SIEMENS

PBASIC® Interpreter

Lehrgang (Tutorial)

Bestell-Nr. C79000-M8776-C31-1

The _Personal BASIC Language Reference Manual_ was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

```
**********************************
*  First Edition:  April 1983  *
**********************************
```

Foreword

This book is a tutorial for the Digital Research interpretive BASIC system, Personal BASIC™ .

The book was written for anyone with no previous programming experience. Its purpose is to teach you how to write computer programs in the Personal BASIC language. The book should be read from the beginning, and you should have access to a computer that can run Personal BASIC.

You should try the many examples throughout the book and any examples that come to mind. You must enter and run the example programs and observe the results to successfully learn BASIC. Curling up on a couch with this book and studying will not do the job.

Section 1 explains why this book was written, and tells how programming and BASIC fit into the computer industry. Section 2 describes using the computer as a calculator and how to save and load programs, as well as other basic concepts. The arithmetic used in Personal BASIC is explained in Section 3.

Because everyone makes at least a few mistakes, Section 4 describes the excellent editing facilities in Personal BASIC. Sections 5, 6, and 7 show many of the Personal BASIC program statements and how they are used in programs. Section 8 covers Personal BASIC's built-in and self-defined functions.

Section 9 explains how to handle large groups of numbers with subscripted variables and arrays. The concepts of files, both sequential and random, are explained in Section 10. Section 11 describes the complete debugging features in Personal BASIC and how to use them when looking for program errors. The techniques of program testing are discussed in Section 11.

Appendix A contains a glossary describing programming and computer words as they relate to Personal BASIC. Appendix B is an annotated bibliography of additional information on computers and computer programming. Appendix C contains the answers to the programming exercises given at the end of some sections. Appendix D explains the error messages you see when you make an error in Personal BASIC.

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Appendixes

Tables and Figures

Section 1
Introduction

1.1 Why a Personal BASIC Language Tutorial?

Not everyone who wants to use a computer is a programmer. Digital Research created Personal BASIC, an easy to use version of the BASIC programming language, and this tutorial book to help those without prior programming knowledge get started in programming.

Another book, The Personal BASIC Language Reference Manual, contains more detailed descriptions of the Personal BASIC statements and commands. Keep the reference manual with you as you continue through this book.

If you have BASIC programming experience, you might need only the reference manual to show you the unique features of Personal BASIC. However, if your BASIC programming experience is not recent, this tutorial is a good manual for you.

Learning a computer language is much like learning a foreign language, although much easier. BASIC has a simple vocabulary, a grammatical structure, and rules of use, just like any other language.

The secret of learning any language, computer or foreign, is to practice speaking it. You will speak to BASIC through your computer or terminal keyboard. Practice as you learn by doing the examples and you will find that remembering is much easier. You might make mistakes as you progress, but they are easy to correct and hopefully, you will not make the same mistake twice.

A Few Things to Remember

- The examples and programs use colored type to indicate what YOU type on your computer and what your computer returns.
- The symbol <cr> means that you should press the carriage return key. It might be called RETURN or Enter on your keyboard. You must press <cr> to signal your computer that you have finished your input and now it is the computer's turn to do some work.
- The control or CTRL key enters codes not available on the keyboard and not visible on your screen. When you see CTRL-C, hold the CTRL key down while you press the C key. Think of the CTRL key as a super shift key with a different function than the SHIFT key.

- Sometimes when typists use terminals for the first time, they use some letters for numbers. DO NOT use a lower-case L for 1, or the letter O for zero.

- Paragraphs beginning with the word **Note:** are very important. Be sure to study these paragraphs.

1.2 What is Programming?

Programming is the writing of descriptions and instructions that tell the computer what operations to perform in solving a problem. The programmer can use many languages to accomplish this purpose. BASIC is just one of the language tools used by programmers to solve problems.

This tutorial teaches you how to use much of the BASIC computer language. As you learn BASIC, remember that learning a language is only part of learning how to program. Knowledge of English does not mean that you can write a best-selling novel. There are hundreds of books available on programming techniques. You must study and practice writing programs to become a good programmer. Analyzing a problem and then planning how a programming language can solve the problem is a major part of the programmer's duties.

In many situations, defining the problem and planning how to solve the problem is more challenging than writing the program instructions.

1.3 What is BASIC?

Before we begin to learn the nuts and bolts of BASIC programming, let's take a few moments to see where BASIC came from and how it compares with other computer languages.

1.3.1 Why BASIC?

Back in the old days of computer programming (the early 60's), there was no easy method of communicating with a computer. Several languages existed, but they were difficult to learn and use. Two professors at Dartmouth College saw the need for a language that could be easily learned and operated. They designed the format of BASIC and with the help of their students created BASIC, primarily to use in teaching programming.

The use of BASIC has increased over the years, and today it is the most widely used computer language. Nearly every computer maker offers a version of BASIC, and it is part of the hardware of many microcomputers.

While BASIC is a universal computer language, all BASICs are not exactly alike. Once you are familiar with a version of BASIC, such as Personal BASIC, you can convert other BASIC programs to run on your system without too much difficulty. Personal BASIC runs under the CP/M® operating system, so many BASIC programs on other CP/M computers will run on your system with few or no changes required.

Because BASIC is used by so many people on so many computers, thousands of programs are available to you. The world of BASIC is indeed an open door into the world of computing.

1.3.2 BASIC Compared to Other Languages

BASIC is one of many computer languages available today. BASIC's main advantages are that it is easy to learn and easy to use. Another big plus for BASIC is that it is available for almost every computer system in use.

You have probably heard of some of the more popular computer programming languages such as FORTRAN, Pascal, COBOL, and PL/I. Here is a brief description of these languages.

FORTRAN

FORTRAN was released by IBM in 1957, making it the first high-level language available. It closely follows mathematical and algebraic notation. FORTRAN is most powerful in solving number crunching problems and is still widely used for scientific calculations.

COBOL

COBOL was developed in 1960 by the Department of Defense and several civilian computer firms. COBOL is the major language used for solving business problems on medium to large computers. Some versions of COBOL will operate on microcomputers. COBOL's strength is in file manipulation and handling large volumes of data. It uses English-like statements and is easy to read.

PL/I

PL/I was introduced by IBM in 1965 as an attempt to combine the best features of FORTRAN, COBOL, and ALGOL. PL/I is very complex and handles scientific processing and business file manipulations very well. Versions of PL/I are used by many software firms for program development.

Pascal

Pascal was made available in 1970. Pascal is a structured language, meaning that programs flow logically from beginning to end without abrupt shifts possible in languages such as BASIC or

FORTRAN. Many educators recommend Pascal as a first programming language.

1.3.3 Features of Personal BASIC

Personal BASIC is one of many in the growing assortment of programming languages offered by Digital Research. Personal BASIC varies from other BASIC systems available from Digital Research. It is an interpretative BASIC. This means that Personal BASIC responds immediately to your input. Each statement is analyzed for correct format at the time it is entered, and an error message explains any error in the line just entered. When the program is completed or partially completed, only a RUN command is necessary to run the program.

Of course, this simplifies program testing. You can make changes very quickly and run your program to be sure the changes work. You can keep changing your program in working storage until it does what it is supposed to. The program can then be put into permanent storage and retrieved whenever you want to use it.

The family of Digital Research BASIC systems is designed for maximum compatibility in the changing 8-bit and 16-bit environment. Staying within the CP/M BASIC family ensures easier program conversion whenever hardware is upgraded or replaced. Changes could be required to convert a program from one Digital Research BASIC to another.

End of Section 1

Section 2
BASIC Basics

Our first venture into programming shows how BASIC can do things for you even without a program. This section reviews the formats of statements and commands. You will learn the concepts of working versus permanent storage, and how BASIC saves and loads programs.

2.1 The BASIC Calculator and Printer

BASIC can do arithmetic and printing operations for you just like a calculator. These examples introduce you briefly to the PRINT statement. PRINT is explained in greater detail in Section 5.

If you want a system disk with Personal BASIC and CP/M on the same disk, see Appendix C in the Personal BASIC Language Reference Manual for instructions on how to create this disk.

Start by bringing Personal BASIC into the memory of your computer. Use the following three steps:

1) Start or boot CP/M following the instructions for your computer.

2) When you see the CP/M prompt A>, type BASIC. If you are using separate disks for Personal BASIC and CP/M, and Personal BASIC is on disk B, return control to disk B, A>B:, before typing BASIC.

3) Personal BASIC should load and give the Ok prompt on your terminal. Personal BASIC responds with an Ok after each operation. This means that everything is OK, and Personal BASIC is ready for another request.

4) Return to CP/M from Personal BASIC by typing the word SYSTEM after the Ok prompt and <cr>.

Personal BASIC is now ready for your instructions. Now type

Ok **PRINT "MOONBEAM"**

and press <cr>. Remember, the computer does not know that you want something done until you press <cr>.

The computer prints the word:

```
MOONBEAM
```

Notice that you typed quotation marks around the word, MOONBEAM. The PRINT statement only prints the information enclosed in quotation marks.

Let's print some more words, or print anything that you would like to see on your terminal. Do not forget the quotation marks. Type the PRINT statements in the next example. The output from the PRINT statement appears on the next line.

```
Ok PRINT "ROSES ARE RED"
ROSES ARE RED
Ok PRINT "VIOLETS ARE BLUE"
VIOLETS ARE BLUE
```

If you receive an error message instead of the beautiful poetry, you could have misspelled PRINT or left out one or more quotation marks.

Let's see how PRINT handles numbers. You can use the CAPS LOCK Key for these and following examples, if there is one on your keyboard. CAPS LOCK on a computer keyboard prints letters in upper-case, and numbers as lower-case numbers. This makes it convenient to use CAPS LOCK when typing upper-case letters and numbers in the program examples to follow. You must use the SHIFT key to type other upper-case characters such as *, +, (,), and ^. Type these PRINT statements and observe the output.

```
Ok PRINT "230"
230
Ok PRINT 230
 230
```

This shows you that quotation marks are not needed to print numbers. The space in front of the number printed in the last example is reserved for the sign. A space means plus. If 230 was minus, it would be printed as -230.

Now we can try some arithmetic with PRINT. Type this:

```
Ok PRINT 5+7
 12
```

The answer, 12, prints on the next line. If nothing printed, did you press <cr>?

Personal BASIC can do the six arithmetic operations listed in Table 2-1.

Table 2-1. Arithmetic Operations

Operation	Symbol
Addition	Use the plus sign (+).
Subtraction	Use the minus sign (-).
Multiplication	Use an asterisk (*).
Division	Use a slash (/).
Exponentiation	Use a caret (^).
Combination	Any or all of these operations can be combined with each other.

Let's try some examples. Enter the PRINT operations as shown. The answer prints on the next line.

```
Ok PRINT 121+130
 251
Ok PRINT 77-23
 54
Ok PRINT 15*5
 75
Ok PRINT 42/7
 6
Ok PRINT 5^3
 125
Ok PRINT 7+2-1*9/3^2
 8
Ok PRINT 456-649
-193
```

2.2 Statement and Command Formats

BASIC statements are the instructions that form the BASIC program. BASIC commands are used outside the program. They tell BASIC how to manipulate the programs in and out of the storage areas and do other useful things like listing your program and renumbering your lines.

Now we will look at a simple BASIC program. The statements and commands are explained in more detail in other sections. Program CALAVG (CALculate AVeraGe) calculates and prints the average of any three numbers you type. The program is then saved to your permanent storage. Type in the program as shown. Press <cr> after each statement and each command.

When you type the RUN command and <cr>, Personal BASIC prints a question mark ?. This means that the program is asking you to type in the first of the three numbers you want to average. Type in the first number, followed by a <cr>; you will then see another question mark. Enter the second number and repeat for the third. When you see "THE AVERAGE IS 43", the program has finished running. Type SAVE and <cr> to save the program for later use.

```
Ok NEW CALAVG ◄──────────────────  NEW clears working storage
Ok 5 REM AVERAGE OF 3 NUMBERS      and names the program
Ok 10 INPUT A
Ok 20 INPUT B
Ok 30 INPUT C
Ok 40 AVG=(A+B+C)/3
Ok 50 PRINT "THE AVERAGE IS";AVG
Ok 60 END
Ok RUN ◄─────────────────────────  RUN tells Personal  BASIC
? 45  ────┐                        to run the program
? 18      │◄── User inputs
? 66  ────┘
THE AVERAGE IS 43
Ok SAVE ◄────────────────────────  SAVE sends program CALAVG
                                   to permanent storage
```

All Personal BASIC programs should have END as their last statement. END closes all files, as we will see later. The use of END is good programming practice.

The REM statement is used only to comment about the Personal BASIC program and has no effect on the operation of the program. REM statements help others understand your program. They also help you remember what is happening in your program when you look at it sometime in the future. An apostrophe ' can be used in place of REM. Line 5 in program CALAVG could have been written:

```
Ok 5 'AVERAGE OF 3 NUMBERS.
```

You can add a remark after your statement. For example,

```
Ok 40 AVG=(A+B+C)/3'Calculate the average
```

Everything after the apostrophe is ignored when the program runs.

2.2.1 Personal BASIC Statement Format

You probably noticed several things about the statements in the previous example. The following are the rules for BASIC statements:

- Statements must start with a line number.
- Statements must be spelled correctly.
- Statements must be separated from the rest of the line by at least one space on either side.
- More than one statement can be written on a line. The statements are separated by a colon, :.

Here are examples of correct and incorrect Personal BASIC statements:

Correct	Incorrect	
20 GOTO 100	GOTO 100	(No line number)
30 READ A	30 READA	(At least one space is required on each side of a statement)
40 PRINT C+D	40 PRNT C+D	(Statement misspelled)
50 A=4:B=32	50 A=4 B=32	(A colon must separate the statements)

The incorrect examples produce an error message immediately after you press <cr> at the end of the line. Correct errors by typing the line again (this erases the original line), or backspace to the error and retype. Section 4, "Editing Your Program," describes the line-editing features of Personal BASIC.

Line Numbers

Each BASIC statement must start with a line number from 0 through 65529. The program runs in numerical order by line number. You can type the statements in any order, but they run in line number order.

Separate your line numbers by some interval. For example, start with 10 and number your lines 10, 20, 30, etc. You will understand the reason for this after you have written a few programs. This practice lets you insert a line between two other lines. In example program CALAVG, you could write a statement using line 25 and it would be inserted between lines 20 and 30.

Note: use LIST after you add or delete lines. LIST arranges your program in numerical order by line number. It is good practice to use LIST to verify program changes and to inspect your program as you are typing it. See Section 2.3.2 for more information on LIST.

When your program is complete, Personal BASIC has a command called RENUM that renumbers your program. You can specify the starting line number and the interval. If no line number is specified, the first line number is 10; line numbers are incremented by 10. The following is an example of a Personal BASIC program before and after a RENUM command:

<u>Program before RENUM</u>

```
 List of EXAM.BAS

     3      AM1 = 16
     12     TOT = 13
     20     SUM = AM1 + TOT
     24     PRINT "SUM IS";SUM
     33     END
 Ok RENUM
 Ok LIST
```

<u>Program after RENUM</u>

```
 List of EXAM.BAS

     10     AM1 = 16
     20     TOT = 13
     30     SUM = AM1 + TOT
     40     PRINT "SUM IS";SUM
     50     END
```

2.2.2 Personal BASIC Command Format

The commands used in the CALAVG program were NEW, RUN, and SAVE. You will learn many other commands in the next sections. Here are the rules for BASIC commands:

- Commands do not have line numbers.
- Commands are followed by a <cr>.
- Commands must be spelled correctly.
- Some commands require additional information.

These are examples of correct and incorrect Personal BASIC commands:

Correct	Incorrect	
NEW	20 NEW	(Line number included)
DELETE 20-50	DELETE	(No line numbers given)
RENUM	RENUMB	(Command misspelled)
DELETE -50	DELETE50	(No space after command)

2.3 How to Save and Load Programs

Before you begin to learn the various Personal BASIC statements and write programs, you should have some knowledge of how Personal BASIC uses your computer's memory and external storage to store and retrieve programs. We will study the concepts of permanent and working storage, and learn how to use the commands controlling program storage.

2.3.1 Working and Permanent Storage

As you type a Personal BASIC program, it is placed into your computer's memory, a temporary or working storage. Temporary means that if you turn off your computer or have a power failure, the program in working storage is lost forever. This internal temporary memory is also called Random Access Memory (RAM).

Often you will want to save your program to use after lunch, tomorrow, or next week. Programs are saved into permanent storage, usually on a floppy or hard disk. Programs saved in permanent storage can be retrieved and run or revised whenever you wish.

Figure 2-1 illustrates how working and permanent storage interact. The programmer is typing a program into working storage. When the program is complete, it moves to permanent storage with the SAVE or REPLACE command. When the program is needed for execution or revisions, it is retrieved from permanent storage and enters working storage with the OLD, or RUN command. The MERGE command combines a program in permanent storage with the program in working storage.

The NEW command clears working storage and makes it ready for a new program. ERA deletes programs from your permanent storage, and NAME renames your program. These commands are discussed in Section 2.3.2.

