IBM BASIC Programming Guide

Program Product

B B B B B

A A A A A

S S S S S

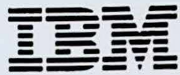
I I I I I

C C C C C

Order Number:
SC26-4027-2

Program Numbers:
5668-996 (VM)
5665-948 (MVS)

Release Number:
2



IBM BASIC Programming Guide

Program Product

Order Number:
SC26-4027-2

Program Numbers:
5668-996 (VM)
5665-948 (MVS)

Release Number:
2

Third Edition (February 1986)

This is a major revision of, and makes obsolete, SC26-4027-1.

This edition applies to Release 2 of IBM BASIC/VM, Program Product 5668-996, and IBM BASIC/MVS, Program Product 5665-948, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Amendments" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370 and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1982, 1983, 1986

About This Manual

This manual describes how to use IBM BASIC to design, develop, test, and run programs written in the BASIC language. IBM BASIC translates BASIC source programs into machine language for processing inside and outside the BASIC environment. IBM BASIC is available as two program products:

- IBM BASIC/VM, which runs under the Virtual Machine/System Product–Conversational Monitor System (VM/SP-CMS), batch and interactive
- IBM BASIC/MVS, which runs under MVS/SP batch, and with the Time Sharing Option (TSO) or the Time Sharing Option/Extended (TSO/E) batch and interactive

This manual provides guidance for using IBM BASIC to develop BASIC programs. It is not intended to be used as a reference manual; that is, it does not give detailed information for coding BASIC programs. For reference information, see *IBM BASIC Language Reference*.

The information in this manual is system-independent; this means it applies to all operating systems under which IBM BASIC runs. For information about a specific operating system, see your IBM BASIC System Services manual for the IBM BASIC product you are using.

The names IBM BASIC and BASIC are used when referring to the common features of both IBM BASIC/VM and IBM BASIC/MVS.

Manual Organization

This manual is designed for the following users:

- Persons who have acquired fundamental skills in BASIC, either through an instructional manual such as IBM's *B is for BASIC*, GC26-4102 or some other form of instruction
- Beginning users with little or no data processing experience, who need documentation introducing them to the principles of BASIC and of computing
- Programmers with or without programming experience in BASIC, who need documentation on how to use IBM BASIC at the full language level

This manual is organized as follows:

- **"Logging On and Connecting to IBM BASIC" on page 5** tells how to log on and off the system and IBM BASIC.

- **"Program Development Inside the BASIC Environment" on page 7** tells how to use the BASIC commands in writing, editing, and processing BASIC programs.

- **"Designing BASIC Programs" on page 37** gives useful tips on designing BASIC programs.

- **"Writing BASIC Programs" on page 39** gives guidance on using BASIC features to help you code BASIC programs.

- **"Performing Input/Output" on page 89** gives guidance on using BASIC features to perform input/output operations.

- **"Handling Exceptions" on page 129** gives guidance on how to handle exception conditions.

- **"Defining and Calling Subprograms" on page 137** tells you how to write and invoke subprograms, how to invoke subprograms written in other languages, and how to access GDDM and relational data bases.

- **"Running Your Programs — Using the RUN Command" on page 189** gives information on running BASIC programs inside the BASIC environment.

- **"Running Your Programs — Compiled Execution" on page 194** gives information on compiling and running BASIC programs both inside and outside the BASIC environment.

- **"Debugging Your Programs" on page 207** gives guidance on finding and correcting errors using the debugging features of BASIC

- **Appendixes A, B, and C on page 219** contain BASIC examples, including examples of calling GDDM and SQL.

- **Appendix D, "The SQL Sample Tables" on page 249** contains the sample tables used in the SQL examples.

- **Appendix E, "The STATS Subprogram" on page 253** contains a general purpose statistics subprogram used in one of the SQL examples.

- **The glossary on page 255** gives definitions for many of the terms used in this book.

This book is organized to reflect the tasks a user must perform in developing BASIC programs.

Note: Some examples in this manual feature interaction between the user and the system. To distinguish between user input and system output, the user input is shown in blue.

Industry Standards

IBM BASIC is designed according to the specifications of the following industry standards, **as understood and interpreted by IBM as of December, 1981:**

American National Standard for Minimal BASIC, ANSI X3.60-1978

International Organization for Standardization proposed standard ISO Minimal BASIC dp ISO-6373

European Computer Manufacturers' Association, Standard ECMA-55 Minimal BASIC, January 1978

These standards are technically equivalent.

IBM BASIC has many additional capabilities not contained in the above standards.

Related Documentation

This manual should be used with the following IBM BASIC manuals:

IBM BASIC Language Reference, GC26-4026, gives the syntax and rules for using BASIC statements, commands, and functions.

IBM BASIC/VM System Services, SC26-4028, provides guidance and reference information specific to using IBM BASIC under VM/SP-CMS.

IBM BASIC/MVS System Services, SC26-4106, provides guidance and reference information specific to using IBM BASIC with MVS, TSO, and TSO Batch.

IBM BASIC Language Reference Summary, SX26-3736, is a convenient, pocket-sized, quick-reference card that summarizes BASIC language elements.

If your BASIC programs communicate with the Graphical Data Display Manager (GDDM), you will find the following manuals useful:

Graphical Data Display Manager Base Programming Reference, SC33-0101.

Graphical Data Display Manager Presentation Graphics Feature: Programming Reference, SC33-0102.

Graphical Data Display Manager Presentation Graphics Feature: Interactive Chart Utility: User's Guide, SC33-0111.

If your BASIC programs communicate with IBM Database 2 (DB2) or Structured Query Language/Data Systems (SQL/DS), you will find the following manuals useful:

IBM DATABASE 2 Introduction to SQL, GC26-4082

IBM DATABASE 2 Application Programming Guide for TSO Users, SC26-4081

IBM DATABASE 2 Messages and Codes, SC26-4113

SQL/Data System Application Programming, SH24-5018

SQL/Data System Messages and Codes, SH24-5019

If your BASIC programs communicate with FORTRAN, COBOL, or PL/I, you will find the following manuals useful:

VS FORTRAN Language and Library Reference, GC26-4119

VS COBOL II Application Programming: Language Reference, GC26-4047

OS and DOS PL/I Language Reference Manual, GC26-3977

Summary of Amendments

IBM BASIC Release 2, February 1986

Product Enhancements

- Full-Screen Editing

  You can now make changes to any program line that appears on your screen.

- Program Function Key Support

  PF keys can be set to execute frequently used commands.

- User Profile

  You can specify commonly used function key settings and program options in a special PROFILE data set. The PROFILE is then processed automatically when you invoke BASIC.

- Real Data

  There are two additional numeric data types, REAL SINGLE and REAL DOUBLE. Mathematical computations performed using these data types execute significantly faster than those using BASIC's DECIMAL data type. The use of REAL data also makes BASIC's interface to FORTRAN more useful.

- DECIMAL Data

  The exponent range has been extended from ±75 to ±999, making DECIMAL data suitable for engineering and scientific users.

- FORTRAN, COBOL, and PL/I Interfaces

  Numeric and character arrays can now be passed to subprograms written in these languages.

- GDDM Interface

You can now pass GDDM function names as well as RCP codes. You can also invoke GDDM's Interactive Chart Utility (ICU) from BASIC.

- Relational Data

  You can now access data managed by IBM's relational data base systems (DB2 and SQL/DS) with the CALL SQL interface.

- MVS/XA Support

  MVS/XA support makes very large programs and data arrays possible. BASIC can reside above the 16 megabyte line and use 31-bit addressing. BASIC can also be installed in the extended link pack area.

- VM Tape Support

  BASIC now supports multi-file tapes.

Major Changes to the Book

- The Programming Primer section (part 1) has been removed. A new IBM BASIC publication, *B is for BASIC*, GC26-4102 will take the place of this section.
- The title of the book has been changed to *IBM BASIC Programming Guide.*
- Sections on full-screen editing have been added.
- Sections on Programmable Function (PF) keys have been added.
- A section on using profiles has been added.
- A section on using the SET commands has been added.
- A section on real data has been added.
- A section on the Graphical Data Display Manager (GDDM) has been added.
- A section on using SQL to access relational data bases has been added.
- The chapter previously named "Writing IBM BASIC Programs" has been reorganized into four chapters: "Writing BASIC Programs," "Performing Input/Output," "Handling Exceptions," and "Defining and Calling Subprograms."

