are given for the bill to be paid. The DATE function is used to capture the current date and compare it against the last billing date, to determine if the bill is overdue.

```
PRINT "ENTER THE CUSTOMER ID : " :
INPUT ID
READV BILLING.DATE FROM CUSTFILE , ID , 8 ELSE
   PRINT "CANNOT READ!"
   STOP
END
TODAY = DATE ( )
PAY.DATE = BILLING.DATE + 30
PAY.DATE.EXT = OCONV(PAY.DATE, "D")
BEGIN CASE
CASE TODAY < PAY.DATE
   PRINT "CUSTOMER'S NEXT BILL IS DUE " : PAY.DATE.EXT : "."
CASE TODAY = PAY.DATE
   PRINT "CUSTOMER'S BILL IS DUE TODAY."
CASE TODAY > PAY.DATE
   PRINT "CUSTOMER'S BILL WAS DUE " : PAY.DATE.EXT : "."
   PRINT "CUSTOMER'S BILL IS " : TODAY–PAY.DATE : " DAYS
                                                    OVERDUE."
END CASE
```

DCOUNT( ): Return the number of fields separated by a delimiter.

The DCOUNT function returns the number of fields separated by a specified delimiter.

DCOUNT(*string,char* **)**

string an expression evaluating to the string to be searched.

char an expression evaluating to the character to be used as a delimiter. If *char* evaluates to more than one character, only the first character will be searched for.

The DCOUNT function returns the number of delimited fields contained within a data string.

DCOUNT differs from COUNT in that it only searches for a single character and returns the number of values separated by delimiters rather than the number of occurrences of a character or string of characters. If the null string is used as the delimiter, the length of the string plus one is returned. If the string evaluates to the null string, zero is returned. If the delimiter is not found within the string, a value of 1 is returned.

In most instances the DCOUNT function returns the same value as performing the COUNT function and adding 1. This should not be relied on, however—note that when performed on the null string (""), DCOUNT returns 0, while COUNT+1 returns 1.

The DCOUNT function is particularly useful for processing dynamic arrays; for example, it can be used to count the number of values or subvalues within an attribute.

Example

In the following application the DCOUNT function is used to determine the number of attributes and the number of values in a dynamic array. Each individual value of the array can then be displayed with imbedded FOR loops, as if it were a matrix.

```
NO.OF.ATTRS = DCOUNT(RECORD , CHAR(254))
FOR I = 1 TO NO.OF.ATTRS
NO.OF.VALS = DCOUNT(RECORD <I> , CHAR(253))
   FOR J = 1 TO NO.OF.VALS
      PRINT RECORD < I , J >
   NEXT J
NEXT I
```

DEBUG: Enter the Pick BASIC debugger.*

The DEBUG statement enters the Pick BASIC debugger at the current line of execution.

DEBUG

The DEBUG statement invokes the interactive Pick BASIC debugger. When this statement is encountered, the execution of the program is stopped and the debugger is entered.

As expected, the DEBUG statement is particularly useful while debugging a program. Alternatives are to try to force a pause in program execution (through a SLEEP or INPUT statement) long enough to hit the BREAK key to enter the debugger, or to run the program with the D option.

Example

In the following application the internal subroutine CALCULATE is executed, and then the value of the variable MONTH is examined. If MONTH contains an unexpected value, an error message is printed and the debugger is entered.

```
GOSUB CALCULATE
IF MONTH > 12 OR MONTH < 1 THEN
   PRINT "ERROR IN CALCULATION OF VARIABLE 'MONTH'"
   DEBUG
END
```

DEL: Delete an element from a dynamic array.*

The DEL statement deletes an attribute, value, or subvalue from a specified dynamic array.

DEL *array* **<***attr#* [**,***value#* [**,***subval#*]]**>**

array	the dynamic array to be changed.
attr#	an expression evaluating to the attribute number.

* Not included in the SMA standards.

value# an expression evaluating to the value number. If *value#* is omitted or equal to 0, the entire attribute is deleted.

subval# an expression evaluating to the subvalue number. If *subval#* is omitted or equal to 0, the entire value is deleted.

DEL is the statement equivalent of the DELETE function.

If the attribute, value, or subvalue expressions evaluate to a negative number or a number greater than the number accessible, no action is taken. If the expression evaluates to a noninteger value, it is truncated to an integer value.

The DEL statement does not have the same effect as using the REPLACE function to replace the element with the null string (""). The REPLACE function removes the data while leaving the delimiters intact, whereas the DEL statement removes the delimiters as well as the data. For example, replacing Attribute 2 of an array with "" leaves an empty Attribute 2, but deleting Attribute 2 forces Attribute 3 to become Attribute 2, Attribute 4 to become Attribute 3, and so on.

Example

To delete value 3 of Attribute 5 of the array RECORD, the code would read:

```
DEL RECORD < 5, 3 >
```

DELETE: Delete a file item from a file.

The DELETE statement deletes an item from a file.

DELETE [*filevar* ,] *item-ID*

filevar the variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without a file variable assignment.

item-ID an expression evaluating to the item ID of the item to be deleted.

The file must have been opened with the OPEN statement before it can be deleted. If the file item is not found, no action is taken.

Example

In the following application the INVENTORY file for a shoe store contains information about each shoe sold. The style number is used as the item ID for each shoe. When a shoe has been discontinued, the DELETE statement is used to delete the record from the INVENTORY file.

```
PRINT "ENTER DISCONTINUED STYLE NUMBER : " :
INPUT STYLE.NO
DELETE INVENTORY , STYLE.NO
```

DELETE( ): Delete an element from a dynamic array.

The DELETE function deletes an attribute, value, or subvalue from a specified dynamic array. It is the function equivalent of the DEL statement.

array = **DELETE(***array, attr#* [,*value#* [,*subval#*]] **)**

array	the dynamic array to be changed.
attr#	an expression evaluating to the attribute number.
value#	an expression evaluating to the value number. If *value#* is omitted or equal to 0, the entire attribute is deleted.
subval#	an expression evaluating to the subvalue number. If *subval#* is omitted or equal to 0, the entire value is deleted.

If the attribute, value, or subvalue expressions evaluate to a negative number or a number greater than the number accessible, no action is taken. If the expression evaluates to a noninteger value, it is truncated to an integer value.

The DELETE function does not have the same effect as using the REPLACE function to replace the element with the null string (""). The REPLACE function removes the data while leaving the delimiters intact, whereas the DELETE function removes the delimiters as well as the data. For example, replacing Attribute 2 of an array with "" leaves an empty

Attribute 2, but deleting Attribute 2 forces Attribute 3 to become Attribute 2, Attribute 4 to become Attribute 3, and so on.

Example

To delete value 3 of Attribute 5 in the array RECORD, the code might read:

```
RECORD = DELETE(RECORD, 5, 3)
```

DIM: Declare array variables.

The DIM statement is necessary to declare the dimensions of array variables before they are used within a program.

DIM[ENSION] *var1* **(** *rows* [**,***columns*] **)** [**,***var2* **(** *rows* [**,***columns*] **)** ...]

var...	is the name of the array.
rows [,*columns*]	contains the maximum dimensions of the array, with *rows* and *columns* being whole number constants.

The DIM statement defines the dimensions of an array variable. It must be used prior to any reference to the array in the program. For a *matrix* (a two-dimensional array), use the DIM statement to set the maximum number of rows and columns available for the elements of the array. For a *vector* (a one-dimensional array), use the DIM statement to set the maximum number of elements in the array. See Chapter 2 in this guide for a complete explanation of matrixes and vectors.

The array name can be any legal variable name.The dimensions can be any positive integer. When specifying the two dimensions of a *matrix*, you must use a comma to separate the row and column expressions. These expressions are referred to as indexes. Note that the dimensions must be written as whole numbers; fractional expressions and variables are not allowed.

You can use a single DIM statement to define multiple arrays. If you define more than one array with a single DIM statement, you must use commas to separate the array definitions. The DIM statement can be broken into two or more lines, provided that the new line occurs directly after the comma.

The DIM statement declares the name and size of the array only: it does not assign values to the elements of the array. Assignment of values to the elements is done with the MAT, MATPARSE, MATREAD, MATREADU, and assignment statements.

The DIM statement is necessary to preassign the specified number of entries for the array variable. Alternatively, arrays can also be preassigned in a COMMON statement, which is used to declare variables that will be shared among programs and external subroutines. See the COMMON statement for more information.

Example

The following application declares a matrix called CUSTOMER with 10 rows and 5 columns (for a total of 50 elements), and a vector called PARTS with 15 elements.

```
DIM CUSTOMER(10,5) , PARTS(15)
```

DISPLAY: Send data to the terminal display screen.*

The DISPLAY statement sends data to the terminal screen. It is similar to the PRINT statement except that it writes only to the terminal. The DISPLAY statement is identical to the CRT statement.

DISPLAY *print-expr*

print-expr is a print expression, optionally combined with commas and colons to designate the format of the output (as described below). If *print-expr* is omitted, a blank line is output.

The DISPLAY statement causes data to be output to the terminal screen, regardless of whether a PRINTER ON statement has been executed.

---

* Not included in the SMA standards.

Formatted Output

Format expressions can be used to provide complex output formatting for variables. In the CRT, DISPLAY, and PRINT statements, however, commas and colons can be used to specify tab stops and suppress line feeds.

- Expressions separated by commas are printed at preset tab positions. Multiple commas can be used together to insert consecutive tabs between expressions. However, tab positions cannot be specified without being surrounded by expressions.
- Colons (:) encountered between expressions are interpreted normally as the string concatenation operator. If the last character of the DISPLAY statement is a colon, however, the line feed and carriage return which usually follow the PRINT statement are suppressed. This is especially useful when an INPUT statement is to follow, or in formatted screen programs.
- The @ function can be used with the DISPLAY statement to send the cursor to a specified location on the screen. In this way formatted screens can be generated within programs.

For examples of using commas and colons to format output, see the PRINT statement page.

Examples

To print the string "HELLO..." to the screen, the code might read:

```
DISPLAY "HELLO..."
```

In the following application the DISPLAY statement is used to send output to the terminal screen even if the PRINTER ON statement is active.

```
FOR I = 1 TO 10
   DISPLAY "RESULT NUMBER " : I : " TO PRINTER (Y OR N)" :
   INPUT ANSWER
   IF ANSWER = "Y" THEN
      PRINTER ON
   END ELSE
      PRINTER OFF
   END
   PRINT "RESULT NUMBER " : I : " IS: " : RESULTS(I)
NEXT I
```

The program execution might look like this (with the user's input in bold):

```
RESULT NUMBER 1 TO PRINTER (Y OR N)?Y

ENTRY # 6
RESULT NUMBER 2 TO PRINTER (Y OR N)?N
RESULT NUMBER 2 IS: 98
RESULT NUMBER 3 TO PRINTER (Y OR N)?Y

ENTRY # 7
RESULT NUMBER 4 TO PRINTER (Y OR N)?
```

EBCDIC( ): Convert a string from ASCII to EBCDIC code.

The EBCDIC function converts a string in ASCII code into EBCDIC code.

EBCDIC (*expr*)

expr an expression evaluating to the string to be converted.

The EBCDIC function converts each character of the given expression from its ASCII representation value to its EBCDIC representation value. The EBCDIC function is the inverse of the ASCII function.

Appendix C supplies a full listing of ASCII codes.

Example

In the following application a file item needs to be converted from ASCII to EBCDIC and written onto a tape to be read by a non-Pick system.

```
STRING = EBCDIC(STRING)
WRITET STRING ELSE
  .
  .
  .
```

ECHO: Turn system echo on or off.

The ECHO statement toggles the system echo on the user's terminal.

ECHO [**ON** | **OFF** | *expr*]

ON	turn system echo on (default).
OFF	turn system echo off.
expr	an expression evaluating to a numeric value. If *expr* evaluates to 0, the echo is turned off; if *expr* evaluates to a number other than 0, the echo is turned on.

The ECHO statement controls the display of input characters on the terminal screen. Normally, all data entered by the user is echoed to the screen as it is typed. The ECHO OFF statement turns off the echo feature, and the ECHO ON statement turns it back on.

If ECHO OFF is specified, subsequent input characters are read by the system as usual, but only control characters (CHAR(1) to CHAR(31)) are displayed on the terminal screen. The ability to turn off character display with ECHO OFF is particularly useful for entering passwords and other confidential information.

The ECHO OFF statement does not affect the echo of control characters for input which is taken directly from the user's terminal.*

☞ The echo feature is not reinstated at the end of a program, so an ECHO OFF statement should be followed by an ECHO ON statement before the end of a program.

Examples

In the following application the user needs to log on to the program and is prompted for a password, which is kept in a file opened to PASSFILE. The ECHO OFF statement is used before accepting the password, so that it does not appear on the screen.

```
READV PASSWD FROM PASSFILE , NAME,1 ELSE
   PRINT "LOGON NOT FOUND!"
   STOP
END
PRINT "PASSWORD : " :
ECHO OFF
INPUT PASSWORD
ECHO ON
```

---

* Some Pick systems allow you to disable control characters entirely from user input. See, for example, ADDS Mentor's INPUT CTRL statement.

```
IF PASSWD # PASSWORD THEN
   PRINT "SORRY."
   ABORT
END
```

Note that the ECHO ON statement is executed immediately after the INPUT statement. If ECHO ON had been included any later, then every time the program aborted because of a bad password, the echo feature would remain off when the user returned to TCL.

In the next application a counter ECHO.COUNT is used to simulate the Break Inhibit Counter. Instead of using ECHO ON, the counter is incremented by one and ECHO ECHO.COUNT is executed instead. Similarly, the counter is decremented by one to turn the echo feature off. By using a counter, programs and subroutines can intermix without danger of a subroutine prematurely reinstating the echo feature.

```
ECHO.COUNT += 1
ECHO ECHO.COUNT
   .
   .
   .
CALL ACCESS.TEST(ECHO.COUNT)
   .
   .
   .
ECHO.COUNT -= 1
ECHO ECHO.COUNT
STOP
```

If the subroutine ACCESS.TEST uses the same method of manipulating the echo, the status of the echo feature is maintained throughout the program.

```
SUBROUTINE ACCESS.TEST(COUNTER)
COUNTER += 1
ECHO COUNTER
   .
   .
   .
COUNTER -= 1
ECHO COUNTER
RETURN
```

ELSE: Initiator used with conditional statements.

For information about the THEN | ELSE construct, see the IF statement.

END: End compilation or a group of THEN | ELSE statements.

The END statement is used to designate the end of a program or the end of a group of statements begun with a THEN or ELSE clause.

END

Use the END statement to terminate a Pick BASIC program or a clause within an IF, file I/O, or tape I/O statement.

An END statement can be the last statement in a Pick BASIC program, but it is not mandatory.* When an END statement that is not associated with an IF, LOCK, or I/O statement is encountered, all subsequent lines of the program are ignored by the Pick BASIC compiler. Note that any subroutines which can have been entered after a program END statement are not recognized.

☞ In using the END statement with multiple conditional statements, take care to use the correct number of END statements. The wrong number of END statements will cause the compilation to abort.

Examples

In the following example, END statements are used with nested IF constructs:

```
SUIT = RND(4)
IF SUIT = 0 THEN
   SUIT = "CLUBS"
END ELSE
   IF SUIT = 1 THEN
      SUIT = "DIAMONDS"
   END ELSE
      IF SUIT = 2 THEN
         SUIT = "HEARTS"
      END ELSE
         IF SUIT = 3 THEN
            SUIT = "SPADES"
         END
      END
   END
END
```

---

* Except on Prime INFORMATION, where END *is* mandatory.

```
NUM = RND(13)
   .
   .
   .
END
```

The last END is interpreted by the compiler as the end of the program.

ENTER: Transfer control to another program.

The ENTER statement can be used to transfer permanent control to another program. Both the ENTERed program and the program containing the ENTER statement should be cataloged.

ENTER { *item-ID* | *@var* }

item-ID	is the item ID of the program to be entered.
@var	*var* is a variable which has been assigned the item ID of the program to be entered.

The ENTER statement transfers program control from the calling program to another program. Program sequence will not return to the calling program. For this reason, the ENTER statement should not be used within a subroutine, nor should it be used to call a subroutine.

Parameters to be passed between programs must be declared through COMMON statements in both programs. All other variables will be initialized upon entering the new program.

Example

In the following application the user is prompted for the program to enter, and then the ENTER statement is used to transfer control to the appropriate program.

```
PRINT "WOULD YOU LIKE TO:"
PRINT " " , "1.  EDIT AN EXISTING ITEM"
PRINT " " , "2.  CREATE A NEW ITEM"
PRINT " " , "3.  DISPLAY AN EXISTING ITEM"
PRINT " " , "(ENTER ANY OTHER PROGRAM)"
PRINT
PRINT "ENTER 1, 2 OR 3":
INPUT ANSWER
```

```
BEGIN CASE
   CASE ANSWER = 1
      ENTER EDIT.ITEM
   CASE ANSWER = 2
      ENTER CREATE.ITEM
   CASE ANSWER = 3
      ENTER DISPLAY.ITEM
   CASE 1
      ENTER @ANSWER
END CASE
```

EQUATE: Assign values at compile time.

The EQUATE statement allows a constant to be assigned at compile time.

EQU[ATE] *var* **TO** *value* [,*var2* **TO** *value2*, ...]

var the name to be assigned to the constant.

value a constant, another variable, or the CHAR function.

The EQUATE statement can be used to define a constant or to make one variable equivalent to another. The value is compiled directly into the object code, so there is no storage location generated for the variable. EQUATE differs from the standard assignment statement (=) in that the assignment is made at compilation and does not have to be reassigned each time the program is executed.

Only literal constants or other variables can be used as the *value*. No operators or functions can be used in *value*, since otherwise it could not be evaluated until run time. The one exception is the CHAR function.

The EQUATE statement is most commonly used to define constants and to provide more meaningful names for variables. Any number of variables can be assigned with a single EQUATE statement, with each clause separated by a comma. An EQUATE statement can be broken into two or more lines, provided that the newline occurs directly after the comma. Multiple EQUATE statements can be used in a program as long as the same variable is not equated twice.

A variable cannot be referred to prior to the EQUATE statement which defines it, or an error similar to the following will occur at compile time:

```
[B115] LINE 9 LABEL 'ERR.BELL' IS USED BEFORE THE EQUATE
                                                    STMT
```

This error message occurs if the variable is used in any way. This includes assignment via the standard assignment operator (=), or if the variable had been previously equated.

Common uses of the EQUATE statement are:

```
EQUATE AM TO CHAR(254),
   VM TO CHAR(253),
   SVM TO CHAR(252),
   TRUE TO 1,
   FALSE TO 0,
   BELL TO CHAR(7)
   ESC TO CHAR(27)
```

Example

In the following application the EQUATE statement is used to give meaningful names to the elements of an array. It is also used to give meaningful names to the bell character (CHAR(7)) and to logical true and false (1 and 0), to make the program more readable.

```
DIM CUST(6)
EQUATE BELL TO CHAR(7), TRUE TO 1, FALSE TO 0
EQUATE NAME TO CUST(1), ADDRESS TO CUST(2),
   CITY TO CUST(3), STATE TO CUST(4),
   ZIP TO CUST(5), PHONE TO CUST(6)
   .
   .
   .
```

EXECUTE: Execute a TCL command and return to the program.*

The EXECUTE statement executes a specified TCL command from within the program, and then returns execution to the statement that follows the EXECUTE statement in the program.

EXECUTE *command-expr* [**CAPTURING** *cvar*]
[**RETURNING** *rvar*]

command-expr any valid TCL command.

---

* The SMA standards define a simpler version of the EXECUTE statement than the one described here. Most implementations of Pick BASIC now support the enhanced version we describe.

CAPTURING *cvar*	place the command output in *cvar*.*
RETURNING *rvar*	place all error message numbers in *rvar*.*

The EXECUTE statement is a powerful statement for executing TCL verbs from within a Pick BASIC program.

The CAPTURING clause can be used to capture the output of the command into a string variable. The output will be in the form of a dynamic array, with each line of output separated by attribute marks (CHAR(254)). This clause is particularly useful for programs that need to access information that is unavailable to the program directly (such as output from the SYSPROG verbs WHAT and WHERE).

The RETURNING clause can be used to capture the error messages from the command into a string variable. This variable will be a string of error message numbers separated by spaces. If a spooler hold file is created by the command, the hold file entry number is also entered into the returning variable.

The DATA statement can be extremely powerful when used with the EXECUTE statement. With the DATA statement, known responses can be stacked for subsequent EXECUTE commands that require operator input. Any leftover data on the stack will be cleared upon return from the EXECUTE statement. See the DATA statement for more information.

EXECUTE can be used to run another Pick BASIC program, which in turn can run a third Pick BASIC program. Each of these constitutes an *execution level*. On some implementations the SYSTEM function can be used to return the current execution level. Up to at least five levels are supported on most systems.

In using the CAPTURING clause, be careful of TCL verbs that clear the screen before displaying output. If the TCL verb clears the screen, the screen-clearing string will also appear in the capturing variable *cvar*. Other terminal manipulation strings might also appear. These strings can be stripped out with the INDEX function.

EXECUTE with Select-Lists

The EXECUTE statement can be used to generate a select-list to be used by a READNEXT statement in the program. Thus EXECUTE provides an

---

* Not all implementations support the CAPTURING and RETURNING clauses.

alternative to the SELECT statement. On many implementations the SYSTEM function can be used to determine whether a select-list is present.

When EXECUTE produces a select-list, the data stack is checked for input before control returns to the program. Any input in the stack is submitted to TCL as a command and executed with the select-list available to it.

On most Pick systems, if a select-list is active when the EXECUTE statement is executed, the select-list is passed to the executed TCL command. On some implementations, however, select-lists which have been returned to the Pick BASIC program cannot be used by subsequent EXECUTE statements. On these systems, the DATA statement can be used to stack a SAVE-LIST command to be executed before returning to the program.

Printing Output from EXECUTE

The output of EXECUTE is sent to the terminal unless otherwise specified with the P option to the command. If the P option is supported by the command, EXECUTE uses the default print file. If this file is already open, the output from EXECUTE is appended to it. The print file, however, will not be closed upon completion of EXECUTE, but will need to be closed by the SP-CLOSE command, by the PRINTER CLOSE statement, or by terminating the program.

Examples

To select a list and then save it, the DATA and EXECUTE statements can be used together as follows:

```
DATA "SAVE-LIST NAMES"
EXECUTE "SSELECT TFILE BY NAME"
```

After the SSELECT command is run, the data stack is examined and the SAVE-LIST command is executed before returning to the program.

In the following application the EXECUTE statement allows the operator to edit and compile another program until it successfully compiles. The RETURNING clause is used to determine if the compile was successful.

The DCOUNT function supplies the number of errors which were returned. The FIELD function extracts the last error. If the last error is "B100" ("COMPILATION ABORTED; NO OBJECT CODE PRODUCED"), then

the program is edited and compiled again; otherwise, the program is executed.

```
SUCCESS = 0
LOOP
   EXECUTE "DE BP " : PROGRAM
   EXECUTE "COMPILE BP " : PROGRAM RETURNING RESULT
   NO.OF.ERRS = DCOUNT(RESULT," ")
   IF FIELD(RESULT," ",NO.OF.ERRS) = "B100" THEN
      PRINT @(-1) : "DIDN'T COMPILE"
   END ELSE
      PRINT @(-1)
      EXECUTE "RUN BP " : PROGRAM
      SUCCESS = 1
   END
UNTIL SUCCESS DO REPEAT
```

In the next example the operator would like to know if a certain user is logged on. The EXECUTE statement executes "WHO A" and then searches through each line for the other user. The FIELD function is used to determine if the other user is logged on, and then the FIELD function is used again to isolate the process number. Thus all processes the other user is logged on to are reported.

```
EQUATE TRUE TO 1, FALSE TO 0,
   BLANK TO " ", NIL TO ""
PRINT "WHO TO SEARCH FOR? ":
INPUT PERSON
EXECUTE "WHO A" CAPTURING LIST
CLEARSTR = @(-1)
LEN.CLEARSTR = LEN(CLEARSTR)
LOOP
   POS = INDEX(LIST,CLEARSTR,1)
UNTIL POS = 0 DO
   LIST = LIST [ 1,POS-1 ] : LIST [ POS + LEN.CLEARSTR,9999 ]
REPEAT
NO.OF.LOGINS = DCOUNT(LIST , AM)
FOUND = FALSE
FOR I = 1 TO NO.OF.LOGINS
   LINE = LIST<I>
   IF FIELD(LINE , BLANK , 2) = PERSON THEN
      PROCESS = FIELD(LINE , BLANK , 1)
      IF FOUND = FALSE THEN
         FOUND = TRUE
         PRINT PERSON : " ON PROCESS " : PROCESS
      END ELSE
         PRINT NIL , "ALSO PROCESS " : PROCESS
      END
   END
NEXT I
```

EXP( ): Return *e* to the specified power.

The EXP function returns the value of the base number *e* raised to the power of the given expression. It is the inverse of the LN function.

EXP(*expr* **)**

expr an expression evaluating to a numeric value.

e is an irrational number, with an approximate value of 2.7183. The EXP function is essentially equal to PWR(2.7183,*expr*), but it produces results of greater accuracy. If the expression is too large or too small, zero is returned.

Example

To assign e^{10} to the variable NUM, the code would read:

```
NUM = EXP( 10 )
```

EXTRACT( ): Return an attribute, value, or subvalue from an array.

The EXTRACT function is used to access a specified attribute, value, or subvalue from a dynamic array string. There are two forms to the EXTRACT function: the first uses the keyword "EXTRACT", and the second uses angle brackets.

EXTRACT(*array,attr#* [**,***value#* [**,***subval#*]] **)**

array<attr# [**,***value#* [**,***subval#*]]**>**

array the dynamic array string.

attr# an expression evaluating to the attribute number.

value# an expression evaluating to the value number. If *value#* is omitted, the entire contents of the attribute is returned.

subval# an expression evaluating to the subvalue number. If *subval#* is omitted, the entire contents of the value is returned.

If an attribute, value, or subvalue expression evaluates to a noninteger value, it is truncated to an integer value. Negative numbers are not valid for *attr#*, *value#*, or *subval#*.

Examples

To assign the contents of Attribute 6 of the string array ADDRESS to the variable ZIP, the code would read:

```
ZIP = EXTRACT(ADDRESS, 6)
```

or

```
ZIP = ADDRESS < 6 >
```

In the following application, a program prints the billing information for a customer. The first and second attributes of the record CUST.REC contain the customer's name, and the sixth, seventh, and eighth attributes contain the customer's billing information.

```
EQUATE BLANK TO " ", NIL TO ""
   .
   .
   .
NAME = CUST.REC< 1 > : BLANK
NAME := CUST.REC< 2 >
PRINT NAME
PRINT "CREDIT CARD: " , CUST.REC< 6 >
PRINT "ACCT. NO. :" , CUST.REC< 7 >
PRINT "EXPIRES:" , OCONV( CUST.REC< 8 >, "D" )
```

FIELD( ): Return a delimited substring of a string.