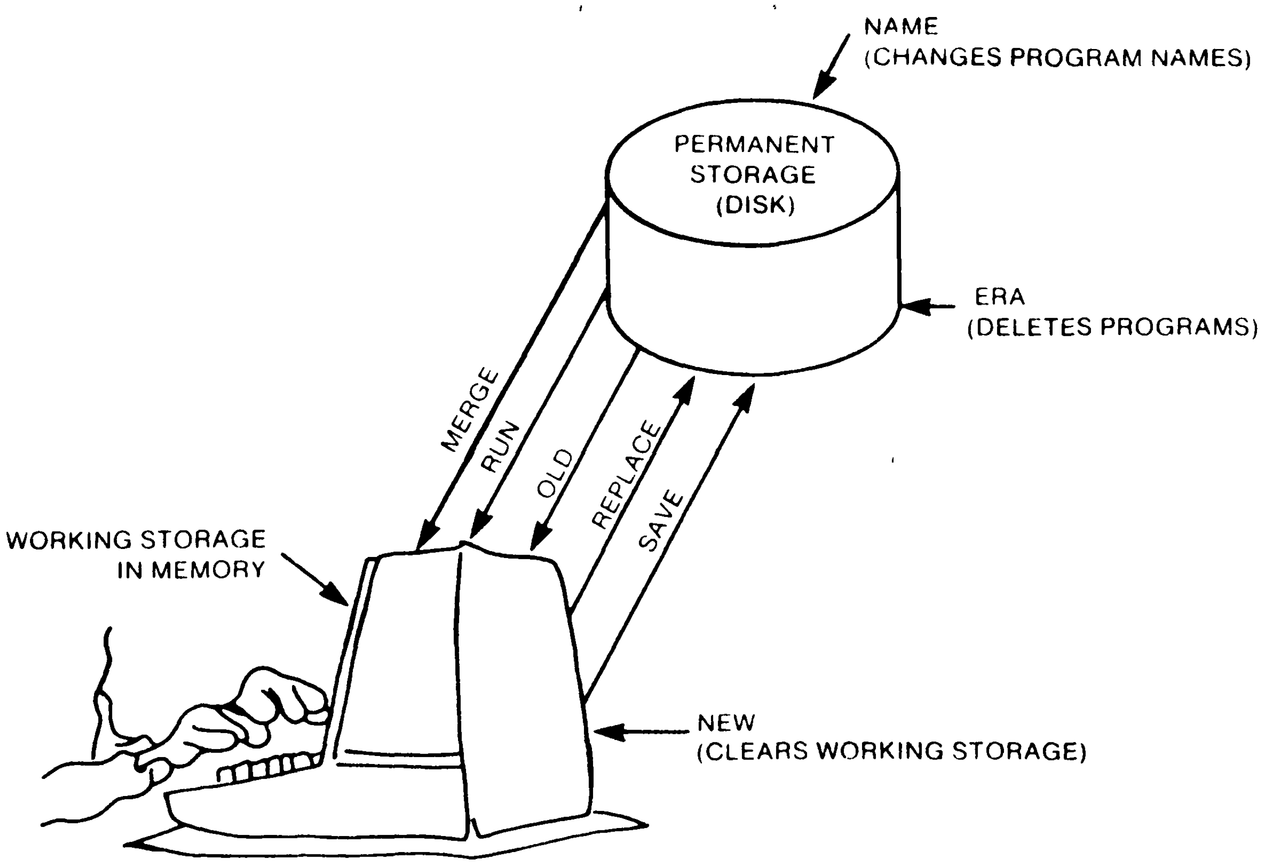


Figure 2-1. Working and Permanent Storage

2.3.2 Personal BASIC Storage Commands

Personal BASIC commands SAVE, REPLACE, MERGE, OLD, and RUN move your program between working and permanent storage. Commands NEW, DELETE, ERA, NAME, DIR, and LIST help you manage your program files. LIST is described because it helps you see what is happening in working storage.

NEW or NEW <filename>

Use the NEW command when you want to start writing a new program. NEW erases anything in working storage and can give your new program a name. If you do not name your program with NEW, you must name it with the SAVE command if you place it into permanent storage.

Type in the following program. NEW clears working storage and names the program "CARDS".

```
Ok NEW CARDS  <------------------- Clears working storage and
                                   names the program "CARDS"
Ok 100 LET CARD=7              --+
Ok 110 LET SUIT$="HEARTS"        |
Ok 120 PRINT CARD,SUIT$          |<-- Program CARDS
Ok 130 END                     --+
Ok
```

Note: it is good practice to use NEW before you start any program. Otherwise, your new program might be mixed with parts of an old program. Personal BASIC treats all the lines in working storage as one program.

SAVE or SAVE <filename>

SAVE moves the Personal BASIC program currently in working storage to permanent storage. The name of the program being saved cannot be in permanent storage when you give the SAVE command. If you want to save a program under the same name as a program already in permanent storage, use the REPLACE command.

SAVE is usually used after you have started a new program with NEW. You must use a program name with SAVE if the program was not named with NEW. In example program CARDS, add a SAVE command after the END statement and CARDS is put into permanent storage. The complete terminal dialogue is listed below:

```
Ok NEW CARDS
Ok 100 LET CARD=7
Ok 110 LET SUIT$="HEARTS"
Ok 120 PRINT CARD,SUIT$
Ok 130 END
Ok SAVE     <-------- SAVE puts program CARDS into
                      permanent storage
```

The program named CARDS is now stored in permanent storage. The command DIR gives you a list of all the files in your permanent storage. Type DIR now to make sure CARDS was stored. The DIR list should look something like the next example. The items will vary, depending on what is in permanent storage.

```
B>DIR

B: BASIC        CMD
B: CALAVG       BAS
B: CARDS        BAS
```

Your first program, CALAVG, is also listed. The DIR list shows your programs as CARDS.BAS and CALAVG.BAS. Personal BASIC adds a filetype, BAS, to all Personal BASIC program filenames in permanent storage. This helps Personal BASIC and you recognize Personal BASIC files. Filetypes can be up to three characters long or omitted.

REPLACE or REPLACE <filename>

REPLACE works just like SAVE, except that REPLACE replaces a program with the same name in permanent storage. If no program name is given, REPLACE uses the name of the program in working storage. REPLACE is usually used after the OLD command. The program in permanent storage being replaced is erased.

Note: it is good programming practice to use the REPLACE command every ten or fifteen minutes when working on a program. This habit keeps your blood pressure down if there is a power failure after you typed 200 lines of a Personal BASIC program. If you remember to use REPLACE, the most you lose is the last ten or fifteen minutes of work.

OLD <filename>

OLD is the opposite of SAVE. OLD clears working storage and then moves a program from permanent storage to working storage. Let's use OLD to retrieve program CALAVG from permanent storage. Give the command:

OLD CALAVG

OLD erased program CARDS from working storage and moved program CALAVG from permanent storage to working storage. Programs CARDS and CALAVG are unchanged in permanent storage. The LIST command, explained in the next section, will prove to you that CALAVG is in working storage.

LIST

How do we know that CALAVG is in working storage? There is a handy command, LIST, that prints the program in working storage. List prints the entire program or specific lines in numerical order by line number. The following is the LIST command format:

LIST	lists all of your program.
LIST 520	lists only line 520.
LIST 40-100	lists lines 40 through 100.
LIST 230-	lists lines from 230 to the end.
LIST -500	lists lines from the beginning through 500.

Now type LIST, and the statements you typed for program CALAVG are printed.

MERGE

MERGE combines the program in working storage with the specified program in permanent storage. If a line number in the program coming from permanent storage is the same as a line number in working storage, the line in working storage is replaced. The following command merges the program NEWPROG with the current program in working storage:

```
Ok MERGE NEWPROG
```

Here is an example of using MERGE. Program ADD is in working storage, and program NEWPROG is brought from permanent storage to merge with program ADD. Each program is listed, the MERGE command given, and then a listing is shown of the merged programs.

If you want to try the example, type programs ADD and NEWPROG and save them with SAVE.

```
Ok OLD ADD
Ok LIST
List of ADD.BAS

    10      A=10
    20      B=20
    30      PRINT A + B
    40      END
```

This is a listing of NEWPROG in permanent storage:

```
    40      FOR E=1 to 10
    50      PRINT "Personal BASIC Merge"
    60      NEXT E
    70      END
```

The merge command is given with program ADD in working storage.

```
Ok MERGE NEWPROG
```

The following is the resulting program, still called ADD.

```
Ok LIST
List of ADD.BAS

    10 A=10
    20 B=20
    30 PRINT A + B
    40 FOR E=1 TO 10
    50 PRINT "Personal BASIC Merge"
    60 NEXT E
    70 END
```

Notice that line 40 in program NEWPROG replaced line 40 in program ADD. Program NEWPROG is still unchanged in permanent storage. A REPLACE command stores the newly merged program ADD into permanent storage, replacing existing program ADD.

MERGE is very useful for adding special-purpose routines or programs to the program you are writing. MERGE saves lots of typing time. Remember that the line numbers must be planned so that you do not erase any lines in working storage that you want to retain in your program.

RUN or RUN <filename>, <line number>

RUN by itself runs the program in working storage. RUN with a program name loads the program from permanent storage and runs it. RUN with a comma and line number runs the program in working storage, starting at the given line number. When RUN is given with a program name, working storage is cleared. These are the RUN command formats:

RUN	runs the program in working storage.
RUN CALAVG	clears working storage, brings program CALAVG into working storage and runs it.
RUN CALAVG, 30	clears working storage, brings program CALAVG into working storage and runs it, starting at line 30.
RUN, 30	runs the program in working storage, starting at line 30.

Bring program CALAVG into your working storage with the OLD command. Type RUN to run CALAVG. You will see the question mark asking for the first number to average, which means that program CALAVG is running. This appears on your screen:

```
Ok RUN
?
```

We are through with the program for now, so return control to Personal BASIC with CTRL-C. CTRL-C stops a running program, and you will see this line:

```
-- Break -- at line 10
Br
```

The program stopped at line 10 with the Break prompt. Return control to Personal BASIC with another CTRL-C.

DELETE <line number> <line number>

DELETE erases lines from your program in working storage. You can delete any one line by typing the line number and <cr>. Here are some examples of DELETE:

DELETE 30-80	deletes lines 30 through 80.
DELETE -70	deletes all lines up to and including line 70.
DELETE 20-50, 100-150	deletes lines 20 through 50 and 100 through 150.
80	deletes line 80 only.

Use DELETE to delete some lines from program CALAVG. Make sure CALAVG is in working storage and type this command:

```
Ok DELETE 20-50
```

Use LIST to prove that lines 20, 30, 40 and 50 were deleted. Your listing is the following:

```
 Ok LIST

List of CALAVG.BAS

   5      REM AVERAGE OF 3 NUMBERS
   10     INPUT A
   60     END
```

Remember that program CALAVG is now labeled CALAVG.BAS because a filetype (BAS for Personal BASIC files) is automatically added to the filename of each program file as it is stored into permanent storage.

Do you know how to recreate the original CALAVG program in your working storage? When you deleted statements 20, 30, 40, and 50, they were deleted from working storage. The original program CALAVG is still unchanged in permanent storage. Bring CALAVG into working storage with the OLD command. LIST the program to make sure all of the statements are still there.

ERA <filename>

ERA erases program files from permanent storage. This command erases program CARDS from permanent storage. Once erased, the program cannot be recovered.

```
Ok ERA CARDS
```

Use DIR to verify that CARDS was erased.

NAME <old filename> AS <new filename>

NAME changes the name of a file in your permanent storage. The old filename must exist and the new filename must not exist. The file is unchanged after this command, but has a new name. The filenames must be enclosed in quotation marks. Let's rename your CALAVG program to AVERAGE. Type in this command:

```
Ok NAME "CALAVG" AS "AVERAGE"
```

Use DIR to see that the program is now named AVERAGE. Now, rename AVERAGE to the original name, CALAVG. The NAME command is

```
Ok NAME "AVERAGE" AS "CALAVG"
```

Verify the change with DIR.

End of Section 2

Section 3
Personal BASIC Arithmetic

Many people believe that you have to be a mathematician to operate or program a computer. Do not believe them. Anyone can master the concepts of programming.

In this section, we will discuss variables and how they are used in BASIC, the LET statement, and the various arithmetic operations possible in BASIC. Then, you learn in what order BASIC computes expressions and how the "E" notation is used to indicate very large or small numbers.

3.1 What are Variables?

A variable is the name given to a quantity that can assume different values during the running of a program. The two types of variables are numeric variables, and string variables. Read this section carefully, because these ideas are used in the following sections.

3.1.1 Numeric Variables

In line number 50 of a Personal BASIC program, R is the name of a numeric variable.

```
50 LET R = 15
```

Variable means that you can assign almost any value to R. In the LET statement shown, R is assigned the value, 15. Personal BASIC does not require the word, LET, so we do not use LET in future examples to save time and fingers. See Section 5.1.1 for more information about the LET statement.

Inside the computer, Personal BASIC assigns a memory space the name R and puts the number 15 into memory space R. The variable R remains equal to 15 until you assign another value to R. Compare a variable to a post office box. The number of the box never changes, but new mail is put into the box almost every day.

The following statements show how memory changes when variables are assigned during a program run.

10 GROSS=1500	GROSS $\boxed{1500} \longleftarrow$	The value, 1500, is placed into a memory location labeled GROSS.
20 TAX=.2*GROSS	TAX $\boxed{300} \longleftarrow$	The tax value is computed and placed into a location labeled TAX.
30 MISC=100	MISC $\boxed{100} \longleftarrow$	Variable MISC is assigned a value of 100.
40 NET=GROSS-TAX-MISC	NET $\boxed{1100} \longleftarrow$	NET is calculated and the value put into location NET.
50 PRINT NET		The variable values are unchanged after a PRINT statement.

Figure 3-1 shows how memory looks after the above statements are run.

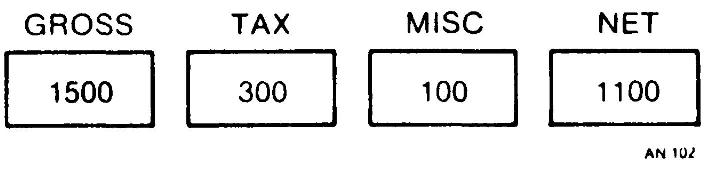


Figure 3-1. Memory Location Assignments

All of the variables retain their values until another value is assigned to the variable. For example, if you wanted to compute GROSS in multiples of 100, add 100 to GROSS and repeat the calculations. Add 100 to GROSS like this:

```
GROSS=GROSS+100
```

The value in the memory location assigned to GROSS is now:

GROSS

$\boxed{1600}$

In BASIC, the equal sign does not mean the same as it does in arithmetic. The equal sign means to compute whatever is on the right of the equal sign and store the value into the variable on the left side.

A name for a variable in Personal BASIC can be up to 31 characters. The characters must be only letters, numbers, or periods. The first character must be a letter. These are legal variables:

A RT357U X4 GROSSPAYLESSFICA Z4Y7.F

Numeric variables can be labeled as integer, single precision, or double precision. Single precision is sufficient for most applications, including business calculations, but double precision is sometimes desired for scientific work.

The labels used to indicate the types of variables are listed in Table 3-1.

Table 3-1. Variable Declaration Labels

Label	Type	Example
!	Single Precision	A! or A
#	Double Precision	SCALE2#
%	Integer	QE5R%

Section 2 of the Personal BASIC Language Reference Manual describes the types of variables in more detail.

If no label is included, the variable is single precision. In the examples in this book, all numeric variables are single precision.

Numeric variable types can be assigned in your program by several Personal BASIC statements. They are DEFINT, DEFSNG, DEFDBL, and DEFSTR. These statements are described in the Personal BASIC Language Reference Manual, Section 5.

Here are few things to keep in mind if you are going to use numeric variable types other than single precision.

- Double precision gives higher accuracy, but requires more storage space.
- The more precision, the slower the computation time.
- Integer variables run faster than the other types.

3.1.2 String Variables

Computers work with letters, symbols, and numbers. A group of letters and symbols is called a string. Numbers can also be used in strings. The variables used with strings are called string variables. String variables can contain up to 255 characters. They have names, just like the numeric variables. The names follow the same rules, except that they end with a dollar sign $. Here are some examples of string variable names:

```
A$        NAME$        RX4632T$        ET$
```

Variable A is not the same as variable A$. Both can be in the same program. When you set a string variable to a value, the value must be enclosed in quotation marks. If you want to store the name "George Washington" in string variable NAME$, the statement is

```
NAME$ = "George Washington"
```

String variables print just like numeric variables. The statement

```
PRINT NAME$
```

produces this output:

```
George Washington
```

The contents of string variables can be manipulated in many ways. Section 7 describes in detail how to use string variables.

3.1.3 Rules for Variables

Here are some rules to remember about variables:

- Any numeric variable is zero until assigned a value.
- When a variable is assigned a value, the previous value is lost.
- Variable names can be up to 31 characters long.
- Variable names must start with a letter.
- Variable names must be made up of only letters, numbers, and periods.
- Numeric variables are single precision unless declared otherwise.