IBM BASIC/MVS Release 1.0

New Program Product

The IBM BASIC Processor and Library are now available under MVS/SP as IBM BASIC/MVS (Program Number 5665-948). Information about the product has been added to this manual.

IBM BASIC/VM Release 1.1

Product Name Changed

The IBM BASIC program product, which runs under VM/SP-CMS, has been renamed to IBM BASIC/VM to differentiate it from the related product, IBM BASIC/MVS, which runs under MVS/SP.

IBM BASIC is used when referring to the common features of both products.

Publication names have been changed where required to indicate information specific to each product.

Subprogram Portion of Text Expanded

The subprogram section has been enlarged to include information on calling Assembler subprograms from IBM BASIC.

Contents

Introduction 1
Terminology Note and Other Notes 2
Inside or Outside the BASIC Environment 2

Logging On and Connecting to IBM BASIC 5

Program Development Inside the BASIC Environment 7
Creating Your Program 8
Entering Line Numbers Yourself 8
Line-by-Line Syntax Checking 9
Numbering Your Lines Automatically — AUTO Command 10
Ending Automatic Line Numbering 11
Writing More Than One Statement on a Line 11
Entering a Statement on Multiple Lines 11
Editing Your Program 12
Editing By Program Line Number 12
Editing Continuation Lines 12
Deleting Continuation Lines 13
Inserting Continuation Lines 14
Full–Screen Editing 14
Tips for Full-Screen Editing 15
PF Keys and Full–Screen Editing 16
Editing With Commands 16
INITIALIZE Command 16
Displaying Your Program — LIST Command 16
Renumbering Your Program — RENUMBER Command 18
Locating Character Strings in Your Program — FIND Command 19
Changing Character Strings in Your Program — CHANGE Command 20
Changing Lines by Line Number — CHANGE Command 21
Duplicating Lines — COPY Command 21
Deleting Lines — DELETE Command 22
Extracting Lines — EXTRACT Command 22
Combining Parts of Programs — MERGE Command 23
Renaming Your Program — RENAME Command 24
Saving and Retrieving Your Programs 24
Saving and Loading Your Program 25
Saving Your Programs — SAVE Command 25
Retrieving Your Programs — LOAD Command 26
Storing and Fetching Your Programs 26
How to Set and Use Programmable Function (PF) Keys 27
Notes for Using the PF Keys 27
Notes for Using RETRIEVE PF Keys 28
Using Profiles 28

PROFILE Option and PROFILE Command 29
SET PROFILE Command 30
SET Commands 30
SET LOG Command 31
SET MSG Command 31
SET OPTION Command 32
SET TERM Command 32
Interrupting Commands 33
Getting Help With the HELP Command 34
Ending an IBM BASIC Session — QUIT Command 35

Designing BASIC Programs 37

Writing BASIC Programs 39
Defining and Using Data Items 39
Defining and Using Numeric Data 39
Use of Numeric Data Types 40
Integer Data 40
Decimal Data 41
Real Data (Real Single and Real Double) 42
Numeric Constants 43
Numeric Variables 43
Numeric Typing Statements 44
Numeric Operators and Expressions 45
Assigning Numeric Values — LET Statement 46
Defining and Using Character Data Items 47
Character Constants 47
Character Variables 48
Character Operators and Expressions 49
Character Substrings 49
Defining and Using Arrays 50
Defining An Array 51
Dimensioning — OPTION And DIM Statements 52
Explicit Dimensioning 53
Implicit Dimensioning 54
Using Subscripts To Access Arrays 54
Subscripting Character Arrays 55
Using Array Expressions 55
Assigning Array Values — MAT Statements 56
Array Addition and Subtraction 56
Array Scalar Multiplication 57
Array Matrix Multiplication 57
Redimensioning Arrays 58
Defining and Using Functions 59
Using Intrinsic Functions 60
Defining and Using Your Own Functions 62
Single-Statement Functions 62
Multi-Statement Functions 62
Using Your Own Functions 63
Comparing and Combining Expressions 63
Relational Expressions 64
Logical Expressions 65
AND Logical Operator 65
OR Logical Operator 65

NOT Logical Operator 65
Priority of Evaluation 66
Controlling Program Flow 67
Using Program Loops 68
DO UNTIL/LOOP Block 68
DO WHILE/LOOP Block 69
FOR/NEXT Blocks 70
Loop Considerations 71
Using Nested Loops 71
IF/ELSE Blocks 72
SELECT/CASE Blocks 73
Branching 76
Line Labels 76
GOTO — Unconditional Branch 76
GOSUB/RETURN — Unconditional Branch 76
Conditional Branching — IF Statement 77
Conditional Branching — ON GOSUB Statement 78
Conditional Branching — ON GOTO Statement 78
Halting Program Execution 79
Using the END Statement 79
STOP Statement 80
PAUSE Statement 80
Chaining BASIC Programs 81
Starting Chained Program Execution — CHAIN Statement 81
Defining Chained Variables — USE Statement 82
Using the OPTION Statement to Change Options 83
Printing Precision for Real Data (SPREC | LPREC) 83
Collating Sequence for Comparing Character Data (COLLATE) 83
Inverting Print Edit Facility (INVP) 84
Levels of Messages (FLAG) 84
Including Comments in Your Program 84
What to Comment 85
How to Comment 85
Avoiding Common Programming Errors 86
Not Saving Your Program 86
Mechanical Errors 87
Logical Errors 87

Performing Input/Output 89
Using Program-Defined Data 89
Specifying Program-Defined Data — DATA Statements 89
Reading Program Defined Data — READ Statements 90
Reusing Program-Defined Data — RESTORE Statement 92
Terminal Input/Output 92
Retrieving Formatted Terminal Input — INPUT Statement 93
Retrieving Unformatted Terminal Input — LINE INPUT Statement 94
Using PROMPT Messages 95
Displaying Items at the Terminal — PRINT Statement 95
Reading and Writing Selected Fields 98
Requesting Formatted Output — IMAGE and FORM Statements 100
Printing Formatted Output — PRINT USING Statement 104
Handling Exceptions — INPUT and PRINT 105
Setting Page Margins — MARGIN Statement 106
Using Files 107
Opening and Closing Files 107

Record Files 108
Using Sequential Files 108
Accessing Internal Sequential Files 109
Accessing Native Sequential Files 109
Accessing Display Sequential Files 110
Positioning a Sequential File 112
Using Relative Files 112
Accessing Relative Files 113
Updating a Relative Record 115
Reading Relative Files Sequentially 115
Positioning Relative Files 115
Using Keyed Files 116
Accessing Keyed Files 116
Updating a Keyed Record 118
Reading Keyed Files Sequentially 119
Positioning Keyed Files 119
Using Stream Files 120
Accessing Stream Files 120
Using Array I/O Statements 122
Exception Recovery 123
Using VSAM 125
Displaying and Purging Your Files 126
Displaying Your Files — QUERY Command 127
Deleting Your Files — PURGE Command 128

Handling Exceptions 129
Exceptions — ON Condition Statement 129
Using Specific ON Conditions 130
Exceptions — I/O Statements 131
Displaying Exception Codes 132
Creating Exception Codes — CAUSE Statement 132
Reprocessing After an Exception — RETRY Statement 133
Continuing After an Exception — CONTINUE Statement 133
Exception Handling Considerations 134
RETRY and CONTINUE Without an Exception 134
Exception Loops 134
Losing the Location of an Exception 135
Exceptions in Functions 136