The FIELD function searches through a string and returns a specified substring, based on a delimiter and the number of occurrences to search for.

FIELD (*expr,delimiter,n*)

expr	an expression evaluating to the string to be searched.
delimiter	an expression evaluating to the delimiter to be searched for. Only one character will be accepted as a delimiter; any subsequent characters are ignored.
n	an expression evaluating to the number of the substring you want to retrieve.

The FIELD function returns the substring in an expression which lies between the *n*–1 and *n*th occurrences of the delimiter. If *n* is 1, the substring from the beginning of the expression to the first occurrence of the delimiter is returned.

Examples

If STRING contains "NOW IS THE TIME", the third word of the string ("THE") can be assigned to the variable THIRD with:

```
THIRD = FIELD(STRING, " ", 3)
```

In the following application, Attribute 1 of the MEMBER array holds the member's full name, and Attribute 8 holds the date on which the customer first joined the health club. The FIELD function is used to isolate the month from the rest of a date in external format, in order to determine if the customer's annual fee is due.

```
NOW = FIELD(TIMEDATE( ) , " " , 4)
MEM.DATE = MEMBER<8>
MEM.DATE = OCONV(MEM.DATE , "D")
MONTH = FIELD(MEM.DATE , " " , 2)
IF MONTH = NOW THEN
   DUES.OWED : = MEMBER<1>
END
```

An easier way to do this, of course, is to use a DM conversion code with OCONV.

FOOTING: Specify the footing for output pages.

The FOOTING statement specifies the text to be printed at the bottom of each page of output.*

FOOTING *expr*

expr — an expression evaluating to the text to be printed as the footing. The full syntax of *expr* is similar to that of the FOOTING modifier in ACCESS:

---

* On some implementations, the HEADING statement is necessary to initialize the page parameters for a program, otherwise the FOOTING statement will have no effect.

[*text*] ['*options* '] [*text*] ['*options* ']...

The following *options* can be included in *expr*, enclosed in single quotes:

P[*n*] Current page number, right-justified in a field of *n* blanks (default is 4).

PN Current page number, left-justified.

L Carriage return and line feed.

T Current time and date.

D Current date.

C Center the line.

Multiple options can be specified within a single set of quotes.

Multiple FOOTING statements can be used to change the footing within a single program, with each FOOTING statement taking effect on the current page of output. Pagination begins with page 1 and increments automatically on generation of each new page.

Output will be suspended after each footing until a carriage return is input. This feature can be disabled by the N option to the HEADING statement.

The HEADING, FOOTING, and PAGE statements affect the same output device that the PRINT statement does. The PRINTER statement toggles the output device between the terminal screen and the printer. If multiple print units are used, the HEADING, FOOTING, and PAGE statements will only affect print unit 0 (the default).

Example

In the following application the FOOTING statement sets up a footing on each page with "Status Report" in the left corner and the page number on the right. The SPACE function is used to generate the correct number of spaces in the footing.

```
FOOTING "Status Report" : SPACE(57) : "Page 'PC' "
```

FOR: Repeat a procedure with an incrementing variable.

The FOR...NEXT construct is designed to allow a set of statements to be repeated and a variable to be incremented until the variable reaches a designated maximum value.

FOR *var* = *start-expr* **TO** *end-expr* [**STEP** *step-expr*]
    [{ **WHILE** | **UNTIL** } *expr*
    statements]
NEXT *var*

var=start-expr	set the specified variable *var* to the value of *start-expr* for the first iteration.
end-expr	stop iterating when *var* exceeds *end-expr*
step-expr	increment *var* by *step-expr* for each iteration. If the STEP clause is not specified, 1 is the default increment.
WHILE *expr*	continue the loop as long as *expr* evaluates to true. Once *expr* evaluates to false, continue program execution after the NEXT statement.
UNTIL *expr*	continue the loop until *expr* evaluates to true. Once *expr* evaluates to true, continue program execution after the NEXT statement.

A FOR...NEXT program loop is a series of statements that execute repeatedly, with a variable *var* incremented by a specified amount for each iteration. (The default increment is 1.) The first time the statements are executed, the specified variable *var* is assigned the value *start-expr*. The second time they are executed, *var* is assigned the value *start-expr* +1 and so on, until *var* exceeds the value of *end-expr*. The NEXT clause defines the end of the loop, forces the increment of the variable *var*, and returns control to the FOR statement.

The body of the loop is skipped if *start-expr* is greater than the *end-expr* and *step-expr* is positive.

Note that although *start-expr* is usually 1 and *end-expr* is usually specified as a positive integer, these are not fixed restrictions: any Pick BASIC expression which evaluates to a numeric value can be used.

Alternative syntax forms involve the STEP clause and the WHILE and UNTIL clauses.

If the STEP *step-expr* clause is specified, *step-expr* is taken as the increment instead of 1, and the *step-expr* is added to the value of *var* for each iteration. Note that *step-expr* does not have to be positive, so *var* can exceed *end-expr* on the onset of a loop with a negative step, and the loop will continue as long as *var* is greater than or equal to *end-expr*. In addition, *step-expr* does not have to evaluate to an integer, so any decimal value can be specified as the step in a FOR loop.

The WHILE and UNTIL clauses can be added to the FOR loop to provide additional flexibility. These clauses will behave just as they do in the standard LOOP structure. See the LOOP reference page for more information.

FOR...NEXT loops can be nested, provided that:

- each loop has a unique variable *var* as its counter.
- the NEXT statement for the inside loop appears before the NEXT statement for the outside loop.

Example

In the following application the DCOUNT function returns the number of attributes in the dynamic array CUST.REC, and a FOR loop is used to print each attribute of the array.

```
NO.OF.ATTRS = DCOUNT(CUST.REC,CHAR(254))
FOR I = 1 TO NO.OF.ATTRS
   PRINT I,CUST.REC < I >
NEXT I
```

Format Expressions: Specify a format for data.

Format expressions return the given data in a specified format. They are particularly useful for generating output in a human-readable form.

expr format-expr

expr	an expression evaluating to the text to be formatted.
format-expr	an expression evaluating to the formatting codes to be used. The syntax for *format-expr* is described below.

Formatted data is produced by simply specifying the format expression directly after the data. In Pick BASIC, if one expression immediately follows another, the second expression is automatically taken as a format expression for the first.

The format expression interprets the given expression as straight numeric data or as the internal format for a date. For an internal date, a D should be entered as the first character in the format expression, and the next two characters determine the number of characters to be taken as the suffix for the year and the character to be used as a delimiter. See the section "Internal Date Conversion" for more information on using format expressions to convert an internal date into external format.

For straight numeric data, the format expression is broken into the following components:

[*format-string*] [[( ] *format-mask* [ )]]

The *format string* formats the data itself, and the *format mask* formats the field in which the result is placed. Although the parentheses around the format mask are optional, they are highly recommended in order to clearly delineate the format mask from the format string.

Be aware that if a format expression does not match the syntax expected, or if the data expression does not evaluate to a numeric value, the results could be bizarre and no error message will be printed.

The formatting codes can be used with the masked input statement, INPUT @, to accept only data matching the specified code and store it in its raw form. See the INPUT @ statement for more information.

Formatting Numbers

The format string for numeric data is a series of characters in the following order:

j Justification code. Justification can be one of the following:

L Left justification (default).

R Right justification.

Justification codes are meant to be used in conjunction with the format masking codes listed below. The justification codes

have no effect unless the format masking codes are used to specify a field size and background character.

n Decimal precision code. A single-digit number (0-9) to be taken as the number of digits to be printed following the decimal point. If *n* is 0, a decimal point will not be printed. If the descaling code (*m*) is specified, the decimal precision *must* be specified first.

m Descaling code. A single-digit number (0-9) to be taken as the descaling factor. A descaling code equal to the current precision (4 by default on most systems) will return the number unchanged, and a descaling code equal to the current precision plus 1 will return the number divided by 10. In general, a descaling code equal to the current precision plus *x* will return the current number divided by 10^x. If the descaling code is specified, the decimal precision (*n*) *must* be specified first.

Z Convert all leading zeros to blanks. This has no effect on leading zeros generated by using the % character in a right-justified format mask.

, Insert commas. Every three digits to the left of the decimal point will be grouped together by commas, each comma representing a thousands position.

c Credit indicator code. One of the following:

- C Place "CR" after negative values and two blank spaces after all other values.
- D Place "DB" after positive values and two blank spaces after all other values.
- E Place negative values between angle brackets (<...>), and all other values between blank spaces.
- M Place a minus sign after negative values.
- N Suppress the minus sign in negative values.

If a decimal precision code (*n*) is not specified with a credit indicator code, a precision of 0 is assumed.

$ Place a dollar sign at the beginning of the resulting data.

Format Masks

The format mask includes any of the following codes:

$	Place a dollar sign at the beginning of the field.
a	Any character to be placed in the field as a filler.
#[*n*]	Place data in a field of *n* blanks. If *n* is not specified, the next character of the resulting data is returned.
*[*n*]	Place data in a field of *n* asterisks. If *n* is not specified, the next character of the resulting data is returned.
%[*n*]	Place data in a field of *n* zeros. For a right-justified field, this code forces leading zeros. If *n* is not specified, the next character of the resulting data is returned.

Although all specifications are optional, Pick BASIC makes certain assumptions about the positioning of format codes. In particular, an initial alphabetic character will be taken as a justification code, and an initial numeric character will be taken as a decimal precision code. Thus, in order to specify either the zero suppression code (Z) or the credit indicator codes (C, D, E, M, N), either a justification code or a decimal precision code must be specified first. Similarly, in order to specify a descaling code, a decimal precision code must immediately precede it.

Internal Date Conversion

For internal date conversion, the syntax of the format expression is as follows:

D [*year*] [{ *separator* | *subcode* }]

where *year* is a single-digit number, *separator* is any character, and *subcode* is a special date subcode if no date is specified. *year* determines how many digits to return from the end of the year, with all digits as the default (for example, "–940" "D" produces "4 JUN 1965", and "–940" "D2" produces "4 JUN 65"). *separator* determines the delimiter for the output fields, in which case the month, date and year are returned in their numeric values delimited by *c* (for example, "–940" "D/" produces "06/04/1965"). Date subcodes are fully described in *Pick ACCESS: A Guide to the SMA/RETRIEVAL Language*.

Examples

Consider a program which reports the balances in a customer's bank account. For accuracy, the current balance and balance at the beginning of the month are kept to five decimal places. Also, the date on which the account was last updated is kept in internal format.

```
BEGIN.BAL = ACCT.REC<7>
BALANCE = ACCT.REC<8>
LAST.UPDATE = ACCT.REC<10>
TODAY = DATE( )
PRINT
ELAPSED = TODAY – LAST.UPDATE
PRINT "LAST UPDATE WAS " : ELAPSED : " DAYS AGO" :
PRINT " ON " : LAST.UPDATE "D2/"
PRINT
PRINT "BALANCE AT START OF MONTH WAS " : BEGIN.BAL "2,$"
PRINT "CURRENT BALANCE IS " : BALANCE "2,$"
PRINT "WITH A CHANGE OF " : BALANCE–BEGIN.BAL "2,$"
```

All dollar values are formatted with the "2,$" code, specifying that:

- only two decimal points should be shown.
- commas should be placed at every thousands point.
- a dollar sign should precede the number.

In addition, the internal date is formatted with the "D2/" code, so that only the last two digits of the year are shown, and slashes (/) are used to delimit the fields.

If a file item from which ACCT.REC was read contains the following data in Attributes 7 through 10:

```
.
.
.
007 78.22545
008 2943.56657
009
010 8096
.
.
.
```

then the following is printed on the screen:

```
LAST UPDATE WAS 14 DAYS AGO ON 03/01/90
BALANCE AT START OF MONTH WAS $78.23
CURRENT BALANCE IS $2,943.57
WITH A CHANGE OF $2865.34
```

Now consider the same program, but with the balances kept as integer values. In actual practice, all numbers on Pick systems are stored as integers, so the file item from which ACCT.REC was read would actually read:

```
.
.
.
007 7822545
008 294356657
009
010 8096
.
.
.
```

To access the data correctly, the integer values need to be divided by 10^5. If the current precision is 4, the last five lines of code would read:

```
PRINT "LAST UPDATE WAS " : ELAPSED : " DAYS AGO " :
PRINT "ON " : LAST.UPDATE "D2/"
PRINT "BALANCE AT START OF MONTH WAS " : BEGIN.BAL "29,$"
PRINT "CURRENT BALANCE IS " : BALANCE "29,$"
PRINT "WITH A CHANGE OF " : BALANCE – BEGIN.BAL "29,$"
```

The only difference to note is that the format string is now "29,$". If the precision is 4, the resulting data shows a decimal point five places from the right. The following is printed on the screen:

```
LAST UPDATE ON ACCOUNT WAS 14 DAYS AGO ON 03/01/90
BALANCE AT BEGINNING OF MONTH WAS $78.23
CURRENT BALANCE IS $2,943.57
WITH A CHANGE IN BALANCE OF $2865.34
```

Note that the results are the same, as expected.

Now imagine that instead of printing out the account balances directly, we would like it tabulated on the screen. Possible source code might read:

```
PRINT "BEGAN MONTH WITH" , "CURRENT  BALANCE" ,
"INCREASE/DECREASE"
PRINT BEGIN.BAL "R29 , $(#16)" , "":
PRINT BALANCE "R29 , $(#15)" , BALANCE – BEGIN.BAL "R29 ,
$(#17)"
```

In parentheses for each of the masking codes is "#*n*", with *n* representing the maximum number of characters in the field, in this case the number of characters in the heading col\umn. "#*n*" places the data in a field of *n* spaces. The "R" at the beginning of the format expression specifies right-justified data (so that if multiple accounts were reported, all decimal points would line up). With the same data as the previous example, the result might be:

BEGAN MONTH WITH	CURRENT BALANCE	INCREASE/DECREASE
$78.23	$2,943.57	$2,865.34

GOSUB: Branch to an internal subroutine.

The GOSUB statement transfers control to the statement with the specified label, where the program will sequentially proceed until it reaches a RETURN statement. The GOSUB statement has two forms of execution: the direct branch to a specified subroutine, and the computed branch to one of several subroutines.

GOSUB *label*
ON *expr* **GOSUB** *label1 label2 ...*

label... statement labels as described in Chapter 2.

expr an expression evaluating to a numeric value.

When the first form is used, program control is transferred to the line starting with the specified label. Once a RETURN statement is encountered, program control transfers back to the statement following the corresponding GOSUB in the main program. In well-structured programs, an *internal subroutine* should comprise only the set of statements confined between the branch label and the RETURN statement.

The ON...GOSUB syntax variation, or *computed GOSUB*, can be used to call different subroutines according to the value of *expr*. *expr* is evaluated and truncated into an integer *n*, and the subroutine beginning with the *n*th label on the statement line is given control of the program. If *expr* evaluates to a number less than 1 or greater than the number of statement labels listed, no action is taken.

The colon is optional in the statement label reference. Furthermore, the statement label can be anywhere in the program; that is, it can either precede or follow the referencing GOSUB statement. If the specified statement label is not found, an error message is printed at compile time.

Examples

The following example shows how a GOSUB statement can be used to allow the user some control of program execution.

```
    PRINT "WOULD YOU LIKE THE RESULTS? (Y OR N)":
    INPUT ANSWER
    IF ANSWER = "Y" THEN GOSUB CALC
* CONTROL RETURNS HERE AFTER CALC
      .
      .
      .
CALC:   * CONTROL TRANSFERS HERE IF ANSWER = Y
      .
      .
      .
    RETURN
```

The next example shows how a computed GOSUB statement can be used in a menu-driven program. Note that the input is tested for a proper value and that execution is suspended until a reasonable response is supplied.

```
    EQU TRUE TO 1, FALSE TO 0
      .
      .
      .
    VALID = FALSE
    LOOP
      PRINT "CHOOSE A MENU (1, 2, OR 3) " :
      INPUT MENU
      IF MENU = 1 ! MENU = 2 ! MENU = 3 THEN
        VALID = TRUE
      END ELSE
        PRINT "ILLEGAL ENTRY."
      END
    UNTIL VALID DO REPEAT
    * BRANCH TO REQUESTED MENU
    ON MENU GOSUB MENU1, MENU2, MENU3
    * CONTROL RETURNS HERE AFTER SUBROUTINE
    .
    .
    .
    STOP
MENU1:  * CONTROL TRANSFERS HERE IF MENU 1 IS CHOSEN
    .
    .
    .
    RETURN
```

```
MENU2:  * CONTROL TRANSFERS HERE IF MENU 2 IS CHOSEN
   .
   .
   .
   RETURN
MENU3:  * CONTROL TRANSFERS HERE IF MENU 3 IS CHOSEN
   .
   .
   .
   RETURN
```

GOTO: Transfer program control to a specified label.

The GOTO statement transfers control to the line with the specified statement label, from which the program will sequentially proceed until it reaches the end of the program or another GOTO statement. The GOTO statement has two forms of execution: the direct branch to a specified label, and the computed branch to one of several labels.

GO[TO] *label*
ON *expr* **GO[TO]** *label1 label2 ...*

When the first form is used, program control is directly transferred to the line starting with the specified label.

The ON...GOTO syntax variation, or *computed GOTO*, can be used to branch to different labels according to the value of *expr*. *expr* is evaluated and truncated into an integer *n*, and the program branches to the statements starting at the *n*th label on the ON...GOTO line. If *expr* evaluates to a number less than 1 or greater than the number of statement labels listed, no action is taken.

The colon is optional in the statement label reference. Furthermore, the statement label can be anywhere in the program; that is, it can either precede or follow the referencing GOTO statement, as long as it is before the end of the program. If the specified statement label is not found, an error message is printed at compile time.

Example

The following example shows how a GOTO statement can be used to conditionally jump to another point in the program.

```
IF CARD(1,1) + CARD(1,2) = 21 THEN
   PRINT "BLACKJACK!  YOU WIN!"
   GOTO GAMEEND
END ELSE ...
   .
   .
   .
GAMEEND:
PRINT "WANT TO PLAY AGAIN?" :
   .
   .
   .
```

HEADING: Initialize parameters, specify heading for output pages.

The HEADING statement specifies the text to be printed at the top of each page of output.*

HEADING *expr*

expr an expression evaluating to the text to be printed as the footing. The full syntax of *expr* is similar to that of the HEADING modifier in ACCESS:

[*text*] ['*options* '] [*text*] ['*options* ']...

The following *options* can be included in *expr*, enclosed in single quotes:

P[*n*]	Current page number, right-justified in a field of *n* blanks (default is 4).
PN	Current page number, left-justified.
L	Carriage return and line feed.
T	Current time and date.
D	Current date.

* On some systems the HEADING statement is also necessary for initializing page parameters for an output device.

N Do not wait for carriage return at end of page.

C Center the line.

Multiple options can be specified within one set of single quotes.

Multiple HEADING statements can be used to change the heading within a single program. The first HEADING statement initializes the page parameters for the program; each subsequent HEADING statement will not take effect until the next page of output. Pagination begins with page 1 and increments automatically on generation of each new page.

If the output device is the screen, initializing the page parameters also enables the no-page feature, which suspends output at the end of each page until a carriage return is input. This feature can be disabled by the N option to the HEADING statement. Unlike the other options to HEADING, the N option will take effect on the current page of output. The N option is usually used when output is not going to the terminal but is being routed through the auxiliary port to the printer.

The HEADING, FOOTING, and PAGE statements affect the same output device that the PRINT statement does. The PRINTER statement toggles the output device between the terminal screen and the printer. If multiple print units are used, the HEADING, FOOTING, and PAGE statements affect only print unit 0 (the default).

Example

In the following application the HEADING statement sets up a heading on each page with "Status Report" in the left corner and the page number on the right. The SPACE function is used to generate the correct number of spaces in the heading.

```
HEADING "Status Report":SPACE(57):"Page 'PC' "
```

ICONV( ): Convert data from external to internal format.

The ICONV function converts data from external format to internal format, according to the conversion code specified.

ICONV(*expr, code* **)**

expr an expression evaluating to the data to be converted.

code an expression evaluating to the conversion code, as described below.

The ICONV function is intended for storing data which would be more consistent or more flexible in internal format.

For example, the internal format for the date is the number of days since December 31, 1967, which is considered day 0. All dates after December 31, 1967, are positive numbers representing the number of days that have elapsed since then. All dates prior to day 0 are negative numbers representing the number of days prior to this date.

When storing a date, it is preferable to store it in internal format, which makes calculations on that date easier to perform. To convert back from internal format to external format, use the OCONV function.

The conversion codes for ICONV correspond to the conversion codes used in ACCESS. Among the more common codes used in Pick BASIC are:

D	Convert date to internal format.
MT[H][S]	Convert time to internal format.
	H 12-hour format.
	S include seconds.
MX	Convert hexadecimal to ASCII representation.

Among the ACCESS codes that *cannot* be called by the ICONV function are "F", "A", and "S". For more information about conversion codes, see *Pick ACCESS: A Guide to the SMA/RETRIEVAL Language.*

Example

To convert the variable TIME into internal format, the code would read:

```
CURR.TIME = ICONV(CURR.TIME, "MT")
```

IF: Perform conditional execution.

The IF construct allows execution of a statement or series of statements if the calculated expression is true, or of a separate set of statements if it is false. The syntax can take a number of forms, depending on how many lines are taken by the complete IF...THEN...ELSE construct. Syntax is fully explained in the next section. The single-line form of the syntax is as follows:

IF *expr* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

expr is any Pick BASIC expression to be calculated for its logical value.

statements is a statement or set of statements to be executed conditionally.

The IF construct calculates the given expression for its logical values. The expression is false if it evaluates to 0 or the null string; it is true if it evaluates to anything else. If the expression is true, the statements following THEN are executed; if the expression is false, the statements following the ELSE are executed, or if there is no ELSE clause, the program continues with the next executable statement.

Both the THEN clause and the ELSE clause are optional; however, one or the other must be included.

IF constructs can be nested. However, it is recommended to use a CASE construct instead, if possible.

Statement Syntax

Although the logistics of the IF construct are relatively simple, the syntax is very exact. The simplest form of the IF construct is the single-line form given in the preceding section. It is possible, however, to include multiple statements in either the THEN or the ELSE clauses. In such cases the program is much easier to read if each statement is entered on its own line. The END statement must be used as a terminator for multiline THEN and ELSE clauses; END should *not* be used, however, to terminate single-line THEN or ELSE clauses.

Single-line and multiline syntax can be combined. The following syntax forms are possible:

```
IF expr THEN statements ELSE
      statements
END

IF expr THEN
      statements
END ELSE statements
```

The full form of the IF construct with multiline THEN and multiline ELSE clauses is as follows:

```
IF expr { THEN
      statements
END [ ELSE
      statements
END ] | ELSE
      statements
END }
```

When multiline THEN and ELSE clauses are used, the following restrictions apply:

- Neither THEN nor ELSE can begin a program line.* For example, the following construct:

  ```
  IF ANSWER="Y"
     THEN...
  ```

  results in an error message at compile time.

- When the statements following THEN or ELSE are kept on a single line, they must be separated by a semicolon (;). That is, the following construct is correct:

  ```
  IF PROFIT THEN GOSUB 100; PRINT PROFIT ELSE GOSUB
                                        200; PRINT LOSS
  ```

- When the statements following the THEN or ELSE are written on more than one line, THEN or ELSE must be the last word on its line

---

* Except on Prime INFORMATION and uniVerse systems, where THEN and ELSE *can* begin a program line.

and an END statement must end the set of statements. For example, the above example can be written:

```
IF PROFIT THEN
   GOSUB 100
   PRINT PROFIT
END ELSE
   GOSUB 200
   PRINT LOSS
END
```

Example

In the following application, IF constructs are nested to calculate the winner in a game of blackjack. It is sometimes difficult to determine which END statement belongs with which THEN or ELSE. A CASE statement would perhaps be more appropriate to this function. See the CASE statement for more information.

```
IF DEALERSCORE > 21 THEN
   PRINT "I WENT OVER.  YOU WIN."
   YOURWINS = YOURWINS + 1
END ELSE
   IF NOT(DEALERSCORE < YOURSCORE) THEN
      PRINT "MY SCORE IS " : DEALERSCORE : ".  I WIN."
      IF DEALERSCORE = YOURSCORE THEN
         PRINT "" ,"HOUSE RULES-DEALER ALWAYS WINS IN A TIE."
         MYWINS + = 1
      END
   END ELSE
      IF NOT(HIT = 11) THEN
         PRINT "MY SCORE IS " : DEALERSCORE : ". I HAVE TO HOLD. "
         PRINT "YOU WIN."
         YOURWINS + = 1
      END ELSE PRINT "5 CARDS.  I WIN."; MYWINS + = 1
   END
END
```

INCLUDE: Read in source code from another file item.

The INCLUDE statement allows source code to be read in from another file item.

INCLUDE [*filename*] *item-ID*

filename the name of the file containing the item. If *filename* is omitted, the current file is assumed.

item-ID the item ID of the item containing the source code

The INCLUDE statement directs the compiler to read in source code from the specified file item and compile it as if it were written in the current item. The $INSERT and $INCLUDE statements are functionally similar to the INCLUDE statement.

The INCLUDE statement differs from the $CHAIN statement in that the compiler returns to the main item and continues compiling with the statement following the INCLUDE.

The INCLUDE statement is particularly useful for reading in items containing COMMON and EQUATE statements, or any statements which a programmer might want to be consistent among several different programs. Be careful, however, of naming conflicts among different file items.

INCLUDE statements can be nested; that is, a program can INCLUDE a file item which INCLUDEs another file item. However, the total object code when compiled cannot exceed the maximum item size supported by your system.

If the source code read in through an INCLUDE statement generates a run-time error message, the error message will display only the number of the line which contains the INCLUDE statement. The line numbers from the external file item are not kept in the object code.

Examples

To read in the source code written in item ID SETUP in file BP, the code might read:

```
INCLUDE BP SETUP
```

In the following application, the INCLUDE statement is used at the beginning of a program to read in common variables, equated constants, and the part of the program which opens the file.

```
INCLUDE STARTUP
PRINT "ENTER THE CUSTOMER ID : ":
INPUT ID
MATREAD PHONE.ARRAY FROM CUSTFILE, ID ELSE
   PRINT "CANNOT READ RECORD!"
   STOP
END
```

The file item STARTUP contains the text:

```
COMMON PHONE.ARRAY(10), PHONEREC
EQUATE TRUE TO 1, FALSE TO 0, AM TO CHAR(254)
   PROMPT " "
OPEN "CUSTOMERS" TO CUSTFILE ELSE
   ABORT 201, "CUSTOMERS"
END
```

INDEX( ): Return the position of a substring within a string.

The INDEX function searches through a string for a specified substring and returns the starting column position of the substring.