Here are some examples of these rules. Type them into your computer if you want to see what they do. Personal BASIC treats upper- and lower-case the same. Variable A means the same as variable a. In the example below, TRASH can be typed trash or TrAsH and still be the same variable.

```
Ok TRASH=16            Variable TRASH is set to 16
Ok R12$="LOVE"         String variable R12$ is set to LOVE
Ok PRINT TRASH
 16
Ok PRINT R12$
LOVE
Ok TRASH=127           Variable TRASH is reset to 127
Ok PRINT TRASH
 127                   The value, 16, was replaced by 127

Ok PRINT F             Variable F is zero, because nothing
 0                     has been assigned to F

Ok TOTAL=TRASH+555     Variable TOTAL is assigned the value
Ok PRINT TOTAL         682, equal to the sum of variable
 682                   TRASH and 555.
```

3.2 Personal BASIC Arithmetic Operations

If you started this book from the beginning, you learned in Section 2.1 that Personal BASIC does six arithmetic operations. They are

- addition
- subtraction
- multiplication
- division
- exponentiation
- combinations of the above

Exponentiation is a number raised to a power. For example, 4 raised to the third power is 4*4*4 or 64, and is written as 4 . Ten to the fourth power is 10 , 10*10*10*10, or 10000.

Personal BASIC calculates arithmetic formulas a little differently than we do with a calculator and pencil. The important thing about Personal BASIC arithmetic is the order in which the calculations are made. This order is sometimes called precedence. In this simple calculation,

```
PRINT 6 + 9/3
```

the answer is 5 or 9, depending on the order used to make the calculations. If you add 6 and 9 and then divide by 3, the answer is 5. If you first divide 9 by 3 and then add 6, the answer is 9. Rules do exist for the order of calculations.

Personal BASIC does arithmetic operations from left to right, in the following order:

1) exponentiation
2) multiplication and division
3) addition and subtraction

Look at this example:

```
PRINT 6+8^2/4-3
```

Personal BASIC first looks for exponentiation. There is one (8^2) and it is calculated first. That leaves us with the following:

```
6+64/4-3
```

Next, Personal BASIC looks for multiplication or division and performs the calculation 64 divided by 4. Now the formula is

```
6+16-3
```

Personal BASIC completes the calculation by performing the addition and subtraction, giving the final answer of 19.

Equally important are the rules used by Personal BASIC for handling parentheses. Parentheses can modify the precedence of calculations. The rules are simple:

- Calculations within parentheses are done first, using the rules of precedence already described.
- If there are parentheses within parentheses, the innermost parentheses are calculated first.

This example shows how Personal BASIC handles parentheses:

```
8+(6+(3+4)-3^2)
```

Personal BASIC scans from left to right and finds a set of parentheses within another set. The inner parentheses are completed first, leaving the following:

```
8+(6+7-3^2)
```

The portion within the parentheses is calculated. The exponentiation is done first, followed by the addition and subtraction. The result is

```
8+4
```

and the final answer is 12.

Almost all versions of BASIC use these rules of order when calculating formulas.

3.3 Scientific Notation

Do not worry too much about scientific, or E notation unless you work with lots of very large or small numbers. You should know something about this subject because sooner or later, one of these numbers will appear in your output.

Personal BASIC can print only six digits of the result of an arithmetic computation. Some way is needed to print very large numbers and very small numbers. Personal BASIC cannot print computation results like these:

```
7943000000000.     0.0000000481
```

Personal BASIC prints these numbers as:

```
7.943E+12 and 4.81E-8
```

For the large number, E+12 means that the decimal point has to be moved 12 places to the right of the existing decimal point to convert to the full number. This is how to make the conversion:

7.943E+12	Original E Notation.
$7.\underbrace{943000000000}_{\text{12 places}\longrightarrow}.$	Count 12 places to the right and add zeros as necessary.
7943000000000.	The converted number.

Use the same method of converting the small number, except move the decimal point to the left of the existing decimal point.

Here are the rules for changing from E to decimal notation.

- If the sign is plus, move the decimal point to the right as many places as the number after E.
- If the sign is minus, move the decimal point to the left as many places as the number after E.

End of Section 3

Section 4
Editing Your Program

4.1 The Need for Editing

In the following sections, you are asked to type several programs. Even experienced programmers press the wrong keys, change lines, and insert lines in a program. There are both time-consuming ways and easy ways to make these changes. Personal BASIC provides a comprehensive editor which saves you time when program revisions are necessary. The Personal BASIC Edit Mode is unique because it can be entered when you are writing program statements or while you are debugging (finding program errors).

You can use an easy, but sometimes time-consuming method of making changes. This method is sometimes the best way. For example, if you want to change line 100, retype the revised line 100, press <cr>, and the original line 100 is replaced by the line 100 you just typed. Or, you can backspace to your error and retype it. EDIT usually saves time over these methods because you do not have to retype an entire line, and you can make changes quickly anywhere in the program. Experience tells you when to use EDIT.

4.2 Editing Subcommands

This section shows you enough EDIT subcommands to help you do the sample programs. A complete description of EDIT is in the Personal BASIC Language Reference Manual.

The Edit Mode subcommands can do the following functions:

- move the cursor left and right
- insert characters
- delete characters
- search for characters
- replace characters
- end and restart Edit Mode

We need a program to practice editing on, so bring program CALAVG into your working storage using the OLD command. Do the editing examples and become familiar with editing techniques. Here is a listing of CALAVG:

```
Ok OLD CALAVG
Ok LIST
List of CALAVG.BAS

    5      REM Average of 3 Numbers
    10     INPUT A
    20     INPUT B
    30     INPUT C
    40     AVG=(A+B+C)/3
    50     PRINT "THE AVERAGE IS";AVG
    60     END
```

Type EDIT to enter the Edit Mode. Follow EDIT with a space and the line number you want to edit. EDIT followed by a space and a period gives you the line number you are currently on or the line number you just entered. This command brings in line 40 for editing:

```
EDIT 40
```

EDIT followed by a space and a period gives you the line just entered or the line you are working on, ready for editing.

```
EDIT .
```

Type EDIT 40. EDIT prints line 40 and an edit line like this:

```
Ok EDIT 40                    <---- EDIT Command
    40    AVG=(A+B+C)/3       <---- Line being edited
Ed                            <---- Edit line
```

Line 40 is now ready for editing. The editing subcommands are placed in the edit line.

4.2.1 Moving the Cursor

Move the cursor to the right with the Space Bar and to the left with the BACKSPACE key (Rubout key on some terminals). There are several edit subcommands that move the cursor. Press <cr> after each subcommand character.

Subcommands L and R

L moves the cursor left to the beginning of the edit line. R moves the cursor right to the end of the edit line.

Subcommand X

X positions the cursor to the end of the line you are editing and enters Insert mode. X is usually used to position the cursor before adding to the end of a line.

Now that we know how to enter the Edit Mode and move the cursor, we can try the various edit subcommands. Return to the Ok prompt with <cr>.

4.2.2 Inserting Characters

Subcommand I

I inserts characters just above where the I is placed in the edit line. Move the cursor with the Space Bar or BACKSPACE until it is under the position where you want to insert a character or characters. Type I and the characters you want to insert. Press <cr> and the insert is complete. In this example, insert the word FINAL before the word "AVERAGE."

```
Ok EDIT 50
   50    PRINT "THE AVERAGE IS";AVG
Ed
```

Type the EDIT command, EDIT 50. Space the cursor to the position under the "A" in "AVERAGE". Type I and the word you want to insert, "FINAL ." Include a space after "FINAL ." Think of the I as reaching up into the line being edited and pushing everything to the right. Now your edit line looks like this:

```
   50     PRINT "THE AVERAGE is";AVG
Ed                  IFINAL
```

Press <cr> and line 50 changes to

```
50 PRINT "THE FINAL AVERAGE IS";AVG
```

This procedure might seem complicated now, but with a little practice you will find that it is easier than retyping the entire line.

4.2.3 Deleting Characters

Subcommand D

D deletes the character directly above each D in the edit line. Line 50 is still available to edit, so let's remove the word FINAL that we just inserted. Move the cursor and type Ds under "FINAL." Press <cr> and line 50 is restored to its original form. The delete operation gives this result:

```
Ok EDIT 50
   50    PRINT "THE FINAL AVERAGE IS";AVG
Ed                  DDDDDD
   50    PRINT "THE AVERAGE IS";AVG
Ed
```

Subcommands I and D can be used together. If we want to replace the word "AVERAGE" with the word "COUNT" in line 50, this edit procedure is used:

```
Ok EDIT 50
   50    PRINT "THE AVERAGE IS";AVG
Ed                 DDDDDDDICOUNT
   50    PRINT "THE COUNT IS";AVG
Ed
```

Now, return line 50 to its original form.

```
   50    PRINT "THE COUNT IS";AVG
Ed                 DDDDDIAVERAGE
   50    PRINT "THE AVERAGE IS";AVG
Ed
```

<u>Subcommand H</u>

H deletes all characters to the right of the cursor and enters Insert mode. You can then add characters to the end of the line. For example, we want to insert the formula for AVG in line 50, which makes line 40 unnecessary. Follow these operations:

```
Ok EDIT 50
   50    PRINT "THE AVERAGE IS";AVG
Ed                             H
Ed 50    PRINT "THE AVERAGE IS";
```

The H Subcommand removed AVG and moved the cursor to the position after the semicolon. Next, add the desired formula:

```
Ed 50    PRINT "THE AVERAGE IS";(A+B+C)/3
   50    PRINT "THE AVERAGE IS";(A+B+C)/3
Ed       Q ←———Finish with a Q.  This cancels the
Ok             changes made in this editing cycle.
```

See Subcommand Q, described in Section 4.2.6, "Ending and Restarting Edit Mode."

4.2.4 Searching for Characters

Subcommand S

S is used to quickly find a specific place in a line. The S subcommand positions the cursor in the edit line under the character being searched for. If the character searched for is not found, the cursor does not move. The format is Sc, where c is the character you are searching for. In this example, we want to jump to the B in line 40. The following steps are necessary:

```
Ok EDIT 40                          Enter EDIT mode
   40     AVG=(A+B+C)/3
Ed        SB                        Type SB and the cursor advances
   40     AVG=(A+B+C)/3             to the position under B
Ed                                  Press <cr>
Ok
```

Subcommand K

K works like S, except that all characters that the cursor passes under are deleted and the remaining characters move to the left. The cursor stops under the searched for character, which now becomes the first character in the line. If the character is not found, the cursor does not move. A KA subcommand given while editing line 50 gives this result:

```
Ok EDIT 50
   50     PRINT "THE AVERAGE IS";AVG
Ed        KA
   50     AVERAGE IS";AVG
Ed        Q <-------------------Cancel the edit with a Q
Ok                              and <cr> to the Ok prompt
```

4.2.5 Replacing Characters

Subcommand C

C deletes the character directly above it and replaces the deleted character with the characters to the right of C. We can change the "THE" in line 50 with the word "THIS", for example.

```
Ok EDIT 50                                      Enter EDIT mode
   50     PRINT "THE AVERAGE IS";AVG
Ed                   CIS                        Type CIS
   50     PRINT "THIS AVERAGE IS";AVG           The edited line
Ed                     Q <----------------------Cancel the edit
Ok
```

4.2.6 Ending and Restarting Edit Mode

<cr>

Press <cr> to save your changes and return to where you entered, either the Ok or Br prompt. (See Section 11.1 for more information about the Br prompt.)

Subcommand E

E performs the same function as <cr>.

Subcommand Q

Q returns you to the Ok or Br prompt and any changes made in the current Edit Mode are not saved.

Subcommand A

A deletes all changes made in the current EDIT and leaves you in EDIT. A allows you to start over on the line being edited.

Note: if you get a syntax error after typing a Personal BASIC line, you can enter Edit Mode by typing EDIT . (EDIT, space, and a period).

End of Section 4

Section 5
Inputs and Outputs

This section tells how to put data into a program and get something out. Section 5.1 describes the data input statements, LET, INPUT, READ, DATA, and RESTORE. Then the most-used output statement, PRINT, is examined in Section 5.2.1. Clear your working storage with the NEW command.

5.1 Personal BASIC Input Statements

5.1.1 LET

There are only three ways to enter numbers into a Personal BASIC program, and you already know one of them. Remember how we assigned values to variables?

```
50 KNEE=1600
```

Here, we set the variable, KNEE, to the value 1600. KNEE remains 1600 until it is set to another value. For example, we could say,

```
60 KNEE=KNEE+100
```

What do you think KNEE equals now? You are right if your answer is 1700. This is called the LET statement, but Personal BASIC does not require the word, LET. These two statements mean the same thing to Personal BASIC:

```
90 LEG=ANKLE+KNEE                  90 LET LEG=ANKLE+KNEE
```

5.1.2 INPUT

Another method of getting numbers into a program was shown to you in the example program, CALAVG, in Section 2.2. Remember the question marks? They were produced by the INPUT statement. Bring the CALAVG program into your working storage area with the OLD command. If it is not available for some reason, please type it again. Here is CALAVG:

```
Ok 5 REM Average of 3 Numbers
Ok 10 INPUT A
Ok 20 INPUT B
Ok 30 INPUT C
Ok 40 AVG=(A+B+C)/3
Ok 50 PRINT "THE AVERAGE IS";AVG
Ok 60 END
```

When we started CALAVG with the RUN command, in Section 2.2, the INPUT statement caused Personal BASIC to stop and print a question mark. The program stopped three times and waited until you had input three numbers for variables A, B, and C. Run CALAVG again. Use the numbers 48, 67, and 56. Your output looks like this:

```
 Ok RUN
 ? 48
 ? 67
 ? 56
THE AVERAGE IS 57
```

Save CALAVG now if it is not in your permanent storage. Use this SAVE command:

SAVE CALAVG

Use INPUT whenever you want the program user to input numbers or words or respond to a question.

There are other variations of INPUT. You can ask for more than one variable in the same statement. The CALAVG program requested values for variables A, B, and C in lines 10, 20, and 30. This line does the same thing:

```
10 INPUT A, B, C
```

INPUT can also print something when it asks for input. Most BASIC programs use this method of requesting input. Let's change your CALAVG program. Enter this statement:

10 INPUT "PLEASE TYPE YOUR NUMBERS";A, B, C

Delete lines 20 and 30 because they are not necessary. To do this, type each line number followed by a <cr>. The command LIST gives you a listing of the program in your working storage. Type LIST now with a <cr>. The listing looks like this:

```
 Ok LIST

List of CALAVG.BAS

   5      REM Average of 3 Numbers
   10     INPUT "PLEASE TYPE YOUR NUMBERS";A,B,C
   40     AVG=(A+B+C)/3
   50     PRINT "THE AVERAGE IS";AVG
   60     END
```

RUN this program. The output is

```
PLEASE TYPE YOUR NUMBERS ? 34, 56, 78
THE AVERAGE IS 56
```

Input any three numbers and separate by commas. Do not save the revised CALAVG. We will need the original version later.

If you do not want a question mark printed after your printed text, substitute a comma for the semicolon. The text to be printed must always be enclosed in quotation marks.

String variables can also be used with INPUT. The next statement uses string variable IN$ and expects an answer of YES or NO.

```
55 INPUT "IS THERE MORE INPUT";IN$
```

The correct answers for line 55 are YES or NO. If you answer YES, the program looks for more input. A NO answer ends the program or it branches to another section. If neither YES or NO is entered, the program prints an error message and returns to line 55. You will learn how to do this kind of decision making in Section 6.1. If you cannot wait to see how this works in a program, see program DICE in Section 8.2.

5.1.3 READ/DATA

There is one more method of bringing numbers and characters into your program, the READ and DATA statements. READ and DATA go together, because you cannot have one without the other.

These statements:

```
45 READ A
50 DATA 5
```

result in variable A being set to the number 5. READ statements can contain both numeric and string variables. Look at these examples and try them on your computer.

Program READ1 sets variable A to 428, the value in the DATA statement.

```
Ok NEW READ1
Ok 10 READ A
Ok 20 PRINT A
Ok 30 DATA 428
Ok 40 END
Ok RUN
 428
```

Program READ2 shows that a string variable can be used the same way.

```
Ok NEW READ2
Ok 10 READ T$
Ok 20 PRINT T$
Ok 30 DATA BUTTERFLY
Ok 40 END
Ok RUN
BUTTERFLY
```

Program READ3 is an example of using more than one DATA statement. Each time a READ statement is run, it takes the next DATA value. READ uses all the DATA statements no matter how many lines they occupy. 386 could have been included in line 70.

```
Ok NEW READ3
Ok 10 READ X
Ok 20 PRINT X
Ok 30 READ Y
Ok 40 PRINT Y
Ok 50 READ Z
Ok 60 PRINT Z
Ok 70 DATA 67, 45
Ok 80 DATA 386
Ok 90 END
Ok RUN
 67
 45
 386
```

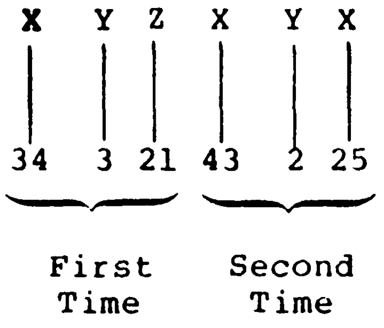
Program READ4 uses several variables in one READ statement. The READ statement in line 10 sets the variables S, R, and U to the values in the DATA statement in line 30.

```
Ok NEW READ4
Ok 10 READ S, R, U
Ok 20 PRINT S+R+U
Ok 30 DATA 45,23,67
Ok 40 END
Ok RUN
 135
```

Here is an example of how READ and DATA are used to obtain data for a Personal BASIC program. Type program COMPUTE and run it.

```
Ok NEW COMPUTE
Ok 30 REM READ/DATA Example
Ok 40 READ X, Y, Z
Ok 60 R=X+(y^3)/Z
Ok 70 PRINT R
Ok 75 GOTO 40
Ok 80 DATA 34, 3, 21, 43, 2, 25
Ok 90 END
Ok RUN
 35.2857
 43.32
READ statement ran out of data at line 40
```

This is how values are assigned in the preceding program:



The message, "READ statement ran out of data at line 40" means that the READ statement went to line 80 for more data the third time around the loop and did not find any. One of the next examples explains how to end a READ/DATA loop. A new statement, GOTO, returns the program to line 40.