Defining and Calling Subprograms 137
Defining a Subprogram — SUB and END SUB Statements 137
Invoking a Subprogram—CALL Statement 138
Transferring Control from a Subprogram—SUBEXIT Statement 139
Using the STOP Statement in a Subprogram 139
Using the CHAIN Statement in Subprograms 139
Program Data in Subprograms 140
Exceptions in Subprograms 140
Using Subprograms Written In Other Languages 141
Using Subprograms Written in Assembler Language 142
Argument List for Assembler Language Subprograms 143
Calling sequence 146
Linking Assembler Language Subprograms 147
Calling System Commands from a Program 147
Using the SET TERM Command from a BASIC Program 147

Calling Graphical Display Data Manager (GDDM) Operations 148
Kinds of Graphics You Can Produce with GDDM 148
Calling GDDM 151
The Pie and Bar Chart Program 153
Calling the Interactive Chart Utility (ICU) 156
Using SQL to Access Relational Data Bases 156
Structure of a Relational Table 157
How to Use the CALL SQL Statement 159
Retrieving Multiple Rows Versus Single Rows 161
Deactivating a SQL Request 162
Checking Return Codes 162
Messages 164
The Value-List: Receiving Data from the Data Base 164
The Value-List: Passing Data to the Data Base System 164
Data Types and Conversions 166
Handling NULL Values 166
Some Things to Keep in Mind... 167
CALL SQL Examples 168
Other SQL Examples 187

Running Your Programs 189
Direct Execution Versus Compiled Execution 189
Running Your Programs — Using the RUN Command 189
Direct Execution of the Program in Your Workspace 190
Direct Execution of Filed Programs 190
Direct Execution with SPREC or LPREC 191
Direct Execution with a Parameter (PARM) 191
Interrupting Execution 191
Performance Considerations 193
Running Your Programs — Compiled Execution 194
Compiled Execution — Inside the BASIC Environment 194
Requesting Compilation — COMPILE Command 194
Requesting Compiled Execution — RUN Command 195
Compiled Execution — PAUSE Statement 196
Compiled Execution — Outside the BASIC Environment 196
Requesting Compilation — Compiler 196
Requesting Execution — Outside the BASIC Environment 197
Requesting Batch Execution 198
Compiler Options 198

Debugging Your Programs 207
Using Debugging Commands 208
Suspending Execution — BREAK Command 208
Resuming Execution — GO Command 209
STEP Mode — RUN and GO Commands 209
Using Debugging Statements 209
Activating Debugging — DEBUG Statement 210
Suspending Execution — BREAK statement 210
Suspending Execution — PAUSE statement 210
Using Immediate Statement 211
Using the Immediate PRINT Statement 211
Using the Immediate LET or MAT Statement 212
Using the Immediate STOP Statement 212
Using Immediate Variables 213
Using the Immediate Typing Statements 213

Using The Immediate DIM Statement 214
TRACE Statement 214
Using the Diagnostic HELP Command 216

Appendix A. Sample Programs 219
Sample Program 1: EDITOR 219
Sample Program 2: LOANS 221
Sample Program 3: CALC 227
Sample Program 4: CUST 232

Appendix B. Sample Program for Calling the GDDM Interactive Chart Utility 237
Using EZICU 237
EZICU Subprogram 240

Appendix C. SQL Examples—Advanced Techniques 241
Example: Retrieval with Multiple Concurrent Selections 241
Example: Retrieval with User-Specified WHERE Clause 243
Example: Retrieval and Update with User-Specified WHERE Clause 246

Appendix D. The SQL Sample Tables 249
The Q.STAFF Sample Table 249
Q.ORG Table 251

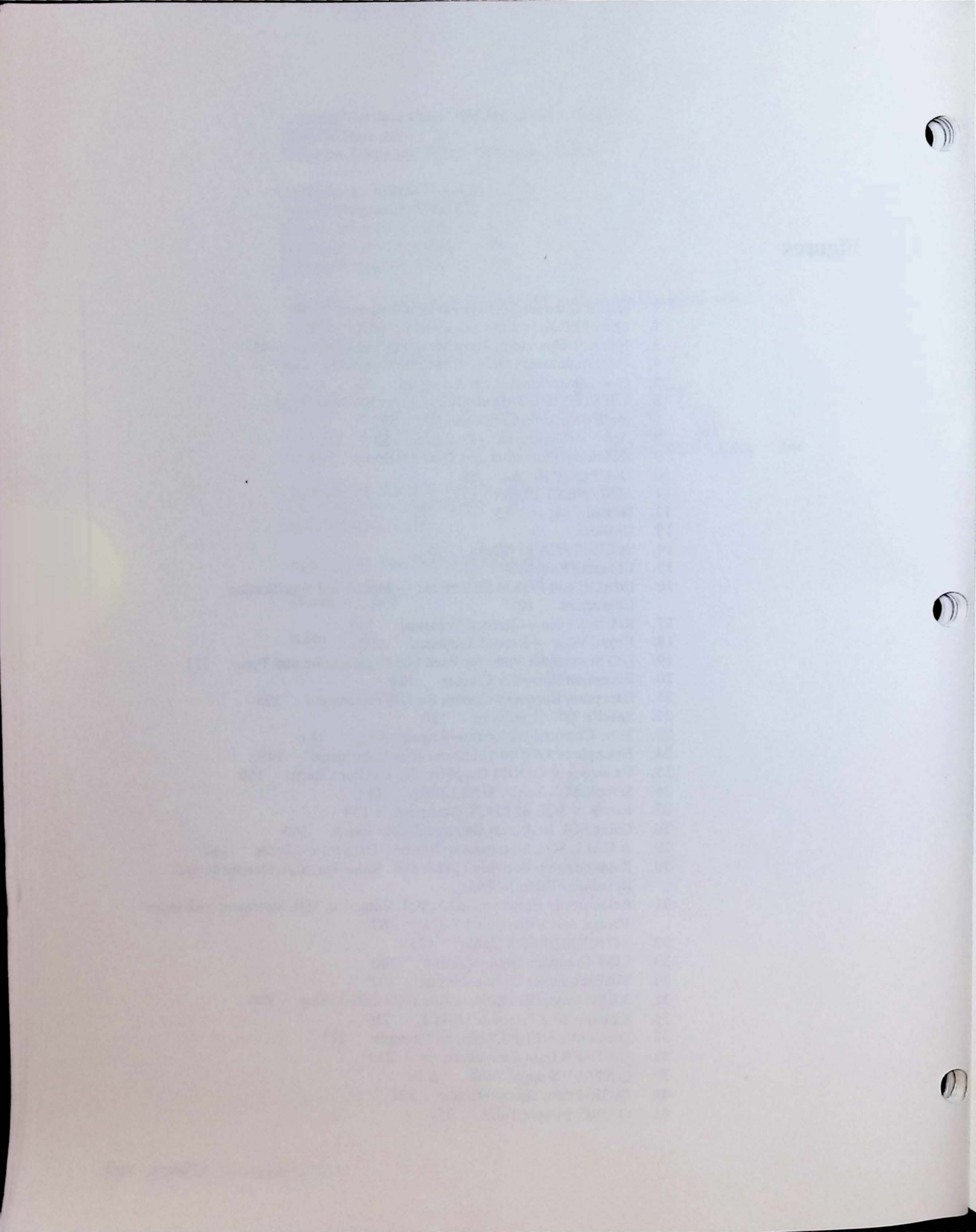
Appendix E. The STATS Subprogram 253

Glossary 255

Index 261

Figures

1. HELP Command—Example of a Response 34
2. Use of PF Keys and Codes while in HELP 35
3. Numeric Operators, Their Meanings and Priorities 45
4. Quotation Marks within Character Constants 48
5. One-Dimensional Array Examples 52
6. WEATHER Two-Dimensional Array Example 52
7. Matrix Multiplication Example 58
8. Array Redimensioning Example 59
9. Relational Operators and Their Meanings 64
10. DO/LOOP Blocks 69
11. FOR/NEXT Blocks 71
12. Nested Loops 72
13. IF Block 73
14. SELECT/CASE Blocks 75
15. Chaining Programs 81
16. IMAGE and FORM Statements — Most-Used Specification Characters 101
17. Relative Files — Record Sequence 113
18. Keyed Files — Record Sequence 116
19. I/O Statements Valid for Each File Organization and Type 121
20. Exception Recovery Clauses 124
21. Exception Recovery Clauses for I/O Statements 125
22. Specific ON Conditions 130
23. Type Conversions for Interlanguage Calls 141
24. Example of GDDM Graphics: Triangular Shape 149
25. Examples of GDDM Graphics: Pie and Bar Charts 150
26. Sample SQL Table: SUPPLIERS 158
27. Result of SQL SELECT Statement 158
28. Using SQL to Access Relational Data Bases 160
29. A CALL SQL Statement to Retrieve Data from a Table 160
30. Relationalship Between CALL SQL Value-list, SQL Statement, and Relational Table 165
31. Relationship Between CALL SQL Value-list, SQL Statement with Index Values, and a Relational Table 167
32. STOCKMARKET Table 175
33. LIST Compiler Option Output 200
34. MAP Compiler Option Output 202
35. XREF Compiler Option Cross-Reference Listing 206
36. Example of a Program TRACE 216
37. Diagnostic HELP Command Example 217
38. Q.STAFF Data Characteristics 250
39. Q.STAFF Sample Table 250
40. Q.ORG data characteristics 251
41. Q.ORG Sample Table 251