INDEX(*string,substring,n* **)**

string	an expression evaluating to the string to be searched through.
substring	an expression evaluating to the substring to be searched for.
n	an expression evaluating to the occurrence of the substring to search for.

The INDEX function returns the starting column position for a specified occurrence of a substring in a string.

If the *n*th occurrence of the substring is found within the string, the starting column position of the substring is returned. If *substring* is null, 1 is returned. If the specified occurrence of the substring cannot be found, a value of zero is returned.

Examples

If the string STRING contains "NOW IS THE TIME", to assign the column number of the beginning of the third word ("THE") to the variable POS, the code would read:

```
EQU BLANK TO " "
   .
   .
   .
POS = INDEX(STRING, BLANK, 3)
```

In this instance POS would contain "8".

In the following application the INDEX function is used within a global change of a string. The LEN function is used to determine the length of the original string. Within a LOOP, the INDEX function returns the column at which the string next occurs.

```
PRINT "GLOBAL CHANGE : "
PRINT
PRINT "STRING TO CHANGE : " :
INPUT ORIG.STR
PRINT "CHANGE TO : " :
INPUT NEW.STR
LENGTH = LEN(ORIG.STR)
LOOP
   POS = INDEX(RECORD , ORIG.STR , 1)
UNTIL POS = 0 DO
   RECORD = [1,POS–1] : NEW.STR : RECORD [POS + LENGTH,9999]
REPEAT
```

In the next example the INDEX function is used with a formatted screen menu. In the ON ... GOSUB statement, the INDEX function returns a 1, 2, or 3, which forces a branch to the first, second, or third subroutine listed.

```
EQUATE TRUE TO 1 , FALSE TO 0
PRINT @(–1) : @(4 , 4) : "WOULD YOU LIKE TO : " :
PRINT @(4 , 7) : "ADD A RESERVATION : " :
PRINT @(4 , 8) : "CHANGE A RESERVATION : " :
PRINT @(4 , 9) : "REMOVE A RESERVATION : " :
PRINT @(0 , 23) : "ENTER A , C OR R : " :
LOOP
   VALID = TRUE
   PRINT @(18 , 23) : @(–4) :
   INPUT CHOICE , 1 :
   IF COUNT("ACR" , CHOICE) # 1 THEN
      VALID = FALSE
   END
UNTIL VALID DO REPEAT
ON INDEX("ACR" , CHOICE , 1) GOSUB ADD , CHANGE , REMOVE
   .
   .
   .
```

INPUT: Request terminal input.

The INPUT statement requests terminal input from the user and assigns the input to a variable.

INPUT *var* [= *expr*] [,*length-expr*] [,*fillchars*] [_] [:]

var [= *expr*]	assign input data to variable *var*. If *expr* is specified, *var* is initially set to *expr* and the current value of *expr* is displayed on the screen as a default value. The user can then modify the default value before pressing the RETURN key, or accept it by pressing the RETURN key without modification.
length-expr	an expression to be interpreted as the maximum number of characters to be input from the terminal. When this number of characters is input, the program automatically interprets a carriage return. If *length-expr* evaluates to 0 or 1, only one character is accepted. If the user does not wish to fill the input field, a carriage return can be entered manually.
fillchars	an expression evaluating to a string of one, two, or three characters. The first character is taken as a format mask for the input field, and the second is used as an overstrike filler for portions of the field left unwritten. The cursor returns to the end of the input data unless there is a third fill character, in which case the cursor remains at the end of the formatted field.
_	an underscore specifies that a carriage return must be entered by the user, even if the input length equals *length-expr*. If the user tries to exceed the maximum length, a bell rings.
:	a colon suppresses the automatic line feed and carriage return when the value is input.

The INPUT statement pauses program execution and prompts for a response. Data entered at the terminal becomes the assigned value of the specified variable *var*.

Use the optional *length-expr* to specify the maximum length, or number of characters, allowed as input. When the specified number of characters is entered, an automatic carriage return and line feed are executed. An underscore (_) disables the automatic carriage return and line feed, and instead forces a beep to sound if the user tries to exceed the maximum

length. When the maximum length is exceeded, all subsequent characters are ignored except for the standard editing control characters, listed below.

Editing Control Characters.

Character	Function
CTRL-H or BACKSPACE	Erase one character.
CTRL-R	Redisplay current input field.
CTRL-W	Erase one word.
CTRL-X	Clear entire input.

A fill character can be used to outline the length of the input field for the user. This allows the user to see the length of the input field. If a second character is specified, it is used to fill the unused portion of the field after the user presses the RETURN key. If a third character is specified, the cursor remains at the end of the formatted field after the excess part of the input field is overstriken; otherwise, it returns to the end of the input data. (The actual value of the third character is not significant.)

The INPUT statement causes only the prompt character to be printed on the screen. The default prompt character is a question mark (?), but it can be reassigned with the PROMPT statement. See the PROMPT statement for more information. A PRINT statement must be used before the INPUT statement in order to tell the user what sort of input is required.

Examples

To ask the user to supply a name, the code might read:

```
PRINT "ENTER YOUR NAME"
INPUT NAME
```

If the name cannot exceed 20 characters, the input line might read:

```
INPUT NAME, 20_
```

If the user tries to enter more than 20 characters, a beep sounds.

To prompt the user to respond either Y for yes or N for no, the code might read:

```
PRINT "DO YOU WANT TO EXIT (Y OR N)?"
INPUT ANSWER, 1
```

The first character the user types is accepted, and the program continues. No carriage return is necessary.

In the following application, a customer is prompted for changes in his billing information. The example also makes use of the PRINT statement with the @ function, generating a formatted screen.

```
LOOP
   PRINT @(10,10) : "ENTER YOUR ACCOUNT NUMBER : " :
   INPUT ACCT, 6 , "?"_
WHILE NOT(NUM(ACCT)) OR LEN(ACCT) # 6 DO
   PRINT @(0,23) : "PLEASE ENTER A 6-DIGIT ACCOUNT NUMBER" :
REPEAT
ON.FILE = 1
READ CUST.REC FROM CUSTFILE,ACCT ELSE
   ON.FILE = 0
END
IF NOT(ON.FILE) THEN
   CUST.REC = ""
   CHANGE = 1
END ELSE
   PRINT @(–1) : @(10,10) : "HAS YOUR BILLING CHANGED (Y OR N)" :
   INPUT ANSWER,1
END
IF ANSWER = 'Y' THEN
   CHANGE = 1
END ELSE
   CHANGE = 0
END
IF CHANGE THEN
   PRINT @(–1) :
   CORRECT = "0"
   LOOP
      PRINT @(5,5) : "MODIFY AS NEEDED : " :
      PRINT @(10,10) : "NAME :  " : @(30,10)  :
      INPUT NAME = CUST.REC<1> :
      PRINT @(10,12) : "ADDRESS :  " : @(30,12)  :
      INPUT ADDRESS = CUST.REC<2> :
      PRINT @(10,14) : "CITY,STATE,ZIP :  " : @(30,14)  :
      INPUT CSZ = CUST.REC<3> :
      PRINT
      CUST.REC<1> = NAME
      CUST.REC<2> = ADDRESS
      CUST.REC<3> = CSZ
      PRINT @(–1) : @(5,5) : "NOW ON FILE : "
      FOR I = 1 TO 3
         PRINT @(15,10 + 2 * I ) : CUST.REC<I> :
      NEXT I
      PRINT @(0,23) : "IS THIS CORRECT (Y OR N)?"  :
      INPUT ANSWER,1
      IF ANSWER = 'Y' THEN
         CORRECT = 1
      END ELSE
         CORRECT = 0
```

```
        END
    UNTIL CORRECT DO REPEAT
END
```

In the preceding example, the INPUT statement is first used to record the customer's account number. The account number must be a number of six digits. For clarification, six question marks are printed to the screen after the prompt, and the user can then write over those question marks. Since it is important that the correct account number be entered, the automatic line feed and carriage return is suppressed with the underscore (_). What the user sees is:

```
ENTER YOUR ACCOUNT NUMBER: ??????
```

If the account is already on file, the INPUT statement is next used to find out if the customer's billing information has changed. Since a yes or no answer will suffice, only one character is accepted and the automatic line feed and carriage return is enabled.

The INPUT statement is then used to allow the customer to change his or her billing information. For each field the current contents are printed and the user has the option of accepting it or modifying it. If the correct information is on file, only pressing the RETURN key is necessary; if the information needs to be changed, the user can use the backspace key to modify the field before pressing the RETURN key. For example, for the first such field what the user might see (with user input in bold) is:

```
MODIFY AS NEEDED:
    NAME: JOHN SMITH_
```

The prompt, represented here by the underscore, waits at the end of the input field. The customer can now use the backspace key to erase the field and enter a different name, or change the one already on file. When the customer is satisfied with the contents of the "NAME" field, the RETURN key can be pressed to record it. This process repeats for each field. At the end of the loop, the modified contents of the array are printed for the user to verify before continuing with the program.

INPUT @: Request terminal input at a specified location.

The INPUT @ statement is a variation of the INPUT statement. INPUT @ maintains formatted screens and provides format masking.

INPUT @(*x,y*) *var* [,*length*] [*format-expr*]

var	the name of the variable to which user input will be assigned.
length	the maximum length of the field to be input. Once *length* characters have been input, an automatic carriage return and line feed are assumed, as in the standard INPUT statement.
format-expr	an expression evaluating to the formatting codes to be used. The syntax for *format-expr* is described below.

The INPUT @ statement is in many ways an enhancement on the INPUT statement in that it allows you to specify the exact location on the screen at which the user is to be prompted. It also provides direct format verification so that input will not be accepted if it does not fit the specified format.

Using the standard INPUT statement, you would have to precede the INPUT statement with the proper PRINT statement (using the @ function) to position the cursor at a specific position on the screen. Also, with the standard INPUT statement, a loop would often have to be used with several tests to ensure that the input matches a particular format.

When the user is prompted, the prompt character appears one space to the left of the position specified by the coordinate (*x*,*y*), and the current value of *var* (if any) is printed (in the specified format). The user can accept the current value by pressing the RETURN key, or can enter another value. The user will continue to be prompted until the input fits the requested format. When input matches the specified format, it is printed in its external format and then stored in the variable *var* in its internal format.

If the original value of *var* does not fit the required format, no error is output.

☞ The INPUT @ statement does not accept any of the other screen manipulation codes available with the @ function (such as clearing the screen or providing text in a standout mode).

Example

In the following application the INPUT @ function is used to prompt for the salary and starting date of a new employee.

```
**** NEXT 3 LINES : PRINT THE 3 FORMATTED ROWS FOR INPUT
PRINT @(-1) : @(10,10) : "YOUR NAME" :
PRINT @(10,12) : "NEW SALARY" :
PRINT @(10,15) : "EFFECTIVE DATE" :
**** INPUT FOR NAME AT PROPER COLUMN, ROW
INPUT @(30,10) NAME
**** INPUT FOR SALARY, WITH MONETARY FORMAT
INPUT @(30,12) SALARY "R2,"
**** INPUT FOR DATE EFFECTIVE, WITH DATE FORMAT
INPUT @(30,15) DATE "D"
```

INPUTCLEAR: Clear the type-ahead buffer.*

The INPUTCLEAR statement clears the type-ahead buffer for the process executing the program.

INPUTCLEAR

Any data in the type-ahead buffer is erased.

Example

In the following application a subroutine GO.CALC is called, which can take several minutes to execute. The user can become impatient during this time and attempt to abort the program by typing control sequences. The GO.CALC subroutine is therefore followed by an INPUTCLEAR statement, to ensure that the when the subroutine is finished, the user has a chance to read the results before continuing with the program.

```
BREAK OFF
ECHO OFF
GOSUB GO.CALC
INPUTCLEAR
PRINT "PRESS ANY KEY TO CONTINUE: ":
```

* Not included in the SMA standards.

INPUTERR: Display an error message on last line of screen.*

The INPUTERR statement prints the specified text at the last line of the screen. It is used in conjunction with the INPUT @ statement.

INPUTERR *print-expr*

print-expr a print expression, optionally combined with commas for tabulation as in the PRINT statement. See the PRINT statement for more information on the format for print expressions.

An entry to the next INPUT @ statement erases any messages previously sent to the bottom of the screen with INPUTERR.

The INPUTERR statement allows the programmer to provide additional error messages to an INPUT @ statement. Like the error messages built into the INPUT @ statement, the message printed by INPUT @ is cleared from the screen when a response to INPUT @ is received.

Examples

To print the message "ERROR! NUMERIC DATA EXPECTED" on the bottom of the screen, the code would read:

```
INPUTERR "ERROR! NUMERIC DATA EXPECTED"
```

The following program asks for the closing date for a property transaction. The closing must occur within 45 days of approving the mortgage, so after the masked INPUT @ statement verifies that a date is entered, the date is verified as being within 45 days from today, and the INPUTERR statement is used if it is not. The program continues to prompt the user for a closing date until a satisfactory one is entered.

```
EQUATE TRUE TO 1, FALSE TO 0
PROMPT " "
PRINT @(0,0) : "ENTER THE CLOSING DATE : " :
LOOP
   INPUT @(24,0) CLOS.DATE "D"
   TODAY = DATE( )
   IF CLOS.DATE >= TODAY AND CLOS.DATE < =TODAY + 45 THEN
      VALID = TRUE
   END ELSE
```

* Not included in the SMA standards.

```
        INPUTERR "CLOSING MUST BE COMPLETED WITHIN 45 DAYS"
        VALID = FALSE
    END
UNTIL VALID DO REPEAT
```

INPUTIF: Capture terminal input from the type-ahead buffer.*

The INPUTIF statement captures input from the type-ahead buffer and assigns the input to a variable.

INPUTIF *var* [= *expr*] [,*length-expr*] [,*fillchars*] [_] [:] { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

var [= *expr*]	assign input data to variable *var*. If *expr* is specified, *var* is initially set to the value of *expr*, and characters in the type-ahead buffer are appended to that value.
length-expr	an expression to be interpreted as the maximum number of characters to assign to *var*.
fillchars	an expression evaluating to a string of one, two, or three characters. The first character is taken as a background mask for the input field, and the second is used as an overstrike filler for portions of the field left unwritten. The cursor will return to the end of the input data unless there is a third fill character, in which case the cursor will remain at the end of the formatted field.
_	an underscore specifies that a carriage return must be entered by the user, even if the input length equals *length-expr*. If the user tries to exceed the maximum length, a bell will ring.
:	a colon suppresses the automatic line feed and carriage return when the value is input.
THEN *statements*	execute *statements* if the type-ahead buffer is not empty. For details about the syntax of THEN clauses, see the IF statement.

* Not included in the SMA standards.

ELSE *statements* execute *statements* if the type-ahead buffer is empty. For details about the syntax of ELSE clauses, see the IF statement.

The INPUTIF statement checks the type-ahead buffer, and if it is non-null, assigns the variable *var* initially with the contents of the type-ahead buffer and executes the THEN part of the statement. Unless a maximum length (*length-expr*) has been specified with no underscore (_), pressing the RETURN key is necessary. If there is no data in the type-ahead buffer, the variable is not assigned and the ELSE clause is executed.

If the type-ahead buffer is empty, the variable is not assigned and the user is not prompted. However, if the type-ahead buffer is not empty, but a carriage return is required to complete the input, the user is prompted with the type-ahead buffer input.

The INPUTIF statement includes many features reflecting those of the standard INPUT statement. See the INPUT statement for more information.

Examples

To assign the variable TYPEA with the current contents of the type-ahead buffer and print "TYPE AHEAD EMPTY" if no data is received, the code would read:

```
INPUTIF TYPEA ELSE
   PRINT "TYPE AHEAD EMPTY"
END
```

In the following application a requested calculation can take several minutes to complete. The user is allowed to use the ESC key to exit from the program, and the INPUTIF statement is repeated at every iteration to see if the ESC key was entered in the type-ahead buffer.

```
PRINT "ENTER <ESC> TO EXIT."
LOOP
   .
   .
   .
   LOOP
      INPUTIF STOPVAR,1 : THEN
         IF STOPVAR = CHAR(27) THEN
            GOSUB EXIT
         END
         TA.EMPTY = 0
      END ELSE
         TA.EMPTY = 1
```

```
    END
  UNTIL TA.EMPTY DO REPEAT
  .
  .
  .
UNTIL ...DO REPEAT
```

INPUTNULL: Establish character as null in input.*

The INPUTNULL statement defines a character to be interpreted as the null string if sent as a response to an INPUT @ statement.

INPUTNULL *expr*

expr an expression evaluating to a single character. Any extraneous characters in the expression are ignored.

The INPUTNULL statement allows the programmer to specify a character to be used as an exit from a masked INPUT @ statement.

Normally the INPUT @ statement continues to prompt the user until a response matching the format expression is entered. The only exceptions to this are the characters specified by the INPUTTRAP statement and the null string. The INPUTNULL statement allows the programmer to specify a character to be interpreted as the null string, thus providing an exit from INPUT @.

Example

In the following application the user is prompted for a price, and the INPUTNULL statement is used to allow the user to specify 0 by entering an X.

```
INPUTNULL " X "
PRINT @(-1) : "ENTER A DOLLAR AMOUNT"
INPUTERR 'ENTER "X" FOR A ZERO AMOUNT'
INPUT @ (23,0) VALUE "L2,$"
```

* Not included in the SMA standards.

INPUTTRAP: Transfer control of program according to input data.*

The INPUTTRAP statement establishes characters that, when input as full responses to an INPUT @ statement, force a branch to the specified labels.

INPUTTRAP *expr* { **GOTO** | **GOSUB** } *label1*, *label2*, *label3*, ...

expr	an expression evaluating to characters to be searched for in the input.
GOTO	branch to the specified label, without returning.
GOSUB	branch to the subroutine starting at the specified label, and return to the line following the INPUTTRAP statement when a RETURN statement is encountered.
*label*n	statement labels to be branched to. If the input is the *n*th character in *expr*, the *n*th label is branched to.

The INPUTTRAP statement provides an escape from masked input statements. It is used to declare characters which will be accepted as responses to an INPUT @ statement, specifying the statement labels to which each character will branch.

The structure of the INPUTTRAP statement is analogous to the ON...GOTO and ON...GOSUB statements, except that the branching variable is taken directly from input instead of from the evaluation of an expression. The primary purpose of the INPUTTRAP statement is that when an INPUT @ statement is used with format expressions, it will continue to prompt for input until the input matches the specified format. The INPUTTRAP statement allows the programmer to provide an exit from the looping structure of the INPUT @ statement.

If the INPUTTRAP statement is used with the GOSUB keyword, program execution returns to the line following the INPUTTRAP statement when the subroutine is finished, *not* to the line following the INPUT @ statement.

* Not included in the SMA standards.

Example

In the following application the user is prompted for a dollar amount. If the user wishes to exit from the program at this point, or would like to start the program over again, the characters "E" or "Q" can be entered as a response. The INPUTTRAP statement ensures that if the characters "E" or "Q" are entered, the program will branch directly to the specified statement labels.

```
PRINT @(0,3) : "ENTER AMOUNT OF PAYMENT :  " :
PRINT @(0,23) : "ENTER 'E' IF YOU MADE AN ERROR, 'Q' TO QUIT" :
INPUTTRAP "EQ" GOTO ASK.QUESTIONS, EXIT
INPUT @(26,3) PAYMENT "L2,$"
```

INS: Insert an attribute, value, or subvalue into an array.*

The INS statement inserts an attribute, value, or subvalue into a specified position in a dynamic array.

INS *expr* **BEFORE** *array* **<***attr#* [**,***value#* [**,***subval#*]]**>**

expr	an expression evaluating to the data to be inserted.
array	the dynamic array to be changed.
attr#	an expression evaluating to the attribute number. If *attr#* is equal to 0, the entire array is replaced. If *attr#* evaluates to a negative number, the attribute is appended to the end of the array. If *attr#* evaluates to a number greater than the number of attributes in the array, the data is appended to the end of the array, with the appropriate number of empty attributes inserted.
value#	an expression evaluating to the value number. If *value#* is omitted or equal to 0, the value is inserted before the attribute. If *value#* evaluates to a negative number, it is appended at the end of the attribute. If *value#* evaluates to a number greater than the number of values in the attribute, the data is appended to the end of the attribute, with the appropriate number of empty values inserted.

* Not included in the SMA standards.

subval# an expression evaluating to the subvalue number. If *subval#* is omitted or equal to 0, the subvalue is inserted before the value. If *subval#* evaluates to a negative number, it is appended to the end of the value. If *subval#* evaluates to a number greater than the number of subvalues in the value, the data is appended to the end of the value, with the appropriate number of empty subvalues inserted.

INS is the statement equivalent of the INSERT function.

If an attribute, value, or subvalue expression evaluates to a noninteger value, it is truncated to an integer value.

The LOCATE statement can be particularly useful with the INS statement to insert array elements in an ascending or descending order. See the LOCATE statement for more information.

Example

In the following application the user creates an alphabetical list of each item ID in the file AIRPORTS. The file is selected, and when each item ID is read, the LOCATE statement is used to find its alphabetical position. The INS statement is then used to insert the current ID in the proper position.

```
EQUATE TRUE TO 1, FALSE TO 0
OPEN "AIRPORTS" TO AIRPORTFILE ELSE
   ABORT 201, "AIRPORTS"
END
SELECT AIRPORTFILE TO LIST
ALPH.LIST = ""
END.OF.LIST = FALSE
LOOP
   READNEXT ID FROM LIST ELSE
      END.OF.LIST = TRUE
   END
UNTIL END.OF.LIST DO
   LOCATE ID IN ALPH.LIST BY "AL" SETTING POSITION THEN
      PRINT ID : " DUPLICATE ENTRY! POSSIBLE FILE CORRUPTION"
      ABORT
   END ELSE
      INS ID BEFORE ALPH.LIST<POSITION>
   END
REPEAT
```

INSERT( ): Insert an attribute, value, or subvalue into an array.

The INSERT function inserts an attribute, value, or subvalue into a specified position in a dynamic array. INSERT( ) is the function equivalent of the INS statement.

array = **INSERT(***array,attr#* [,*value#* [,*subval#*]] { , | ; } *expr* **)**

array — the dynamic array to be changed.

attr# — an expression evaluating to the attribute number. If *attr#* is equal to 0, the entire array is replaced. If *attr#* evaluates to a negative number, the attribute is appended to the end of the array. If *attr#* evaluates to a number greater than the number of attributes in the array, the data is appended to the end of the array, with the appropriate number of empty attributes inserted.

value# — an expression evaluating to the value number. If *value#* is omitted or equal to 0, the value is inserted before the attribute. If *value#* evaluates to a negative number, it is appended at the end of the attribute. If *value#* evaluates to a number greater than the number of values in the attribute, the data is appended to the end of the attribute, with the appropriate number of empty values inserted.

subval# — an expression evaluating to the subvalue number. If *subval#* is omitted or equal to 0, the subvalue is inserted before the value. If *subval#* evaluates to a negative number, it is appended to the end of the value. If *subval#* evaluates to a number greater than the number of subvalues in the value, the data is appended to the end of the value, with the appropriate number of empty subvalues inserted.

expr — an expression evaluating to the data to be inserted.

Either a comma or a semicolon can separate the subvalue expression from the replacement data, but if the subvalue is omitted, then a semicolon must be used.

If an attribute, value, or subvalue expression evaluates to a noninteger value, it is truncated to an integer value.

The LOCATE statement can be particularly useful with the INSERT function to insert array elements in an ascending or descending order. See the LOCATE statement for more information.

Example

In the following application the user creates an alphabetical list of each item ID in the file AIRPORTS. The file is selected, and when each item ID is read, the LOCATE statement is used to find its alphabetical position. The INSERT function is then used to insert the current ID in the proper position.

```
EQUATE TRUE TO 1, FALSE TO 0
OPEN "AIRPORTS" TO AIRPORTFILE ELSE
   ABORT 201, "AIRPORTS"
END
SELECT AIRPORTFILE TO LIST
ALPH.LIST = ""
END.OF.LIST = FALSE
LOOP
   READNEXT ID FROM LIST ELSE
      END.OF.LIST = TRUE
   END
UNTIL END.OF.LIST DO
   LOCATE ID IN ALPH.LIST BY "AL" SETTING POSITION THEN
      PRINT ID : " DUPLICATE ENTRY! POSSIBLE FILE CORRUPTION"
      ABORT
   END ELSE
      ALPH.LIST = INSERT(ALPH.LIST , POSITION; ID)
   END
REPEAT
```

INT( ): Return the integer portion of an expression.

The INT function truncates an expression to its integer value.

INT(*expr* **)**

expr an expression evaluating to a numeric value.

Use the INT function to return the integer portion of an expression *expr*. The fractional portion of the value is truncated (not rounded), and the integer portion remaining is returned.

Example

In the following application the INT function is used to calculate the number of items that can be bought for a given amount at a set price:

```
PRINT "HOW MUCH DO YOU HAVE TO SPEND? $" :
INPUT AVAILABLE
NUMBER = INT(AVAILABLE/PRICE)
PRINT "EACH ITEM COSTS " : PRICE"2,$" : "."
PRINT "FOR " :AVAILABLE"2,$" : "YOU CAN GET " : NUMBER :"ITEMS."
```

LEN( ): Return the length of an expression.

The LEN function returns the length of a string expression.

LEN(*expr* **)**

expr any string expression.

The LEN function reports the number of characters in a string. All blank spaces (including trailing blanks) are included in the calculation.

Examples

To assign the number of characters in the string STRING to the variable LENGTH, the code would read:

```
LENGTH = LEN(STRING)
```

If STRING contains "HI, MOM", LENGTH will contain "7".

In the following application the INDEX function is used within a global change of a string. The LEN function is used to determine the length of the original string.

```
PRINT "GLOBAL CHANGE : "
PRINT
PRINT "STRING TO CHANGE : " :
INPUT ORIG.STR
PRINT "CHANGE TO : " :
INPUT NEW.STR
LENGTH = LEN(ORIG.STR)
```

```
LOOP
   POS = INDEX(RECORD , ORIG.STR , 1)
UNTIL POS = 0 DO
   RECORD = [1,POS–1] : NEW.STR : RECORD [POS + LENGTH,9999]
REPEAT
```

LN( ): Return the natural log of an expression.

The LN function returns the log base *e* of an expression. The LN function is the inverse of the EXP function.

LN (*expr***)**

expr an expression evaluating to a numeric value.

The LN function returns the natural log of an expression. The natural log is the the log base *e* of a given expression. *e* is an irrational number, with an approximate value of 2.7183.

Example

To assign the natural log of 10 to the variable NLOG, the code would read:

```
NLOG = LN(10)
```

LOCATE( ): Find an attribute, value, or subvalue in a string.

The LOCATE statement is used to find the occurrence of an attribute, value, or subvalue within a specified dynamic array. If it is not found, LOCATE finds the proper position at which the string should be inserted.