The READ statement reads three numbers from the DATA statement and then does the calculation and printing. This is how READ and DATA work together:

1) READ assigns the first value in DATA (34) to the first variable in READ (X).

2) The program returns to the READ statement and assigns the next value in DATA (3) to the second variable in READ (Y).

3) The program returns again to the READ statement and assigns the last variable in DATA (21) to the last variable in READ (Z).

4) Because there are no more variables to assign values to, the program continues and does the calculation and printing.

5) The program returns to read the next set of numbers.

Here are some rules to remember for the READ and DATA statements.

- One READ statement can access several DATA statements. Data statements are accessed in the same order they are listed in the program.
- More than one READ statement can use the same DATA statement.
- If there are more variables in the READ statement then items in the DATA statements, this error message is printed: "READ statement ran out of data at line xx."
- If there are more items in the DATA statement than variables in the READ statement, the extra items are not read and there is no error condition. If there is a later READ statement, it starts with the first unread item.
- The DATA statements can appear anywhere in the program.
- There can be any number of DATA statements. READ uses them in order until there are no more variables.

5.1.4 RESTORE

RESTORE tells the READ statement to access the first DATA statement in the program. If you use RESTORE with the number of a DATA statement, the READ statement accesses the first item in that DATA statement. Try this example:

```
Ok NEW RESTORE
Ok 100 READ RM, S3, ET
Ok 110 RESTORE
Ok 120 READ X, B, R
Ok 130 DATA 56, 78, 34
Ok 140 PRINT RM;S3;ET;X;B;R
Ok 150 END
Ok RUN
 56  78  34  56  78  34
```

What happened here?

Things start out just like they did in program COMPUTE. Variables RM, S3, and ET are assigned the values in DATA, 56, 78, and 34. RESTORE then returns the second READ statement (line 120) to the first item in DATA. X, B, and R are also assigned the values 56, 78, and 34. All of the numbers print on the same line because of the semicolons. You will learn more about how PRINT spaces output in Section 5.2.1. Type the RESTORE program and run it both with and without the RESTORE statement. Type 110 and press <cr> to remove the RESTORE statement.

When you run with the RESTORE statement removed, this output results:

```
Ok RUN
READ statement ran out of data at line 120
```

This means that the READ statement at line 120 was unable to find any data because the READ statement in line 100 had used up the data. With program RESTORE, READ in line 120 was able to use the same data already read by the READ statement in line 100.

5.2 Personal BASIC Output Statements

5.2.1 PRINT

PRINT is the most-used output statement and has many variations. This book shows you how to do most of the PRINT operations. Full details of PRINT and another statement, PRINT USING, are in the Personal BASIC Language Reference Manual.

Let's start with several PRINT statements and a program containing PRINT statements and the resulting output. Then we will examine each PRINT operation in greater detail and show you how to use PRINT in your programs.

You can use a question mark ? in place of PRINT when writing a Personal BASIC program. The next example program uses both conventions. Look at the examples and then type and run program PRINTEX.

```
PRINT                          Displays a blank line

PRINT 230                      Prints a number
 230

PRINT "RED SNAPPER"            Prints whatever is in quotes
RED SNAPPER
```

```
 Ok NEW PRINTEX
 Ok 20 REM PRINT Format Samples
 Ok 30 A=5:B=3:C=8
 Ok 40 PRINT A+B
 Ok 50 PRINT "THE TOTAL IS";A+B
 Ok 60 PRINT
 Ok 65 REM PRINT Spacing
 Ok 70 ? A,B,C
 Ok 80 ? A;B;C
 Ok 90 ? A*B*C;
 Ok 100 ? A+B+C,
 Ok 110 ? "PRINT IS EASY!"
 Ok 120 END
 Ok RUN
```

The output from program PRINTEX is produced by these lines:

```
 8                                      Line 40
THE TOTAL IS 8                          Line 50
                                        Line 60
 5             3             8          Line 70
 5  3  8                                Line 80
 120  16      PRINT IS EASY!            Lines 90, 100, and 110
```

Here is how the items in program PRINTEX were printed:

1) Line 40 is the total of A+B.

2) Line 50 prints the text enclosed in quotation marks, followed by the value of A+B. The semicolon (;) made the 8 print next to the word "IS."

3) Line 60 PRINT by itself produces a blank line.

4) Line 70 prints the values of A, B, and C in columns. A print line is divided into print zones of 14 spaces each. The commas tell PRINT to print each item at the beginning of a print zone. The value of A prints in the first zone, the value of B in the second zone, and the value of C in the third zone.

5) Line 80 is the same as line 70, except that a semicolon is used instead of a comma. The semicolon tells PRINT to print the values with no separation. Then why are there spaces between the values? Printed numbers are followed by a space. Positive printed numbers are preceded by a space. Negative numbers are identified by a preceding minus sign.

   In the example, there are two spaces between values 5 and 3 and 3 and 8 for these reasons.

6) Line 90 prints the product of A, B, and C. The semicolon tells PRINT to put the next PRINT output on the same line, with no space between.

7) Line 100 prints the sum of A, B, and C on the same line with the product. The comma after C tells PRINT to put the next PRINT output on the same line and space it to the next print zone.

8) Line 110 prints the text, "PRINT IS EASY!" on the same line with the output produced by lines 90 and 100 because of the comma after the "C" in line 100.

5.2.2 TAB

The TAB function works like the tabs in a typewriter, except the tabs must be set each time you PRINT. Type this example program and RUN it. Do not forget NEW.

```
Ok NEW TABS
Ok 50 PRINT TAB(10)"This"
Ok 60 PRINT TAB(15)"is how"
Ok 70 PRINT TAB(20)"TAB works"
Ok 80 PRINT TAB(25)"with PRINT."
Ok 90 END
Ok RUN
```

The output of TABS looks like this:

```
         This
              is how
                   TAB works
                        with PRINT.
```

Each PRINT command positions the output to start at the line position of the TAB function. The "T" in the word "This" started at line position 10, the "i" in the word "is" started at line position 15, and each following PRINT command advances the output five spaces.

The value for TAB must be between 1 and 255. If the current print position is greater than the TAB value, TAB positions the PRINT output on the next line. Value 1 is the leftmost position on the line; the rightmost position is the defined line width minus 1.

TAB is used mainly for column headings and to position numbers on business and scientific reports.

5.2.3 PRINT USING

PRINT USING is a very powerful PRINT command that prints strings or numbers using a specified format.

When printing strings, PRINT USING can print only the first character of a string, specific characters, or exactly as input.

When printing numbers, PRINT USING can round the number, add asterisks, dollar signs, or use various other formats. See your Personal BASIC Language Reference Manual, Section 4.2, for more detailed information on PRINT USING.

5.3 Exercises

1) Write a Personal BASIC program to print the sum and average of these numbers: 474, 651, 562, 701, 631, and 568. Use 999 to tell the program that there is no more data to process. Use READ/DATA statements to input the numbers. The output should be in this format:

```
SUM = XXXX
AVERAGE = XXX.XX
```

2) Repeat exercise 1, but use INPUT instead of READ/DATA and use this output format:

```
THE SUM IS XXXX  AND THE AVERAGE IS XXX.XX.
```

3) Write a Personal BASIC program to print your name and address in this format:

```
Robert S. Jones
     140 Oak Avenue
          Sand City, CA  94562
```

   Use INPUT statements and input separately your name, street address, city, state, and zip code.

   See Appendix C for the suggested solutions.

End of Section 5

Section 6
Decisions and Looping

Let's stop a minute and see where we are in the BASIC learning process. The following sections use the information already learned, so it is important that you understand what we have covered so far. You should remember these statements and commands:

Statements	Commands
END	DELETE
GOTO	DIR
LET	ERA
PRINT	LIST
READ/DATA	NAME
REM	NEW
RESTORE	OLD
	RENUM
	REPLACE
	RUN
	SAVE

If you are not sure of any of the statements or commands listed, review the previous sections or refer to the Personal BASIC Language Reference Manual. Keep doing the examples. If you wonder, "What would happen if I tried .....?", go ahead and try it. The computer will not blow up and your curiosity will be satisfied.

6.1 Decisions, Decisions

So far, all of the example programs were run in the order of increasing line numbers. When we used GOTO, the program jumped to the line number after GOTO. One of the most useful and necessary features of Personal BASIC is the ability to jump to a line number only if certain conditions are true.

6.1.1 IF/THEN/ELSE

The IF/THEN/ELSE statement is best introduced with examples. The next program inputs numbers and prints "Over 100" for numbers over 100 and "100 or less" for numbers equal to 100 or less. Type the program and run it with various numbers.

```
Ok NEW NUMBER
Ok 10 INPUT "ENTER A NUMBER";N
Ok 20 IF N > 100 THEN GOTO 50
Ok 30 PRINT "100 OR LESS"
Ok 40 GOTO 10
Ok 50 PRINT "OVER 100"
Ok 60 GOTO 10
Ok 70 END
```

This is what happened in program NUMBER:

1) Line 10	Variable N is set to the number input.
2) Line 20	The IF statement sends the program to line 50 if N is greater than 100. If N is 100 or less, the program goes to the next statement, line 30.
3) Line 30	N was 100 or less, so "100 OR LESS" is printed.
4) Line 40	The program returns for another number.
5) Line 50	The program branched here from line 20, because N was greater than 100.
6) Line 60	The program returns for another number.
7) Line 70	The end of the program. The program never reaches this line, because the GOTO statement at line 60 always returns the program to line 10.

Return control to Personal BASIC with CTRL-C.

6.1.2 IF/THEN Variations

```
100 IF R45=16 THEN B3=42
```

In line 100, a variable is assigned a value after THEN instead of a GOTO, as in line 20 of program NUMBER.

```
300 IF A4=3.4 AND ERT=67 AND PU=567 THEN GOTO 350
```

More than one condition has to be true to branch to line 350. If any one condition is not true, the program run is continued with the next line after line 300.

The ELSE statement is even more flexible. Examine this statement.

```
245 IF COMM >SALARY THEN PRINT "COMM" ELSE IF SALARY>COMM
THEN PRINT "SAL" ELSE PRINT "EQUAL"
```

In this statement, three different things can be printed, depending on the values of the variables COMM and SALARY.

If this is true:	The printed output is
COMM > SALARY	COMM
SALARY > COMM	SAL
SALARY = COMM	EQUAL

CTRL-J

Note: did you notice that line 245 extends more than one physical line? Remember that logical lines of one or more statements are terminated by <cr>. It is possible to extend a logical line more than one physical line by using the line feed key. With line feed, you can continue typing a logical line on the next physical line without using a <cr>. Personal BASIC can have a maximum of 255 characters in each program line.

Program GRADES uses the concepts we just learned for IF/THEN/ELSE. The program asks you to input student grades and then prints "PASS" after every grade 65 or over and "FAIL" after every grade under 65. Indicate the end of the grades by inputting an impossible grade, 999. This is sometimes called the end-of-file. Examine program GRADES carefully before you continue.

```
Ok NEW GRADES
Ok 5 REM PASS or FAIL
Ok 10 INPUT GRADE
Ok 20 IF GRADE=999 THEN END
Ok 30 IF GRADE>100 or GRADE<1 THEN GOTO 60
Ok 40 IF GRADE <65 THEN PRINT GRADE;"FAIL" ELSE PRINT GRADE;
   "PASS"
Ok 50 GOTO 10
Ok 60 PRINT "GRADE SHOULD BE FROM 1 THROUGH 100"
Ok 70 GOTO 10
Ok 80 END
```

Enter program GRADES and run it using various number grades. Here is a sample run of GRADES:

```
 Ok RUN
 ? 45
45 FAIL
 ? 65
65 PASS
 ? 120
GRADE SHOULD BE FROM 1 THROUGH 100
 ? 95
95 PASS
 ? 999
 Ok
```

Two new concepts are introduced here: a test for program termination and error messages.

Every program has to end some time. How does the program know when to end? The program designer must provide a way for the program to recognize the end of the input. For example, a payroll file could have as the last name, Mr. XX.XX, a very improbable name. The program checks each name and when it finds Mr. XX.XX, it knows that there is no more input. A social security number of 999-99-9999 could also be used.

For program GRADE, a specific number outside the legal grade range tells the program the input has terminated. Notice that the test for the end of input is made in line 20, before the test for a legal grade.

The program also checks to make sure that each grade input is within the limit of 1 through 100 (see line 30). If the grade is not within the range, an error message is printed. Error messages are very important in programs. The programmer must anticipate errors by the program user and provide error messages to clearly explain the problem and tell the user how to proceed. Good error handling is a necessary part of the programming task.

Note: IF/THEN/ELSE is one statement. ELSE cannot be separated from the rest of the IF statement by a colon. ELSE cannot be on another program line. This example is invalid:

```
50 IF COWS=MILK THEN GRAPES=WINE
60 ELSE ET=3.1416
```

ELSE ET=3.1416 should be in line 50 after WINE.

6.2 Looping Around - WHILE/WEND, FOR/NEXT

In the preceding sections, some of the programs performed loops under the control of GOTO and IF/THEN statements. GOTO can send a program into a loop with no way of getting out. IF/THEN can end loops by checking for a particular value or condition. Personal BASIC makes life easier for programmers by providing many ways to handle looping.

6.2.1 WHILE/WEND

The WHILE/WEND statements run a series of statements in a loop as long as the stated condition of WHILE is true (not zero). Here is an example:

```
Ok NEW
Ok 40 C=4
Ok 50 WHILE C
Ok 60 PRINT "I'm trapped in a loop"
Ok 70 C=C-1
Ok 80 WEND

continuation of program
```

How many times will the text be printed? The WHILE/WEND logic is to execute the loop from WHILE to WEND until C=0. Here is how it runs through the loop:

1) C starts as 4 and is reduced by 1 each time the loop runs.

2) WEND sends Personal BASIC to WHILE until the WHILE value is zero.

3) The statement C=C-1 at line 70 reduces C by one each time the WHILE/WEND loop is run.

4) When the WHILE value is zero, the program run continues with the line after WEND.

5) The program prints the text, "I'm trapped in a loop" four times.

An expression can be used after WHILE, such as VM>PQ7. Run this example:

```
Ok NEW
Ok 200 VM=10
Ok 210 PQ7=5
Ok 220 WHILE VM>PQ7
Ok 230 PRINT "LOOP COUNTER"
Ok 240 VM=VM-1
Ok 250 WEND

continuation of program
```

The loop runs five times, until VM is reduced to five, the value of PQ7. The program then continues with the statement after WEND.

WHILE/WEND is useful for controlling the number of times a program goes through a loop. If a program decision is made not to enter the loop, set the WHILE value to zero before the program reaches the WHILE statement.

WHILE/WEND loops can be nested (enclosed within each other). Each WEND matches the most recent WHILE. Each WHILE statement requires a WEND statement.

6.2.2 FOR/NEXT

The FOR/NEXT statement uses a loop to set a variable to a series of values and terminates the loop when all the values are used. This is the format of a FOR/NEXT statement:

```
30 FOR X = 1 to 10
```

where X is the variable name; 1 is the lower limit (value of X when loop starts); and 10 indicates the upper limit (value of X when the loop ends). The statement, or statements, to be run using the values of X are inserted after the FOR command and before the NEXT command.

```
40 PRINT X,X^2,X^3

50 NEXT X
```

The NEXT statement adds 1 to the value of X and branches to line 50 until X=10

Here is the complete program. Write the output you think SQRCUBE will give on a piece of paper and then type SQRCUBE and run it. Were you right?

```
Ok NEW SQRCUBE
Ok 10 PRINT "NUMBER","SQUARE","CUBE"
Ok 20 PRINT
Ok 30 FOR X=1 to 10
Ok 40 PRINT X,X^2,X^3
Ok 50 NEXT X
Ok 60 END
```

Program SQRCUBE computes the squares and cubes of the numbers from 1 to 10 and prints the results. This is the output:

```
Ok RUN
NUMBER         SQUARE         CUBE

 1              1              1
 2              4              8
 3              9              27
 4              16             64
 5              25             125
 6              36             216
 7              49             343
 8              64             512
 9              81             729
 10             100            1000
```

The FOR statement in program SQRCUBE increases the value of X by 1 until the values from 1 through 10 are used. The values for the variable can be increased or decreased for values other than 1. In program SQRCUBE, we can easily change it to print the squares and cubes of all numbers from 0 through 100 ending in 5 by changing line 30 to

```
30 FOR X=5 TO 95 STEP 10
```

Here is the output with this change:

```
Ok 30 FOR X=5 TO 95 STEP 10
Ok RUN
NUMBER        SQUARE        CUBE

 5             25            125
 15            225           3375
 25            625           15625
 35            1225          42875
 45            2025          91125
 55            3025          166375
 65            4225          274625
 75            5625          421875
 85            7225          614125
 95            9025          875375
```

X starts at 5 and is increased by 10 for each loop until X equals 95.