Introduction

The BASIC language is an easy-to-learn, easy-to-use computer language applicable to a wide range of scientific and business applications. Many people introduced to programming for the first time get that introduction through BASIC, and for good reasons. BASIC is not only easy to learn and use, it has the programming capabilities needed for much more than mere problem solving.

Now IBM BASIC has added many functions to previous BASIC program products, to make BASIC an even more versatile programming tool than ever before. IBM BASIC has the power needed for production programming, language constructs that allow lengthy and sophisticated programs, self-documenting capabilities that make programs easy to maintain, versatile input/output and file access methods that make complicated input/output easier to achieve than before, and all this without losing the simplicity that has always been BASIC's strongest point.

IBM BASIC provides:

- A set of statements, designed for easy understandability
- An easy-to-use command set, designed especially for editing, running, and debugging BASIC programs
- A set of immediate statements, designed both for use in desk-calculator mode and for program debugging

These facilities ensure that you are for the most part free of operating system concerns when developing and running programs.

Source programs in BASIC consist of statements you write to solve your problem; these statements must conform to BASIC programming rules.

As you enter each statement, its syntax is immediately examined by BASIC. If there are errors, you are immediately given an error message and can then make any needed corrections.

After your program is complete, you can run it immediately, or you can save it for later execution.

To assist you in these activities, BASIC provides you with a set of commands, developed especially for BASIC. These commands let you edit your programs, place them in permanent storage, correct errors in them, and execute them directly from the terminal.

The BASIC immediate statements also assist you in these activities.

With few exceptions, a BASIC program entered or compiled under one supported operating system will run and produce identical results under any other supported operating system, except as described below. The exceptions are:

- System-dependent functions invoked in the CALL SYSTEM statement
- System-dependent file names in the OPEN and CHAIN statements and in commands
- System-dependent exception codes

Terminology Note and Other Notes

IBM BASIC runs under two different operating systems, each of which uses different terminology to refer to stored data. For example, under VM/SP CMS, data is stored in files, and under MVS and TSO, data is stored in data sets. This manual uses *file* to refer to stored data. See your IBM BASIC System Services manual for information on using files.

Some examples in this manual feature interaction between the user and the system. To distinguish between user input and system output, the user input is shown in blue.

Note for remote-display terminal users: If you are experiencing slow response time, switching to LINE mode may help. The original setting is determined by the installation option, and is usually SCROLL mode. All the examples in this book assume you are in SCROLL mode. In SCROLL mode, the input line is at the bottom of the screen; the lines you enter are scrolled up the screen. To switch modes, use the SET TERM command. (See "SET TERM Command" on page 32.) You can also switch modes from a program, using the CALL BASIC statement. (See "Using the SET TERM Command from a BASIC Program" on page 147.)

Many examples show a program by listing it using the LIST command. That program is assumed to be the one in the workspace for the example.

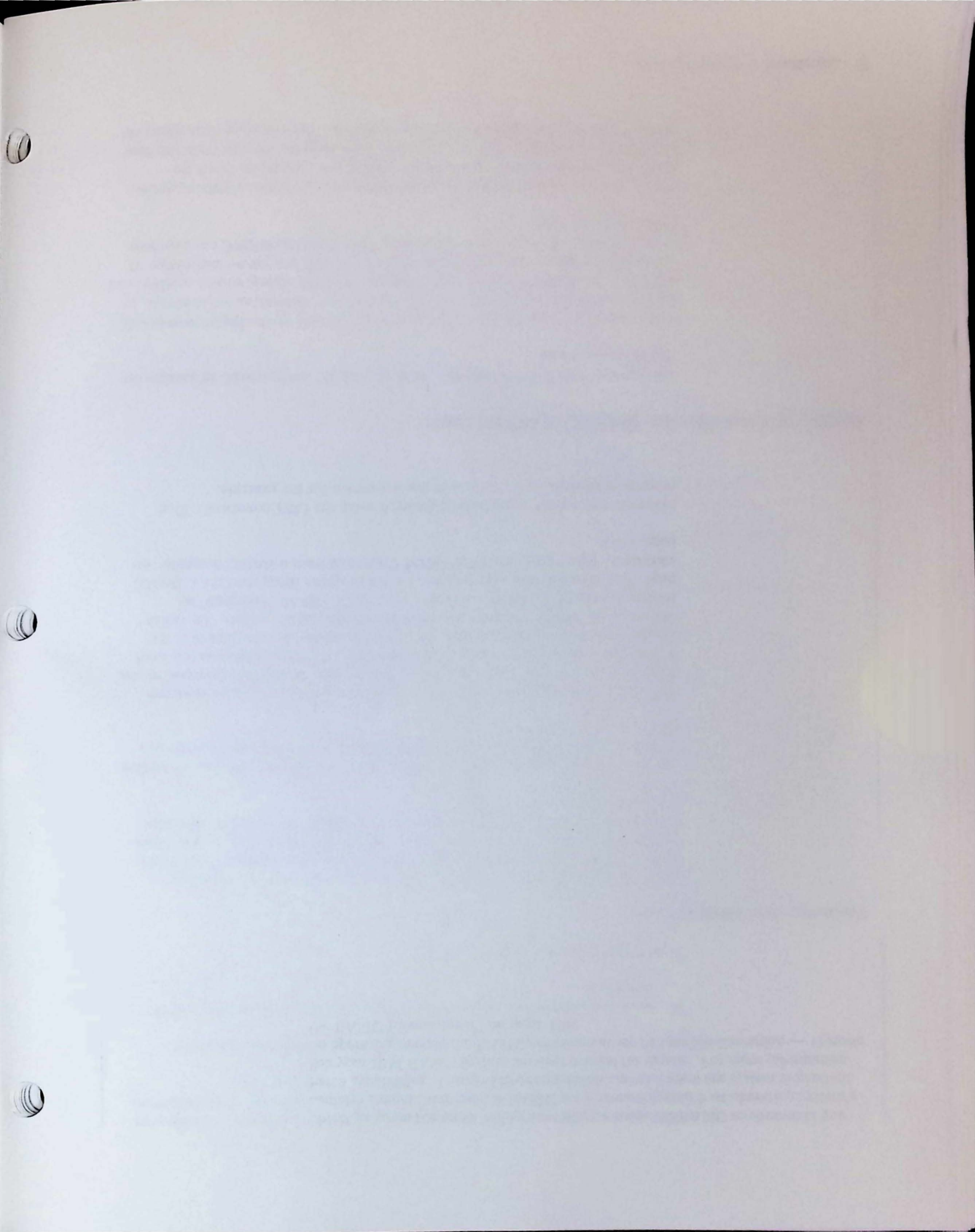
Inside or Outside the BASIC Environment

You can use BASIC in two modes: inside the BASIC environment, or outside the BASIC environment.

When operating *inside the BASIC environment*, you have all the facilities of BASIC. BASIC commands, editing features, and "immediate" statements are available, in addition to the BASIC programming language. You can create, debug, modify, and run BASIC programs using an interactive terminal with little or no knowledge of the host system. This document assumes you are inside the BASIC environment unless otherwise stated.