There are two forms of the LOCATE statement, both using the LOCATE keyword. The second form, which is the only version included in the SMA standards, allows the programmer to specify where in the dynamic array to start searching, and uses the angle bracket syntax similar to that of the EXTRACT function.

LOCATE(*string,array* [**,***attr#* [**,***val#*]] ; *var* [; *'seq '*]) { **THEN**
statements [**ELSE** *statements*] | **ELSE** *statements* }

LOCATE *string* **IN** *array* [**<***attr#* [**,***val#*]**>**] **,***start* [**BY** *seq*]
SETTING *var* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

string an expression evaluating to the string to be searched for in the dynamic array.

array an expression evaluating to the dynamic array to be searched.

attr# an expression evaluating to the attribute number to be searched. The first value number matching the string will be placed in *var*. If *attr#* is omitted or equal to 0, the entire dynamic array is searched and the first attribute number which matches the string is placed in *var*.

val# an expression evaluating to the value number to be searched within the specified attribute. The first subvalue number matching the string will be placed in *var*. If *val#* is omitted or equal to 0, the entire attribute is searched and the first value number which matches the string is placed in *var*.

var the variable to be assigned.

start position to start searching with (default is 1). If *string* is in the first *start*–1 positions, it will not be found; however, if *string* is found, *var* will still be assigned a value as if it had started searching with 1.

seq an expression evaluating to the sequence in which elements are sorted. In the first syntax line given above, *seq* must be enclosed in single quotes. Possible values are:

A ascending order.

D descending order.

AL ascending order, left-justified.

DL descending order, left-justified.

AR ascending order, right-justified.

DR descending order, right-justified.

THEN *statements*
: if found, execute the specified statements. For details about the syntax of THEN clauses, see the IF statement.

ELSE *statements*
: if not found, execute the specified statements. For details about the syntax of ELSE clauses, see the IF statement.

The LOCATE statement searches a dynamic array for an attribute, value, or subvalue, and places an integer in *var* depending on whether it is found. If it is found, the integer indicates the position where the expression was found; if it is not found, the integer indicates where it should be inserted.

The LOCATE statement can be used with the INSERT statement: if the string is not found, the number placed in *var* can be used with a subsequent INSERT statement to place the string in the proper sequence.

Unless the BY clause is used, the sequence is assumed to be random: if the string is not found, *var* is assigned the number of the last position plus 1. Thus a subsequent INSERT statement places the string at the end of the sequence. However, if the elements to be searched are already sorted into an ascending or descending ASCII sequence, the BY clause can be used to specify the order to be maintained. If the string is not found and the BY clause is used, *var* is therefore assigned the position which would maintain the order, and a subsequent INSERT statement places the string in its proper place. Note, however, that if the BY clause is used and the elements are not in the expected sequence, an element which is out of order is not found.

Either the THEN or the ELSE clause must be used with the LOCATE statement. A common use of the ELSE clause is to insert the string into the proper position, using the position returned in *var*.

Example

In the following application the user creates an alphabetical list of each item ID in the file AIRPORTS. The file is selected, and each item ID is read with READNEXT. The LOCATE statement does not actually find the ID in the list, but returns into the variable POSITION the position where it should have been found. (If the ID *is* found in the list, the program aborts since it would imply duplicate item IDs.) The INS statement is then used to insert the current ID in the proper position.

```
EQUATE TRUE TO 1, FALSE TO 0
OPEN "AIRPORTS" TO AIRPORTFILE ELSE
   ABORT 201, "AIRPORTS"
END
SELECT AIRPORTFILE TO LIST
ALPH.LIST = ""
END.OF.LIST = FALSE
LOOP
   READNEXT ID FROM LIST ELSE
      END.OF.LIST = TRUE
   END
UNTIL END.OF.LIST DO
   LOCATE ID IN ALPH.LIST BY "AL" SETTING POSITION THEN
      PRINT ID : " DUPLICATE ENTRY! POSSIBLE FILE CORRUPTION"
      ABORT
   END ELSE
      INS ID BEFORE ALPH.LIST<POSITION>
   END
REPEAT
```

The LOCATE statement in this example might also have read:

```
LOCATE( ID, ALPH.LIST; POSITION; "AL") THEN...
```

LOCK: Set an execution lock.

The LOCK statement sets an execution lock for the current process.

LOCK *expr* [**ELSE** *statements*]

expr	*expr* evaluates to a number designating as the lock number.
ELSE *statements*	if a lock has already been set by another user, execute *statements*. If the ELSE clause is not specified, the program pauses until the execution lock is lifted. For details about the syntax of ELSE clauses, see the IF statement.

The significance of each execution lock is designated by the application developer. By establishing a unique execution lock for a specific procedure, the programmer can ensure that two users cannot run the same procedure simultaneously.

At least 64 execution locks are available; some Pick systems support more. The TCL verb WHAT produces a list of all current execution locks and the process numbers which have set them.

For protecting file items, the READU, READVU, and MATREADU statements use a completely different method of item locking.

Execution locks, like item locks, are automatically released at the end of the program. The UNLOCK statement can be used, however, to unlock a procedure before terminating the program.

Example

In the following application the external subroutine REMOTE.MAIL uses a modem to send an electronic message to a remote system and then reports to the user that the message was successfully sent. By using the LOCK statement, it is ensured that two users will not be using the modem at once. In this example, execution lock number 56 is the one that has been established for this subroutine.

The LOCK statement is actually called twice in the example. The first time it is called, an ELSE clause is included to give the user the chance to exit. If the user chooses not to exit, the LOCK statement is called again without an ELSE clause, and the program will wait for the modem to be freed before continuing.

```
SUBROUTINE REMOTE.MAIL (MESSAGE, MODEMLINE)
LOCK 56 ELSE
   PRINT "MODEM IN USE.  ABORT MESSAGE? (Y OR N)":
   INPUT ANSWER , 1
   IF ANSWER = "Y" THEN
      LOCK 56
   END ELSE
      GOSUB EXIT
   END
END
GOSUB 20; * Encumber modem line
GOSUB 30; * Send modem initialization string
   .
   .
   .
UNLOCK 56
RETURN
```

LOOP: Structure for program looping.

The LOOP construct provides a two-tiered structure for repeated execution of a group of statements.

```
LOOP
      [statements ]
{ WHILE | UNTIL } expr DO
      [statements ]
REPEAT
```

The LOOP statement starts a program loop. The first group of statements is executed, and when the WHILE or UNTIL clause is reached, a test is performed. Depending on the result, execution continues either with the second group of statements or with the statements following the REPEAT clause.

The WHILE clause allows a positive condition to be specified: as long as this condition is true (i.e., evaluates to 1), the loop repeats. The UNTIL clause allows a negative condition to be specified: the loop repeats as long as the specified condition is *not* true (i.e., evaluates to 0).

Statements included between the LOOP and the WHILE or UNTIL clause are executed at least once; they are repeated again each time the loop is iterated. The statements between the WHILE and UNTIL are executed only if the condition passes; they are repeated each time the loop is iterated, but only if the condition passes in each iteration.

Although it is possible to exit the loop by means other than the conditional WHILE and UNTIL statements (for example, by using GOTO or GOSUB in the DO statements), it is not recommended to do so.

Example

In the following application, numbers are prompted for and then multiplied. The program continues to prompt for a number until a valid number is entered. The program continues to prompt for numbers to multiply until a zero is entered as the first number.

```
LOOP
   LOOP
      PRINT "ENTER A NUMBER (ENTER 0 TO EXIT) : " :
      INPUT NUM1
   UNTIL NUM(NUM1) DO
      PRINT NUM1 : " : NOT A NUMBER. USE 0 TO EXIT"
```

```
   REPEAT
UNTIL NUM1 = 0 DO
   LOOP
      PRINT "ENTER A SECOND NUMBER :  " :
      INPUT NUM2
   UNTIL NUM(NUM2) DO
      PRINT NUM2 : " :  NOT A NUMBER"
   REPEAT
   PRINT NUM1 : " TIMES " : NUM2 : " IS " : NUM1*NUM2 : "."
PRINT
REPEAT
```

MAT: Assign values to elements of an array.

The MAT statement is used either to assign a single value to all elements of an array, or to copy all elements of one array into another.

MAT *array* = *expr*
MAT *array1* = **MAT** *array2*

MAT *array* = *expr* assign the evaluated value of *expr* to all elements of the array.

MAT *array1* = MAT *array2*
assign each element of *array2* to the corresponding element in *array1* The arrays thus become equivalent. Note that each array must have been dimensioned via the DIM statement to have the same number of elements, although the actual dimensions do not need to be identical. The elements are assigned in consecutive order, regardless of whether the dimensions are the same.

The MAT statement can be used only for assigning a value to all elements of an array or to make two arrays equivalent in data. It cannot be used to perform matrix arithmetic.

In passing a dimensioned array to an external subroutine, the MAT keyword must precede the array name. See the CALL statement for more information.

Example

If the array ARR1 has been dimensioned to be a 12-element vector and the array ARR2 has been dimensioned to a 3 by 4 matrix, ARR1 can be assigned the elements of ARR2 with:

```
MAT ARR1 = MAT ARR2
```

If ARR2 contained:

	1	2	3	4
1	FIRST	SECOND	THIRD	FOURTH
2	FIFTH	SIXTH	SEVENTH	EIGHTH
3	NINTH	TENTH	ELEVENTH	TWELFTH

then ARR1 will contain:

1	FIRST
2	SECOND
3	THIRD
4	FOURTH
5	FIFTH
6	SIXTH
7	SEVENTH
8	EIGHTH
9	NINTH
10	TENTH
11	ELEVENTH
12	TWELFTH

MATBUILD: Create a dynamic array from a dimensioned array.*

The MATBUILD statement writes the consecutive elements of a dimensioned array into a dynamic array. It is the inverse of the MATPARSE statement; that is, a string converted into a dimensioned array with MATPARSE can be reconstructed, using the same delimiters, with MATBUILD.

---

* Not included in the SMA standards.

We give two syntaxes. The first syntax line shows how MATBUILD is implemented on several R83 systems; the second shows how it is implemented on uniVerse systems.

MATBUILD *string* **FROM** *array, delim*

MATBUILD *string* **FROM** *array* [,*start* [,*end*]] [**USING** *delim*]

string	the string or dynamic array to be created.
array	the dimensioned array from which the elements are read.
delim	an expression evaluating to the characters used to delimit elements in *string*. The behavior of the MATBUILD statement is dependent on the number of characters specified as delimiters.
start	the element of the array to begin with. If not specified, or if out of range, the default is 1.
end	the element of the array to end with. If not specified, or if out of range, the default is the size of the array. In IDEAL and INFORMATION flavor accounts on uniVerse systems, to retain overflow elements from *array*, *end* must be either less than or equal to zero, or greater than the size of *array*.
USING	specifies the characters used to delimit elements in *string*. If not specified, the default is an attribute mark. To specify a null delimiter, specify USING with no option.

The behavior of the MATBUILD statement is dependent on the number of characters specified as delimiters:

Number of Delimiters	Result
0 (*delim* = "")	Each element of *array* is placed into *string* without delimiters.
1	Each element of *array* is placed into *string* separated by the delimiter character *delim*.
2 or more	The fields of *array* will be loaded in the dynamic array. They will be alternating data and delimiter elements.

On uniVerse systems, PICK flavor arrays store overflow elements in the last element (see example 3 below); IDEAL and INFORMATION flavors store overflow elements in element zero.

Examples

Using Two Delimiters

If array ARR1 is dimensioned to be a 5 by 2 matrix and ARR1 contains:

	1	2
1	THIS	-
2	IS	-
3	A	-
4	STRING	
5		

then:

```
MATBUILD STRING FROM ARR1, "--"
```

produces as STRING:

```
THIS-IS-A-STRING
```

Using One Delimiter

If ARR1 contains:

	1	2
1	THIS	IS
2	A	STRING
3		
4		
5		

then:

```
MATBUILD STRING FROM ARR1, "-"
```

will also produce:

```
THIS-IS-A-STRING
```

Using No Delimiters

If ARR1 contains:

	1	2
1	T	H
2	I	S
3	-	I
4	S	-
5	A	-STRING

then:

```
MATBUILD STRING FROM ARR1, ""
```

will produce:

```
THIS-IS-A-STRING
```

The three arrays shown above are representative of arrays which were composed with the MATPARSE statement using the same delimiters.

MATPARSE: Create a dimensioned array from a dynamic array.*

The MATPARSE statement separates the elements of a string expression into consecutive elements of a dimensioned array. It is the inverse of the MATBUILD statement. The array must be named and dimensioned in a DIM or COMMON statement before it is used in this statement.

We give two syntaxes. The first syntax line shows how MATPARSE is implemented on several R83 systems and on Prime INFORMATION; the second shows how it is implemented on uniVerse systems.

MATPARSE *array* **FROM** *string, delim* [**SETTING** *var*]

MATPARSE *array* [,*start* [,*end*]] **FROM** *string* [{ , | **USING**} *delim*] [**SETTING** *var*]

array	the dimensioned array to be created.
string	an expression evaluating to the string or dynamic array from which to read the elements.

* Not included in the SMA standards.

delim	an expression evaluating to the characters used to delimit elements in *string*. The behavior of the MATPARSE statement is dependent on the number of characters specified as delimiters.
SETTING *var*	assign to *var* the number of fields separated by *delim* in the string. The resulting *var* can be tested to determine if it is larger than the dimensions of the array. (The SETTING phrase is not available on Prime INFORMATION systems.)
start	an optional starting position within *array*. If *start* is less than 1, the default is 1.
end	an optional ending position within *array*. If *end* is less than 1 or greater than the size of *array*, the default is the size of the array.
USING	specifies the characters used to delimit elements in *string*. If not specified, the default is an attribute mark. To specify a null delimiter, omit the comma or USING after *string*.

The behavior of the MATPARSE statement is dependent on the number of characters specified as delimiters:

Number of Delimiters	Result
0 (*delim* = "")	Each character of *string* is placed into a separate element of *array*.
1	Each field separated by the delimiter character *delim* is loaded into a separate element of *array*. The delimiter character itself will not be stored.
2 or more	Fields and delimiters alternate as elements of *array*. Fields will occupy all odd-numbered elements of the array, and delimiters will occupy all even-numbered elements. Consecutive delimiters will be placed in the same element if they are identical, but will be placed in separate elements if they are different, with a null field element in between.

If the dimensions of the dimensioned array are too small to accommodate the parsed string, the leftover portion is appended to the last element of the array, delimiters intact.

On uniVerse systems, PICK flavor arrays store overflow elements in the last element (see the first example below); IDEAL and INFORMATION flavors store overflow elements in element zero.

Examples

Using No Delimiters

If a string STRING contained:

```
THIS-IS-A-STRING
```

and the array ARR1 were dimensioned to a 7-element vector, then:

```
MATPARSE ARR1 FROM STRING, ""
```

(0 delimiters) would produce as ARR1:

1	T
2	H
3	I
4	S
5	-
6	I
7	S-A-STRING

Using One Delimiter

If the single delimiter "-" was specified:

```
MATPARSE ARR1 FROM STRING, "-"
```

would return into ARR1:

1	THIS
2	IS
3	A
4	STRING
5	
6	
7	

Using Two Delimiters

If two delimiters are specified:

```
MATPARSE ARR1 FROM STRING, "--"
```

then ARR1 would contain:

1	THIS
2	-
3	IS
4	-
5	A
6	-
7	STRING

For each of these examples, MATBUILD with the same delimiters can be used to reconstruct the string.

MATREAD: Read a file item as a dimensioned array.

The MATREAD statement reads a file item and assigns each attribute to an element of a dimensioned array.

MATREAD *array* **FROM** [*filevar,*] *item-ID* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

array — the dimensioned array to be assigned. The array must have been dimensioned with a DIM or COMMON statement before it can be assigned with the MATREAD statement.

filevar	the file variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the last file opened without a file variable assigned.
item-ID	an expression evaluating to the item ID to be read. If the item is not found, the contents of *array* remains unchanged.
THEN *statements*	execute *statements* if the item ID is found. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if the item ID is not found. For details about the syntax of ELSE clauses, see the IF statement.

The MATREAD statement assigns the attributes of a file item to consecutive elements of the specified dimensioned array. The first attribute of the item becomes the first element of *array*, the second attribute of the item becomes the second element of *array*, and so on. The array must be named and dimensioned in a DIM or COMMON statement before it is used in this statement.

A MATREAD statement does not set an update lock on the specified record. That is, the record remains available for update to other users. To prevent other users from updating the record until it is released, use a MATREADU statement. See the MATREADU statement for more information.

If the number of attributes in the file item is greater than the dimensions of the array, the remainder of the attributes are placed into the last element of the array, separated by attribute marks (CHAR(254)). If the number of elements in the array is greater than the number of attributes in the item, the extra elements in the array are assigned a null value.

The MATREAD statement functionally yields the same result as using the READ statement to read a dynamic array and then using the MATPARSE statement to assign a dimensioned array to the same elements.

Example

In the following application, the item containing an employee's statistics is read into the dimensioned array EMP. The MATREAD statement reads the elements of the array from the EMPLOYEES file.

```
EQU TRUE TO 1, FALSE TO 0
   .
   .
   .
ITEM.ABORTED = FALSE
ITEM.ON.FILE = TRUE
READ.COMPLETED = FALSE
LOOP
   MATREAD EMP FROM EMPLOYEES,EMP.ID THEN
      PRINT @(-1):"*EMPLOYEE'S DATE OF BIRTH IS ":
      OCONV (EMP(11),'D')
   END ELSE
      PRINT "NOT ON FILE"
   END
```

MATREADU: Read a dimensioned array, setting an item lock.

The MATREADU statement performs a MATREAD, simultaneously setting an item lock.

MATREADU *array* **FROM** *[filevar,] item-ID* [**LOCKED** *statements*] { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

array	the dimensioned array to be assigned. The array must have been dimensioned with a DIM or COMMON statement before it can be assigned with the MATREADU statement.
filevar	the file variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the last file opened without a file variable assigned.
item-ID	an expression evaluating to the item ID to be read. If the item is not found, the contents of *array* remains unchanged.

LOCKED *statements*	execute statements if the item was already locked by another process. The statements of the LOCKED clause follow the syntax of statements in THEN or ELSE clauses.
THEN *statements*	execute *statements* if the item ID is found. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if the item ID is not found. For details about the syntax of ELSE clauses, see the IF statement.

The behavior of the MATREADU statement is identical to that of the MATREAD statement, except that a lock is placed on the item to be read.

When an item is locked, it cannot be read by another READU, READVU, or MATREADU statement until the lock is removed. The lock is removed by exiting the program, updating the file with the WRITE, WRITEV, or MATWRITE statements, or releasing the lock with the RELEASE statement. The file can be updated without removing the item lock with the WRITEU, WRITEVU, or MATWRITEU statements. Although there is no theoretical limit to the number of items that can be locked, the size of the item lock table will vary from implementation to implementation.

If the MATREADU statement is executed on an item already locked by another user, the program hangs until the lock is released, unless the LOCKED clause is specified. The LOCKED clause allows the user to exit from the MATREADU statement without waiting for the item to be released. Note that if the LOCKED clause is used, the array variable *array* will not be assigned.

If the item does not exist, the item lock is still set, and the ELSE clause is executed. Thus the MATREADU statement can be used to reserve an item for use by the program even if it does not yet exist.

The MATREADU statement performs the same function as using the READU statement to read a dynamic array and then using the MATPARSE statement to assign a dimensioned array to the same elements.

Example

In the following example, the MATREADU statement is used to read a file item from the file opened as EMPLOYEES. This item is placed into the

dimensioned array EMP with Attribute 1 going into EMP(1), Attribute 2 going into EMP(2), etc. If the item is currently locked by another user, a message is displayed and the user can try again or quit. Once the MATREADU has successfully completed, either the item is modified, if it was on file, or is added. The MATWRITE statement is then used to write the item back to the file, releasing the item lock in the process.

```
EQU TRUE TO 1, FALSE TO 0
    .
    .
    .
ITEM.ABORTED = FALSE
ITEM.ON.FILE = TRUE
READ.COMPLETED = FALSE
LOOP
    MATREADU EMP FROM EMPLOYEES,EMP.ID LOCKED
        PRINT "** ITEM LOCKED.  PRESS RETURN TO"
        PRINT "TRY AGAIN, ENTER 'Q' TO QUIT = ":
        INPUT RESP
        IF OCONV(RESP, "MCU") = "Q" THEN ITEM.ABORTED = TRUE
    END ELSE
        ITEM.ON.FILE = FALSE
        READ.COMPLETED = TRUE
    END
UNTIL READ.COMPLETED OR ITEM.ABORTED DO REPEAT
IF ITEM.ABORTED THEN
    NULL ;* Skip processing
END ELSE
    IF ITEM.ON.FILE THEN
        GOSUB PROCESS.EXISTING.ITEM
    END ELSE
        GOSUB PROCESS.NEW.ITEM
    END
END
MATWRITE EMP TO EMPLOYEES,EMP.ID
    .
    .
    .
```

MATWRITE: Write a dimensioned array into a file item.

The MATWRITE statement writes the elements of a dimensioned array as attributes of a file item.

MATWRITE *array* **ON** [*filevar,*] *item-ID*

array an expression evaluating to the dimensioned array to write into the file item.

filevar the variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.

item-ID an expression evaluating to the item ID of the item to be written.

The MATWRITE statement writes a dimensioned array onto a file item, overwriting any data previously stored in that item. If the file item does not exist, a new item is created. Element 1 of the dimensioned array becomes Attribute 1 of the file item, element 2 becomes Attribute 2, etc.

Example

To write the dimensioned array EMPLOYEE into an item with *item-ID* NAME in the file opened as EMP.FILE, the code would read:

```
MATWRITE EMPLOYEE ON EMP.FILE , NAME
```

MATWRITEU: Write an array into a file item, retaining item locks.

The MATWRITEU statement writes the elements of a dimensioned array as attributes of a file item, leaving item locks intact.

MATWRITEU *array* **ON** [*filevar,*] *item-ID*

array an expression evaluating to the dimensioned array to write into the file item.

filevar the variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.

item-ID an expression evaluating to the item ID of the item to be written.

The MATWRITEU statement is identical to the MATWRITE statement, except that item locks are not lifted by the MATWRITEU statement.

Element 1 of the dimensioned array becomes Attribute 1 of the file item, element 2 becomes Attribute 2, etc.

Example

In the following application the MATWRITEU statement is used to update inventory information which is still being processed, and is therefore not available to other users for updating.

```
.
.
.
INPUT QTY.ORD
INV.ITEM (10) += QTY.ORD ; *Update qty committed
MATWRITEU INV.ITEM ON INV.FILE, PN
.
.
.
```

MOD( ): Return remainder of one expression divided by another.*

The MOD function divides one expression by another and returns the remainder. It is functionally equivalent to the REM function.

MOD(*expr1*, *expr2* **)**

The MOD function returns the value of the remainder after division is performed on *expr1* by *expr2*. This is also called *expr1* modulo *expr2*. The expressions can evaluate to any numeric value, with the exception that *expr2* cannot be zero.

Examples

To place the remainder in the variable NUMB when 17 is divided by 5, the code would read:

```
NUMB = MOD(17,5)
```

In this instance, NUM would contain "2".

---

* Not included in the SMA standards.

In the following application the MOD function is used to calculate a customer's change after a purchase.

```
PRINT "HOW MUCH DO YOU HAVE TO SPEND? $" :
INPUT AMOUNT
NUMBER = INT(AVAILABLE/PRICE)
CHANGE = MOD(AVAILABLE,PRICE)
PRINT "EACH ITEM COSTS " : PRICE"2,$" : "."
PRINT "FOR " : AMOUNT "2,$" : " YOU CAN GET " : NUMBER : "
ITEMS."
PRINT "YOUR CHANGE WILL BE " : CHANGE "2,$" : "."
```

A sample run might appear as follows (with the operator's input in bold):

```
HOW MUCH DO YOU HAVE TO SPEND? $4
EACH ITEM COSTS $1.25.
FOR $4.00 YOU CAN GET 3 ITEMS.
YOUR CHANGE WILL BE $0.25.
```

NEXT: Terminator used with FOR...NEXT loops.

For information about the NEXT terminator, see the FOR statement.

NOT( ): Return the logical inverse of an expression.

The NOT function returns the logical inverse of a given expression.

NOT(*expr* **)**

expr the expression to be logically inverted.

The NOT function returns the logical complement of the value of *expr*. If the expression evaluates to true, the NOT function returns a value of false. If the value of *expr* is false, the NOT function returns a value of true.

An expression is considered false if it evaluates to 0 or the null string (""). If an expression evaluates to any other value, it is considered true. See Chapter 2 for more information on logical values.

Example

The following example demonstrates how the NOT function can be used to verify that meaningful data has been input by the user:

```
LOOP
   PRINT "ENTER YOUR ACCOUNT NUMBER : " :
   INPUT ACCT,6, "?"
WHILE NOT (NUM(ACCT)) DO
   PRINT "MUST BE NUMERIC."
REPEAT
```

The account number input must be numeric in order to be valid. If it is not, the program is aborted.

NULL: Null statement.

The NULL statement results in no action. It is meant as a filler for clauses which require a statement, regardless of whether action is necessary.

NULL

The NULL statement has been devised for situations in which a statement is required but no operation is to be performed. It is seldom necessary, but is often used for program readability and consistency.

Example

The following example demonstrates how the NULL statement can be used in an IF construct.

```
IF ZIP MATCHES "5N" THEN
   NULL
END ELSE
   .
   .
   .
END
```

In the preceding example the NULL statement is used as a placeholder. The THEN clause requires at least one statement, so the NULL statement is used to allow the program to compile.

Note that in Pick BASIC this construction is not required; the THEN clause is not mandatory in an IF statement, as long as an ELSE clause

exists. However, if the THEN clause is omitted, the statement would make sense to the compiler but it might not to the reader. The NULL statement is used to make clear to the reader that if the match returns a value of true, no action is taken.

NUM( ): Determine if an expression is numeric.

The NUM function is used to test a given expression for numeric data.

NUM(*expr* **)**

expr the expression to be tested.

The NUM function determines whether the *expr* is a numeric or a nonnumeric string. If *expr* contains only numeric characters, the function evaluates to true and a value of 1 is returned. If *expr* contains nonnumeric characters, the expression evaluates to false and a value of 0 is returned.

A single decimal point within a numeric string is considered a numeric character; that is, if the expression contains only numbers and a decimal point, the expression evaluates to true (1). However, if a second decimal point is encountered in the string, the expression evaluates to false (0) since a second decimal point is meaningless in a mathematical expression.

Commas and dollar signs are never considered numeric characters; that is, a numeric string containing a comma or dollar sign evaluates to false, regardless of whether all other characters are numbers.

Example

In the following application, the NUM function is used to verify that a valid numeric response was input:

```
LOOP
   PRINT "ENTER YOUR ACCOUNT NUMBER : " :
   INPUT ACCT,6, "?"
WHILE NOT (NUM(ACCT)) DO
   PRINT "MUST BE NUMERIC."
REPEAT
```

Note that the decimal precision used by the program is a consideration in this example. For example, if the precision is 4 (the default) and the user enters a value with five decimal positions (such as 3.14159), the NUM

function returns false (0) and the program stops. See the PRECISION statement for more information.