The value of the FOR variable can also be decreased. In the same program, if we want to print the same squares and cubes but start with the largest, we can use this statement:

```
30 FOR X=95 TO 5 STEP -10
```

Here is the output with this change:

```
     Ok 30 FOR X=95 TO 5 STEP -10
     Ok RUN
     NUMBER        SQUARE        CUBE

     95             9025          875375
     85             7225          614125
     75             5625          421875
     65             4225          274625
     55             3025          166375
     45             2025          91125
     35             1225          42875
     25             625           15625
     15             225           3375
     5              25            125
```

X starts with 95 and is decremented by 10 until X=5.

The value of the FOR variable can be set outside of the FOR/NEXT loop. This is necessary if the number of times through the loop has to be determined by another part of the program.

Many programs use nested FOR/NEXT loops. Two or more FOR/NEXT loops must not have crossing paths.

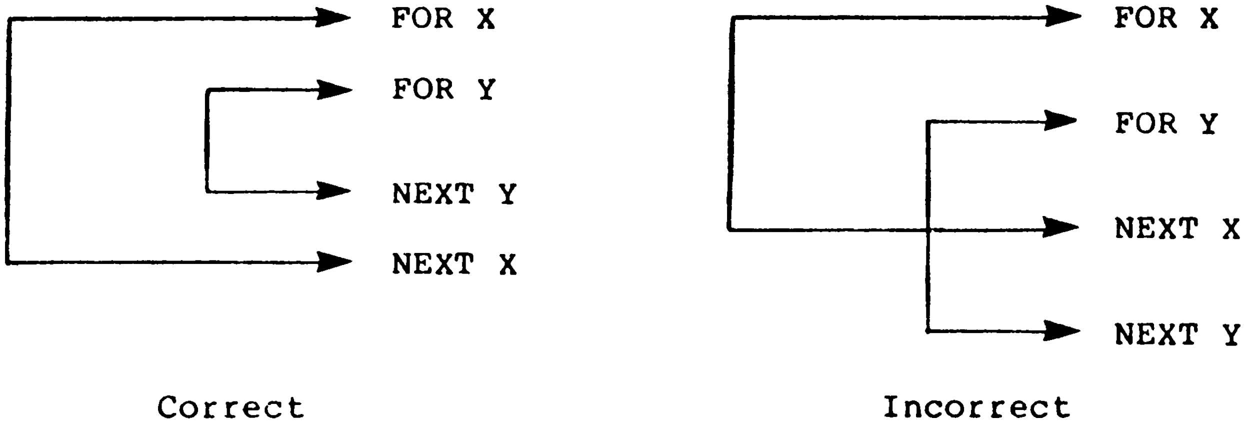
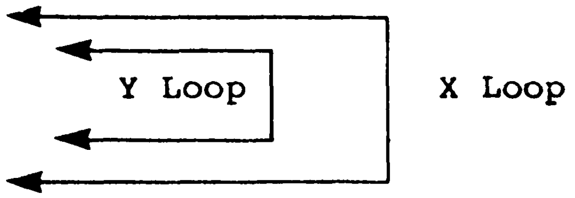


Figure 6-1. Nested FOR/NEXT Loops

The inner loop (Y) must be completed each time before the outside (X) is started again.

Program POWERS calculates and prints the value of X to the second, third, fourth, and fifth power for X = 1 to 10.

```
Ok NEW POWERS
Ok 10 FOR X=1 to 10
Ok 20 FOR Y=2 to 5
Ok 30 PRINT X, X^Y
Ok 40 NEXT Y
Ok 50 NEXT X
Ok END
Ok RUN
 1             1
 1             1
 1             1
 1             1
 2             4
 2             8
 2             16
 2             32
 3             9
 3             27
 3             81
 3             243
 4             16
 4             64
 4             256
 4             1024
```



The program continues for the values of X from 1 through 10.

1) Line 10 sets up the X FOR/NEXT loop.
2) Line 20 sets up the Y FOR/NEXT loop.
3) Line 30 calculates various powers of X.
4) Lines 40 and 50 repeat the loops.

Program INTRATE computes interest and shows the practical use of some of the statements we have learned. Type the program, save it as program INTRATE, and run it. The program and a sample run are listed below. Try to analyze how the program works before you read the explanation.

```
Ok NEW INTRATE
Ok 10 REM Monthly Interest Compounding Program
Ok 20 INPUT "PRINCIPAL";P
Ok 30 INPUT "YEARLY INTEREST RATE (IN %)";R
Ok 40 INPUT "HOW MANY MONTHS";M
Ok 50 PRINT
Ok 60 ? "MONTH","PRINCIPAL","INTEREST","P + I"
Ok 65 ?
Ok 70 FOR K=1 to M
Ok 80 I=(P*(R/100))/12
Ok 90 ? K,P,I,P+I
Ok 100 P=P+I
Ok 110 NEXT K
Ok 120 END
Ok RUN
PRINCIPAL? 5000
YEARLY INTEREST RATE (IN %)? 12
HOW MANY MONTHS? 6

MONTH         PRINCIPAL     INTEREST      P + I

 1             5000          50            5050
 2             5050          50.5          5100.5
 3             5100.5        51.005        5151.51
 4             5151.51       51.515        5223.03
 5             5203.03       52.0302       5255.06
 6             5255.06       52.5505       5307.61
```

There are programming methods to round numbers and print only two digits following the decimal point. These techniques are beyond the scope of this tutorial. Here is how the program does the interest report:

1) Lines 20, 30, and 40 set the variable values by user input.
2) Line 60 prints the heading.
3) Line 70 sets up the FOR/NEXT loop to the number of months (M).
4) Line 80 is the interest formula.
5) Line 90 prints the values for each month.
6) Line 100 adds the last interest calculated to the principal.
7) Line 110 returns the program to line 70 until K=M.

Make sure you saved INTRATE and then use OLD to bring our old friend, program CALAVG, into working storage. List it. CALAVG prints the average of any three numbers. What if we want to find the average of 100 numbers? If we use the method in CALAVG, the program would be very long. Can you write a program to input 100 numbers and print the average using a FOR/NEXT loop? Try this without looking at the solution.

```
Ok NEW
Ok 10 FOR C=1 to 100
Ok 20 INPUT N
Ok 30 SUM=SUM+N
Ok 40 NEXT C
Ok 50 PRINT "THE AVERAGE IS";SUM/100
Ok 60 END
```

Here is an explanation of the preceding example:

1) Line 10 sets up the FOR/NEXT loop to input one hundred numbers.
2) Line 20 inputs the numbers.
3) Line 30 keeps a running total of the numbers input.
4) Line 40 restarts the FOR/NEXT loop until C=100.
5) Line 50 computes and prints the average.

We will be using the decision and looping statements in some of the coming example programs. If you are uncertain about the things you learned in this section, review the explanations and examples.

6.3 Exercises

1) Use a FOR/NEXT loop to print the sum of all the even numbers from 1 through 1000.

2) Use nested FOR/NEXT loops to print all combinations of the numbers 1, 2 and 3. The start of your output will be

```
111
112
113
121
122
123
131
|||
```

3) Write a Personal BASIC program that inputs a series of numbers (any numbers). Signal the end of data with a 999. Compute and print the largest number, the smallest number, and the number of numbers input. Use this output format:

```
THE SMALLEST NUMBER IS XX
THE LARGEST NUMBER IS XXX
 XX NUMBERS WERE INPUT
```

Hint: Use the first number input as the largest and smallest number so far.

End of Section 6

Section 7
Working with Words and Letters

7.1 What are Strings?

As we learned in Section 3, a string is a series of characters. This sentence is a string. The number 100 is a string three characters long. The words One Hundred are also a string, eleven characters long.

The variable name for a string follows the same naming rules as other variables, except that the name of a string variable is followed by a dollar sign, $. These are examples of string variable names:

```
X$          YOUNGER$          T23$          NAME$
```

We can set string variable NAME$ equal to the characters "Mr. Harry Frankenstein" with this statement:

```
NAME$ = "Mr. Harry Frankenstein"
```

The characters in a string must be enclosed in quotation marks. The quotation marks are not a part of the string.

String variables can be printed just like other variables. The statement, PRINT NAME$, gives this output:

```
Mr. Harry Frankenstein
```

String variables are time savers if you are printing long words or names. Several Personal BASIC statements control the manipulation of strings.

7.2 String Statements

7.2.1 LEN

LEN tells you how many characters are contained in a string. The count includes spaces. You must put parentheses around the string. This statement:

```
PRINT LEN("Mr. Harry Frankenstein")
```

or this statement:

```
PRINT LEN(NAME$)
```

prints the number of characters and spaces in the string, 22.

There are ways to print only part of a string variable. The statements used are LEFT$, RIGHT$, and MID$.

7.2.2 LEFT$, RIGHT$, MID$

We can print the first nine characters of NAME$ by using LEFT$. The statement to do this is

```
PRINT LEFT$(NAME$,9)
```

and the print output is

```
Mr. Harry
```

RIGHT$ works just like LEFT$, except the characters printed are counted from the right side. This statement

```
PRINT RIGHT$(NAME$,5)
```

produces this output:

```
stein
```

MID$ prints characters starting anywhere in the string, instead of at the left or right side. This statement

```
PRINT MID$(NAME$,11)
```

prints this output:

```
Frankenstein
```

The F in Frankenstein is the 11th character in the string. MID$ printed the string, starting at character 11. MID$ can also print a specified number of characters starting at any location. This statement

```
PRINT MID$(NAME$,13,3)
```

gives you this output:

```
ank
```

The first number after NAME$ is the position in the string where printing starts. The second number is the number of characters to be printed.

Here is a program that uses some of the new statements we just learned. Write what you think the output will be and then type the program and RUN it to see if you were right.

```
Ok NEW NAMES
Ok 50 NAMES$="BOBMAEANNSUELOU"
Ok 60 FOR C=1 TO LEN(NAMES$) STEP 3
Ok 70 PRINT MID$(NAMES$,C,3)
Ok 80 NEXT C
Ok 90 END
```

The program works this way:

1) Line 50 sets string variable NAMES$.

2) Line 60 starts a FOR/NEXT loop for C, beginning with C=1 and incrementing C by 3.

3) Line 70 prints three characters from the string, starting at C.

4) Line 80 continues the FOR/NEXT loop until C=15, the value of LEN(NAMES$).

This program might look a little different to you from the previous examples, but you have learned all of the statements and variations in the program. You should have obtained this output:

```
BOB
MAE
ANN
SUE
LOU
```

7.2.3 VAL

You can use numbers as strings, but they cannot be used for arithmetic computations. Study these examples.

```
Ok NEW CON
Ok 30 N$="45"
Ok 40 RT$="672"
Ok 50 PRINT N$+RT$
Ok 60 END
Ok RUN
45672
```

In the preceding example, program CON places the two strings together end-to-end. This is a useful feature called concatenation, but the numbers in the string are not added mathematically. Notice what happens in the following example:

```
Ok NEW
Ok 100 CASH$="50.50"
Ok 110 PRINT CASH$ + 70.55
                     ^

Types of values do not match
```

Line 110 produces an error message because the number 70.55 is not a string.

The VAL statement changes the numbers in strings to values that can be used mathematically. VAL gives you the numeric value of a string. The first nonblank character of the string must be +, -, &, ., or a digit. The string can have leading blank characters. Any other first nonblank character produces a VAL result of zero.

Look at the last example. If we change it by adding VAL, the result is

```
100 CASH$="50.50"
110 PRINT VAL(CASH$)+70.55
```

The correct total, 121.05, is printed. The string variable name must be in parentheses in the VAL statement. You can set a variable using the value of a string variable, as in the following example:

```
CAR = VAL(AMOUNT$) + CQ7 + 49.67
```

Another statement, STR$, does the reverse of VAL.

7.2.4 STR$

STR$ takes a number and converts it into a string, the opposite of VAL. In this example, the string variable AMOUNT$ is set to 45.38, the value of variable A.

```
Ok NEW
Ok 60 A = 45.38
Ok 70 AMOUNT$ = STR$(A)
Ok 80 PRINT AMOUNT$
Ok RUN
 45.38
```

STR$ could be used to determine the number of digits in a number. Here is the method:

```
Ok NEW
Ok 100 INPUT NUMBER
Ok 110 N$ = STR$(NUMBER)
Ok 120 PRINT LEN(N$)-1
Ok RUN
? 4867
 4
```

The variable "NUMBER" is input. STR$ converts the numbers in variable "NUMBER" to string variable N$. LEN prints the total number of digits in N$, orginally input as "NUMBER." The 1 was subtracted from LEN(N$) because a space is included for the sign.

7.3 Comparing and Joining Strings

Strings can be compared like numbers and joined together end to end, to form a longer string.

7.3.1 Comparing Strings

String variables are compared to each other using the same operations as numbers:

Table 7-1. Operators for String Comparisons

Symbol	Meaning
=	equals
<>	does not equal
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to

Strings are compared by comparing the numerical ASCII codes of the characters in the string. The ASCII code is the numbers used by most computers to represent characters. The ASCII codes are listed in the Personal BASIC Language Reference Manual, Appendix B.

If one string has fewer characters than the other, it is the smaller. Blanks are considered characters to be compared. Here are some examples of strings when their values are compared:

```
"RED LIPS" = "RED LIPS"
"SWEET " > "SWEET"
"rjm" > "RJM"
"ANN" < "ANNE"
"AA" < "AB"
DATE$ < "3/3/83"     (if DATE$ = "2/2/83")
"XYZ" <> "xyz"
```

Here is a program named COLORS that uses string comparisons to convert English color names to French. Type in the program and run it with names of colors. A sample run is included. Save the program. Program COLORS is also a good example of the use of READ/DATA and RESTORE.

```
Ok NEW COLORS
Ok 40 REM ENGLISH TO FRENCH COLOR CONVERSION
Ok 50 INPUT "ENTER A COLOR";C$
Ok 60 READ E$,F$
Ok 70 IF E$ = "END" THEN PRINT "TRY ANOTHER COLOR":GOTO 100
Ok 80 IF E$<>C$ THEN GOTO 60
Ok 90 PRINT C$; " IN FRENCH IS ";F$
Ok 95 PRINT
Ok 100 RESTORE
Ok 110 GOTO 50
Ok 120 DATA BLACK,NOIR,BROWN,BRUN,GREEN,VERT
Ok 125 DATA BLUE,BLEU,GREEN,VERT,RED,ROUGE
Ok 130 DATA ORANGE,ORANGE,YELLOW,JAUNE,WHITE,BLANC,END,END
Ok 140 END
Ok RUN
ENTER A COLOR? RED
RED IN FRENCH IS ROUGE

ENTER A COLOR? PINK
TRY ANOTHER COLOR
```

Do you understand how the program works? Here is an explanation:

1) Line 50 An English color name is input.

2) Line 60 The English and French colors are read from DATA, beginning with line 120.

3) Line 70 If the program reaches END, there is no English color by that name in either DATA line.

4) Line 80 If the specified color is not found, the program branches back to read another set of colors. <> means "not equal to."

5) Line 90 A match was found and the answer printed.

6) Line 100 RESTORE makes the READ statement start with the first DATA value (BLACK) if the color was not in DATA or if a match was found.

7) Line 110 Returns the program to the READ statement.

8) Lines 120, 125 and 130 DATA statements.

This programming method could be used to design an educational game to learn the state capitals.

7.3.2 Joining Strings

Joining strings together is called concatenation. With concatenation, you can make strings up to 255 characters long. Return to the Ok prompt with a CTRL-C and type and run program JOIN.

```
Ok NEW JOIN
Ok 10 B$ = "OVER"
Ok 20 C$ = "EASY"
Ok 30 PRINT B$ + " AND OUT"
Ok 40 PRINT B$ + C$
Ok 50 PRINT C$ + B$
Ok 60 END
Ok RUN
OVER AND OUT ◄──────── "AND OUT" added to the end of B$(OVER)
OVEREASY ◄──────────── C$ (EASY) added to the end of B$(OVER)
EASYOVER ◄──────────── B$ (OVER) added to the end of C$(EASY)
```

Here is an interesting program called REVERSE that reverses the order of characters in a string. It illustrates the use of concatenation, MID$, and uses a FOR/NEXT loop. Study the program, type and run it with different inputs.

```
Ok NEW REVERSE
Ok 10 REM Reverses order of input string
Ok 20 INPUT "ENTER STRING TO BE REVERSED ",STRNG$
Ok 30 LENGTH=LEN(STRNG$)
Ok 40 FOR X=LENGTH TO 1 STEP -1
Ok 50 CHARACTER$=MID$(STRNG$,X,1)
Ok 60 NEWSTRING$=NEWSTRING$+CHARACTER$
Ok 70 PRINT NEWSTRING$, CHARACTER$
Ok 80 NEXT X
Ok 85 PRINT
Ok 90 PRINT "REVERSED STRING IS ";NEWSTRING$
Ok 100 CLEAR
Ok 110 INPUT "MORE";M$
Ok 120 IF M$="YES" THEN GOTO 20
Ok 130 END
Ok RUN
ENTER STRING TO BE REVERSED ARTICHOKE
E              E
EK             K
EKO            O
EKOH           H
EKOHC          C
EKOHCI         I
EKOHCIT        T
EKOHCITR       R
EKOHCITRA      A

REVERSED STRING IS EKOHCITRA
MORE? NO
Ok
```

Line 70 was included to give you a picture of how the reversed string, NEWSTRING$ (left column) is developed as each character (right column) is added to it.