When operating *outside the BASIC environment*, you can both compile programs (with the BASIC command) and execute pre-compiled programs (with the BASICRUN command). You invoke BASIC as a compiler directly from the host system for each source program file to be compiled. You can elect to produce an

object program file and a listing file. When outside the BASIC environment you can also compile and execute BASIC programs using your host operating system's batch capabilities. Using batch, compilation and execution are system dependent. See your IBM BASIC System Services manual for details. For more information on operating outside the BASIC environment see "Compiled Execution — Outside the BASIC Environment" on page 196.



Logging On and Connecting to IBM BASIC

The easiest way to create a BASIC program is at the terminal, using the interactive editing facilities provided inside the BASIC environment.

Before you can begin building a BASIC program from the terminal, you must log on to the system, using your organization's logon procedure. See your IBM BASIC System Services manual for more details on logging on and invoking BASIC.

After you've logged on to your operating system, you can invoke the BASIC environment by issuing the following command:

```
basic
```

This activates IBM BASIC allowing you to operate inside the BASIC environment. (Your organization may have another invocation procedure; check with your system administrator.)

BASIC then displays the following message:

```
IBM BASIC/s VERSION x RELEASE y.z  yyyy/mm/dd  hh:mm
 *
  _
```

to indicate it is waiting for your input. (The *s* is the operating system identification, VM or MVS, for the IBM BASIC product you are using. The *x* is the version, and *y.z* is the release of IBM BASIC you are using. The *yyyy/mm/dd* is the current date, and *hh:mm* is the current time.) The asterisk prompt is IBM BASIC's indication that you are expected to enter a BASIC command, immediate statement, or statement beginning with a line number. The underline shows the cursor position (that is, the place where the next character can be entered).

Program Development Inside the BASIC Environment

This chapter tells you how to use the terminal to create a program, how to test the program after it's created, and how to make corrections to it. You can do this through the BASIC commands and editing facilities.

BASIC programs are composed of program statements. To edit and debug your programs, you can use immediate statements and commands.

You can enter information in uppercase, lowercase, or mixed case. In this book, most examples are shown in lowercase. Names of statements, commands, and other items are shown in uppercase when not in examples.

Program Statements: Every BASIC program consists of program statements. The first program statement on each line of a program must be preceded by a line number. For example:

```
150 let a = b * c
```

When you enter program statements, they are checked for errors and then saved for future reference. Only when you compile or run the program are the lines processed.

Immediate Statements: Immediate statements are executed as soon as you enter them. Immediate statements do not have a line number; for example:

```
let a = b * c
```

You can use immediate statements in *calculator mode* — that is, use them for solving problems in the same way you would use a desk or pocket calculator. You can also use immediate statements to inspect or change program variables during debugging.

Commands: BASIC commands are also executed immediately. Commands do not have line numbers; for example:

```
delete 100 to 140
```

tells BASIC to erase workspace lines 100 through 140.

You can use the BASIC commands to edit and debug your program after you've initially entered it.

Creating Your Program

Every BASIC program you write is composed of statements preceded by a line number. Program statements require line numbers; commands and immediate statements do not begin with numbers. Ordinarily, you'll write one statement for each line of the program. These statements are also known as source statements and the program as a source program.

In a BASIC program, each line begins with a line number. The line numbers identify each line in your program; you can use them as reference points within the program itself, or when you're editing the program later. You can enter the line numbers yourself or use the AUTO command to have them entered automatically. The line number also determines the order in which the statements are to be processed. All statements are processed in numeric sequence, regardless of the order in which you entered them, unless the sequence is altered by branches, loops, or subroutines. The allowable range of line numbers is 1 to 9999999.

BASIC maintains the program you are currently working on in the *workspace*—an area of storage reserved for a user's exclusive use. When you log on, your workspace is empty. When you create a program it is entered, line by line, into the workspace. When you have finished working on the current program, you can save it. The workspace is the collection of information which completely describes the program you are currently editing and/or executing. This includes the program itself, the data being acted upon, and other information concerning the status of the program.

Entering Line Numbers Yourself

When you first enter the BASIC environment, BASIC displays an asterisk to prompt you to enter a line.

```
* _
```

The underscore represents the screen cursor, which shows where you will begin entering input.

If you enter a line number, your entry is a program statement rather than a command or immediate statement and is retained in your workspace for processing later.

When you're entering program line numbers, it's a good idea to number the first line with some number other than 1 (say 100), and to use a line increment larger than 1 (say 10); this gives you some room for inserting new lines between existing ones. For example:

```
 * 100 print "have"
 * 110 print "a"
 * 120 print "day"
 * 115 print "nice"
 * 130 end
 * run
have
a
nice
day
END AT LINE 130.
```

You can see that when we realized that we had forgotten to include "NICE" in our program, it was a simple matter to include it. Notice that line 115 is entered out of order. There is no need to be concerned. If you type in the lines out of order, BASIC rearranges them in the proper sequence by line number.

If you try to enter two different program lines using the same line number, BASIC only remembers the last entry with that number. For example:

```
 * 100 print "this is number one"
 * 100 print "this is number two"
 * 110 end
 * run
this is number two
END AT LINE 110.
```

The only line number 100 you will have in your program is the last one you entered.

In addition, during program creation, you can enter a line number with no other input and then press the ENTER key and BASIC deletes that line.

```
 * 500
LINE 500 DELETED.
```

Line-by-Line Syntax Checking

As you type each line, BASIC checks it for syntax errors.

Example:

```
 * 100 let a = b* - c
                  1
###1) BAS30122S NUMERIC TERM EXPECTED.
```

This is incorrect, as you cannot follow a multiplication sign with a minus sign.

BASIC informs you of the error as follows:

- A 1 is printed beneath the point in the line at which the first error is detected.
- The message is keyed to that number and tells what the error is.

You can now correct the statement:

```
 * 100 let a = b * (-c)
```

If you enter a line with more than one error, a 2 appears under the second error, a 3 under the third error, etc. For example:

```
 * 100 s$ =(a$b$)(4:7)(2:3)
               1     2    3
###1) BAS30131S END OF IDENTIFIER OR CONSTANT EXPECTED.
###2) BAS30161S STATEMENT KEYWORD, VARIABLE, OR ARRAY ELEMENT EXPECTED.
###3) BAS30161S STATEMENT KEYWORD, VARIABLE, OR ARRAY ELEMENT EXPECTED.
```

The HELP command can be used to get more information about error messages; simply enter HELP or HELP followed by the 8-character message code. See "Using the Diagnostic HELP Command" on page 216 for more information. (The HELP command can also be used when you want more information about a specific command or statement or if you want tutorial assistance. See "Getting Help With the HELP Command" on page 34.)

Numbering Your Lines Automatically — AUTO Command

You can automatically number your program lines as you enter them if you use the AUTO command. The line number will then appear as a prompt:

```
* auto
* 100 _
```

Enter the statement after the line number. When the workspace is empty, BASIC begins line numbering at 100 and displays a number 10 higher after you have entered each line. Each time you enter a line, BASIC prompts you by displaying the next line number:

```
* auto
* 100 print "have"
* 110 _
```

If you don't want to begin numbering at 100 and want the line increment to be 5, you can do this by entering an optional beginning line number and an optional increment:

```
* auto 500 step 5
* 500 print "have"
* 505 print "a"
* 510 print "nice"
* 515 print "day"
* 520 end
* 525 _
```

This AUTO command specifies that you want numbering to begin at 500 and to increase by 5.

If you already have a program in the workspace and do not specify a starting line number, the AUTO command begins numbering from the highest number in your workspace plus the increment.

For example:

```
* 515 print "day"
* 520 end
* auto step 5
* 525 _
```

Ending Automatic Line Numbering

To stop automatic line numbering, respond to the line number prompt with a null line by pressing the ENTER key instead of entering text. BASIC stops automatic line numbering and responds with its normal prompt.