OCONV( ): Convert data from internal to external format.

The OCONV function converts data from internal format to external format, according to the conversion code specified.

OCONV(*expr, code* **)**

expr an expression evaluating to the data to be converted.

code an expression evaluating to a conversion code.

Data is stored in internal format for consistency and flexibility. The OCONV function converts the data back to external format. For example, dates are generally kept as the number of days since December 31, 1967—so June 4, 1965 would be stored as –940, and OCONV(–940,"D") produces "4 JUN 1965."

To convert back from external format to internal format, use the ICONV function.

Conversion Codes

The conversion codes for OCONV correspond to the conversion codes used in ACCESS. Among the more common codes used in Pick BASIC are:

D	Convert date to external format.
MT[H][S]	Convert time to external format.
	H 12-hour format.
	S include seconds.
M{L | R}	Format numbers.
MX	Convert ASCII to hexadecimal representation.

Among the ACCESS codes that *cannot* be called by the OCONV function are "F", "A", and "S". For more information about conversion codes, see *Pick ACCESS: A Guide to the SMA/RETRIEVAL Language.*

Examples

To convert data in the string STRING into hexadecimal format, the code might read:

```
STRING = OCONV(STRING, "MX")
```

ON: Conditionally branch to a subroutine.

For information about the ON conditional phrase, see the GOSUB and GOTO statements.

OPEN: Open a file.

The OPEN statement is necessary to access a file in the current program.

OPEN [*dict*] *file* [**TO** *filevar*] { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

dict	an expression evaluating to the keyword DICT. If the dictionary expression is not specified, the data file is assumed.
file	an expression evaluating to the filename to be opened. If the file is one of several data files associated with a single file dictionary, it can be opened by the syntax, '*dictname,file*', with *dictname* the name of the file dictionary.
TO *filevar*	define *filevar* as the file variable name by which the file will be accessed. If the TO *filevar* clause is not specified, the file can be accessed only as the default file variable.
THEN *statements*	execute *statements* if *filename* is opened successfully. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if *filename* cannot be opened. This clause is generally used to cause print error messages or to stop or abort the program. For

details about the syntax of ELSE clauses, see the IF statement.

The OPEN statement prepares a file for use by the current Pick BASIC program. All references to a file within a Pick BASIC program must be preceded by a separate OPEN statement for that file.

If a file variable is not assigned with the TO keyword, the file is assigned to the default file variable. Any subsequent file I/O statements that do not specify a file variable will default to this file. Note that default file variables are not local to the program from which they are executed: when a subroutine is called, the current default file variable is shared with the calling program.

There is no limit to the number of files which can be open at a given time. However, if multiple files are opened and accessed concurrently, file variables must be used. The default file variable can represent only one file at a time.

Example

In the following application, the OPEN statement is used to open a reservation file, and the operator is asked to enter the customer's last name, to be used as an item ID. If the reservation file is not found, the program will abort. A READ statement is then used to find the file item. If the item is found, any current reservations will be shown; if not, a new reservation can be entered.

```
OPEN "RESERVATIONS" TO RES.FILE ELSE
   ABORT 201,"RESERVATIONS"
END
   .
   .
   .
LOOP
   PRINT "LAST NAME : " :
   INPUT ITEM.ID
   READ RECORD FROM RES.FILE,ITEM.ID THEN
      PRINT ITEM.ID : " ON FILE."
      GOSUB SHOW.RES
   END ELSE
      PRINT ITEM.ID : " NOT ON FILE"
      GOSUB ENTER.RES
   END
UNTIL LAST.NAME = "" DO REPEAT
```

PAGE: Advance the page on the output device.

The PAGE statement causes the current page to be ended and the footing to print at the bottom of the page.*

PAGE [*expr*]

expr an expression evaluating to the page number to appear on the next page.

The PAGE statement ends the current page of output by sending the footing to the bottom of the page and waiting for a carriage return to continue output. The N option to the HEADING statement suppresses waiting for the carriage return.

If a page number is specified, the next page of output is numbered with the new page number *expr*. Each subsequent page number will increment by one. Note that the current page number is numbered *expr*–1, which appears in the current page footing if a page number is included in the footing specifications. The page heading is not affected until the next page of output.

The HEADING, FOOTING, and PAGE statements affect the same output device that the PRINT statement does. The PRINTER statement toggles the output device between the terminal screen and the printer. If multiple print units are used, the HEADING, FOOTING, and PAGE statements affect only print unit 0 (the default).

Example

In the following application the contents of each item in a file are printed with the internal subroutine PRINTOUT. The PAGE statement is used before PRINTOUT is called to ensure that each report starts on page 1.

```
LOOP
   READNEXT ITEM ELSE
      END.OF.LIST = 1
   END
UNTIL END.OF.LIST DO
   MATREAD REPORT FROM LOGFILE,ITEM ELSE ...
   PAGE 1
```

* On some implementations, the HEADING statement is necessary to initialize the page parameters for a program, otherwise the PAGE statement will have no effect.

```
    GOSUB PRINTOUT
    PRINTER CLOSE
REPEAT
```

PRECISION: Declare decimal precision.

The PRECISION statement determines the decimal precision of numeric values used in the current program.

PRECISION *digits*

digits an expression evaluating to the number of fractional digits to which numeric values will be calculated. Any fractional digits in the result of a conversion that exceed *digits* are rounded off. *digits* can be any integer from at least 6,* with a default value of 4. A precision of zero implies that all values will be treated as integers throughout the program.

Precision can be declared anywhere within the program.

☞ Only one PRECISION declaration is allowed in a program; if more than one is encountered, a warning message is printed and the second PRECISION declaration is ignored. However, if external subroutines are called, each external subroutine should include a PRECISION statement matching that of the main program. If the precision differs between the calling program and the subroutine, a warning message is printed and the second declaration is ignored.

Trailing fractional zeros are dropped during output. Therefore, when an internal number is converted to an ASCII string, the result might appear to have fewer decimal places than the precision setting allows. However, regardless of the precision setting, the calculation always reflects the maximum accuracy of which the computer is capable.

The range of possible numeric values varies from system to system. If the precision is set to a value less than the default (4 on most systems), the range of allowable numbers is increased accordingly.

---

* Some implementations allow a greater precision to be set; some older implementations allow only a maximum of 4 (the default).

Example

The following example demonstrates how the precision statement affects the output of a calculation of π:

```
PRECISION 2
CIRC = 25.13276208
RAD = 4
PI = CIRC / RAD / 2
PRINT PI
```

The following output is displayed:

```
3.14
```

PRINT: Send data to the output device.

The PRINT statement sends data to the terminal screen or to another specified print unit.

PRINT [**ON** *unit#*] *print-expr*

ON *unit#* specifies that data should be output to a spooler print unit *unit#*. *unit#* can be any integer in the range 0 to 254, with 0 as the default. Print unit 0 is interpreted as either the terminal screen or the printer, depending on previous use of the PRINTER statement. When the program is terminated or when a PRINTER CLOSE statement is used, all print units are output to the printer. This option is used when several different reports are being generated by the program simultaneously.

print-expr is a print expression optionally combined with commas and colons to designate the format of the output (as described below). If *print-expr* is omitted, a blank line is output.

The PRINTER statement determines the output device to which data is to be written by the PRINT statement. See the PRINTER statement for more information. There is also a CRT (or DISPLAY) statement available in Pick BASIC, which is identical to the PRINT statement except that it always prints its output to the screen, regardless of whether a PRINTER ON statement had been issued.

Formatted Output

Format expressions can be used to provide complex formatting specifications for output. See *Format Expressions* for more information. In the PRINT statement, however, commas and colons can be used to specify tab stops and suppress line feeds.

- Expressions separated by commas are printed at preset tab positions. Multiple commas can be used together to cause multiple tabulations between expressions. However, tab positions cannot be specified without being surrounded by expressions.
- Colons (:) encountered between expressions are interpreted normally as the string concatenation operator. If the last character of the PRINT statement is a colon, however, the line feed and carriage return which usually follow the print statement is suppressed. This is especially useful when an INPUT statement is to follow, or in formatted screen programs.
- The @ function can be used with the PRINT statement to send the cursor to a specified location on the screen. (Don't try to use the @ function when sending output to the printer!)

Examples

In the following application, the full contents of a dimensional array are printed via a FOR...NEXT loop. Only one element is printed at a time, however; hence, to imitate the actual structure of the array, tab stops are generated with commas and new lines are suppressed with colons. Before a new row is begun, the carriage return is generated by a null PRINT statement.

```
FOR I=1 TO 4
   FOR J=1 TO 3
      PRINT EMPLOYEE(I,J), " " :
   NEXT J
   PRINT
NEXT I
```

If the array EMPLOYEE contains the name, telephone number, and marital status of each employee, the output might be:

```
FRED HENKEL          555-1234     SINGLE
ARCHIE ANDREWS       555-4321     MARRIED
MARGARET WOOD        867-5309     SEPARATED
LUCY RICARDO         338-6887     MARRIED
```

In the next example, the @ function is used with the PRINT statement to print an error in blinking text onto the bottom of the screen. It also clears the rest of the line, in case there had already been text on that line. Note that the cursor remains at the bottom of the screen after the error message is printed.

```
PRINT "ENTER YOUR SOCIAL SECURITY NUMBER : "
PROMPT ""
INPUT ANSWER,11,"*#"_
IF ANSWER MATCHES "3N-2N-4N" THEN
   GOSUB SOCSEC
END ELSE
   PRINT @(0,23) : @(-5) : ANSWER : " : DOES NOT MATCH SS #" :
   PRINT @(-6) : @(-4) :
END
   .
   .
   .
```

PRINTER: Specify the output device.

The PRINTER statement is necessary to redirect output to either the terminal screen or the printer. By default, all output will be sent to the terminal screen unless a PRINTER ON statement is issued or the P option is used at run time.

PRINTER ON | OFF | CLOSE

ON — directs all subsequent program output sent by the PRINT, HEADING, FOOTING, or PAGE statements to a print unit. The contents of the print unit are printed when the program terminates. If output is requested from the printer before program termination, the PRINTER CLOSE statement must be used to flush the output.

OFF — directs all subsequent program output immediately to the terminal screen.

CLOSE — sends all contents of the print unit (that is, all output sent while the PRINTER ON statement was in effect) to the printer immediately.

By default, all output is sent to print unit 0, which is immediately written to the terminal display screen unless the PRINTER ON statement is used.

If the PRINTER ON statement is used, the contents of print unit 0 is sent to the spooler, where it is held as an open print file until the program has been terminated or until a PRINTER CLOSE statement is used, at which point the print job is spooled to the printer.

The program output device can also be established at run time. The TCL command RUN has a P option that allows the program to run as if a PRINTER ON statement had been issued at the beginning of the program.

The Pick BASIC statement CRT (or DISPLAY), is identical to the PRINT statement except that it is not affected by the PRINTER statement. That is, all data which is output via the CRT statement is sent to the terminal screen, regardless of whether the PRINTER ON statement is active. The CRT statement is designed to facilitate programs in which some output is sent to the terminal and some to the printer, since it allows printing to the terminal without toggling PRINTER OFF and PRINTER ON statements.

Example

To direct subsequent PRINT output to the printer, enter:

```
PRINTER ON
```

The output is sent to print when the program is exited.

PROCREAD: Read the primary input buffer of the calling proc.*

The PROCREAD statement assigns the string value of the primary input buffer of the calling proc to a variable.

PROCREAD *var* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

var	the variable to be assigned the current value of the primary input buffer.
THEN *statements*	execute *statements* if buffer is read successfully. For details about the syntax of THEN clauses, see the IF statement.

* Not included in the SMA standards.

ELSE *statements* execute *statements* if program was not called by a proc, or if the input buffer was empty. For details about the syntax of ELSE clauses, see the IF statement.

The PROCREAD statement can be used to access the primary input buffer of a calling proc. On some implementations the SYSTEM function can determine whether the program was called by a proc.

To write to the primary input buffer, use the PROCWRITE statement.

Example

In the following application, the program DIALOUT should be called only by a proc. If the program was called directly, an error message is printed and execution is stopped.

```
PROCREAD ARGUMENTS ELSE
   PRINT "ERROR! "
   PRINT "'DIALOUT' MUST BE CALLED FROM A PROC"
   ABORT
END
```

PROCWRITE: Write to the primary input buffer of the calling proc.*

The PROCWRITE statement writes data into the primary input buffer of the calling proc.

PROCWRITE *expr*

expr an expression evaluating to the string to be written into the primary input buffer.

The PROCWRITE statement writes to the primary input buffer of a calling proc. Any previous data in the primary input buffer is erased and replaced by the given data.

Some Pick systems include an optional ELSE clause in their PROCWRITE syntax. Statements in the ELSE clause are executed if the program was *not* called by a proc.

---

* Not included in the SMA standards.

To read from the primary input buffer, use the PROCREAD statement.

Example

In the following application, the program asks the operator if reports should be printed out. The response is used in the program, but is also passed to the calling proc with PROCWRITE.

```
PRINT "SEND RESULTS TO THE PRINTER? (Y OR N)" :
INPUT ANSWER, Y
IF ANSWER = "Y" THEN
   PRINTER ON
      .
      .
      .
END
PROCWRITE ANSWER
```

PROMPT: Assign the prompt character.

The PROMPT statement can be used to reassign the prompt character from its default setting of "?".

PROMPT *expr*

expr an expression evaluating to a single character, to be taken as the prompt character. If *expr* evaluates to more than one character, only the first character is used as the prompt.

The character specified by a PROMPT statement is sent to the output device whenever an INPUT statement is used, to signal the operator that input is requested before execution may continue. The default prompt is "?".

Note that the prompt character is generated by each INPUT statement and not by PRINT statements.

Example

In the following application, the colons (:) at the end of each PRINT statement do not designate a prompt but simply suppress the line feed or

carriage return for display purposes. The prompt character is generated by the INPUT statement, not by the PRINT statement.

```
PRINT "DO YOU WANT TO KNOW YOUR SCORES (Y OR N)" :
INPUT ANSWER
IF ANSWER = "Y" THEN
   PROMPT ":"
   PRINT "ENTER GAME NUMBER" :
   INPUT GAME
   PRINT "YOUR SCORE IN GAME " :
   PRINT GAME :
   PRINT " WAS " :
   PRINT SCORES(GAME)
END
```

The resulting output (with the operator's input in bold) is:

```
DO YOU WANT TO KNOW YOUR SCORES (Y OR N)?Y
ENTER GAME NUMBER:2
YOUR SCORE IN GAME 2 WAS 78
```

PWR( ): Returns an exponential value.

The PWR function returns the value of one number raised to the power of a second number.

PWR(*expr1*,*expr2* **)**

expr1 an expression evaluating to a numeric value.

expr2 an expression evaluating to the exponent that *expr1* is to be raised to.

The PWR function raises the first expression to the power of the second expression. Any number raised to the power of 0 will return 1.

Example

To assign MAX.LEN the value of 2^{15}, enter:

```
MAX.LEN = PWR(2,15)
```

READ: Read a file item as a dynamic array.

The READ statement assigns the string value of a file item to a variable.

READ *array* **FROM** [*filevar*,] *item-ID* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

array	the variable, in dynamic array form, to which the string value of the item is assigned.
filevar	the file variable name to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.
item-ID	an expression evaluating to an item ID. If the specified item ID does not exist, *var* is assigned the value of the null string ("").
THEN *statements*	execute *statements* if *item-ID* is read successfully. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if *item-ID* cannot be read. For details about the syntax of ELSE clauses, see the IF statement.

Before a file can be accessed in a READ statement, it must be opened with an OPEN statement or an error will occur at run time. See the OPEN statement for more information.

In Pick BASIC there are also READU, READV, and READVU statements available. The READU statement sets an item lock on the file item before reading it, the READV statement reads a single attribute from a given file item, and the READVU statement sets an item lock and then performs a READV. See the READU, READV, and READVU statement pages for more information.

Example

In the following application, the OPEN statement opens a reservation file and the operator is asked to enter the customer's last name, to be used as an item ID. If the reservation file is not found, the program aborts. A READ statement is then used to find the file item. If the item is found, any current

reservations are shown; if it is not found, a "NOT ON FILE" message is displayed.

```
OPEN "RESERVATIONS" TO RES.FILE ELSE
   ABORT 201,"RESERVATIONS"
END
   .
   .
   .
LOOP
   PRINT "LAST NAME : " :
   INPUT ITEM.ID
   READ RECORD FROM RES.FILE,ITEM.ID THEN
      PRINT ITEM.ID : " ON FILE."
      GOSUB SHOW.RES
   END ELSE
      PRINT ITEM.ID : " NOT ON FILE"
   END
UNTIL LAST.NAME = "" DO REPEAT
```

READNEXT: Read the next value in a select-list.

The READNEXT statement reads the next sequential value in a select-list.

READNEXT *var1* [,*var2*] [**FROM** *select-var*] { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

var1	read the next value in the select-list and assign it to *var1*.
var2	assign the value mark count to *var2*. This option is applicable only to select-lists constructed through the TCL SSELECT verb.
select-var	read values from the named select-list variable *select-var*. If *select-var* is not specified, the default select-list variable is used.
THEN *statements*	execute *statements* unless at the end of the list. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if at the end of the list. For details about the syntax of ELSE clauses, see the IF statement.

The READNEXT statement assigns the next value from an active select-list to the specified variable. If it is a select-list of item IDs, the variable can then be used in a READ statement to read the file item. (The SELECT statement might also create a select-list of attributes from a dynamic array.)

See the SELECT statement for more information on creating select-lists. If a value is successfully read from the select-list, *var* is assigned to the value, and the THEN statements are executed; when the end of the select-list is reached, *var* is set to the null string and the ELSE statements are executed.

Example

In the following application we create an alphabetical list of each item ID in the file AIRPORTS. The file is selected to the select variable LIST, and each item ID is read from LIST with READNEXT. The END.OF.LIST variable is set to true when the READNEXT statement fails to read any more item IDs, and is used to complete the loop. The actual alphabetizing is accomplished with the LOCATE and INS statements.

```
EQUATE TRUE TO 1, FALSE TO 0
OPEN "AIRPORTS" TO AIRPORTFILE ELSE
   ABORT 201, "AIRPORTS"
END
SELECT AIRPORTFILE TO LIST
ALPH.LIST = ""
END.OF.LIST = FALSE
LOOP
   READNEXT ID FROM LIST ELSE
      END.OF.LIST = TRUE
   END
UNTIL END.OF.LIST DO
   LOCATE ID IN ALPH.LIST BY "AL" SETTING POSITION THEN
      PRINT ID : " DUPLICATE ENTRY! POSSIBLE FILE CORRUPTION"
      ABORT
   END ELSE
      INS ID BEFORE ALPH.LIST<POSITION>
   END
REPEAT
```

READT: Read next record from magnetic tape.

The READT statement reads the next record (block) on the magnetic tape unit, assigning its value to the specified variable.

READT *var* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

var	the variable, in dynamic array form, into which the next record is read.
THEN *statements*	execute *statements* if record is successfully read. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if record cannot be read. For details about the syntax of ELSE clauses, see the IF statement.

If a record is read, its value is assigned to the specified variable and the THEN statements are executed. If the record cannot be read, the specified variable is assigned the null string and the ELSE statements are executed.

A record might not be read because the tape has not been attached or because an End-Of-File mark was encountered. To determine why a tape could not be read, the SYSTEM function is often used in the ELSE portion of a READT statement. See the SYSTEM function for more information.

Example

The program segment in the following example reads data off a tape and prints it in a readable format. The item IDs are printed and then each attribute is printed on a separate line, preceded by the attribute number.

```
LOOP
   READT NEWRECORD ELSE
      IF SYSTEM(0) = 2 THEN
         END.OF.TAPE = TRUE
      END ELSE
         PRINT "SYSTEM ERROR --"
         GOSUB EXIT
      END
   END
   REC.NUM += 1
UNTIL END.OF.TAPE DO
   PRINT
```

```
    PRINT "PRESS ANY KEY TO READ RECORD ": REC.NUM : ":":
    INPUT CHAR,1
    PRINT
    NO.OF.ATTRS = DCOUNT(NEWRECORD , AM)
    FOR I = 1 TO NO.OF.ATTRS
        PRINT I, NEWRECORD< I >
    NEXT I
REPEAT
```

READU: Read a file item as a dynamic array, locking the item.

The READU statement performs a READ, simultaneously setting a lock on the item to be read.

READU *array* **FROM** [*filevar*,] *item-ID* [**LOCKED** *statements*] { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

array	the variable, in dynamic array form, to which the string value of the item is assigned.
filevar	the file variable name to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.
item-ID	an expression evaluating to an item ID. If the specified item ID does not exist, *var* is assigned the value of the null string ("").
LOCKED *statements*	execute *statements* if the item was already locked by another process. The *statements* of the LOCKED clause follow the syntax of *statements* in THEN or ELSE clauses.
THEN *statements*	execute *statements* if *item-ID* is read successfully. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if *item-ID* cannot be read. For details about the syntax of ELSE clauses, see the IF statement.

The behavior of the READU statement is identical to that of the READ statement, except that a lock is placed on the file item to be read. When an item is locked, it cannot be read by another READU, READVU, or MATREADU statement until the lock is removed. The lock is removed by exiting the program, updating the file with the WRITE, WRITEV, or MATWRITE statements, or releasing the lock with the RELEASE statement. The file can be updated without removing the item lock with the WRITEU, WRITEVU, or MATWRITEU statements. Although there is no theoretical limit to the number of items that can be locked, the size of the item lock table will vary from implementation to implementation.

If the READU statement is executed on an item already locked by another user, the program hangs until the lock is released, unless the LOCKED clause is specified. The LOCKED clause allows the user to exit from the READU statement without waiting for the item to be released. Note that if the LOCKED clause is used, *var* is not assigned.

If the item does not exist, the item lock is still set and the ELSE clause is executed. Thus the READU statement can be used to reserve an item for use by the program even if it does not yet exist.

Example

In the following application, the file CUSTOMERS contains customer billing information. The program tries to read the a customer's record. If the item is not locked, the program goes on to update addresses; otherwise the user is asked to try later.

```
OPEN "CUSTOMERS" TO CUSTFILE ELSE
   ABORT 201 , "CUSTOMERS"
END
READU RECORD FROM CUSTFILE , NAME LOCKED
   PRINT "ITEM IS LOCKED!  TRY AGAIN LATER."
END ELSE
   .
   .
   .
```

READV: Read a single attribute of a file item.

The READV statement reads a single attribute of a file item and places it into a dynamic array variable.

READV *array* **FROM** *[filevar,*] *item-ID* , *attr#* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

array	the variable, in dynamic array form, to which the string value of the attribute is assigned.
filevar	the file variable name to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.
item-ID	an expression evaluating to an item ID. If the specified item ID does not exist, *var* is assigned the value of the null string ("").
attr#	an expression evaluating to an attribute number in the specified item. If the specified attribute number does not exist, *var* is assigned the value of the null string ("").
THEN *statements*	execute *statements* if the attribute is read successfully. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if the item cannot be found or if the attribute cannot be read. For details about the syntax of ELSE clauses, see the IF statement.

Before a file can be accessed in a READV statement, it must be opened with an OPEN statement or an error will occur at run time. See the OPEN statement for more information.

In Pick BASIC, there are also READ, READU, and READVU statements available. The READVU statement sets an item lock on the item before reading the attribute. The READ statement reads the entire file item, and the READU statement sets an item lock on the item before reading it. See the READU, READ, and READVU statement pages for more information.

Example

In the following application the file represented by CUSTFILE contains the name, address, and phone number of each customer. The phone number is stored in Attribute 6 of each item. To retrieve only the customer's phone number, the READV statement is used to read Attribute 6.

```
PRINT "ENTER CUSTOMER ID : " :
INPUT ID
READV PHONE FROM CUSTFILE,ID,6 ELSE
   PRINT "ERROR!"
   STOP
END
PRINT ID , PHONE
```

READVU: Read an attribute of a file item, setting an item lock.

The READVU statement performs a READV, simultaneously setting a lock on the item from which the attribute is to be read.

READVU *array* **FROM** [*filevar*,] *item-ID*, *attr#* [**LOCKED** *statements*] { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

array	the variable, in dynamic array form, to which the string value of the attribute is assigned.
filevar	the file variable name to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.
item-ID	an expression evaluating to an item ID. If the specified item ID does not exist, *var* is assigned the value of the null string ("").
attr#	an expression evaluating to an attribute number in the specified item. If the specified attribute number does not exist, *var* is assigned the value of the null string ("").

LOCKED *statements*	execute *statements* if the item was already locked by another process. The *statements* of the LOCKED clause follow the syntax of *statements* in THEN or ELSE clauses.
THEN *statements*	execute *statements* if the attribute is read successfully. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if the item cannot be found or if the attribute cannot be read. For details about the syntax of ELSE clauses, see the IF statement.

The behavior of the READVU statement is identical to that of the READV statement, except that a lock is placed on the file item to be read. When an item is locked, it cannot be read by another READU, READVU, or MATREADU statement until the lock is removed. The lock is removed by exiting the program, updating the file with the WRITE, WRITEV, or MATWRITE statements, or releasing the lock with the RELEASE statement. The file can be updated without removing the item lock with the WRITEU, WRITEVU, or MATWRITEU statements. Although there is no theoretical limit to the number of items that can be locked, the size of the item lock table will vary from implementation to implementation.

If the READVU statement is executed on an item already locked by another user, the program hangs until the lock is released, unless the LOCKED clause is specified. The LOCKED clause allows the user to exit from the READVU statement without waiting for the item to be released. Note that if the LOCKED clause is used, *var* is not assigned.

If the item does not exist, the item lock is still set and the ELSE clause is executed. Thus the READVU statement can be used to reserve an item for use by the program even if it does not yet exist.

Example

In the following application, the file CUSTOMERS is read and locked by READVU.

```
OPEN "CUSTOMERS" TO CUSTFILE ELSE
   ABORT 201, "CUSTOMERS"
END
PRINT "ENTER CUSTOMER ID: ":
```

```
INPUT ID
FOUND = 0
READVU PHONE FROM CUSTFILE,ID,6 FOUND THEN
   PRINT "ITEM IS FOUND."
   FOUND = 1
END ELSE
   PRINT "ERROR!"
   STOP
END
```

RELEASE: Release item locks in a file.

The RELEASE statement releases item locks in a file, without performing an update.

RELEASE [[*filevar*,] *item-ID*]

filevar the file variable to which the file had been opened. If *filevar* is omitted and *item-ID* is specified, the default file variable is used, which is the last file opened without a file variable assigned.

item-ID an expression evaluating to the item ID to be released. If *item-ID* is not specified, then *filevar* cannot be specified, and all item locks set by the current program are released.