This is what happens in program REVERSE:

1) Line 20 The string to be reversed is input.

2) Line 30 The length of the input string is determined.

3) Line 40 The FOR/NEXT loop is initialized.

4) Line 50 A character of the input string is obtained, starting with the rightmost.

5) Line 60 The character obtained in line 50 is concatenated to the reversed string.

6) Line 70 PRINT statement to illustrate method.

7) Line 80 End of FOR/NEXT loop.

8) Line 90 PRINT statement for printing reversed string.

9) Line 100 This is a new statement for you. CLEAR sets all variables to zero or null. The program is unchanged. Without CLEAR, string variable NEWSTRING$ would contain the string entered from the previous run. Run REVERSE without CLEAR a few times and you will understand why CLEAR is necessary.

10) Lines 110 and 120 User asked if rerun is desired.

Other statements dealing with strings are described in the Personal BASIC Language Reference Manual. They are MKI$, OCT$, SPACE$, and STRING$.

7.4 Exercises

1) Write a Personal BASIC program that counts and prints the number of B's (or any other character) in any string. Hint: Review program NAMES.

2) Input the string "ONE TWO THREE" and reverse the words to produce this output: THREE TWO ONE.

End of Section 7

Section 8
Personal BASIC Functions

We stopped at the beginning of Section 6 to review the statements and commands that you should be familiar with. Well, it's time to do this again. The statements added in Sections 6 and 7 are marked with an asterisk (*).

Statements	Commands
* CLEAR	DELETE
END	DIR
* FOR/NEXT	ERA
GOTO	LIST
* IF/THEN/ELSE	NAME
* LEFT$	NEW
* LEN	OLD
LET	RENUM
* MID$	REPLACE
PRINT	RUN
READ/DATA	SAVE
REM	
RESTORE	
* RIGHT$	
* STR$	
* VAL	
* WHILE/WEND	

Review these statements and commands if necessary. Continue entering and running the examples. Keep the Personal BASIC Language Reference Manual handy.

8.1 Definition of Functions

Many computing tasks are required on a regular basis. Personal BASIC includes many functions to automatically do these tasks. Functions are like formulas that manipulate numbers and strings.

Personal BASIC provides a full set of preprogrammed built-in functions. With these built-in functions and user-defined functions, you can perform complicated operations with minimum difficulty.

Only the most-used functions are described in this tutorial. All of the functions are described in detail in the Personal BASIC Language Reference Manual.

8.2 Personal BASIC Built-in Functions

8.2.1 SQR(X)

The format for all mathematical functions is FUNCTION(X). X is called the argument of the function. Think of the argument as what the function works on. Function SQR gives the square root of the argument. SQR output looks like this:

```
Ok PRINT SQR(36)
 6
Ok PRINT SQR(10)
 3.16228
```

The argument of SQR can be as complicated as you like, but it cannot be a negative number. For example, this argument is permitted:

```
PRINT SQR(C+(U^3)+66)
```

8.2.2 INT(X)

The result of INT is the integer part of X. Integer is another way of saying whole number. This is how INT takes the integer of numbers:

```
Ok PRINT INT(45.678)
 45
Ok PRINT INT(0.418)
 0
Ok PRINT INT(6)
 6
Ok PRINT INT(-.647)
-1
Ok PRINT INT(-8)
-8
```

The integer part of a number is the first integer less than the number. INT does not round the number.

8.2.3 SGN(X)

SGN tells you the sign of the argument. If the argument is positive, the SGN output is 1; if the argument is negative, the output is -1. If the argument is zero, the SGN output is 0. Look at these examples:

```
Ok PRINT SGN(56.78)
 1
Ok PRINT SGN(-45.6)
-1
Ok PRINT SGN(0)
 0
```

8.2.4 ABS(X)

ABS removes the sign from the argument and leaves what is left. Here is ABS at work:

```
Ok PRINT ABS(78.23)
 78.23
Ok PRINT ABS(-537.8)
 537.8
Ok PRINT ABS(0)
 0
```

8.2.5 RND(X)

You can have fun with RND, especially if you are a gambler. RND produces a number generated at random between 0 and just under 1. Random numbers are used in simulations and in generating gambling situations, like rolling dice. If X is positive or not included, RND generates the same sequence of random numbers. If X is zero, the last number generated is repeated.

The numbers obtained from RND look like this:

```
.7349128     .1429684     .5980341     .9854673
```

If you want to see a lot of random numbers, enter the next program and run it. CTRL-C stops the output, <cr> restarts it. Return to the Ok prompt with CTRL-C given while the output is stopped.

```
Ok NEW
Ok 20 PRINT RND(8);
Ok 30 GOTO 20
Ok 40 END
```

If you want random numbers other than the decimal numbers generated, something must be done to change the output from RND. If you need a random number from 1 through 10, multiply the RND output by 10, take the INT and then add 1. Any other required number may be done in a similiar way. This program produces a random number from 1 through ten. Remember, the argument of RND can be any positive number.

```
Ok NEW
Ok 20 N=RND(4)
Ok 30 N=INT (N*10) + 1
Ok 40 PRINT N
Ok 50 END
```

In the example, assume that RND generates the number, .5129042.

```
N * 10 is 5.129942
INT (N * 10) is 5
5 + 1 is 6
```

The addition of 1 is necessary because the random numbers must be from 1 through 10. Program RAN generates 10 random numbers ranging from 1 through 100. The RANDOMIZE statement produces a different set of numbers each time program RAN is run. RANDOMIZE asks for a number seed which it uses to generate the random numbers. Enter any number in the range specified. Program RAN without the RANDOMIZE statement will produce the same set of numbers for each run.

Here is the program with sample runs. Run it a few times and notice the variations in the numbers generated.

```
Ok NEW RAN
Ok 290 RANDOMIZE
Ok 300 FOR I = 1 to 10
Ok 310 PRINT INT(RND*100) + 1;
Ok 320 NEXT I
Ok 330 END
Ok RUN
Random number seed (-32768 to +32767)? 6
 84  80  40  59  14  40  28  76  53  70
Ok
```

Try program DICE. DICE simulates throwing a pair of dice and prints the result of each throw.

```
Ok NEW DICE
Ok 10 B=INT(6*RND(1)) + 1
Ok 20 PRINT "Black die rolls ";B
Ok 30 W=INT(6*RND(1)) + 1
Ok 40 PRINT "White die rolls ";W
Ok 50 PRINT
Ok 60 INPUT "Roll again ";Q$
Ok 70 IF Q$ = "YES" THEN GOTO 10
Ok 80 IF Q$ = "NO" THEN END
Ok 90 PRINT "Please answer YES or NO"
Ok 100 GOTO 60
Ok 110 END
Ok RUN
Black die rolls  6     ───────────┐  Program output should
White die rolls  1                │◄ look like this.
                                  │
Roll again ?           ───────────┘  GOOD LUCK !
```

If you are not sure how the formula in lines 10 and 30 works, try it out by hand with a few numbers. Included in the program is the option of continuing more dice throws or stopping. Without this option, you must type RUN to simulate another throw. Here is the program logic:

1) Lines 10 and 30 simulate the dice rolling
2) Lines 20 and 40 print the results
3) Lines 60 through 100 allow for another throw

When you finish throwing dice, we will continue our discussion of Personal BASIC built-in functions.

Personal BASIC contains many mathematical functions not included in our tutorial. They are listed here for your information. See the Personal BASIC Language Reference Manual for details.

Table 8-1. Personal BASIC Math Functions

Function	Description
ATN(X)	Gives the arctangent of X in radians
CDBL(X)	Converts X to a double precision number
CINT(X)	Converts X to an integer by rounding
COS(X)	Gives the cosine of X in radians
CSNG(X)	Converts X to a single precision number
EXP(X)	Gives e to the power of X
FIX(X)	Gives the truncated integer part of X
SIN(X)	Gives the sine of X in radians
TAN(X)	Gives the tangent of X in radians

8.3 User-defined Functions

Personal BASIC lets you to create do-it-yourself functions with DEF statements. You define the function and then the function is used just like the functions we learned, such as SQR or INT. A DEF statement has the following format:

DEF (FNname)(argument) = definition

- name is a valid variable name.
- argument is a variable name in the definition that is replaced by a value when the function is used.
- definition is an expression that describes the function.

Program TEMPCON creates a DEF function to convert Celsius temperature to Fahrenheit.

```
Ok NEW TEMPCON
Ok 5 REM Celsius to Fahrenheit conversion
Ok 10 DEF FNCTOF(C)=C*1.8+32
Ok 20 INPUT "CELSIUS ";C
Ok 30 PRINT C;"DEGREES CELSIUS IS" FNCTOF(C) "DEGREES FAHRENHEIT."
Ok 40 END
Ok RUN
CELSIUS ? 45
 45 DEGREES CELSIUS IS 113 DEGREES FAHRENHEIT
```

Here is an explanation of program TEMPCON:

1) Line 10 shows the DEF FN statement that defines a function to convert Celsius to Fahrenheit. CTOF is the variable name, C is the argument, and C*1.8+32 is the definition.

2) Line 20 is a normal INPUT statement asking for C.

3) Line 30 is the PRINT statement that calls function FNCTOF.

DEF statements can have more than one variable, as seen in program CARPET. The program computes the total cost of carpeting a room, including five percent for trimming. Type the program and run it with various inputs.

```
Ok NEW CARPET
Ok 10 INPUT "ENTER LENGTH AND WIDTH IN FEET ";L,W
Ok 20 INPUT "ENTER COST IN DOLLARS PER YARD ";C
Ok 30 DEF FNCOST(L,W,C) = 1.05*(L/3*W/3*C)
Ok 40 PRINT
Ok 50 PRINT "TOTAL COST IS $";FNCOST(L,W,C)
Ok 60 INPUT "DO YOU WANT ANOTHER ESTIMATE ";Q$
Ok 70 IF Q$ = "YES" THEN GOTO 10
Ok 80 END
Ok RUN
ENTER LENGTH AND WIDTH IN FEET ? 25,18.5
ENTER COST IN DOLLARS PER YARD ? 15.95

TOTAL COST IS $ 860.635
```

In program CARPET, the function was defined in line 30 and the total cost printed in line 50.

Notice that the input was requested in the most used units, feet and dollars per square yard. We usually measure in feet and price a carpet per square yard. Program users should be able to enter familiar values into programs. Let the program handle any necessary conversions. In program CARPET, feet were converted into yards in the function formula in line 30.

In line 70, we assumed that anything other than "YES" meant that the user did not want to continue.

A DEF FN statement can be placed anywhere in a program. The function only needs to be defined once.

End of Section 8

Section 9
Working with Groups of Data — Arrays

Who needs ARRAYs? You need arrays if you want to work with more than a few numbers or characters at a time. Arrays sometimes scare beginners, but if you follow the examples and text, you will find that they are no more difficult than the material already covered.

In the programming examples in previous sections, only a few numbers were used to illustrate the various statements and commands. In the real world, programs usually operate on many numbers related in some way. Consider this problem:

A teacher wants to compute grade averages for three tests and print the results for 100 students. READ/DATA statements are impractical in this example because of their length, and the difficulty of assigning and keeping records on 100 variables. This program could do the job, but has several disadvantages:

```
10 NEW
20 PRINT "NAME","GRADE AVERAGE"
30 PRINT:PRINT
40 FOR S=1 to 100
50 INPUT "Enter name and grades";N$,A,B,C
60 PRINT N$,(A+B+C)/3
70 PRINT
80 NEXT S
90 END
```

The program gives us the required output, but what if you want to change a name, a grade, or add a student? The tedious job of inputting 100 names and grades has to be repeated. Arrays with subscripted variables let us handle large groups of data comparatively easily.

9.1 Subscripted Variables

A group of data is called an array. Each item in an array must be defined separately. Subscripts are used to do this. This is an array:

```
T(0) =  4
T(1) = 12
T(2) = 32
T(3) = 20
T(4) = 18
T(5) =  6
T(6) = 26
```

The name of this array is T. The size of array T is 7, since it has seven elements (numbers). The numbers 4, 12, 32, 20, 18, 6, and 26 are the elements in array T. The numbers after the array name are subscripts. Subscripts normally start with zero. An array name follows the same naming rules as other variables. Array A and variable A are not the same and both can be used in one program.

Arrays are described like this when reading them in a program: the fifth element in array T is called, "T sub five". Array T is called a one-dimensional array, because only one number (or subscript) is required to locate any element.

Arrays can have more than one dimension. The following array M is a two-dimensional array:

ARRAY M (R, C)

```
M(0,0) =  6     M(0,1) = 21     M(0,2) =  8
M(1,0) = 13     M(1,1) = 34     M(1,2) = 17
```

Array M has six elements. The first subscript gives the row number and the second, the column number. M(1,2) is read as "M sub one two." Array M looks like this in row and column form:

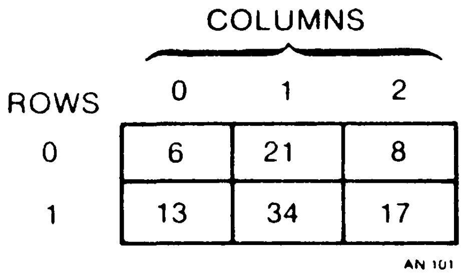


Figure 9-1. Two-dimensional Array

Use subscripted variables like other variables in BASIC statements. The statement PRINT M(1,2) prints the number 17. PRINT M(1,0) + M(0,2) prints the number 21. Looking at array T, PRINT T(3) + T(6) prints the sum of 20 and 26, 46.

FOR/NEXT statements are sometimes used to set up and load arrays. Program ARRAY sets up an eight by five array and then loads 9s into each element.

```
Ok NEW ARRAY
Ok 10 DIM B(7,4)
Ok 20 FOR R = 0 to 7
Ok 30 FOR C = 0 to 4
Ok 40 B(R,C) = 9
Ok 50 NEXT C
Ok 60 NEXT R
Ok 70 END
```

If you are unsure how the nested FOR/NEXT loops work, refresh your memory by reviewing Section 6.2. The row and column values are reset during each FOR/NEXT loop and a 9 is inserted into each element. The substitutions in line 40 start like this: B(0,0), B(0,1), B(0,2), B(0,3), B(0,4), B(1,0), and so on. B(7,4) is the last element.

If you want to make sure that array B was loaded with 9s, print some of the elements. For example, PRINT B(3,3); PRINT B(5,1).

9.2 Array statements

9.2.1 DIM

Personal BASIC must know the size of an array so that enough memory space can be reserved. The DIM statement is required before an array can be specified. If an array variable name is used and no DIM statement exists with that name, the maximum number of subscripts possible is 11. The subscripts are numbered zero through 10. If the option base is 1 (see the option base description below), 10 subscripts is the maximum in the range 1 through 10. If a subscript over the maximum is used without a DIM statement, the error message, "Subscript refers to element outside the array" is printed. Here is an example of a DIM (dimension) statement:

```
100 DIM B(4,19), Y(79), Z(33)
```

Three arrays are dimensioned in line 100. B is a two-dimensional array with five rows and twenty columns. Y is a one-dimensional array with eighty elements, and Z is a one-dimensional array with 34 elements. DIM statements should be first in your program so that you will know easily how much space your arrays are using. It is good practice to use DIM for small arrays, just to make your bookkeeping easier.

9.2.2 OPTION BASE

The minimum value for a subscript is zero unless changed by the OPTION BASE statement. This statement changes the minimum value from zero to 1. When this statement:

```
OPTION BASE 1
```

is used, the lowest value for an array subscript is 1. The option base can be redefined any number of times.

9.2.3 ERASE

ERASE does just what you might guess--it erases the specified array or arrays from your program. The statement:

```
560 ERASE RT, G45
```

erases all traces of arrays RT and G45. An array can be redimensioned by DIM after being erased. Arrays cannot be redimensioned unless ERASE is first used. If this is attempted, error message "You defined an array more than once" is printed.

End of Section 9

Section 10
Disk Input and Output — File Processing

This section is a brief introduction to disk input/output and files using Personal BASIC. For a complete explanation of files and how to use them, refer to the Personal BASIC Language Reference Manual.

10.1 File Concepts

Computers would not be very useful if they only used their internal memory (RAM) for storage. The programs and data in RAM are lost when the computer is switched off. The amount of storage in RAM is limited compared to the storage available in permanent storage (disk and tape). We have been using the SAVE and REPLACE commands to move programs from temporary storage (RAM) to permanent storage.

Permanent storage of data and the manipulation of that data is done by various BASIC commands and statements that pertain to files. The creation and manipulation of files is the purpose of most data processing. A file is a series of records relating to the same subject. The records of all the inventory items make up a file; the records containing the names and addresses of employees could be a file.