BASIC will also end automatic line numbering if:

- Your next line number overlaps an existing line. This could happen if you are adding lines in the middle of your program. You'll get an error message.
- Your next line number exceeds 9999999 (the highest line number allowed). You'll get an error message.
- The line you have just entered has a syntax warning or error. The warning or error in the line is flagged, and you'll get a message. The line may then be corrected by reentering the line, and you may resume automatic numbering with a new AUTO command.
- The line you have just entered has a different line number or no line number.

Writing More Than One Statement on a Line

You can enter more than one statement on a single program line. Your first statement follows the line number. If you enter a colon (:) at the end of your statement, you can then enter an additional statement on the same line. For example:

```
* 100 let a=5 : let b=10 : let c=15
```

is functionally equivalent to the three following separate LET statements:

```
* 100 let a=5
* 110 let b=10
* 120 let c=15
```

Entering a Statement on Multiple Lines

You can continue BASIC statements from one line to the next. This is convenient for long statements like OPEN or FORM.

To continue a statement on the next line, type an ampersand (&) as your last nonblank character on the line. BASIC prompts for the next line with an ampersand. For example:

```
* 500 print "have", &
*   &_
```

You can continue statements for as many lines as you need. However, you cannot continue a keyword, identifier, constant, or comment.

The maximum length of each line in a BASIC program line is 156 characters including the line number and/or the continuation symbol (&). The total number of characters in a BASIC program line is limited only by the storage available.

Editing Your Program

If you want to edit a program that is already in your workspace, you can begin immediately. However, if the program has been saved, you must first move it into your workspace from your library.

Your library is an area of auxiliary storage reserved for your use in which you can place your programs (either while you are developing them or after they've been completely debugged) for future reference.

To move a program into the workspace from your library, you issue a LOAD command. For example:

```
* load oldpgm
```

loads the previously saved program, named OLDPGM, into your workspace.

If you want to know what programs are saved in your library, you can issue the QUERY command. For example:

```
* query program
```

gives you a display of all the BASIC program files in your library. You can also query for the presence of a specific file, for all the data files, and for all files of all types.

After the program is in your workspace, you can edit the program by line number, with full-screen editing, or by using commands. Each method is discussed below.

Editing By Program Line Number

If you want to replace a program line you have already entered, enter a new line with the same line number. For example:

```
* 510 x=y+1
* 510 x=y
```

replaces the original statement X=Y+1 with the statement X=Y.

You can delete existing program lines by typing the line number and then pressing the ENTER key. For example:

```
* 510
```

removes the statement(s) at line 510 from your program.

Editing Continuation Lines

Continuation lines (also called line segments) within a multiline statement are assigned segment numbers relative to the original line number. For example:

Program Line	Segment Number
500 print "have", &	500.0
& "a", &	500.1
& "nice", &	500.2
& "day"	500.3

You can replace individual continuation lines by typing the line number and the relative segment number. For example:

```
* 500.2& "wonderful", &
```

changes "nice" to "wonderful".

Whenever you change any part of a line, BASIC checks the syntax of the entire line and reports any error. In particular, if you modify a continuation line so that the continuation ampersands are incorrect, you get an error message, and the incorrect continuation lines are still present even though they are not syntactically correct.

You can refer to continuation lines by segment number when:

- You are editing by line number, by following your line number with the segment number.
- You are using a special form of the CHANGE command (described later) to recall a particular continuation line to the terminal screen so you can edit it with the terminal's editing keys to insert and delete characters.

All other references to line numbers within your program and in BASIC commands are to simple line numbers and designate the entire line, continuation lines included.

Deleting Continuation Lines

You can delete continuation lines by entering the line number and the relative continuation number. For example:

```
* 500.2
```

deletes the continuation line at 500.2.

If the continuation lines originally were:

Program Line	Segment Number
500 print "have", &	500.0
& "a", &	500.1
& "very", &	500.2
& "nice", &	500.3
& "day"	500.4

The continuation lines after the delete are:

Program Line	Segment Number
500 PRINT "have", &	500.0
& "a", &	500.1
& "nice", &	500.2
& "day"	500.3

Inserting Continuation Lines

To insert a new continuation line after the continuation line 500.1, you enter:

```
* 500+2& "very",&
```

and the complete line now lists as

```
500 print "have",&
  & "a",&
  & "very",&
  & "nice",&
  & "day"
```

If you try to insert a continuation line after the last continuation line, by entering:

```
* 500+5& "all day"
```

you will get a syntax error; the fourth continuation line does not end with an ampersand. Therefore, attempting to add another continuation line will cause a syntax error. To avoid this error, you should edit the last continuation line so it ends with an ampersand, then add the continuation line.

If the last line ends with an ampersand *and* the entire line up to that point is syntactically correct, BASIC assumes you wish to add another continuation line and prompts you with its leading ampersand. Thus the last example could be done as follows:

```
* 500.4& "day",&
*    & "all day"
* list
500 print "have",&
  & "a",&
  & "very",&
  & "nice",&
  & "day",&
  & "all day"
 *
   _
```

Full–Screen Editing

If you have a 327x type display terminal, you can use the BASIC facility for *full-screen editing*, in addition to line number editing as explained above. In essence, this editing facility works like other full-screen editing facilities; you use the cursor to change, replace, or delete characters anywhere on the screen. Although the line numbers and continuation ampersands inherent to BASIC make the editing process a little different, you will find full-screen editing to be an extremely helpful and efficient tool.

Full-screen editing allows you to change a BASIC program line that is displayed on the terminal. (You cannot change commands or immediate statements.) To use it, you need a 327x type terminal with SET TERM SCROLL in effect, and you must be prompted by the *__ (command prompt) symbol. You make changes to lines on the screen by moving the cursor under a line and typing over the characters, or by using special keys to insert or delete characters. You need only change a portion of the line if that is sufficient to accomplish your purpose, and you may change as many lines as you like before pressing the ENTER key.

When you press enter, BASIC will replace the changed program lines in your workspace, starting at the top of the screen. A message will appear in the right-hand side of your input area at the bottom of the screen, telling you how many program lines were changed. You can also enter data in the input area at the bottom of the screen, as you normally would while in scroll mode. Whatever you enter in the input area will be processed last.

Tips for Full-Screen Editing

The entire line (or continuation line) must be on the screen to be edited. If only a partial line is on the screen (this can only occur at the top of the screen), it is in a protected area, and cannot be changed by using full-screen editing.

If any errors occur as a result of your changes, the lines in error are scrolled onto the screen with the appropriate messages, in the order in which they are found. Anything entered in the input area will remain there; it will not be acted upon. The lines in error will be entered into the workspace as is.

Use the LIST command to display your program so that you can edit it. If you want to edit a previous page, use the LIST BACK command to get back to that page. Similarly, the LIST FORWARD command will move the page forward. (For more information on the LIST command see "Displaying Your Program — LIST Command" on page 16 and *IBM BASIC Language Reference.*)

Although you can change every line on the screen before pressing ENTER, we suggest that you enter only a few changes at a time. This is because (1) it is awkward to deal with a great number of error messages; and (2) if the list of lines with error messages is longer than your screen, those scrolled out of sight cannot be corrected. However, you can easily recall those lines onto the screen with the LIST ERROR command (see "Displaying Your Program — LIST Command" on page 16).

If you press the CLEAR key while making changes (before pressing the ENTER key), the screen will be returned to the state before you made the changes. Also, if you clear a line using the ERASE EOF key that line will reappear unchanged (if no further changes are made before pressing the ENTER key).

If you run a program that executes an INPUT FIELDS or PRINT FIELDS statement, any lines remaining on the screen when the

*_

prompt comes up, cannot be edited. If you wish to edit those lines, use the LIST command to put another copy of the lines on the screen.

PF Keys and Full–Screen Editing

You can use PF keys in conjunction with full-screen editing (see "How to Set and Use Programmable Function (PF) Keys" on page 27).

If you press a PF key, processing of the entire screen takes place (all changes on the screen are entered for processing) before the PF key is acted upon.

Editing With Commands

In addition to line number and full-screen editing, you can also use BASIC editing commands. Some of these commands edit the "current line."