The RELEASE statement unlocks any item locks set by the READU, READVU, and MATREADU statements.

Item locks can be released three ways: through a WRITE, WRITEV, or MATWRITE statement, through a RELEASE statement, or by ending the program execution.

Example

In the following application, item locks are set with the READU statement at the onset of the program, but once it is verified that the item will not be updated during the execution of the program, the RELEASE statement is used to free the item so that other users can update it.

```
READU INFO.REC FROM INFOFILE,ID ELSE
   PRINT ID : " NOT FOUND."
   STOP
```

```
END
PRINT "CURRENT INFORMATION ON THAT CUSTOMER IS: "
NO.OF.ATTRS = DCOUNT(INFO.REC , CHAR(254))
FOR I = 1 TO NO.OF.ATTRS
   PRINT I
END
PRINT "IS THIS CORRECT (Y OR N)"
INPUT ANSWER,1
IF ANSWER = "Y" THEN
   RELEASE INFOFILE,ID
END ELSE
   .
   .
   .
```

REM: Enter a remark in the source code.

The REM statement begins a comment line in the program.

REM *anything*

anything any text can be placed after a REM statement.

The REM statement can be used to begin a comment line in the source code. It is functionally identical to the ! and * statements. Comment lines should be used to thoroughly document source code.

Comment lines can be inserted into the object code with the $* statement. See the $* statement for more information.

Examples

A program might be documented as follows:

```
REM
REM   GET ATTRIBUTE DEFINITIONS FROM DICT OF INVENTORY
                                                    FILE
      OPEN 'DICT', 'INV' TO INV.DICT ELSE PRINT 'CANT OPEN
                                          "DICT INV"'; STOP
      READV DESC.AMC FROM INV.DICT,2 ELSE
         PRINT 'CANT READ "DESC" ATTR'; STOP
      END
      READV QOH.AMC FROM 'QOH',2 ELSE
         PRINT 'CANT READ "QOH" ATTR'; STOP
      END
REM   OPEN DATA PORTION OF INVENTORY FILE
```

```
        OPEN '', 'INV' TO INV.FILE ELSE PRINT 'CANNOT OPEN
                                                    "INV""; STOP
REM     PROMPT FOR PART NUMBER
100     PRINT
        PRINT 'PART-NUMBER ':
        INPUT PN
        IF PN = '' THEN PRINT '--DONE--'; STOP
        READ INV.ITEM FROM INV.FILE,PN ELSE
           PRINT 'CANT FIND THAT PART'; GOTO 100
        END
        DESC = INV.ITEM<DESC.AMC>
        QOH = INV.ITEM<QOH.AMC>
REM     PRINT DESCRIPTION AND QUANTITY-ON-HAND
        PRINT 'DESCRIPTION -': DESC
        PRINT 'QTY-ON-HAND -': QOH
        PRINT
        GOTO 100
END
```

REM( ): Return remainder of one expression divided by another.

The REM function divides one expression by another and returns the remainder. Note that the REM *function* is not the same as the REM *statement*, which is used for documentation of a program. The REM function is functionally equivalent to the MOD function.

REM(*expr1*, *expr2*)

The REM function returns the value of the remainder after division is performed on *expr1* by *expr2*. This is also called *expr1* modulo *expr2*. The expressions can evaluate to any numeric value, with the exception that *expr2* cannot be zero.

Examples

To place the remainder in the variable NUMB when 17 is divided by 5, the code would read:

```
NUMB = REM(17,5)
```

In this instance, NUM would contain "2".

In the following application the REM function is used to calculate a customer's change after a purchase.

```
PRINT "HOW MUCH DO YOU HAVE TO SPEND? $" :
INPUT AMOUNT
NUMBER = INT(AVAILABLE/PRICE)
CHANGE = REM(AVAILABLE,PRICE)
PRINT "EACH ITEM COSTS " : PRICE"2,$" : "."
PRINT "FOR " : AMOUNT "2,$" : " YOU CAN GET " : NUMBER : "
ITEMS."
PRINT "YOUR CHANGE WILL BE " : CHANGE "2,$" : "."
```

A sample run might appear as follows (with the operator's input in bold):

```
HOW MUCH DO YOU HAVE TO SPEND? $4
EACH ITEM COSTS $1.25.
FOR $4.00 YOU CAN GET 3 ITEMS.
YOUR CHANGE WILL BE $0.25.
```

REPEAT: Terminator used with LOOP statements.

For information about the REPEAT terminator, see the LOOP statement.

REPLACE( ): Replace an attribute, value, or subvalue in an array.

The REPLACE function replaces a specified attribute, value or subvalue in an array with another expression. There are two forms of the REPLACE function listed below. The first form uses the REPLACE keyword, and the second form uses angle brackets similar to the EXTRACT angle brackets.

array = **REPLACE(***array,attr#* [,*value#* [,*subval#*]] { , | ; } *expr* **)**

array **<***attr#* [,*value#* [,*subval#*]] **>** = *expr*

array — the dynamic array to be changed.

attr# — an expression evaluating to the attribute number. If *attr#* is equal to 0, the entire array is replaced. If *attr#* evaluates to a negative number, a new attribute with the new data is appended to the end of the array. If *attr#* evaluates to a number greater than the number of attributes in the array, a new attribute with the new data is appended to the end of the array, with the correct number of empty attributes inserted.

value# — an expression evaluating to the value number. If *value#* is omitted or equal to 0, the entire contents of the

attribute is replaced. If *value#* evaluates to a negative number, a new value with the new data is appended to the end of the attribute. If *value#* evaluates to a number greater than the number of values in the attribute, a new value with the new data is appended to the end of the attribute, with the correct number of empty values inserted.

subval# an expression evaluating to the subvalue number. If *subval#* is omitted or equal to 0, the entire contents of the value is replaced. If *subval#* evaluates to a negative number or a number greater than the number of subvalues in the value, a new subvalue with the new data is appended to the end of the value. If *subval#* evaluates to a number greater than the number of subvalues in the value, a new subvalue with the new data is appended to the end of the value, with the correct number of empty subvalues inserted.

expr the string expression with which to replace the specified attribute, value, or subvalue.

In the first form of REPLACE, either a comma or a semicolon can delimit the subvalue expression from the replacement data, but if the subvalue is omitted, then a semicolon must be used.

If an attribute, value, or subvalue expression evaluates to a noninteger value, it is truncated to an integer value.

Examples

To replace Attribute 8 of the array RECORD with the string "NEW YORK", the code would read:

```
RECORD = REPLACE(RECORD, 8, "NEW YORK")
```

or:

```
RECORD<8> = "NEW YORK"
```

In the following application the dynamic array CUST.REC has the customer's billing information (credit card, account number, and expiration date) in Attributes 6, 7, and 8. The REPLACE function is used to change the customer's billing information.

```
PRINT "YOUR CURRENT BILLING IS:" :
PRINT
PRINT "CREDIT CARD: ",CUST.REC< 6 >
PRINT "ACCOUNT NO. ",CUST.REC< 7 >
PRINT "EXPIRES",CUST.REC< 8 >
PRINT "DO YOU WANT TO CHANGE THIS (Y OR N)?":
INPUT ANSWER,1
IF ANSWER = "Y" THEN
   PRINT "CREDIT CARD: ":
   INPUT CARD
   CUST.REC< 6 > = CARD
   PRINT "ACCOUNT NO.: :
   INPUT ACCT.NO
   CUST.REC< 7 > = ACCT.NO
   PRINT "EXPIRATION DATE: ":
   INPUT EXP.DATE
   CUST.REC< 8 > = EXP.DATE
END ELSE
   .
   .
   .
END
```

RETURN: Return control to the main program.

The RETURN statement returns program control from a subroutine to the main program or to a specified statement label.

RETURN [**TO** *label*]

TO *label* — pass control of the specified program to the line marked by the specified *label*.

A RETURN statement terminates a subroutine. Without the TO clause, a RETURN statement returns execution to the line following the GOSUB or CALL statement. The TO clause can be used to exit only internal subroutines called by GOSUB statements.

☞ Care should be exercised when using the TO clause. Any other GOSUBs or CALLs that were active at the time the GOSUB was executed remain active, and errors can result.

Example

In the following application a subroutine for performing all the final calculations in an accounting program is called CALCULATE, and the subroutine for exiting the program is called EXIT. The EXIT subroutine clears the screen, calls another subroutine called REPORT to print out a final report of transactions, and prints "--EOJ" before exiting.

```
    PRINT "CONTINUE WITH ALL CALCULATIONS?":
    INPUT ANS,1
    IF ANS = "Y" THEN
       GOSUB CALCULATE
    END ELSE
       GOSUB EXIT
    END
        .
        .
        .
    STOP
EXIT:
    PRINT @(-1) : "REPORT OF TRANSACTIONS:":
    GOSUB REPORT
    PRINT "-- EOJ"
    STOP
    RETURN
CALCULATE :
        .
        .
        .
    RETURN
```

REWIND: Rewind a magnetic tape to the beginning.

The REWIND statement rewinds the attached magnetic tape to the Beginning-of-Tape.

REWIND { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

THEN *statements* — execute *statements* if tape is attached and successfully rewound. For details about the syntax of THEN clauses, see the IF statement.

ELSE *statements* — execute *statements* if tape is not attached. For details about the syntax of ELSE clauses, see the IF statement.

If the tape is not attached, the ELSE clause is executed.

Example

In the following application, all records on a tape are to be displayed on the screen. Before the records can be read via a READT statement, the REWIND statement is used to ensure that the tape is positioned at the beginning.

```
REWIND ELSE
   PRINT "TAPE NOT ATTACHED!"
   GOSUB EXIT
END
LOOP
   READT RECORD ELSE
      .
      .
      .
```

RND( ): Return a random number.

The RND function returns a random positive integer.

RND *expr*

expr an expression evaluating to an integer. The random number generated will be between 0 and *expr*–1.

If *expr* evaluates to a negative value, the RND function returns 0. If *expr* evaluates to a noninteger value, the decimal part to *expr* is truncated.

Example

In the following application, a user is asked to guess a number from 1 to 10:

```
PRINT "GUESS A NUMBER FROM 1 TO 10"
INPUT GUESS
ANSWER = RND(10) + 1
IF GUESS = ANSWER THEN
   PRINT "YOU WIN"
```

```
END ELSE
   PRINT "NO, IT WAS" : ANSWER
END
```

RQM: Sleep for a specified number of seconds.*

The RQM statement causes program execution to pause for a specified number of seconds or until a given time of day. RQM stands for "release quantum": it terminates the executing program's current time-slice, or quantum. The SLEEP statement is functionally the same as the RQM statement.

RQM [*expr*]

expr an expression evaluating to the number of seconds to sleep for, or to a specific time in the form of a 24-hour clock. To specify a time, *expr* should be of the form:

hh:*mm* [:*ss*]

representing the hour, minute, and (optionally) second to sleep until. If no expression is specified, the current time-slice is released.

If the BREAK key is enabled (ON) and the BREAK key is pressed during a sleep, the debugger is entered. When "G" is pressed at the debugger prompt, the program resumes at the next statement after the RQM statement. If the BREAK key is disabled (OFF) and BREAK is pressed, it terminates the sleep without entering the debugger, and the program resumes at the next statement after RQM.

Example

In the following application, the program sleeps until 3PM and then sends a beep to the screen as a reminder to the user.

```
RQM 15:00
PRINT CHAR(7)
```

---

* Not included in the SMA standards.

SELECT: Create a select-list.

The SELECT statement selects all item IDs from a file, or the first value of all attributes in a dynamic array variable. The data selected is placed in a select-list, to be accessed by a subsequent READNEXT statement.

SELECT [*var* | *filevar*] [**TO** *select-var*]

var — the dynamic array to be selected. The select-list will contain only the first value of each multivalued attribute in the variable.

filevar — a file variable, previously assigned by an OPEN statement. The select-list will contain all item IDs in the system file opened to *filevar*.

select-var — assign the name *select-var* to the select-list. If *select-var* is not specified, the default select-list variable will be used.

The SELECT statement forms a list of all item IDs from the specified file. The file must have been opened by an OPEN statement before data can be selected. The list created by the SELECT statement can then be accessed by a subsequent READNEXT statement.

Alternatively, the SELECT statement can be used to form a select-list of the first value of each attribute in a dynamic array.

For a select-list of item IDs, either the SELECT statement can be used, or the TCL select-list generators (SELECT, SSELECT, FORM-LIST, etc.) can be called with the EXECUTE statement. The SELECT statement performs the same function as the TCL SELECT verb without any selection or sort expressions, and in that respect is less versatile than using the EXECUTE statement to access TCL.

Example

In the following application we create an alphabetical list of each item ID in the file AIRPORTS. The file is selected to the select variable LIST with the SELECT statement. Each item is then read from LIST with the READNEXT statement. The actual alphabetizing is accomplished with the LOCATE and INS statements.

```
EQUATE TRUE TO 1, FALSE TO 0
OPEN "AIRPORTS" TO AIRPORTFILE ELSE
   ABORT 201, "AIRPORTS"
END
SELECT AIRPORTFILE TO LIST
ALPH.LIST = ""
END.OF.LIST = FALSE
LOOP
   READNEXT ID FROM LIST ELSE
      END.OF.LIST = TRUE
   END
UNTIL END.OF.LIST DO
   LOCATE ID IN ALPH.LIST BY "AL" SETTING POSITION THEN
      PRINT ID : " DUPLICATE ENTRY! POSSIBLE FILE CORRUPTION"
      ABORT
   END ELSE
      INS ID BEFORE ALPH.LIST<POSITION>
   END
REPEAT
```

SEQ( ): Return the decimal value of an ASCII character.

The SEQ function returns the decimal value for the supplied character.

SEQ(*expr* **)**

expr an expression evaluating to a single character. If *expr* evaluates to a string longer than one character, only the first character is evaluated.

The SEQ function returns the decimal value for the ASCII character given. It is the inverse of the CHAR function.

See Appendix C for ASCII character codes.

Examples

To return the ASCII decimal value for a space into a variable SPACESEQ, the code would read:

```
SPACESEQ = SEQ (" ")
```

In this instance, the variable SPACESEQ would contain "32".

In the following application, a string has standard control characters (decimal 1 through decimal 31) converted into printable equivalents. The

SEQ( )

SEQ function is used to return the ASCII decimal value of each character, and that value is then used to determine whether the character falls into the ASCII range of control characters.

```
NO.OF.CHARS = LEN(STRING)
NEW.STRING = ""
FOR I = 1 TO NO.OF.CHARS
   CHARACTER = STRING[ I ,1 ]
   ASC = SEQ(CHARACTER)
   IF (ASC > 0 AND ASC < 32 ) THEN
      NEWCHAR = "^" : CHAR( ASC + 64 )
   END ELSE
      NEWCHAR = CHARACTER
   END
   NEWSTRING := CHARACTER
NEXT I
STRING = NEW.STRING
```

SIN( ): Return the sine of the expression.

The SIN function returns the trigonometric sine of the expression.

SIN(*expr* **)**

The expression *expr* is treated as an angle expressed as a numeric value in degrees. Values outside the range of 0 to 360 degrees are interpreted as modulo 360.

Example

In the following application the SIN function is used with a standard trigonometric formula to calculate the cosine of an angle without using the COS function.

```
COSINE = SQRT(1 – SIN(ANGLE) * SIN(ANGLE) )
PRINT " THE COSINE IS CALCULATED AS : " : COSINE
```

SLEEP: Sleep for a specified number of seconds.

The SLEEP statement causes program execution to pause for a specified number of seconds, or until a given time of day. SLEEP terminates the executing program's current time-slice, or quantum (see the RQM

statement). The RQM statement is functionally the same as the SLEEP statement.

SLEEP [*expr*]

expr an expression evaluating to the number of seconds to sleep for, or to a specific time in the form of a 24-hour clock. To specify a time, *expr* should be of the form:

hh :*mm* [:*ss*]

representing the hour, minute, and (optionally) second to sleep until. If no expression is specified, the current time-slice is released.

If the BREAK key is enabled (ON) and the BREAK key is pressed during a sleep, the debugger is entered. When "G" is pressed at the debugger prompt, the program resumes at the next statement after the SLEEP statement. If the BREAK key is disabled (OFF) and BREAK is pressed, it terminates the sleep without entering the debugger, and the program resumes at the next statement after SLEEP.

Example

In the following application, the program sleeps until 3PM and then sends a beep to the screen as a reminder to the user.

```
SLEEP 15:00
PRINT CHAR(7)
```

SOUNDEX( ): Convert a string into its phonetic equivalent.*

The SOUNDEX function uses the soundex algorithm to convert a string into its phonetic equivalent.

SOUNDEX(*expr* **)**

expr an expression evaluating to a string value. Any nonalphabetic characters in *expr* are ignored.

---

* Not included in the SMA standards. Prime INFORMATION implements the soundex algorithm with the ICONV SOUNDEX (S) conversion code.

The SOUNDEX function evaluates the *expression* and returns the first alphabetic letter of the string, followed by a 1- to 3-digit phonetic code. The soundex algorithm is used to analyze the input string. The codes are assigned as follows:

Code	Letters
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Any letters not included in the above assignments are ignored by the SOUNDEX function, as are all nonalphabetic characters.

Consecutive duplicate codes are not repeated by the SOUNDEX function. That is, SOUNDEX("LLLLLLLL") will return L4, regardless of how many Ls are repeated, and SOUNDEX("DAMNATION") will return "D535" (the M and N are considered phonetically equivalent).

The purpose of the SOUNDEX function is to provide protection against strings not being matched because one was misspelled. However, it is not recommended to accept its conclusions without some confirmation. Some of the phonetic matches which the SOUNDEX function generates can be unrealistic.

Example

In the following application, each item ID in a file SOUNDFILE is a SOUNDEX code, and contains a list of the customers whose names are SOUNDEX equivalents to that code. When a customer's name is entered, the SOUNDEX code is taken and the corresponding file item is read into a dimensioned array XREF for cross-reference.

```
DIM XREF(1000)
FOUND = TRUE
PRINT "ENTER NAME:" :
INPUT NAME
LIKE = SOUNDEX(NAME)
MATREAD XREF FROM SOUNDFILE, LIKE ELSE
   PRINT "NO LIKE ENTRIES"
```

```
END ELSE
   .
   .
   .
```

SPACE( ): Generate a specified number of spaces.

The SPACE function produces a string of the specified number of blank spaces.

SPACE(*expr* **)**

expr an expression specifying the number of spaces to be generated.

Use the SPACE function to return a string composed of blank spaces. It is essentially the same as STR(" ",*expr*)

Example

To print Attribute 1 of a dynamic array indented five spaces, enter:

```
PRINT SPACE(5) : RECORD<1>
```

SQRT( ): Return the square root of an expression.

The SQRT function returns the square root of an expression, given that the expression evaluates to a positive number.

SQRT(*expr* **)**

expr an expression evaluating to a numeric value.

If *expr* evaluates to a negative value, the SQRT function returns a value of 0.

Example

The following example demonstrates how the SQRT function can be used to calculate the diagonal of a given rectangle:

```
DIAGONAL = SQRT(HEIGHT ^ 2 + WIDTH ^ 2)
PRINT "THE DIAGONAL OF THE RECTANGLE IS " : DIAGONAL : "."
```

STOP: Terminate execution of a program.

The STOP statement terminates the program and returns the user to the calling environment.

STOP [*errmsg* [,*parameter1*, *parameter2*, ...]

errmsg is an integer corresponding to an error message from the system message file (ERRMSG). The message is output upon termination of the program.

parameter1, *parameter2*, ...
are parameters passed to the error message.

The STOP statement terminates program execution and returns system control to the calling environment, which can be a menu, another Pick BASIC program, or a proc. When an error message is specified, it is printed on the terminal screen before the STOP statement is executed, with the specified parameters inserted.

The STOP statement can be used in any part of a program. It is generally used at the logical end of a program or in the ELSE part of a file I/O or tape I/O statement.

To stop the program and return directly to TCL, use the ABORT statement.

Examples

The following example demonstrates how the STOP statement can be used to terminate a program when it fails to locate a file item.

```
READ CUST.REC FROM CUSTFILE,ID ELSE
   PRINT "ITEM " : ID :" NOT FOUND!"
   STOP
```

```
END
   .
   .
   .
```

In the next example, the STOP statement is used to signify the end of program execution. The text below the STOP is not executed directly as part of the program sequence but can be accessed through GOTO or GOSUB statements. At run time the program stops at the STOP statement, but the source lines have been compiled and can be executed nonsequentially.

```
   .
   .
   .
   GOSUB ADDEMUP
   .
   .
   .
STOP
ADDEMUP:  * THIS SUBROUTINE DOES THE COMPUTATION.
   .
   .
   .
```

Had an END statement been used instead of a STOP statement, the ADDEMUP routine would not have been compiled.

STR( ): Repeat a character string *n* times.

The STR function repeats the given character string a specified number of times.

STR(*string,number***)**

string — the string expression to be repeated in the generated string.

number — an expression specifying the number of times *string* is to be repeated in the result. If *number* does not evaluate to an integer, it is truncated. If *number* evaluates to a value less than 1, no value is returned.

The STR function produces a specified number of repetitions of a particular character string.

Examples

To print ten asterisks on the screen, the code might read:

```
PRINT STR(" * ", 10)
```

In the following application, a table is printed listing an initial mortgage, interest rate, amount of payments, and the number of years and months it will take to pay off the mortgage at that rate. The STR function is used to produce a line across an 80-column screen between the heading and the data.

```
PRINT "MORGAGE","RATE","PAYMENTS","YEARS","MONTHS"
PRINT STR("_",80)
PRINT
PRINT ORIG.MORG"L,$",ORIG.RATE,PAYMENT"L2,$",YEARS MONTH
```

SUBROUTINE: Identify a subroutine.

The SUBROUTINE statement must be the first statement in an external subroutine.*

SUBROUTINE [*name*] [(*var1* [,*var2*] ...)]

name is the name of the subroutine.

var ... variables to be assigned by the corresponding CALL statement. If *var* is an array variable, it must be preceded by the MAT keyword.

An external subroutine is a subroutine which has been compiled and cataloged separately from the programs that call it. It can then be called by any program with the CALL statement. When a RETURN statement is encountered without a corresponding GOSUB, program control returns to the main program. If there is no ending RETURN statement, control will not return to the main program. See the CALL statement for more information.

The SUBROUTINE statement must be the first line in an external subroutine. The name of the subroutine can be supplied, but it is not necessary since the subroutine is addressed by its item ID.

The variables *var* must make a one-to-one correspondence with the parameters supplied on the corresponding CALL statement line. Each of the

---

* Some implementations allow comment lines before the SUBROUTINE statement.

parameters listed in the CALL syntax line are passed into the corresponding variable list on the SUBROUTINE syntax line. Other than their positions on the CALL and SUBROUTINE syntax lines, there is no correspondence between variable names in the calling program and the subroutine.

An alternative way of passing variables between programs and subroutines is to use COMMON statements in both program and subroutine. See the COMMON statement for more information.

If an array variable is passed, it must be preceded by the MAT keyword and dimensioned in both the main program and the subroutine. See the CALL statement for information on passing arrays.

Examples

The first line of the subroutine ADDTHEM might read:

```
SUBROUTINE ADDTHEM (X, Y, Z)
```

and the corresponding line for calling ADDTHEM might read:

```
CALL ADDTHEM (A, B, C)
```

Variable A is passed to variable X, B is passed to Y, and C is passed to Z. When the subroutine has finished, these values are passed in the opposite direction.

SUM( ): Add elements of a dynamic array.*

The SUM function calculates the sum of numeric data in a dynamic array string.

SUM(*string* **)**

string the dynamic array string.

Data is added by the SUM function only at the lowest delimiter level in the string. The SUM function adds each group of lowest-level elements into a single number, removing the lowest-level delimiters.

If the string contains all three levels (subvalues, values, and attributes), the SUM function will have to be called three times to produce a single numeric

---

* Not included in the SMA standards.

value; if the string contains two of the three levels, the SUM function will have to be called twice.

Examples

If STRING contains:

```
10^20\30\40^50]60\70]80^90
```

then:

```
STRING= SUM(STRING)
```

produces:

```
10^90^50]130]80^90
```

If we once again performed STRING= SUM(STRING), we would have:

```
10^90^260^90
```

and STRING = SUM(STRING) again would finally give us:

```
450
```

In the following application, a program to maintain inventory for a shoe store keeps a record of each shoe style in a separate file item. The format for each file item is that each attribute represents a color that the shoe style is manufactured in, and within each attribute are eight multivalues representing different sizes in that color, ranging from 5 1/2 to 9.

A program to perform a quick calculation of the number of shoes still available in that style might read:

```
EQUATE AM TO CHAR(254)
OPEN "SHOES" TO STYLEFILE ELSE ...
PRINT "ENTER STYLE NO."
INPUT ID
READ STYLEREC FROM STYLEFILE, ID ELSE ...
COLORREC = SUM(STYLEREC)
NO.OF.COLORS = DCOUNT(STYLEREC, AM)
FOR I = 1 TO NO.OF.COLORS
   PRINT "COLOR " : I, COLORREC< I >
NEXT I
PRINT "TOTAL ON HAND", SUM(COLORREC)
```

SYSTEM( ): Return general status information about the system.

The SYSTEM function returns status information about the operator's terminal, the current output device, the tape drive, or a variety of other system variables. Like the @ function, the SYSTEM function is implemented differently on different Pick systems. Check the documentation supplied with your system for complete details about the use of SYSTEM function codes.

SYSTEM (*expr*)

expr an expression evaluating to a system function code, some of which are listed in the tables of system function codes.

The SYSTEM function checks on the status of the system function specified. Table 5-3 lists the SMA standard system function codes accepted by the SYSTEM function, and gives the significance of the returned values. If *expr* evaluates to a number not listed below, 0 is assumed.

Table 5-3. SMA Standard System Function Codes.

Code	Function
0	The error code of the last READT, WRITET, WEOF, or REWIND statement. Error codes are as follows: 1 Tape unit is not attached. 2 EOF read from tape unit. 3 Attempt was made to write a null string to tape. 4 Attempt was made to write a record longer than tape record length.
1	0 if program output is being sent to the terminal, 1 if program output is being sent to a printer device.
2	Current page width on the screen.
3	Current number of lines on the screen.
4	Number of lines remaining on current page of output.
5	Number of lines already printed on the current page of output. (On most systems, 5 gives the current page number.)
6	Current page number. (On most systems, 6 gives the number of lines already printed on the current page of output.)
7	A single character representing the terminal type code. See the TERM verb for possible values.
8	Current tape record length.

Table 5-4 lists some additional system function codes that follow the same conventions on many Pick systems. They are not included in the SMA standards, and not all implementations (Ultimate, for example) follow these conventions.

Table 5-4. Other System Function Codes.