Files are used to store numeric data and string data into a permanent place for use at any time. The data in files can be updated, inspected, deleted, or sorted. Files can be very small--your telephone list, for example--or very large--such as the government's list of Social Security recipients.

We have been using BASIC statements like INPUT, LET, and READ/DATA to enter program data. Using files, you can enter and store data using one program and then access the same data with a different program. Programs can read from or write onto files. Imagine an inventory file. In a business, programs such as these might use this file:

- Inventory Control
- Accounting
- Forecasting
- Parts Lists
- Purchasing
- Scheduling

There are two types of files that can be created and used by a Personal BASIC program, sequential and random access.

10.2 Sequential Files

Sequential files are easier to use than random access files, but they have fewer features and are slower. Sequential files store information as a continuous series of data. For example,

```
BOB SMITH,345-8496/ANN JONES,563-890/JERRY WHITE,540-7436/
```

Sequential files are stored and searched on the basis of a key item in each record. The records are read, one at a time, until the desired record is found. The key of the record is not related to its location in the file. If we want to search for an employee record with an employee number key of 4267, the computer has to search the sequential file from the beginning. The key of each record is compared to 4267 until the record for employee 4267 is found or the end-of-file is reached.

The Personal BASIC statements used with sequential files are

```
OPEN        WRITE#          INPUT#

PRINT#      PRINT#USING     LINE INPUT#

CLOSE       EOF             LOC
```

These are the steps necessary to create a sequential file and access the data in the file:

1) OPEN the file for output.
2) Write data to the file using PRINT#, WRITE#, or PRINT#USING.
3) CLOSE the file and then OPEN it to access the data in the file.
4) Read data into your program with INPUT# or LINE INPUT#.

Program FILES illustrates these four steps. A file named "DATA" is created and string variables are written to the file. The file is closed and then reopened to read the string variables into the program. The values of the string variables are printed.

```
Ok NEW FILES
Ok 190 A$="APPLE":B$="BEAN":C$="CHERRY"
Ok 200 OPEN "O",#1,"DATA"                   Step 1
Ok 210 WRITE#1,A$,B$,C$                     Step 2
Ok 220 CLOSE #1                             Step 3
Ok 230 OPEN "I",#1,"DATA"                     "
Ok 240 INPUT#1,X$,Y$,Z$                     Step 4
Ok 250 PRINT X$,Y$,Z$
Ok 260 END
Ok RUN
APPLE        BEAN          CHERRY
Ok
```

Here is another example. Program PARTS creates an inventory list from information input at the terminal.

```
Ok NEW PARTS
Ok 10 OPEN "O",#1,"PARTS"
Ok 20 INPUT "NAME";NAME$
Ok 30 IF NAME$="HALT" THEN END
Ok 40 INPUT "NUMBER";NO$
Ok 50 INPUT "QUANTITY";Q
Ok 60 WRITE#1,NAME$,NO$,Q$
Ok 70 PRINT:GOTO 20
Ok 80 END
Ok RUN
NAME? T CLAMP
NUMBER? 36932N
QUANTITY? 45

NAME? TORSION BOLT
NUMBER? 68154AD
QUANTITY? 200

NAME? RETRACTOR
NUMBER? 31930AT
QUANTITY? 95

NAME? LOCK NUT
NUMBER? 84613W
QUANTITY? 2000

NAME? HALT
Ok
```

The END statement closes all open files if no CLOSE statement is used in the program.

Now we have a sequential file created and closed according to the first three steps. Step 4 is necessary to use the file in a program. Program SEARCH reads file PARTS and prints all part numbers with quantities under 100.

```
Ok NEW SEARCH
Ok 5 REM Print part numbers when quantity is less than 100
Ok 10 OPEN "I",#1,"PARTS"
Ok 20 IF EOF(1) THEN END
Ok 30 INPUT #1,NA$,NO$,Q
Ok 40 IF Q<100 THEN PRINT NO$
Ok 50 GOTO 20
Ok 60 END
Ok RUN
36932N
31930AT
```

Statement 20 checks for the end-of-file. An error statement "You Have Reached End-of-File" is printed without this statement. An end-of-file mark is added to the end of a sequential file when the file is closed.

Data can be added to sequential files, but another variation of the OPEN command is used. If a sequential file is opened for output in the "O" mode, the contents are destroyed. When you want to add data to an existing sequential file, you must OPEN the file for APPEND. The explanation of the OPEN statement in the Personal BASIC Language Reference Manual explains this in more detail.

Now that we have seen some examples of sequential files, we can examine how random files work and how they differ from sequential files.

10.3 Random Files

Random files should be used if frequent changes to the file are necessary or if data in the file must be accessed in minimum time. Some examples of files that have to be random are

- Airline Reservations
- Credit Card Data
- Instant Bank Teller
- On-line Inventory

Random file records each have an assigned number. Each record can be compared to a small sequential file. Records are found directly or randomly without reading through the entire file. Records can be changed easily without the involved procedures required by sequential files. Random files use less space than sequential files because they are stored in a different format. The statements used with random files are

OPEN	FIELD	LSET/RSET
GET	PUT	CLOSE
MKI$	MKS$	MKD$
CVI	CVS	CVD
LOC	LOF	

These are the steps necessary to create a random file and access the data in the file. The random buffer is an intermediate storage area between RAM and the random file.

1) OPEN the file for random access (R) mode. The next program, MAIL, creates a random file named MLIST, and specifies a record length of 55 bytes (characters).

2) Use the FIELD statement to allocate space in the random buffer for the variables to be written into the random file.

3) Use LSET to move the data into the random buffer set up by FIELD. Numeric values must be converted to strings prior to being placed in the buffer.

4) Use PUT to write the data from the buffer to the disk.

Program MAIL illustrates these four steps. A mailing list andom file of name, address, and city/state is created from erminal inputs in lines 330, 340, and 350. The PUT statement in ine 390 writes a record to the file each time it is run. The two igit code input in line 330 becomes the record number.

```
                              random file  Aufz.-Länge : 55 byte
Ok NEW MAIL                        ↑
Ok 300 OPEN "R",#1,"MLIST",55                                Step 1
Ok 310 FIELD #1,20 AS N$, 20 AS A$, 15 AS CS$                Step 2 Zuweisung d. erforderlichen
Ok 320 INPUT "TWO DIGIT CODE";C                              Record No. Platzes
Ok 325 IF C=99 THEN END                                            im Random
Ok 330 LINE INPUT "NAME";X$                                        Puffer
Ok 340 LINE INPUT "ADDRESS";Y$
Ok 350 LINE INPUT "CITY/STATE";Z$
Ok 360 LSET N$=X$                                     Einlesen der Daten in den
Ok 370 LSET A$=Y$                                     ◄──────── Step 3
Ok 380 LSET CS$=Z$                                    Random Puffer
Ok 390 PUT #1,C     -> Daten werden vom Puffer auf die    Step 4
Ok 400 PRINT           Diskette geschrieben
Ok 410 GOTO 320
Ok 420 END
Ok RUN
TWO DIGIT CODE? 11
NAME? BUFFORD BLIMP
ADDRESS? 68000 CHIP ROAD
CITY/STATE? BIG HORN, WYOMING

TWO DIGIT CODE? 23
NAME? PRIMROSE PLUM
ADDRESS? 4289 ROLLING DRIVE
CITY/STATE? TWO DOT, MONTANA

TWO DIGIT CODE? 18
NAME? CONSTANCE CORTISONE
ADDRESS? 920 HOSPITAL STREET
CITY/STATE?  CANOE, MICHIGAN

TWO DIGIT CODE? 99
Ok
```

These are the steps necessary to access the random file just created:

1) OPEN the file in the R mode.

2) Use FIELD to allocate space in the random buffer for the variables to be read from the file.

3) Use the GET statement to move the record you want to read into the random buffer.

4) The data in the buffer is now available to the program. Numeric values must be converted back to numbers using CVI, CVS, or CVD.

The next program, RSEARCH, accesses the random file, MLIST, created in program MAIL. The two digit code is entered and the record input with that code is read from the file and printed. This is the basic idea behind information retrieval systems.

```
Ok NEW RSEARCH
Ok 50 OPEN "R",#1,"MLIST",55                          Step 1
Ok 60 FIELD #1, 20 AS N$, 20 AS A$, 15 AS CS$         Step 2
Ok 70 INPUT "TWO DIGIT CODE";C                        Record No.
Ok 80 IF C=99 THEN END
Ok 90 GET #1, C    → Daten von File in Random Puffer   Step 3
Ok 95 PRINT
Ok 100 PRINT N$:PRINT A$:PRINT CS$:PRINT
Ok 110 GOTO 70
Ok 120 END
Ok RUN
TWO DIGIT CODE? 23

PRIMROSE PLUM
4289 ROLLING DRIVE
TWO DOT, MONTANA

TWO DIGIT CODE? 99
Ok
```

You might want to experiment with various inputs using the MAIL program and then do some searching with RSEARCH. As an optional activity, you could modify the programs to include the ZIP code.

End of Section 10

Section 11
Testing and Debugging Your Program

Bugs are not desirable in your home or in your program. Bugs are program errors. Newly written program usually have a bug or two, even if written by an experienced programmer. Debugging is the procedure used to find and correct program errors.

Personal BASIC checks for syntax errors as you enter each line of your program. This means that syntax errors will not slow you down when you run your program.

Once the program is debugged and running, you can begin program testing. Program testing is simple or complex, depending on how sure you want to be that the program will handle most of the data input to it.

11.1 Program Debugging

The best way to avoid program bugs is to plan your program in advance. Do not sit down at your keyboard and start keying in statements at a furious pace unless you have done some planning. Flow charts, even a written description of what your program should do, are better than no planning at all. The program inputs and outputs are vital and should be specified before programming begins.

Personal BASIC has a complete set of debugging tools to help you find program errors. Personal BASIC's Break Mode lets you run your program in step and various trace modes. From the Break Mode, you can restart your program run, list your program, run it one line at a time with the STEP command, trace any line number with the TRACE command, or obtain a list of line numbers as they run with TRON.

11.1.1 Break Mode

The Break Mode is a powerful debugging tool, and even beginners find it easy to use. The best way to learn how the Break Mode works is to use it.

Type program BUGS for use in the following examples and save it. Do you see something wrong? You are right if you noticed that BUGS will never stop when run. An endless loop is not acceptable in a program, but we use it to help explain the debugging commands.

```
Ok NEW BUGS
Ok 20 N = 5
Ok 30 FOR X = 1 to N
Ok 40 PRINT X,X^2,X^3
Ok 50 NEXT X
Ok 55 GOTO 20
Ok 60 END
```

Run program BUGS. While it is looping, press CRTL-C to enter the Break Mode. The program stops at some line number and displays this output:

```
Ok RUN
   |          |       |
   |          |       |
   |          |       |
-- Break   -- at line 50
Br
```

The line number can be any line in your program, depending on what line was being run when you pressed CTRL-C. The break prompt is indicated by the letters Br. Now you are in the Break Mode and can use the commands listed below. All of these commands except CONT can also be given after the Ok prompt. A CTRL-C given while in the Break Mode returns you to Personal BASIC and the Ok prompt.

11.1.2 STEP

STEP lets you step through your program, line by line, as each line is run. It shows you what the program is doing as it operates in slow motion. Run BUGS and enter Break Mode with CTRL-C. Type STEP after the Br prompt and press <cr>. Each time you press <cr>, the program prints the next line before it is run and stops. The next STEP prints the output of the line, if any, and the next line. If you stopped at line 50, STEP produces this output:

```
Ok RUN
   |                 |                  |
   |                 |                  |
   |                 |                  |
 3                  9                 27
 4                  16                64
 5                  25                125
-- Break -- at line 50
Br STEP
s  55       GOTO 20
Br
s  20       N = 5
Br
s  30       FOR X=1 to N
Br
```

Line 55 follows line 50 every five times through the loop, when N is equal to 5. Continue running the program by giving the command CONT after a Br prompt. Run BUGS a few times, break it with CTRL-C and use STEP until you are familiar with its operation. Return to the Ok prompt with CTRL-C given while in the Br prompt.

11.1.3 CONT

CONT means continue the program run. When you are in Break Mode, program execution continues with the CONT command. Just type CONT after a Br prompt. CONT continues running with the next line number, not from the beginning of the program. CONT is the only debugging command that cannot be used after the Ok prompt.

11.1.4 BREAK/UNBREAK

BREAK causes the program to stop at any line number just before running the line. The program line and any output are printed. The program resumes with <cr> and runs until the next line number with a break is reached. BREAK with no line number stops the program at every line number, much like STEP. The UNBREAK command with no line number specified removes all BREAKS. Insert a BREAK at line 55 in program BUGS and run BUGS. This is the output:

```
Ok BREAK 55
Ok RUN
 1            1            1
 2            4            8
 3            9            27
 4            16           64
 5            25           125
b   55 GOTO 20
Br
```

A <cr> restarts the program until it reaches line 55 again. BREAK is useful when you want to see what happened after a specific line runs. Return to the Ok prompt with CTRL-C and type UNBREAK to remove the break.

11.1.5 TRACE/UNTRACE

The TRACE command produces the same output as BREAK, but TRACE does not stop the program run. Each line traced is printed with the output, but the program continues running until it ends or until you enter Break Mode with a CTRL-C. TRACE output can be stopped with CTRL-C and restarted with <cr>. TRACE is turned off with the UNTRACE command. UNTRACE also turns TRON off.

11.1.6 TRON/TROFF

The TRON command prints each program line number in the order it is run. Program output is also printed. The line numbers print in square brackets, [50]. TRON is useful if you suspect your program is taking an incorrect path. It is also a good training aid in understanding how a program runs statements. Delete line 55 from program BUGS and run with the TRON command. This is your output:

```
Ok 55
Ok TRON
Ok RUN
[20][30][40] 1                    1                   1
[50][40] 2      4                 8
[50][40] 3      9                 27
[50][40] 4      16                64
[50][40] 5      25                125
[50][60]
Ok TROFF
```

The line numbers in square brackets printed in the same order they were run. The program output was printed just after each PRINT statement in line 40.

If your program is long and the TRON output extends more than one screen, press CTRL-C to stop the TRON output. Press <cr> to continue the TRON output. Use TROFF to turn off TRON for future program runs. UNTRACE and NEW also turn off TRON.

11.1.7 FOLLOW/UNFOLLOW

FOLLOW is especially useful if your program variables have changing values. FOLLOW tells you whenever a variable you specify has changed value and what the new value is. The FOLLOW output is variable name, line number, and value. FOLLOW is turned off by the command, UNFOLLOW.

We can use FOLLOW to follow the value of X in program BUGS. Put line 55 (GOTO 20) back into program BUGS and run BUGS. Stop the output with CTRL-C. Output can be resumed with <cr>. Depending on where you stopped, the output looks like this:

```
Ok 55 GOTO 20
Ok FOLLOW X
Ok RUN
[20][30]
Var X! =  1  At line 30          <-------- The first value of
[40] 1          1           1              variable X
[50]
Var X! =  2  At line 50          <-------- The   value   of   X
[40] 2          4           8              changes to 2
[50]
Var X! =  3  At line 50          <-------- X   changes   again
[40] 3          9           27             and again and ....
[50]
Var X! =  4  At line 50
-- Break -- at line 40
Br
```

The FOLLOW output includes the type of numeric variable. We discussed these types in Section 3.1. In the example, the numeric variable X has the data type of single precision, shown by an exclamation mark, !. Single precision is also indicated if the variable has no label following it, as in our examples.

Try FOLLOW with some of the other programs in your library. You will see that FOLLOW can be very valuable in finding bugs around variables.

<u>Some Things to Remember about Break Mode</u>

- END returns you to Personal BASIC and the Ok prompt.
- A STOP statement in your program puts you into Break Mode at the line number after the STOP line. A <cr> continues the program.
- LIST and RUN can be used from the Br prompt.
- LIST your program to see what lines are set for TRACE or BREAK. A t or b is printed to the left of each line number.
- All Break Mode commands except CONT can be given after the Ok prompt.

11.2 Program Testing

A program is really not completed until it has been tested. Some very large programs may never be completely free of program errors. For example, a specific path in a program is never reached until input X is made to the program. The program is tested, but input X is not a part of the test input, since this input might never happen. The program runs fine for six months and input X is given. The program enters the untested area and cannot handle X correctly.

It has been said that the only real problem in programming is getting the program to work correctly and then prove that it does. In programming, there are no small errors; even the lack of a period or hyphen could cause the entire program to fail. It is difficult to define an objective for program testing, except to eliminate all errors--sometimes an impossible task.

Explaining the psychology and science of program testing is best left to the numerous programming texts available at your book store. We will leave you with some hints that should help you test your programs. The more time you spend in program testing, the more confident you will be that the program is doing the task it was designed to accomplish.

- As you plan and write the program, jot down testing ideas.
- Test every branch in the program.
- Test inputs should contain a wide sampling of both legal and illegal inputs.
- If possible, use live or actual input samples.
- Let someone unfamiliar with the program use it. This also tests your documentation (do not forget that).
- When you remove a bug, run all the tests again, because sometimes a program change in one area affects another area.
- If everything looks fine in early testing, do not stop the test procedure.
- If you are about to give up your search for a bug, get up and walk around, eat, listen to music, watch the sky--anything but looking at your program. Then make a fresh start.