The *current line* is the last program line processed by a BASIC command. However, when you use the editing commands, you may modify the setting of the current line. The current line number is the line number associated with the current line.

The effect of each command on current line is explained with each command. Unless stated otherwise, "line" means a line beginning with a line number, including all the associated continuation lines.

See *IBM BASIC Language Reference* for reference documentation for the BASIC commands.

INITIALIZE Command

Use the INITIALIZE command before you start entering a new program. It clears your workspace and closes all files that may be open.

You can name the new program by entering the name as part of the INITIALIZE command. For example:

```
* initialize abc
```

clears the workspace, closes any open files, and names your workspace ABC. You can now enter the statements for the program ABC.

Displaying Your Program — LIST Command

You can display your program by using the LIST command. There are various ways you can use LIST.

If you enter the LIST command without any options, your entire program in the workspace is displayed. If your program is too long to fit within a single screen, as much of the program as possible is displayed, and you are given a message asking if you want to continue. If you do, press the ENTER key and the display area is filled with the next group of program lines. You can continue this until the end of your program, just as though you were turning the pages of a book. If you do not want to continue, you can use an attention interrupt to end LIST command processing. To find out how to generate an attention interrupt on your system, see your IBM BASIC System Services manual.

If you want to display only one line, enter the LIST command and a line number. For example:

```
 * list 500
500 a=b+c
```

If you want to display a group of lines, enter the first and last number of the lines to be displayed. For example:

```
 * list 500 to 600
```

displays lines 500 through 600.

The LIST command, with or without options, sets the current line to the last line displayed.

Use the LIST command with the XREF option to obtain a cross-reference listing of the program in your workspace. For example:

```
 * list xref
```

This listing will consist of:

- referenced line numbers in numeric sequence
- line labels in alphabetic sequence
- variables, arrays, user-defined functions, and intrinsic functions in alphabetic sequence.

Use the LIST command with the ERROR option to display those lines with syntax warnings or errors and their syntax messages. If you enter ERROR ALL, all lines in the requested range(s) with syntax warnings or errors are displayed, along with their messages. For example:

```
 * list error all
```

If you enter ERROR without ALL, only the first line with syntax warnings or errors is displayed, along with its messages.

Note: You cannot specify the XREF and ERROR options at the same time.

Although listings are normally displayed at your terminal, you may use the OUT option when you want to direct your output to a file. For example:

```
 * list error all out (myfile)
```

directs a listing of all errors in the workspace program to the file named MYFILE.

Renumbering Your Program — RENUMBER Command

With the RENUMBER command you can "clean up" your program by changing the line numbers of all or part of your program.

Suppose you have the following program:

```
100 print "count to 500"
101 let n=n+1
102 print n
110 if n <500 then goto 101
120 print "we now have 500"
130 end
```

The original line increment was 10, but during editing we inserted additional program lines. Now we can use a RENUMBER command to restore the original increment:

```
 * renumber
RENUMBER COMPLETED.
 * list
100 print "count to 500"
110 let n=n+1
120 print n
130 if n <500 then goto 110
140 print "we now have 500"
150 end
```

Using the RENUMBER command, you can also change the beginning line number and the increment with the STEP clause. For example, the following RENUMBER command gives the results shown:

```
 * renumber at 200 step 5
RENUMBER COMPLETED.
 * list
200 print "count to 500"
205 let n=n+1
210 print n
215 if n <500 then goto 205
220 print "we now have 500"
225 end
```

Notice that all the line number references within your program are changed to refer to the new line numbers; not only was line 110 changed to line 205, but the line number reference in the GOTO part of the IF statement was changed from GOTO 110 to GOTO 205.

You can also specify a range of lines to be renumbered. For example:

```
 * renumber 150 to 225 at 500 step 20
```

Only the lines from 150 to 225 in your program are renumbered. The first renumbered line will be 500. Subsequent lines are increased by 20. Any references to the renumbered lines will be adjusted accordingly.

If the renumbering you specify would cause an overlap of lines outside the specified range, BASIC does not make the changes, and you get an error message.

RENUMBER does not change the position of the current line, but it may change the line number associated with it. For example, if the last statement in the program was 150 before renumbering and 225 after renumbering, the current line is still the last statement but its line number is now 225 rather than 150.

Locating Character Strings in Your Program — FIND Command

You can locate specific character strings in your program with the FIND command. You can find the first occurrence of a character string, or all occurrences of a character string in a single line or a group of lines. For example:

```
* find 'their'
```

finds and displays the first occurrence of *their* in your program.

The apostrophe (') delimits the string within your FIND command. For the delimiter, you can use any special character (not 0 through 9 or A through Z) that is not contained within the character string you want to locate. You could not, for example, use the apostrophe (') as a delimiter when you want to find THEY'RE, because in this case the apostrophe is part of the character string you are searching for.

Also, if you enter

```
* find 'THEIR'
```

and the word *their* appears in lowercase letters in your program, BASIC will not find it. To the FIND command, THEIR and *their* are different character strings; they must appear identical if a match is to be made.

```
* find 200/THEY'RE/
```

finds the first occurrence of THEY'RE anywhere from line 200 to the end of your program.

```
* find 200 to 300/THEY'RE/all
```

finds and displays all the occurrences of the lines containing THEY'RE from line 200 through line 300, inclusive.

```
* find first to 350/THEY'RE/all
```

The search is from the beginning of your program and continues to line 350, finding and displaying all occurrences of THEY'RE.

```
* find /90 days overdue/all
```

finds and displays all occurrences of the lines containing the character string *90 days overdue.*

FIND sets the current line equal to the last line inspected. If you do not request FIND ALL, and the search is successful, the current line is set to the line at which the match was found. If ALL is specified, the current line is set to the last line in the range of lines inspected.

Changing Character Strings in Your Program — CHANGE Command

You can locate and change specific character strings in your program with the CHANGE command. CHANGE operates much like FIND in searching for and locating character strings.

As with the FIND command, the search argument must be identical to the character string to be located and changed, with respect to upper- and lowercase letters. "THEY'RE" and "they're" are different character strings to the CHANGE command.

You can change the first occurrence of a character string or all occurrences of a character string in a single line or a group of lines.

```
* change/their/they're/
```

changes the first occurrence of *their* to *they're* in the current line.

```
* change 200 to 400/their/they're/
```

changes the first occurrence of *their* to *they're* in every line between 200 and 400, inclusive.

```
* change/THEIR/THEY'RE/all
```

changes all occurrences of THEIR to THEY'RE in the current line.

```
* change 200/their/they're/all
```

changes all occurrences of *their* to *they're* in line 200.

```
* change 200 to 400/THEIR/THEY'RE/all
```

changes all occurrences of THEIR to THEY'RE in every line between 200 and 400, inclusive.

```
* change first to last/their/they're/all
```

changes all occurrences of *their* to *they're* in every line of your program.

To delete a character string from the program, you specify the new string as not having any characters (that is, type two successive delimiters). Thus, to delete the first occurrence of THEY'RE on every line from 100 through 200, you specify:

```
* change 100 to 200/THEY'RE//
```

When you issue a CHANGE command, BASIC displays each changed line, and, when all changes are completed, displays the total number of changed lines. In addition, as each line is changed, BASIC checks the syntax of the new line; if there is a syntax error, BASIC flags it and gives you an error message.

If the old character string you have specified cannot be located, BASIC gives you the message:

```
STRING NOT FOUND.
```

When you change continuation lines within a multiline statement, remember to include the same leading and/or trailing ampersands in the new character string as in the old character string. If you don't, you'll get a syntax error message.

After execution of the CHANGE command, the current line is set to the last line accessed by CHANGE. If you specify a range of lines, this is the last line in the range.

Changing Lines by Line Number — CHANGE Command

You can also use the CHANGE command to place a program line in the input area of your screen. For example:

```
* change 110
```

places program line 110 into the input area of the screen. You can then retype, or insert and delete, only that part of the line you want to change.

Note: You must be in SCROLL mode and on a 327x type terminal to use this form of the CHANGE command. See the note for remote-display terminal users on page 2. It explains the SET TERM command, which you must use when you want to change modes.