Code	Function
9	Current number of charge units.
10	0 if the secondary output buffer is empty, 1 if it contains data.
11	1 if the unassigned select-list is present, 0 otherwise.
12	System time in milliseconds.
15	Options used to invoke execution of current program, in the form: *options*^*n*^*m* where ^ is an attribute mark (CHAR(254)) and: *options* are the letter options used. *n* is the first number specified. *m* is the second number specified.

The SYSTEM function can be very useful for programs which create reports and perform tape handling.

Examples

In the following application designed for a GA system, the program uses the SYSTEM function to check the user's privilege level.

```
SYSPRIV = SYSTEM(20)
IF SYSPRIV < 2 THEN
   PRINT "Your privilege level is insufficient "
   PRINT "to run this program."
   ABORT
END
```

In the next application, a WRITET statement is used to write a record onto a tape. Although the SYSTEM(0) function's response of "4" is reserved for data strings longer than the tape record length, the WRITET statement avoids this eventuality by truncating the data and printing an error message. The SYSTEM(8) function is therefore used to determine if the data is too long, and to truncate it into the appropriate number of segments if necessary. In the ELSE clause of the WRITET statement the SYSTEM(0)

function is then used to determine the error code and print the corresponding message.

```
MAX.LEN = SYSTEM(8)
REC.LEN = LEN(CURR.REC)
NO.OF.SEGS = REC.LEN / MAX.LEN
NO.OF.SEGS = INT(NO.OF.SEGS)
IF REM(REC.LEN,MAX.LEN) THEN
   NO.OF.SEGS + = 1
END
FOR I = 1 TO NO.OF.SEGS
   BEGIN.COL = (MAX.LEN * (I–1)) + 1
   SEGMENT = CURR.REC[BEGIN.COL, MAX.LEN]
   WRITET SEGMENT ELSE
      ERROR.CODE = SYSTEM(0)
      BEGIN CASE
         CASE ERROR.CODE = 1
            PRINT "ERROR! TAPE NOT ATTACHED"
            GOSUB EXIT
         CASE ERROR.CODE = 3
            PRINT ID:": ITEM IS EMPTY"
      END CASE
   END
NEXT I
```

The next example prints a string of asterisks across the screen:

```
COLS=SYSTEM(2)
PRINT STR("*",COLS)
```

Finally, if the programmer needed to determine if the I option to RUN was used, the code might read:

```
EXEC.STRING = SYSTEM(15)
LETTERS = EXEC.STRING<1>
IF COUNT(LETTERS, "I") THEN ...
```

TAN( ): Return the tangent of the expression.

The TAN function returns the trigonometric tangent of the expression.

TAN(*expr* **)**

The expression *expr* is treated as an angle expressed as a numeric value in degrees. Values outside the range of 0 to 360 degrees are interpreted as modulo 360.

TAN( )

Example

To assign the variable TANGENT to the tangent of an angle ANGLE, enter:

```
TANGENT = TAN(ANGLE)
```

THEN: Initiator used with conditional statements.

For information about the THEN | ELSE construct, see the IF statement.

TIME( ): Return the time of day in seconds.

The TIME function returns the current system time in internal format.

TIME( )

The internal time expression represents the number of seconds, to the nearest second, that have passed since midnight.

The DATE function can be used to return the current date in internal format. To receive both the time and the date in external format, use the TIMEDATE function.

Example

In the following application, the program checks the time of day and informs the user if it is too late to record the transaction that afternoon.

```
TIME = TIME( )
FIVE.PM = 3600 * 17
IF TIME > FIVE.PM THEN
   PRINT "IT IS CURRENTLY AFTER FIVE PM."
   PRINT "YOUR TRANSACTION WILL NOT BE RECORDED" :
   PRINT "UNTIL THE NEXT BUSINESS DAY."
END ELSE
   GOSUB RECORD
END
```

TIMEDATE( ): Return the time and date in external format.

The TIMEDATE function returns the current system time and date in external format.

TIMEDATE( )

The current time and date is returned by the TIMEDATE function in the following form:

```
HH:MM:SS   DD MMM YYYY
```

where:

HH	hours (based on 24-hour clock)
MM	minutes
SS	seconds
DD	day
MMM	month
YYYY	year

To receive just the current time in internal format, use the TIME function. To receive just the current date in internal format, use the DATE function. The TIME function returns the number of seconds since midnight, and the DATE function returns the number of days since December 31, 1967.

Example

In the following application, the current time is shown at the beginning of the program. To get the current time, the TIMEDATE function is used and the date is stripped out with the FIELD function.

```
CURR.TIME = TIMEDATE( )
CURR.TIME = FIELD(CURR.TIME," ",1)
PRINT "BEGINNING THIS SESSION AT " : CURR.TIME
```

TRIM( ): Remove extraneous blanks from a string.

The TRIM function removes any extraneous blank spaces from the given string.

TRIM(*expr* **)**

expr an expression evaluating to the string to be trimmed.

The TRIM function reduces multiple consecutive blank spaces and tabs in *expr* to one blank space or tab, and removes all leading and trailing spaces.

When creating a new file item, it is a good practice to TRIM the name of the new item, to provide consistency and avoid later confusion.

Example

In the following example the TRIM function is used as soon as the operator enters a last name which will be used as an item ID. Any extra spaces that the operator might have entered in the name are then stripped out, making the item ID more likely to be matched successfully.

```
PRINT "ENTER LAST NAME: " :
INPUT NAME
NAME = TRIM(NAME)
READ RECORD FROM GUESTS,NAME ELSE
   FOUND = 0
END
```

TRIMB( ): Remove trailing blanks from a string.*

The TRIMB function removes any trailing blank spaces from the given string.

TRIMB(*expr* **)**

expr an expression evaluating to the string to be trimmed.

The TRIMB function removes all trailing spaces. Any spaces embedded in the string or leading the string are left intact.

Example

In the following application the TRIMB and TRIMF functions are used on the address input by the operator.

```
PRINT "ENTER YOUR ADDRESS : " :
INPUT ADDRESS
ADDRESS = TRIMF(ADDRESS)
ADDRESS = TRIMB(ADDRESS)
```

TRIMF( ): Remove leading blanks from a string.*

The TRIMF function removes any leading blank spaces from the given string.

TRIMF(*expr* **)**

expr an expression evaluating to the string to be trimmed.

The TRIMF function removes all leading spaces. Any spaces embedded in the string or trailing the string are left intact.

Example

In the following application, the customer's name is entered last name first. The last name is used as an item ID, and the first name becomes the first attribute of the file item. The FIELD function separates the last name from the first name, and the TRIMF function removes the leading space that generally follows the comma.

```
PRINT "ENTER YOUR NAME (LAST NAME, FIRST NAME): ":
INPUT NAME
ITEM.ID = FIELD(NAME , " , " , 1)
ITEM.ID = TRIM(ITEM.ID)
FIRST.NAME = FIELD(NAME , " , " , 2)
FIRST.NAME = TRIMF(FIRST.NAME)
```

* Not included in the SMA standards.

UNLOCK: Release execution locks.

The UNLOCK statement releases execution locks set by the program.

UNLOCK [*expr*]

expr an expression evaluating to the lock number to be released. Valid lock numbers are from 0 to at least 63.* If *expr* is omitted, all execution locks set by the program are released.

The UNLOCK statement releases execution locks that were set by a LOCK statement in the program. It can be used either to release a specific lock or to release all locks currently set.

In writing a subroutine you should unlock any execution locks explicitly with the lock number, otherwise locks which might have been established by the main program could be lifted prematurely.

The UNLOCK statement does not affect item locks established with the READU, READVU, or MATREADU statements. Item locks are released with the RELEASE statement or with a WRITE, WRITEV, or MATWRITE statement.

If there are no current execution locks set, or if the lock number specified has not been locked, no action will be taken.

Example

In the following application the external subroutine REMOTE.MAIL uses the LOCK statement for execution lock number 55. The UNLOCK statement is used at the end of the subroutine. Note that lock number 55 is specified for the UNLOCK statement, so only the specific lock set by the current program is affected.

```
SUBROUTINE REMOTE.MAIL (MESSAGE, MODEMLINE)
LOCK 55 ELSE
   .
   .
   .
UNLOCK 55
RETURN
```

* Many Pick systems support more than 64 locks.

UNTIL: Initiator used with FOR and LOOP statements.

For information about the UNTIL initiator, see the FOR and LOOP statements.

WEOF: Write an End-of-File mark.

The WEOF statement writes an End-of-File mark at the current position of an attached magnetic tape.

WEOF { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

THEN *statements* — execute *statements* if tape is attached and EOF is successfully written. For details about the syntax of THEN clauses, see the IF statement.

ELSE *statements* — execute *statements* if tape is not attached. For details about the syntax of ELSE clauses, see the IF statement.

Example

After writing a tape, the code might read:

```
WEOF THEN
   WRITET RECORD ELSE
   PRINT "EOF CANNOT BE WRITTEN"
   GOSUB EXIT
END
```

WHILE: Initiator used with FOR and LOOP statements.

For information about the WHILE initiator, see the FOR and LOOP statements.

WRITE: Write an item to a file.

The WRITE statement writes an item to a file.

WRITE *array* **ON** [*filevar*,] *item-ID*

array	an expression evaluating to the dynamic array representing the contents of the file item.
filevar	the file variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.
item-ID	an expression evaluating to the item ID of the item to be written.

The WRITE statement writes a new value to the specified item of a file. Any data previously stored in the specified file item is overwritten. If the item does not exist, a new item is created.

When updating a file, the WRITE statement releases the item lock set with a READU statement. To maintain the item lock as it was set by the READU statement, use the WRITEU statement instead of WRITE. See the WRITEU statement for more information.

Example

In the following application, Attribute 6 contains the customer's phone number. A new phone number is prompted for and then placed into Attribute 6. The new item is then written back to the file.

```
PRINT "ENTER THE NEW PHONE NUMBER: ":
INPUT NEW.PHONE
CUST.REC<6> = NEW.PHONE
WRITE CUST.REC ON CUSTFILE,ITEM.ID
```

WRITET: Write a tape record onto a magnetic tape.

The WRITET statement writes the given text as the next record on the attached magnetic tape.

WRITET *expr* { **THEN** *statements* [**ELSE** *statements*] | **ELSE** *statements* }

expr	an expression evaluating to the text to be written to the tape.
THEN *statements*	execute *statements* if tape is successfully written. For details about the syntax of THEN clauses, see the IF statement.
ELSE *statements*	execute *statements* if tape cannot be written. For details about the syntax of ELSE clauses, see the IF statement.

The WRITET statement writes a record at the current position on an attached magnetic tape. If the data to be written is longer than the maximum record length on the tape, the data is truncated and an error message is printed.

A tape might not be writable because the tape has not been attached or because the record to be written evaluates to the null string. To determine why a tape record could not be written, the SYSTEM function is often used in the ELSE portion of a WRITET statement. See the SYSTEM function for more information.

Example

In the following application, each item of a file is written to tape. After SELECTing the item IDs of the file, the READNEXT statement is used to read each item ID and each item is then read and written to tape. If the item is not written, the SYSTEM function determines the error message, and an appropriate error message is printed.

```
SELECT TEMPFILE
END.OF.LIST = 0
LOOP
   READNEXT ID ELSE
      END.OF.LIST = 1
   END
```

```
UNTIL END.OF.LIST DO
   READ CURR.REC FROM TEMPFILE,ID THEN
      WRITET CURR.REC ELSE
         ERROR.CODE = SYSTEM(0)
         BEGIN CASE
            CASE ERROR.CODE = 1
               PRINT "ERROR!  TAPE NOT ATTACHED"
               GOSUB EXIT
            CASE ERROR.CODE = 3
               PRINT ID:" : ITEM IS EMPTY"
         END CASE
      END
   END ELSE
      PRINT "ERROR IN READING RECORD ":ID
      PRINT "PRESS RETURN TO ACKNOWLEDGE" :
      INPUT RESP
   END
REPEAT
```

WRITEU: Write an item to a file, retaining item locks.

The WRITEU statement writes an item to a file, leaving item locks intact.

WRITEU *array* **ON** [*filevar,*] *item-ID*

array — an expression evaluating to the dynamic array representing the contents of the file item.

filevar — the file variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.

item-ID — an expression evaluating to the item ID of the item to be written.

The WRITEU statement is identical to the WRITE statement, except that item locks are not affected by the WRITEU statement. The new value is written to the specified item of a file, but any item locks remain intact.

Example

In the following application the WRITEU statement is used to update inventory information which is still being processed, and is therefore not available to other users for updating.

```
.
.
.
INPUT QTY.ORD
INV.ITEM <10> += QTY.ORD ; *Update qty committed
WRITEU INV.ITEM ON INV.FILE, PN
.
.
.
```

WRITEV: Write the value of one attribute to a file.

The WRITEV statement writes or updates a single attribute of an item.

WRITEV *expr* **ON** [*filevar,*] *item-ID,attr#*

expr — an expression evaluating to the dynamic array variable representing the contents of the attribute.

filevar — the file variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.

item-ID — an expression evaluating to the item ID of the item to be written.

attr# — an expression evaluating to the number of the attribute to be written. If *attr#* evaluates to a noninteger, it is truncated. If *attr#* evaluates to 0, *expr* is written to Attribute 1 and all existing attributes are pushed up one (i.e., Attribute 1 becomes Attribute 2, Attribute 2 becomes Attribute 3, and so on). If *attr#* evaluates to a negative value, *expr* is written as the last attribute in the item, and all previous attributes remain unchanged. In this respect the behavior of the WRITEV statement parallels that of the INS statement.

The WRITEV statement writes a new value to the specified attribute of a file item. Any data previously stored in the specified attribute is overwritten. If the item or attribute does not exist, a new attribute or item is created.

When updating a file, the WRITEV statement releases the item lock set with a READVU statement. To maintain the item lock as it was set by the READVU statement, use a WRITEVU statement instead of WRITEV. See the WRITEVU statement for more information.

Example

In the following application, Attribute 6 of the file opened as CUST.FILE contains the customer's phone number. A new phone number is prompted for, and the new item is then written back to the file.

```
READV PHONE FROM CUST.FILE,ID,6 ELSE
   PRINT "CREATING A NEW RECORD"
END
PRINT "ENTER THE NEW PHONE NUMBER: ":
INPUT NEW.PHONE
WRITEV NEW.PHONE ON CUSTFILE,ITEM.ID, 6
```

WRITEVU: Write an item to a file, retaining item locks.

The WRITEVU statement writes a single attribute of an item to a file, leaving item locks intact.

WRITEVU *expr* **ON** [*filevar,*] *item-ID,attr#*

expr	an expression evaluating to the dynamic array variable representing the contents of the attribute.
filevar	the file variable to which the file was opened. If *filevar* is not specified, the default file variable is used, which is the file most recently opened without an assigned file variable.
item-ID	an expression evaluating to the item ID of the item to be written.
attr#	an expression evaluating to the number of the attribute to be written. If *attr#* evaluates to a noninteger, it is

truncated. If *attr#* evaluates to 0, *expr* is written to Attribute 1 and all existing attributes are pushed up one (i.e., Attribute 1 becomes Attribute 2, Attribute 2 becomes Attribute 3, and so on). If *attr#* evaluates to a negative value, *expr* is written as the last attribute in the item, and all previous attributes remain unchanged. In this respect the behavior of the WRITEVU statement parallels that of the INS statement.

The WRITEVU statement is identical to the WRITEV statement, except that item locks are not affected by the WRITEVU statement. The new value is written to the specified attribute of a file, but any item locks remain intact.

Example

In the following application the WRITEVU statement is used to update inventory information which is still being processed, and is therefore not available to other users for updating.

```
.
.
.
INPUT QTY.ORD
INV.ITEM <10> += QTY.ORD ; *Update qty committed
WRITEVU INV.ITEM<10> ON INV.FILE, PN, 10
.
.
.
```

APPENDIXES

APPENDIX A

Pick BASIC Program Examples

Appendix A provides examples of actual Pick BASIC programs. The programs are diverse in application in order to show a variety of programming techniques.

Programming Example 1: Triples

Example 1 demonstrates the use of internal subroutine branching. This program finds pythagorean triples.

```
        PRINT
        PRINT 'SOME PYTHAGOREAN TRIPLES ARE:'
        PRINT
        FOR A=1 TO 40
           FOR B=1 TO A-1
              CC=A*A+B*B
              GOSUB 50
              IF C = INT(C) THEN PRINT B,A,C
           NEXT B
        NEXT A
        STOP
*       SQUARE ROOT SUBROUTINE
50      C=CC/2
        FOR I=1 TO 20
           X=(C+CC/C)/2
           IF C = X THEN RETURN
           C=X
        NEXT I
        RETURN
END
```

Programming Example 2: Guess

Example 2 demonstrates the use of conditional branching within a loop. This program is a guessing game.

```
        HEADING ''
        HISSCORE=0; YOURSCORE=0
10      PAGE
        PRINT 'GUESS NUMBERS BETWEEN 0 AND 100'
        PRINT 'MACHINE: ':HISSCORE:'    ':'YOU:':YOURSCORE
        PRINT
        NUM=RND(101)
        FOR I=1 TO 6
           PRINT 'GUESS ':I:' ':
           INPUT GUESS
           IF GUESS=NUM THEN
              PRINT
              PRINT 'CONGRATULATIONS, YOU WON!'
              YOURSCORE=YOURSCORE+1
              GOTO 60
           END
           IF GUESS<NUM THEN PRINT 'HIGHER'
           IF GUESS>NUM THEN PRINT 'LOWER'
        NEXT I
        PRINT
        PRINT 'YOU LOST YOU DUMMY, YOUR NUMBER WAS ':NUM
        HISSCORE=HISSCORE+1
60      PRINT
        PRINT 'AGAIN?':
        INPUT X
        IF X = 'NO' THEN STOP
        GOTO 10
END
```

Programming Example 3: INV-INQ

Example 3 demonstrates the use of file inquiry logic. This program queries an inventory file. It reads the dictionary of file INV to get the attribute numbers of DESC (description) and QOH (quantity on hand). The program then prompts the user for a part number, which is the item ID of an item in INV, and uses the attribute numbers to read and display the part description and quantity on hand. The program loops until a null part number is entered.

```
*---
*---    GET ATTRIBUTE DEFINITIONS FROM DICT OF INVENTORY
                                                          FILE
        OPEN 'DICT', 'INV' TO INV.DICT ELSE PRINT 'CANT OPEN
                                                "DICT INV""; STOP
        READV DESC.AMC FROM INV.DICT,2 ELSE
           PRINT 'CANT READ "DESC" ATTR'; STOP
        END
        READV QOH.AMC FROM 'QOH',2 ELSE
           PRINT 'CANT READ "QOH" ATTR'; STOP
        END
*---    OPEN DATA PORTION OF INVENTORY FILE
        OPEN '', 'INV' TO INV.FILE ELSE PRINT 'CANNOT OPEN
                                                    "INV""; STOP
*---    PROMPT FOR PART NUMBER
100     PRINT
        PRINT 'PART-NUMBER ':
        INPUT PN
        IF PN = '' THEN PRINT '--DONE--'; STOP
        READ INV.ITEM FROM INV.FILE,PN ELSE
           PRINT 'CANT FIND THAT PART'; GOTO 100
        END
        DESC = INV.ITEM<DESC.AMC>
        QOH = INV.ITEM<QOH.AMC>
*---    PRINT DESCRIPTION AND QUANTITY-ON-HAND
        PRINT 'DESCRIPTION -': DESC
        PRINT 'QTY-ON-HAND -': QOH
        PRINT
        GOTO 100
END
```

Programming Example 4: Format

Example 4 demonstrates the use of structured block format. This program formats a Pick BASIC program to display block structuring by indenting lines.

```
*---- DEFINITIONS
10      SP = 6      ; * LEFT MARGIN COLUMN NUMBER
        ID = 3 ; * NUMBER OF SPACES TO INDENT
*---- INITIALIZATION
        SPX = SP
        LINE.NO = 0
*---- INPUT FILE NAME AND PROGRAM NAME
        PRINT
        PRINT
        PRINT 'PICK.BASIC FILE NAME    - ':; INPUT FILE
        IF FILE = '' THEN STOP
        OPEN '', FILE ELSE PRINT 'CANT OPEN FILE -': FILE; GOTO 10
        PRINT 'PICK.BASIC PROGRAM NAME - ':; INPUT NAME
        IF NAME = '' THEN GOTO 10
        NEWITEM = ''
        READ ITEM FROM NAME ELSE
            PRINT 'CANNOT FIND THAT PROGRAM'
            GOTO 10
        END
*---- GET NEW LINE, IF NONE - THEN DONE
100     LINE.NO = LINE.NO + 1
        LINE = EXTRACT(ITEM,LINE.NO,0,0)
        IF LINE = '' THEN
            WRITE NEWITEM ON NAME
            PRINT; PRINT; PRINT '--DONE--'; GOTO 10
        END
        LABEL = ''
*---- STRIP OFF LEADING/TRAILING SPACES
200     IF LINE[1,1] = ' ' THEN LINE = LINE[2,32767]; GOTO 200
210     IF LINE[LEN(LINE),1] = ' ' THEN
            LINE = LINE[1,LEN(LINE)-1]; GOTO 210
        END
*---- LOOK FOR A COMMENT ('*', '!', OR 'REM')
        IF LINE[1,1] = '*' THEN GOTO 1500
        IF LINE[1,1] = '!' THEN GOTO 1500
        IF LINE[1,3] = 'REM' THEN GOTO 1500
*---- LOOK FOR 'FOR'
        IF LINE[1,4]='FOR ' AND INDEX(LINE,'NEXT ',1)>0 THEN
            GOTO 2000
        END
        IF LINE[1,4]='FOR ' AND INDEX(LINE,'NEXT ',1)=0 THEN
            GOTO 1000
        END
*---- LOOK FOR 'END'
```

```
        IF LINE = 'END' THEN GOTO 1100
        IF LINE[1,4] = 'END ' THEN
           IF LINE[LEN(LINE)-4,5] = ' ELSE' THEN GOTO 1200
        END
*---- LOOK FOR 'NEXT'
        IF LINE[1,5] = 'NEXT ' THEN GOTO 1100
*---- EXTRACT LEADING NUMERIC LABEL
        IF LINE[1,1] MATCHES '1N' THEN
           L = 2
300        IF LINE[L,1] MATCHES '1N' THEN L=L+1;  GOTO 300
           LABEL = LINE[1,L-1]
           LINE = LINE[L,32767]
           GOTO 200
        END
*---- LOOK FOR LINE ENDING IN ' ELSE' OR ' THEN' ('IF' OR 'READ')
        X = LINE[LEN(LINE)-4,5]
        IF X = ' THEN' THEN GOTO 1000
        IF X = ' ELSE' THEN GOTO 1000
*---- THIS IS JUST ANOTHER LINE, THEREFORE NO CHANGE
        GOTO 2000
*---- INDENT ON SUBSEQUENT LINES
1000    SP = SP - ID
        GOTO 2000
*---- OUTDENT ON THIS AND SUBSEQUENT LINES
1100    SP = SP - ID
*---- OUTDENT THIS LINE ONLY
1200    SPX = SPX - ID
        GOTO 2000
*---- PRINT WITH NO INDENTATION
1500    SPX = 0
*---- WRITE NEW LINE
2000    NEW.LINE = LABEL :  STR(' ', SPX-LEN(LABEL)) :  LINE
        PRINT NEW.LINE
        NEWITEM = REPLACE(NEWITEM,LINE.NO,0,0,NEW.LINE)
        SPX = SP
        GOTO 100
END
```

Programming Example 5: Lot-Update

Example 5 demonstrates the use of file update logic. This program updates data on lots in a housing tract. Item IDs in the file LOT are TRACT.NAME*LOT.NUMBER.

```
100   * INITIALIZATION
      PROMPT '='
      CLEAR
      DIM DESC(30), TYPE(30)
      OPEN 'DICT', 'LOT' ELSE
         PRINT "CAN'T OPEN DICT LOT"
         STOP
      END
*
200   * GET DESCRIPTIONS, CONVERSIONS
      FOR I = 1 TO 30
         READ DICT ITEM FROM I ELSE
            PRINT "DICTIONARY ITEM '":I:"' NOT FOUND"
            GOTO 250
         END
         D = EXTRACT(DICT.ITEM,3,0,0) ; *S/NAME--DESCRIPTION
         IF D # '' THEN DESC(I) = D:STR('.',15-LEN(D))
         IF C[1,2] = 'MD' THEN
            TYPE(I) = 'NUM'
            GOTO 250
         END
         IF C[1,1] = '0' THEN TYPE(I) = 'DATE'
250   *
      NEXT I
*
      OPEN '','LOT' ELSE
         PRINT "CAN'T OPEN LOT FILE."
         STOP
      END
*
300   * GET THE TRACT NAME
      PRINT
      PRINT "TRACT NAME.....":
      INPUT TRACT
      IF TRACT = 'STOP' OR TRACT = 'END' THEN STOP
      IF TRACT = '' THEN GOTO 300
      READ INFO FROM TRACT ELSE
         PRINT "TRACT '":TRACT:"'LOT ON FILE"
         GOTO 300
      END
*
```

```
400     * GET A VALID LOT NUMBER
        PRINT
        PRINT "LOT NUMBER.....":
        INPUT NUMBER
        IF NUMBER = '' THEN GOTO 400
        IF NUMBER = 'END' OR NUMBER = 'STOP' THEN GOTO 300
        IF NUM(NUMBER) = 0 THEN
           PRINT "MUST BE A NUMBER"
           GOTO 400
        END
        NUMBER = TRACT:'*':NUMBER
        READ ITEM FROM NUMBER ELSE
           ITEM = ''
           PRINT "NEW LOT"
        END
*
450     *
        NOT.SOLD = 0
        FOR I = 1 TO 30
           GOSUB 1000  ;  * UPDATES THE I'TH ATTRIBUTE
           IF I = 10 THEN
              IF EXTRACT(ITEM,10,0,0) = '' THEN
                 NOT.SOLD = 1
                 I = 19
              END
           END
*
           IF I = 21 THEN
              IF NOT.SOLD THEN GOTO 500
           END
        NEXT I
*
500     * VERIFY DATA & STORE
        PRINT
        PRINT"       OK     ":
        INPUT OK
        IF OK = '' THEN
           WRITE ITEM ON NUMBER
           GOTO 400
        END
        IF OK = 'L' THEN
           PRINT
           FOR L = 1 TO 30
              ATT = EXTRACT(ITEM,I,0,0)
              IF ATT = ''THEN GOTO 550
              PRINT DESC(L):
              IF TYPE(L) = 'DATE' AND NUM(DATE) THEN
                 ATT =  OCONV(ATT,'DO')
              END
              IF TYPE(L) = 'NUM' AND NUM(ATT) THEN
                 ATT = 0.01 * ATT
              END
```

```
            PRINT ATT 'R###############'
550         NEXT L
            GOTO 500
        END
        GOTO 400
*
1000    * UPDATE'S THE I'TH ATTRIBUTE OF "ITEM"
        IF DESC(I) = '' THEN RETURN ;*NOT NEEDED OR NOT FOUND
        PRINT DESC(I):
        CURRENT = EXTRACT(ITEM,I,0,0)
*
        IF TYPE(I) = 'NUM' THEN
1100    * NEED A NUMBER (AMOUNT)
            PRINT CURRENT*.01  'R###############':
            INPUT RESPONSE
            IF RESPONSE = '' THEN RETURN     ;* JUST LOOKING
            IF RESPONSE # '' THEN
                ITEM = REPLACE(ITEM,I,0,0,'')
                RETURN                       ;* DELETE THIS ATT.
            END
            IF NUM(RESPONSE) = 0 THEN
                PRINT "MUST BE A NUMBER"
                GOTO 1100
            END
*
            ITEM = REPLACE(ITEM,I,0,0,RESPONSE*100)
            RETURN
        END
*
        IF TYPE(I) = 'DATE' THEN
1200    * NEED A DATE
            PRINT OCONV(CURRENT,'DO') 'R###############':
            INPUT RESPONSE
            IF RESPONSE = '' THEN RETURN     ;* JUST LOOKING
            IF RESPONSE = 'T' THEN
                DATE = DATE()
                GOTO 1250
            END
            IF RESPONSE = '' THEN
                ITEM = REPLACE(ITEM,I,0,0,'');* DELETE THIS ATT.
                RETURN
            END
            DATE = ICONV(RESPONSE,'D')
            IF DATE = '' THEN
                PRINT "USE DATE FORMAT 'MONTH/DAY/YEAR'"
                GOTO 1200
            END
1250    *
            ITEM = REPLACE(ITEM,I,0,0,DATE)
            RETURN
        END
```

```
1300  * NO NECESSARY FORMATS
      PRINT CURRENT 'R###############':
      INPUT RESPONSE
      IF RESPONSE = '' THEN RETURN
      IF RESPONSE # '' THEN RESPONSE = ''
      ITEM = REPLACE(ITEM,I,0,0,RESPONSE)
      RETURN
END
```

APPENDIX B

Error Messages

Appendix B lists some of the more common Pick BASIC error messages that are found in the ERRMSG file of most Pick systems. Please note that the list is representative rather than complete, since each manufacturer has made additions and changes to its own ERRMSG file to suit its particular needs. See the documentation supplied with your system for more complete information about system-specific error messages.