End of Section 11

Dear Reader,

Here we are at the end of this tutorial book on Personal BASIC. If you have been using your computer and doing the examples, you should have a good start in learning how to program in BASIC. Even though this is the end of the book, you have not arrived at the end of BASIC. There is more to learn.

Look at the other Personal BASIC capabilities in the Personal BASIC Language Reference Manual. Visit your local book stores, computer stores, and library. You will find many books on BASIC programming and programming in general. Some of the computer magazines offer excellent articles and columns on programming techniques.

We told you before that learning a programming language is like learning how to speak a new language. Learn new statements and commands (words), combine them into programs (sentences) and practice writing programs (speaking), until you know the language like a native.

When you have mastered Personal BASIC, consider learning another programming language. Digital Research has a complete assortment of languages tailored for your computer.

We hope to meet you again through the written word in another Digital Research language programming book. Goodbye and happy programming!

Appendix A
User's Glossary

address: Location in memory.

argument: Variable that the built-in function works on.

array: Collection of data items of the same data type. Term that describes a form of storing and accessing data in memory, visualized as matrices. The number of elements in an array is the number of dimensions of the array. A one-dimensional array can be called a list.

ASCII: Acronym for American Standard Code for Information Interchange. ASCII is a standard code for the representation of the numbers, letters, and symbols that appear on most keyboards.

assignment statement: Statement that assigns the value of an expression on the right side of an equal sign to the variable name on the left side of the equal sign. (A = B + 3).

boot: Initial start-up of a computer operating system after the computer is turned on or after a system error.

Break Mode: Debugging mode for Personal BASIC. Entered with a CTRL-C given while program is operating. The Br prompt indicates that you are in Break Mode.

buffer: Area of memory that temporarily stores data during the transfer of information.

bug: Error in a computer program.

built-in function: Subroutine that is part of Personal BASIC to which you can pass values and receive the computed values. For example, INT, RND.

byte: Unit of memory or disk storage that usually contains eight bits.

CAPS LOCK key: Keyboard key used when typing BASIC lines. Produces upper-case letters and lower-case numbers.

central processing unit: Brain of the computer, usually called CPU. The central processing unit contains a control unit, an arithmetic/logic unit and a storage unit.

character: Single symbol output and input. A character is usually represented by a byte inside the computer or storage device. For example, F, 6, +, #

code: Sequence of statements of a given computer language that make up a program.

command: Instruction given to BASIC outside the program. For example, SAVE, LIST, OLD.

compiler: Language translator that translates the text of a high-level language into machine code (1s and 0s) that the computer understands.

concatenation: The joining of two or more strings together, end-to-end.

constant: String or numeric value that does not change throughout program execution.

control character: Nonprinting character combination that sends a simple command to the operating system, Personal BASIC, or an applications program. To enter a control character, press the control (CTRL) key and the specified character at the same time.

CP/M: The operating system controlling the operation of Personal BASIC. CP/M stands for Control System for Microprocessors.

CPU: See **Central Processing Unit.**

<cr>: Symbol meaning press the carriage return, RETURN, or Enter key.

cursor: One-character symbol that can be moved anywhere on the video screen. The cursor indicates where the next keystroke will be placed.

data: Numbers, figures, names, symbols, etc.

debug: Find and remove errors from a program.

dimension: Number of elements in an array. A one-dimensional array is a list of the elements in the array. A two-dimensional array can be visualized as a matrix of rows and columns of storage space for the elements of the array. A three-dimensional array can be thought of as a geometric solid having volume.

disk, diskette: Magnetic media used to store information. Programs and data are stored and retrieved like music on a record. The term diskette usually refers to floppy disks 8 or 5 1/4 inches in diameter. The term disk can refer to a diskette, a removable cartridge disk, or a fixed hard disk.

disk drive: Peripheral device that reads and writes on hard or floppy disks. CP/M assigns a letter to each drive.

element: Individual data item in an array.

execute a program: Start a programming running. When the program is running, a sequence of instructions, or statements, is executing.

expression: Algebraic combination of variables, constants, operators, and function references that evaluates to an integer, real, or string value.

file: Collection of related records containing characters, instructions, or data; usually stored on a disk under a unique file specification.

file number: Unique identification number you assign to a file with the OPEN statement. File numbers can be any numeric expression.

filename: Name assigned to a file. The filename can be 1 to 8 alpha, numeric, and/or special characters. The filename should tell something about the file.

filetype: Letters following the filename indicating the type of file. Filetypes can be up to three characters long or omitted.

floating point: Value expressed in decimal notation that can include exponential notation; a real number.

floppy disk: Flexible magnetic disk used to store information. Sizes are 5 1/4 and 8 inches in diameter.

flowchart: Graphic diagram using special symbols to indicate the input, processing, output, and flow of a program.

function: See built-in function and user-defined function.

high-level language: Computer instructions written in procedural form or in the language of the problem. Many machine instructions are generated for each high-level statement. For example, BASIC, COBOL, Pascal.

I/O: Abbreviation for input/output.

input: Data entered into an executing program from the terminal or from external storage, or from READ/DATA statements.

integer: Positive or negative whole number with no decimal point.

interpreter: Computer program that translates and executes each source language statement in turn every time the source program is executed. Personal BASIC is an interpretive BASIC.

key: Specific field of a record on which processing is performed.

line number: The first item in a line of BASIC coding. Legal range is 0 to 65529.

listing: List of the source program. The LIST command produces a list of the Personal BASIC program in working storage.

load: Move programs or data from permanent storage into memory.

loop: Series of program instructions repeated a specific number of times. An endless loop is a program error.

numeric constant: Real or integer quantity that does not vary in the program.

numeric variable: Real or integer identifier to which various numeric quantities can be assigned during program execution.

open: Announcement to the operating system that a resource, usually a disk file, is to be used.

operating system: Collection of programs that supervises the execution of other programs and the management of computer resources. An operating system provides an orderly input/output environment between the computer and its peripheral devices. Personal BASIC runs under the CP/M operating system, which is compatible with many different computer systems.

output: Data that the processor sends to the console, printer, disk, or other storage media, after processing is complete.

peripheral device: Devices external to the CPU. Terminals, printers, and disk drives are peripheral devices.

permanent storage: Area to store programs and data outside of RAM. Usually is on disk or tape.

precedence: the order that arithmetic formulas are processed by Personal BASIC.

program: Series of coded instructions that tell a computer what operations to perform in solving a problem.

prompt: Characters displayed on the video screen to help the user decide what action to take. The Personal BASIC prompts are Ok, Br, and Ed.

RAM: See **random access memory.**

random access: Method of entering a file at any record number, without search from the first record.

random access file: File structure where data can be accessed in a random manner, no matter where it is located in the file.

random access memory: Temporary storage internal to your computer. Also called working storage. Common term for random access memory is RAM. Size of RAM is measured in Ks, one K = 1024 bytes.

random number: Number selected at random from a set of numbers. The RND function returns a random number in Personal BASIC.

real number: Numeric value specified with a decimal point, same as floating point notation.

record: Portion of a file containing related information such as a name and address. A file contains one or more records and is usually stored on disk.

record number: Position of a specific record in a fixed-length file, relative to record number 1. A key by which a specific record in a fixed file is accessed randomly.

reserved word: Keyword that has specific meaning to a given language or operating system. Usually, variables cannot use a reserved word as a name.

run a program: Start a program executing. When the program is running, the statements are being executed by the Personal BASIC interpreter.

sequential access: File structure where data is accessed serially, one record at a time, from the first record. Data can only be added to the end of the file and cannot be deleted. All magnetic tape files and some disk files are sequential access files.

SHIFT Key: Keyboard key that causes printing of upper-case characters. Affects all key positions.

source program: The program instructions as typed by the programmer.

statement: Coded instruction using specific keywords in a defined format. Examples in Personal BASIC are PRINT, INPUT, READ.

storage: See **permanent storage** and **temporary storage.**

string variable: Identifier of a group of characters to which varying values can be assigned during program execution. Examples, NAME$, R367$, X$

subroutine: Portion of a program that performs a specific task, but is logically separate from the main program flow. Subroutines can be used when the same sequence of code is used more than once. The main program flow branches to the subroutine, continues through the subroutine, and then branches back to the main program flow.

subscript: Integer expression that specifies the position of an element in an array.

syntax: Rules for structuring statements and commands for an operating system or programming language.

syntax error: Results from entering instructions not according to format rules (syntax).

temporary storage: See **random access memory.**

trace: Option used for debugging during a program run. The trace option usually lists each line of code as it executes to help the programmer debug the program.

user-defined function: Expression created and given a function name by the user. The function performs a specific task and is called into action by referencing the function by name.

value: Quantity expressed by an integer or real number.

variable: Name to which the program can assign a numeric value or a string. Examples, A, TEST$, G849, E$

variable name: Same as variable.

working storage: See **random access storage.**

End of Appendix A

Appendix B
Annotated Bibliography

B.1 Book References

Finkel, Leroy, and Jerald R. Brown. Data File Programming in BASIC. New York: John Wiley & Sons, Inc., 1981.

A text with detailed explanations of data file programming and how to set up your files on disk.

Lien, David. The BASIC Handbook. 2nd ed. San Diego: Compusoft Publishing Co., 1982.

A handbook explaining the BASIC statements and commands used in various versions of BASIC. Very useful if you are translating from one BASIC to another.

Moulton, Peter. Foundations of Programming Through BASIC. New York: John Wiley & Sons, Inc., 1979.

One of the few texts that teaches BASIC using the structured approach. This book might be hard to locate.

Nagin, Paul, and Henry Ledgard. BASIC With Style. Rochell Park, NJ: Hayden Book Company, Inc., 1978.

A BASIC text motivated by Strunk's and White's The Elements of Style. A light-hearted approach to writing carefully constructed, readable, BASIC programs.

Weinberg, Gerald. The Psychology of Computer Programming. New York: Van Nostrand Reinhold Co., 1971

A classic programming text for anyone serious about programming.

B.2 Magazine References

"BYTE Magazine." BYTE Publications, Inc. 70 Main Street, Peterborough, NH 03458

A thick magazine devoted to microcomputers. It is more technical than most magazines. The advertisements are numerous and educational. BYTE contains product reviews, columns, and articles.

"Creative Computing Magazine." Ahl Computing, Inc. P.O. Box 789-M, Morristown, NJ 07960

A microcomputing magazine with many product reviews, programs, and programming articles. Most of the content is understandable to the beginner.

"Datamation." 875 Third Avenue, New York, NY 10022

Datamation deals with the computer field in general. The emphasis is on mainframes, but there are some articles on microcomputers. The magazine is aimed at practicing professionals, but beginners will find parts of it interesting.

"Infoworld Magazine." Popular Computing, Inc. 375 Cochituate Road, Framingham, MA 01701

A weekly magazine devoted to the microcomputer. It contains news, columns, and reviews of hardware and software.

"Interface Age Magazine." 16704 Marquardt Avenue, Cerritos, CA 90701

A magazine featuring the business use of microcomputers. The magazine has articles on business applications and reviews of hardware and software.

End of Appendix B

Appendix C
Answers to Exercises

C.1 Answers to Exercises in Section 5

1.

```
    10 REM EXERCISE 1, SECTION 5
    20 READ A
    30 IF A=999 THEN 60
    40 SUM=SUM+A
    50 GOTO 20
    60 PRINT "SUM IS"SUM
    70 PRINT "AVERAGE IS"SUM/6
    80 DATA 474,651,562,701,631,568,999
    90 END
Ok RUN
SUM IS 3587
AVERAGE IS 597.833
```

2.

```
    10 REM EXERCISE 2, SECTION 5
    20 INPUT A
    30 IF A=999 THEN 60
    40 SUM=SUM+A
    50 GOTO 20
    60 PRINT "THE SUM IS"SUM"AND THE AVERAGE IS"SUM/6
    70 END
Ok RUN
? 474
? 651
? 562
? 701
? 631
? 568
? 999
THE SUM IS 3587 AND THE AVERAGE IS 597.833
Ok
```

3.

```
    10 REM EXERCISE 3, SECTION 5
    20 INPUT "NAME";NAME$
    30 INPUT "STREET ADDRESS";STREET$
    40 INPUT "CITY";CITY$
    50 INPUT "STATE";STATE$
    60 INPUT "ZIP";ZIP$
    70 PRINT
    80 PRINT NAME$
    90 PRINT TAB(5)STREET$
    100 PRINT TAB(10)CITY$", ";
    110 PRINT STATE$" ";
    120 PRINT ZIP$
    130 END
Ok RUN
NAME? ROBERT S. JONES
STREET? 140 OAK AVENUE
CITY? SAND CITY
STATE? CA
ZIP? 94562

ROBERT S. JONES
    140 OAK AVENUE
         SAND CITY, CA 94562
```

C.2 Answers to Exercises in Section 6

1.

```
    10 REM EXERCISE 1, SECTION 6
    20 FOR I=2 TO 1000 STEP 2
    30 SUM=SUM+I
    40 NEXT I
    50 PRINT "THE SUM IS";SUM
    60 END
Ok RUN
THE SUM IS 250500
```

2.

```
    10 REM EXERCISE 2, SECTION 6
    20 FOR A=1 TO 3
    30 FOR B=1 TO 3
    40 FOR C=1 TO 3
    50 PRINT A;B;C
    60 NEXT C
    70 NEXT B
    80 NEXT A
    90 END
```

3.

```
10 REM EXERCISE 3, SECTION 6
20 REM SET 1ST NUMBER AS LARGEST AND SMALLEST SO FAR
30 INPUT N:C=1
40 LARGEST=N:SMALLEST=N
50 INPUT N
60 IF N=999 THEN 120
65 REM COUNT OF NUMBERS
70 C=C+1
80 REM IS NUMBER LARGEST OR SMALLEST SO FAR?
90 IF N>LARGEST THEN LARGEST=N
100 IF N<SMALLEST THEN SMALLEST=N
110 GOTO 50
120 PRINT "THE SMALLEST NUMBER IS";SMALLEST
130 PRINT "THE LARGEST NUMBER IS";LARGEST
140 PRINT C"NUMBERS WERE ENTERED"
150 END
```

C.3 Answers to Exercises in Section 7

1.

```
    10 REM EXERCISE 1, SECTION 7
    20 INPUT ST$
    30 L=LEN(ST$)
    40 FOR X=1 to L
    50 C$=MID$(ST$,X,1)
    60 IF C$="B" THEN N=N+1
    70 NEXT X
    80 PRINT "THERE ARE";N;"B'S IN THE STRING."
    90 END
Ok RUN
? RABBIT
THERE ARE 2 B'S IN THE STRING.
```

2.

```
    10 REM EXERCISE 2, SECTION 7
    20 INPUT ST$
    30 A$=LEFT$(ST$,3)
    40 B$=RIGHT$(ST$,5)
    50 C$=MID$(ST$,4,5)
    60 PRINT B$+C$+A$
    70 END
Ok RUN
? ONE TWO THREE
THREE TWO ONE
```

End of Appendix C

Appendix D
Personal BASIC Error Messages

Table D-1 lists error numbers and their meanings.

Table D-1. Personal BASIC Error Messages

Number	Message
1	Your NEXT statement needs a matching FOR
2	Something is wrong
3	RETURN statement needs a matching GOSUB
4	READ statement ran out of data
5	Function call not allowed
6	The number is too large
7	Program is too large for memory
8	Not used
9	Subscript refers to element outside the array
10	You defined an array more than once
11	You cannot divide by zero or raise zero to a negative power
12	Statement is illegal in direct mode
13	Types of values do not match
14	Not used
15	Strings cannot be over 255 characters long
16	String expression is too long or too complex
17	CONT works only in Break Mode
18	Function needs prior definition with DEF FN
19	Not used
20	RESUME statement found before error routine entered

Table D-1. (continued)

Number	Message
21	Not used
22	Expression has operator with no following operand
23	Program line too long
24-49	Not used
50	FIELD statement caused overflow
51	Not used
52	File number or filename invalid
53	File not found on disk drive specified
54	File mode is not valid
55	You cannot OPEN or KILL a file already open
56	File number in use
57	Disk input/output error, restart your operation (MP/M™)
58	Filename exists
59	Not used
60	Not used
61	Disk is full
62	You have reached end-of-file
63	The record number in PUT or GET is more than 32767 or 0
64	The filename is invalid
65	Invalid character :X: in program file
66	File being read has statement with no line number
67	Not used
68-98	Not used
99	--Break--

Table D-1. (continued)

Number	Message
101	Program exceeds memory size
102	ON statement is out of range
103	A line number is expected here
104	A variable is required
105	Not used
106	Line number does not exist
107	Number is too large for an integer
108	Input data is not valid, restart input from first item
109	STOP
110	You have nested subroutine calls too deep
111-201	Not used
202	Command not allowed here
203	Line number is required
204	Your FOR statement needs a NEXT or WHILE needs a WEND
205	Your NEXT statement needs a FOR or WEND needs a WHILE
206	A comma is expected
207	A parenthesis is expected
208	OPTION BASE must be 0 or 1
209	Statement end is expected
210	Too many arguments in your list
211	Not used
212	Cannot redefine variable(s)
213	Function defined more than once

Table D-1. (continued)

Number	Message
214	You are trying to jump into a loop
221	System error #X, please restart (1 through 4)

End of Appendix D

Index