Compiler Messages

Error messages generated by the COMPILE or BASIC verbs are printed along with the error number and the cause of the error when not self-explanatory.

When a compile-time error occurs, the error number is printed followed by the line number in which it was found and the associated error message. For example:

```
[B115] LINE 2 LABEL BELL IS USED BEFORE THE EQUATE STMT
```

Error number B115 was detected on line 2 of the program. The error message is taken from the system ERRMSG file.

See *A Guide to the Pick System* for more information about the ERRMSG file.

Table B-1. Pick BASIC Compiler Error Messages

Error #	Error Message	Cause
B100	COMPILATION ABORTED; NO OBJECT CODE PRODUCED.	Compilation was not completed. This message is printed after all other error messages, to warn the user that the object code was not updated.
B101	MISSING "END", "NEXT", "WHILE", "UNTIL", "REPEAT", OR "ELSE"; COMPILATION ABORTED, NO OBJECT CODE PRODUCED.	
B102	BAD STATEMENT	The statement was not recognized as a valid Pick BASIC statement.
B103	LABEL '*label*' IS MISSING	Label indicated by GOTO or GOSUB was not found.
B104	LABEL '*label*' IS DOUBLY DEFINED	More than one statement was found beginning with the same label.
B105	'*array*' HAS NOT BEEN DIMENSIONED	Variable *array* was referred to with dimensioned array syntax, but was not dimensioned in the program.
B106	'*array*' HAS BEEN DIMENSIONED AND USED WITHOUT SUBSCRIPTS	Dimensioned array *array* was used without subscripts.
B107	"ELSE" CLAUSE MISSING	
B108	"NEXT" STATEMENT MISSING	A FOR loop was begun without a corresponding NEXT statement.
B109	VARIABLE MISSING IN "NEXT" STATEMENT	Iteration variable is missing in NEXT statement.
B110	'END' STATEMENT MISSING	
B111	"UNTIL" OR "WHILE" MISSING IN "LOOP" STATEMENT	

Error #	Error Message	Cause
B112	"REPEAT" MISSING IN "LOOP" STATEMENT	
B113	TERMINATOR MISSING	Garbage following a legal statement, or quote missing.
B114	MAXIMUM NUMBER OF VARIABLES EXCEEDED	Using the default descriptor size of 10, the maximum number of variables (including array elements) is 3223.
B115	LABEL '*var*' IS USED BEFORE THE EQUATE STMT	The equate-variable *var* is referenced before it has been defined.
B116	LABEL '*var*' IS USED BEFORE THE COMMON STATEMENT	COMMON variable *var* has been referenced before it has been defined.
B117	LABEL '*array*' IS MISSING A SUBSCRIPT LIST	Dimensioned array is referenced without a subscript list.
B118	LABEL '*var*' IS THE OBJECT OF AN EQUATE STMT AND IS MISSING.	
B119	WARNING - PRECISION VALUE OUT OF RANGE - IGNORED!	A PRECISION statement specified a value other than 0 through at least 6.
B120	WARNING - MULTIPLE PRECISION STATEMENTS - IGNORED!	When two PRECISION statements are included, only the first one is read.
B121	LABEL '*c*' IS A CONSTANT AND CAN NOT BE WRITTEN INTO	The program attempted to assign an EQUATEd variable to another value.
B122	LABEL '*c*' IS IMPROPER TYPE	
B124	LABEL '*array*' HAS LITERAL SUBSCRIPTS OUT OF RANGE.	An attempt has been made to access an element of a dimensioned array beyond its dimensions.
B154	FOR statement with no NEXT statement.	

Error #	Error Message	Cause
B199	FORMAT ERROR IN SOURCE FILE DEFINITION	The file that the program resides in does not have a DC-type pointer.
B209	FILE IS UPDATE PROTECTED.	
B210	FILE IS ACCESS PROTECTED.	
B220	'CSYM' IS NOT A FILE NAME OR NEEDS A DATA LEVEL.	

Run-Time Messages

There are two types of message that can be encountered during run-time: warning messages and fatal messages. Warning messages indicate that illegal conditions have been smoothed over (by making an appropriate assumption), and do not result in program termination. Fatal error messages result in program termination and deposit the user in the Pick BASIC debugger.

When an error occurs, the error number is printed followed by the error message. For example:

```
[B27] LINE 87 RETURN EXECUTED WITH NO GOSUB
```

Error number B27 was detected on line 87 of the program. The error message is taken from the system ERRMSG file.

See *A Guide to the Pick System* for more information about the ERRMSG file.

Table B-2. Run-Time Error Messages (Warning).

Error #	Error Message	Cause
B9	WRITE, DELETE or CLEARFILE operation attempted on read only file.	An unassigned variable was referenced. A value of 0 is assumed.
B10	VARIABLE HAS NOT BEEN ASSIGNED A VALUE; ZERO USED!	An unassigned variable was referenced. A value of 0 is assumed.

Error #	Error Message	Cause
B13	NULL CONVERSION CODE IS ILLEGAL; NO CONVERSION DONE!	A conversion was attempted with ICONV or OCONV with no conversion code.
B16	NON-NUMERIC DATA WHEN NUMERIC REQUIRED; ZERO USED!	A string variable was encountered when a number was required. A value of 0 is assumed.
B19	ILLEGAL PATTERN	Illegal pattern used with MATCH of MATCHES operator.
B20	COL1 OR COL2 USED PRIOR TO EXECUTING A FIELD STMT; ZERO USED!	COL1 or COL2 function used before FIELD function used. A value of 0 is assumed.
B21	MATREAD: NUMBER OF ATTRIBUTES EXCEEDS VECTOR SIZE	The number of attributes in the item exceeds the dimensioned size of the array; the remaining attributes are placed as a dynamic array in the last element of the array.
B24	DIVIDE BY ZERO ILLEGAL; ZERO USED!	Division by zero attempted. The operation will return 0 (not ∞).
B26	'UNLOCK *c*' ATTEMPTED BEFORE LOCK!	An attempt was made to unlock a lock which the program did not lock.
B209	FILE IS UPDATE PROTECTED.	
B210	FILE IS ACCESS PROTECTED.	

Table B-3. Run-Time Error Messages (Fatal).

Error #	Error Message	Cause
B1	RUN-TIME ABORT AT LINE *n*	The program is aborted. The debugger is not invoked.
B12	FILE HAS NOT BEEN OPENED	File indicated in I/O statement has not been previously opened via an OPEN statement.
B14	BAD STACK DESCRIPTOR	Either the length of the input-lists or output-lists in the CALL and SUBROUTINE statements are different; an attempt is made to execute an external subroutine as a main program; a file variable is used as an operand; or a variable has been assigned a value with a precision greater than program allows.
B15	ILLEGAL OPCODE: "*C*"	Object code for item indicated by RUN verb contains garbage or external subroutine without SUBROUTINE statement.
B17	ARRAY SUBSCRIPT OUT-OF-RANGE	Array subscript is less than or equal to zero or exceeds the dimensions indicated by a DIM statement.
B18	ATTRIBUTE NUMBER LESS THAN -1 IS ILLEGAL	An attribute number less than -1 was specified in a READV or WRITEV statement.
B25	PROGRAM '*prog*' HAS NOT BEEN CATALOGED	The specified external subroutine must be cataloged before referenced in a CALL statement.
B27	RETURN EXECUTED WITH NO GOSUB	
B28	NOT ENOUGH WORK SPACE	Not enough work space assigned.

Error #	Error Message	Cause
B29	CALLING PROGRAM MUST BE CATALOGED	An external call cannot be made unless the calling program is also catalogued.
B30	ARRAY SIZE MISMATCH	Array sizes in MAT Copy statement, or in CALL and SUBROUTINE statements, do not match.
B31	STACK OVERFLOW	The program has attempted to call too many nested subroutines.
B32	PAGE HEADING EXCEEDS MAXIMUM OF 1400 CHARACTERS	Page heading is too long.
B33	PRECISION DECLARED IN SUBPROGRAM '*c*' IS DIFFERENT FROM THAT DECLARED IN THE MAINLINE PROGRAM	Precision must be same between calling program subroutines.
B34	FILE VARIABLE USED WHERE STRING EXPRESSION EXPECTED	
B41	LOCK NUMBER IS GREATER THAN 47	Maximum number of locks available is 47.

Debugger Messages

The following messages are used by the Pick BASIC debugger.

Table B-4. Debugger Messages.

Message	Description
*E *x*	Single step breakpoint at line number *x* .
*B*n x*	Table breakpoint at line number *x*; *n* equals number of breakpoint.
**v* = *x*	Value of variable at breakpoint
*N*var*	Variable not found in statement.
CMND?	Command not recognized.
NSTAT#	Statement number out of range of program.

Message	Description
SYM NOT FND	Symbol not found in table.
UNASSIGNED VAR	Variable not assigned a value.
STACK EMPTY	The subroutine stack is empty.
STACK ILL	Illegal subroutine stack format.
TBL FULL	Trace or break table full.
ILLGL SYM	Illegal symbol.
NOT IN TBL	Not in trace break table.
NO SYM TAB	Symbol table not in file.

APPENDIX C

List of ASCII Codes

Decimal	Hex	Character	Symbol	Remark
0	0		NUL	Null prompt character
1	1	^A	SOH	Cursor home
2	2	^B	STX	
3	3	^C	ETX	
4	4	^D	EOT	End of Transmission
5	5	^E	ENQ	
6	6	^F	ACK	Cursor forward
7	7	^G	BEL	Bell
8	8	^H	BS	Backspace
9	9	^I	HT	Hard Tab
10	A	^J	LF	Cursor down
11	B	^K	VT	Vertical address
12	C	^L	FF	Screen erase
13	D	^M	CR	Carriage return
14	E	^N	SO	
15	F	^O	SI	
16	10	^P	DLE	Horizontal address
17	11	^Q	DC1	
18	12	^R	DC2	Redisplay input field
19	13	^S	DC3	
20	14	^T	DC4	
21	15	^U	NAK	Cursor back

Decimal	Hex	Character	Symbol	Remark
22	16	^V	SYN	
23	17	^W	ETB	Erase word
24	18	^X	CAN	Erase character
25	19	^Y	EM	
26	1A	^Z	SUB	Cursor up
27	1B		ESC	
28	1C		FS	
29	1D		GS	
30	1E		RS	
31	1F		US	
32	20		SPACE	
33	21		!	Exclamation Point
34	22		"	Quotation Mark
35	23		#	Sharp Sign
36	24		$	Dollar Sign
37	25		%	Percent Sign
38	26		&	Ampersand
39	27		'	Apostrophe
40	28		(	Left Parenthesis
41	29		)	Right Parenthesis
42	2A		*	Asterisk
43	2B		+	Plus Sign
44	2C		'	Comma
45	2D		-	Hyphen
46	2E		.	Period
47	2F		/	Slash (Virgule)
48	30		0	
49	31		1	
50	32		2	
51	33		3	
52	34		4	
53	35		5	
54	36		6	
55	37		7	
56	38		8	

Decimal	Hex	Character	Symbol	Remark
57	39		9	
58	3A		:	Colon
59	3B		;	Semicolon
60	3C		<	Left Angle Bracket
61	3D		=	Equal Sign
62	3E		>	Right Angle Bracket
63	3F		?	Question Mark
64	40		@	"At" Sign
65	41		A	
66	42		B	
67	43		C	
68	44		D	
69	45		E	
70	46		F	
71	47		G	
72	48		H	
73	49		I	
74	4A		J	
75	4B		K	
76	4C		L	
77	4D		M	
78	4E		N	
79	4F		O	
80	50		P	
81	51		Q	
82	52		R	
83	53		S	
84	54		T	
85	55		U	
86	56		V	
87	57		W	
88	58		X	
89	59		Y	
90	5A		Z	
91	5B		[	Left Square Bracket

Decimal	Hex	Character	Symbol	Remark
92	5C		\	Backslash
93	5D		]	Right Square Bracket
94	5E		^	Caret
95	5F		_	Underscore
96	60		`	Back Quote
123	7B		{	Left Curly Brace
124	7C			Vertical Bar
125	7D		}	Right Curly Brace
126	7E		~	Tilde
127	7F	DEL		
251	FB	SB	^[	Start buffer (ESC, CTRL-[)
252	FC	SVM	^\	Subvalue Mark (CTRL-\)
253	FD	VM	^]	Value Mark (CTRL-])
254	FE	AM	^^	Attribute Mark (CTRL-^)
255	FF	SM	^_	Segment Mark (CTRL-_)

APPENDIX D

Pick BASIC Statements, Functions, and Operators

Appendix D lists all Pick BASIC statements, functions, and operators described in this guide, along with some others that are available on selected Pick and Pick-related systems. Statements, functions, and operators that are included in the SMA standards are indicated in the left-most column.

SMA	Statements, Functions, and Operators	Pick	ADDS	GA	Ultimate	Prime	uniVerse
•	^	•			•	•	•
•	! [comment]	•	•	•	•	•	•
•	! [OR]	•	•		•	•	•
•	# [NE]	•	•	•	•	•	•
	#< [GE]					•	•
	#> [LE]					•	•
	$* [comment]		•		•		
	$CHAIN		•	•	•		•
	$INCLUDE		•	•	•		•
	$INSERT		•	•		•	•
•	& [AND]	•	•		•	•	•
•	* [comment]	•	•	•	•	•	•
•	* [multiplier]	•	•	•	•	•	•
	** [exponentiation]					•	•
•	+	•	•	•	•	•	•
•	–	•	•	•	•	•	•

SMA	Statements, Functions, and Operators	Pick	ADDS	GA	Ultimate	Prime	uniVerse
•	/	•	•	•	•	•	•
•	:	•	•	•	•	•	•
•	<	•	•	•	•	•	•
•	< > [arrays]	•	•	•	•	•	•
	< > [NE]	•	•	•	•	•	•
•	<=	•	•	•	•	•	•
•	=	•	•	•	•	•	•
	=<		•	•		•	•
	=>		•	•		•	•
•	>	•	•	•	•	•	•
	> < [NE]	•	•		•	•	•
•	>=	•	•	•	•	•	•
•	@( )	•	•	•	•	•	•
	[ ]=		•	•		•	•
•	ABORT	•	•	•	•	•	•
•	ABS( )	•	•	•	•	•	•
	ALPHA( )	•	•	•	•		•
•	AND	•	•	•	•	•	•
•	ASCII( )	•	•	•	•	•	•
•	BEGIN CASE	•	•	•	•	•	•
	BREAK [KEY]		•	•		•	•
•	BREAK	•	•	•	•	•	•
•	CALL	•	•	•	•	•	•
•	CASE	•	•	•	•	•	•
•	CAT	•	•		•	•	•
•	CHAIN	•	•	•	•	•	•
•	CHAR( )	•	•	•	•	•	•
•	CLEAR	•	•	•	•	•	•
	CLEAR COMMON					•	•
•	CLEARFILE	•	•	•	•	•	•
•	COL1( )	•	•	•	•	•	•
•	COL2( )	•	•	•	•	•	•
•	COM[MON]	•	•	•	•	•	•
	CONVERT		•			•	•
•	COS( )	•	•	•	•	•	•

SMA	Statements, Functions, and Operators	Pick	ADDS	GA	Ultimate	Prime	uniVerse
•	COUNT( )	•	•	•	•	•	•
•	CRT	•	•	•		•	•
•	DATA	•	•	•	•	•	•
•	DATE( )	•	•	•	•	•	•
•	DCOUNT( )	•	•	•	•		•
	DEBUG	•	•			•	•
	DEL		•	•	•	•	•
•	DELETE	•	•	•	•	•	•
•	DELETE( )	•	•	•	•	•	•
•	DIM[ENSION]	•	•	•	•	•	•
	DISPLAY		•		•		•
•	DO	•	•	•	•	•	•
•	EBCDIC( )	•	•	•	•	•	•
•	ECHO	•	•	•	•	•	•
•	ELSE	•	•	•	•	•	•
•	END	•	•	•	•	•	•
•	END CASE	•	•	•	•	•	•
•	ENTER	•	•	•	•		•
•	EQ	•	•	•	•	•	•
•	EQU[ATE]	•	•	•	•	•	•
•	EXECUTE	•	•	•	•	•	•
•	EXP( )	•	•	•	•	•	•
•	EXTRACT( )	•	•	•	•	•	•
•	FIELD( )	•	•	•	•	•	•
•	FOOTING	•	•	•	•	•	•
•	FOR...NEXT	•	•	•	•	•	•
•	GE	•	•	•	•	•	•
•	GO [TO]	•	•	•	•	•	•
•	GOSUB	•	•	•	•	•	•
•	GOTO	•	•	•	•	•	•
•	GT	•	•	•	•	•	•
•	HEADING	•	•	•	•	•	•
•	ICONV( )	•	•	•	•	•	•
•	IF	•	•	•	•	•	•
•	INCLUDE	•		•			

SMA	Statements, Functions, and Operators	Pick	ADDS	GA	Ultimate	Prime	uniVerse
•	INDEX( )	•	•	•	•	•	•
•	INPUT	•	•	•	•	•	•
•	INPUT @	•	•	•	•	•	•
	INPUTCLEAR		•	•	•		•
	INPUTERR	•	•	•			•
	INPUTIF		•				•
	INPUTNULL	•	•	•			•
	INPUTTRAP	•	•	•			•
	INS		•	•	•	•	•
•	INSERT( )	•	•	•	•	•	•
•	INT( )	•	•	•	•	•	•
•	LE	•	•	•	•	•	•
•	LEN( )	•	•	•	•	•	•
	LET	•	•	•	•	•	•
•	LN( )	•	•	•	•	•	•
•	LOCATE( )	•	•	•	•	•	•
•	LOCK	•	•	•	•	•	•
•	LOOP	•	•	•	•	•	•
•	LT	•	•	•	•	•	•
•	MAT	•	•	•	•	•	•
	MATBUILD		•	•			•
•	MATCH[ES]	•	•	•	•	•	•
	MATPARSE		•	•		•	•
•	MATREAD	•	•	•	•	•	•
•	MATREADU	•	•	•	•	•	•
•	MATWRITE	•	•	•	•	•	•
•	MATWRITEU	•	•	•	•	•	•
	MOD( )	•	•	•	•	•	•
•	NE	•	•	•	•	•	•
•	NEXT	•	•	•	•	•	•
•	NOT( )	•	•	•	•	•	•
•	NULL	•	•	•	•	•	•
•	NUM( )	•	•	•	•	•	•
•	OCONV( )	•	•	•	•	•	•
•	ON	•	•	•	•	•	•

SMA	Statements, Functions, and Operators	Pick	ADDS	GA	Ultimate	Prime	uniVerse
•	OPEN	•	•	•	•	•	•
•	OR	•	•	•	•	•	•
•	PAGE	•	•	•	•	•	•
•	PRECISION	•	•	•	•	•	•
•	PRINT [ON]	•	•	•	•	•	•
•	PRINTER	•	•	•	•	•	•
	PROCREAD	•	•	•	•		•
	PROCWRITE	•	•	•	•		•
•	PROMPT	•	•	•	•	•	•
•	PWR( )	•	•	•	•	•	•
•	READ	•	•	•	•	•	•
•	READNEXT	•	•	•	•	•	•
•	READT	•	•	•	•	•	•
•	READU	•	•	•	•	•	•
•	READV	•	•	•	•	•	•
•	READVU	•	•	•	•	•	•
•	RELEASE	•	•	•	•	•	•
•	REM [comment]	•	•	•	•	•	•
•	REM( )	•	•	•	•	•	•
•	REPEAT	•	•	•	•		•
•	REPLACE( )	•	•	•	•	•	•
•	RETURN [TO]	•	•	•	•	•	•
•	REWIND	•	•	•	•	•	•
•	RND( )	•	•	•	•	•	•
	RQM	•	•	•	•	•	
•	SELECT	•	•	•	•	•	•
•	SEQ( )	•	•	•	•	•	•
•	SIN( )	•	•	•	•	•	•
•	SLEEP	•	•	•			•
	SOUNDEX( )		•	•			•
•	SPACE( )	•	•	•	•	•	•
•	SQRT( )	•	•	•	•	•	•
•	STOP	•	•	•	•	•	•
•	STR( )	•	•	•	•	•	•
•	SUBROUTINE	•	•	•	•	•	•

SMA	Statements, Functions, and Operators	Pick	ADDS	GA	Ultimate	Prime	uniVerse
	SUM( )		•	•		•	•
•	SYSTEM( )	•	•	•	•		•
•	TAN( )	•	•	•	•	•	•
•	THEN | ELSE	•	•	•	•	•	•
•	TIME( )	•	•	•	•	•	•
•	TIMEDATE( )	•	•	•	•	•	•
•	TRIM( )	•	•	•	•	•	•
	TRIMB( )		•	•		•	•
	TRIMF( )		•	•		•	•
•	UNLOCK	•	•	•	•	•	•
•	WEOF	•	•	•	•	•	•
•	WHILE | UNTIL	•	•	•	•	•	•
•	WRITE	•	•	•	•	•	•
•	WRITET	•	•	•	•	•	•
•	WRITEU	•	•	•	•	•	•
•	WRITEV	•	•	•	•	•	•
•	WRITEVU	•	•	•	•	•	•

I N D E X

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

About the Author

Linda Mui is a technical writer at O'Reilly & Associates. She has worked in UNIX system administration and text processing (troff), and co-authored the Nutshell Handbook on UNIX terminal databases (termcap and terminfo). She has also worked as a consultant at Digital Equipment Corporation and the Open Software Foundation.

About the Editor

W. Clifton Oliver is the technical editor of the Pick Series. He is a consultant who has worked with the Pick system since 1974, using it to implement a wide range of applications. His clients have included Fortune 500 corporations, individual end users, and Pick system manufacturers. Mr. Oliver is also an instructor, speaker, and columnist.

Overseas Distributors

Effective January 1, 1990, customers outside the U.S. and Canada will be able to order Nutshell Handbooks and the X Window System Series through distributors near them. These overseas locations offer international customers faster order processing, more local bookstores and local distributors, an increased representation at trade shows worldwide, as well as the high level, quality service our customers have always received.

AUSTRALIA & NEW ZEALAND
(orders and inquiries)
Addison-Wesley Publishers, Pty. Ltd.
6 Byfield Street
North Ryde, N.S.W. 2113
AUSTRALIA
Telephone: 61-2-888-2733
FAX: 61-2-888-9404

UNITED KINGDOM & AFRICA
(orders and inquiries)
Addison-Wesley Publishers, Ltd.
Finchampstead Road
Wokingham, Berkshire RG11 2NZ
ENGLAND
Telephone: 44-734-794-000
FAX: 44-734-794-035

EUROPE & THE MIDDLE EAST
(orders and inquiries)
Addison-Wesley Publishers, B.V.
De Lairessestraat 90
1071 PJ Amsterdam
THE NETHERLANDS
Telephone: 31-20-764-044
FAX: 31-20-664-5334

ASIA inquiries (excluding Japan)
Addison-Wesley Singapore Pte. Ltd.
15 Beach Road #05-09/10
Beach Centre
Singapore 0718
SINGAPORE
Telephone: 65-339-7503
FAX: 65-339-9709

ASIA orders (excluding Japan)
Addison-Wesley Publishing Co., Inc.
International Order Department
Route 128
Reading, Massachusetts 01867 U.S.A.
Telephone: 1-617-944-3700
FAX: 1-617-942-2829

JAPAN
(orders and inquiries)
Toppan Company, Ltd.
Ochanomizu Square B, 1-6
Kanda Surugadai
Chiyoda-ku, Tokyo 101
JAPAN
Telephone: 81-3-295-3461
FAX: 81-3-293-5963

Pick BASIC

A REFERENCE GUIDE

Pick BASIC: A Reference Guide is comprehensive documentation for applications programmers. The large reference section covers all Pick BASIC statements and functions. This book's material is organized for easy access, as a reference manual should be. Explanations are clearly written and can be understood in a single reading. Each statement or function is illustrated by a nontrivial sample program. The debugger, which is thinly documented elsewhere, is fully covered here.

Contents include:

- Creating Pick BASIC programs
- Format, data, and expressions
- Functional overview of statements and functions
- Using the Pick BASIC debugger
- Statement and function reference
- Appendixes, which provide program examples, error messages, ASCII codes, and a table of statements, functions, and operators supported by SMA and selected Pick implementations
- Index

The Pick Series is for users who want more out of Pick documentation—understanding a passage at first reading; speedily looking up an option; finding complete coverage of a topic; having a guide you can give to a first-time user. The Pick Series offers a complete Pick documentation set for all users, based on a mature implementation of the Pick operating system (R83), with notes on SMA standards and specific differences among major Pick implementations.

ISBN # 0-937175-42-0

O'Reilly & Associates, Inc.