Duplicating Lines — COPY Command

You may have a line, or block of lines, that you want to use more than once in your program. Rather than reentering these lines, you can use the COPY command. To use the COPY command, you need only specify the line or lines you want copied and the point in the program at which you want the copied lines placed. For example:

```
* copy 200 at 500
```

copies line 200 at line 500.

```
* copy 100 to 200 at 500 step 5
```

copies lines 100 through 200 beginning at line 500. The copied lines are increased by 5.

Any statements within the block of lines which are copied to a new block, have their line numbers changed. For example:

```
 * copy 100 to 107 at 250
3 LINE(S) COPIED.
 * list
100 if x=y then 107
105 cause n
107 let x=0
.
.
.
250 if x=y then 252
251 cause n
252 let x=0
.
.
.
```

copies lines 100, 105, and 107 at 250, 251, and 252. If you don't specify the increment, (that is, omit the STEP clause) the default increment is one.

COPY changes the current line to be the last line of the new block of lines.

Deleting Lines — DELETE Command

The DELETE command allows you to delete a single line or a group of lines in your program. Request the lines you want deleted as follows:

```
 * delete 210
LINE 210 DELETED.
```

deletes line number 210 only.

```
 * delete 200 to 300
LINES 200 TO 300 DELETED.
```

deletes all the lines from 200 through 300 inclusive.

```
 * delete first to 300
LINES 10 TO 300 DELETED.
```

deletes all lines from the first line in your program through line 300, inclusive.

```
 * delete 250 to last
LINES 250 TO 550 DELETED.
```

deletes all lines from line 250 through the last line in your program.

```
 * delete first to last
ALL LINES DELETED.
```

deletes all the lines in your program.

```
 * delete 100 to 200, 310, 450
LINES 100 TO 200 DELETED.
LINE 310 DELETED.
LINE 450 DELETED.
```

deletes all lines from line 100 to 200 and then deletes line 310 and line 450.

The current line is set to the next line after the highest line number deleted. If you enter DELETE 500 TO 750 and your program only has statements from 100 to 600, the current line is set to the last line remaining in your workspace, in this case the statement immediately preceding statement 500.

Extracting Lines — EXTRACT Command

Suppose you want to delete most of the lines in your program, except for only a few. Instead of using the DELETE command, specifying the lines you want to delete, you can use the EXTRACT command, specifying the line you want to keep.

For example:

```
 * extract 110
LINE 110 EXTRACTED.
```

deletes all the lines in your program *except* line 110.

```
* extract 200 to 300
LINES 200 TO 300 EXTRACTED.
```

deletes lines 1 to 199 and lines 301 to the end of your program, keeping lines 200 through 300.

EXTRACT sets the current line to the last line in your workspace.

Combining Parts of Programs — MERGE Command

Use MERGE when you want to extract a sequence of statements (or all statements) from a program saved in a file and insert them into the program in your workspace. You can use this command to take previously written subprograms from one program and place them into a new program. You don't have to enter all the statements again.

MERGE has two capabilities: The first allows you to transfer statements from the file to the *empty* line positions in your workspace; the second allows you to transfer statements from the file and *replace* the statements at the line positions in your workspace.

Merging Without Replacing Lines: To merge an entire file into empty line positions in your workspace, you enter:

```
* merge filea
```

which copies all the lines in FILEA to the program in your workspace. If the program in your workspace already contains any lines with line numbers in the range of FILEA's line numbers, the lines from FILEA are not copied and a message is issued.

To merge part of a file into empty line positions in your workspace, you enter:

```
* merge filea 100 to 200
```

which copies lines 100 to 200, inclusive, from FILEA to the program in your workspace but does not renumber them. If the program in your workspace already contains any lines with line numbers between 100 and 200, the lines from FILEA are not copied and a message is issued.

To merge part of a file to a specific position in your workspace without replacing lines, you enter:

```
* merge filea 600 to 800 at 300
```

which copies lines 600 to 800, inclusive, from FILEA to the workspace renumbering them starting with 300 and increasing by the increments that exist in FILEA. If the program in your workspace already contains any lines in the range of the new line numbers (that is, lines 300 and following), the lines from FILEA are not copied and a message is issued.

Merging and Replacing Lines: To replace lines in the workspace with lines from a file, you enter:

```
* merge filea 350 to 500 at 600 step 5 (replace
```

which copies lines 350 to 500, inclusive, from FILEA to the workspace, renumbering them starting with 600 and increasing by 5. If there are any line numbers in your workspace which are in the range of the new line numbers, they are replaced by the lines from FILEA.

Renaming Your Program — RENAME Command

You can change the name of the program in your workspace with the RENAME command. For example:

```
* rename nuname
```

changes the name of your current program to NUNAME.

Now, if you reissue the RENAME command without specifying a new name, the old name is displayed:

```
* rename
RENAME 'NUNAME' TO (NULL LINE FOR NO CHANGE): _
```

At this point, you can enter another name for NUNAME, or you can press the ENTER key to cancel the command and retain NUNAME as the name of the program.

Saving and Retrieving Your Programs

When you're developing a program, you'll usually want to save it for future reference. You can use the SAVE and STORE commands for this purpose.

Each command creates a different kind of file:

- The SAVE command creates a file containing your program lines.

  You should use the SAVE command if you want:

  - line-by-line syntax checking of this program when you load it into the workspace
  - to transfer it to tape or diskette for use at some other location
  - to edit the file using some other editor (for example, XEDIT or the SPF editor).

  Use the LOAD command to retrieve a program that was saved.

- The STORE command creates a file containing both your program lines and the BASIC internal representation of your program.

You should use the STORE command if you do not need to have your program syntax checked, line by line, when it is fetched into your workspace. For example, use the STORE command when your program is ready for operational use, as syntax checking should no longer be necessary.

Use the FETCH command to retrieve a program that was stored.

Another consideration you should note is that the STORE/FETCH commands are faster than the SAVE/LOAD commands.

Remember, if you issue a QUIT command before you save or store the program in your workspace, the contents of your workspace will be lost. The QUIT command clears the workspace before transferring control to the system.

Saving and Loading Your Program

Use the SAVE and LOAD commands in conjunction with each other. SAVE writes a program from your workspace to a file. LOAD reads this program from the file into your workspace.

Saving Your Programs — SAVE Command

Use the SAVE command to save the program in your workspace in a source file. There are several options to the SAVE command.

You can enter the SAVE command without specifying a file name. For example, if ABLE is the current name of your workspace and you enter:

```
* save
```

BASIC prompts you with the name of your workspace:

```
ENTER NAME (NULL LINE TO CANCEL): ABLE
```

The name ABLE is in the input area and may be modified by typing over it or using the Insert or Delete keys. You can enter a new name and then press ENTER, or you can press ENTER without changing anything and accept the current name, or you can erase everything after the colon and press ENTER to cancel the SAVE command. The program in your workspace is written to a file with the associated name.

If your workspace doesn't have a name, you are asked to enter one. If you enter a null response to this prompt, the SAVE command is canceled.

You can use a file name with the SAVE command. For example:

```
* save stat
```

saves the workspace program in a source file named STAT.

If the name you specify in the SAVE command is the same as the name of your workspace (whether entered as a response to the prompt or as part of the command), the program is always saved.

If the name of your workspace is BAKER and you enter SAVE STAT, BASIC responds differently depending on whether a source file named STAT exists in your library:

- If a source file named STAT does **not** exist, the BAKER program is saved as STAT.
- If a source file named STAT does exist, BASIC asks you if you want to replace it. If you answer YES, the program in your workspace replaces the source file STAT. If you answer NO, the program in your workspace is not saved.

However, if you use the keyword REPLACE, the program in your workspace is saved whether a source file with that name exists or not:

```
* save abc (replace)
```

writes the program in your workspace to the file that was previously saved as ABC. (The current program *replaces* ABC.)

SAVE does not change the current line number, the program in the workspace, or the name of the workspace.

Retrieving Your Programs — LOAD Command

Use the LOAD command when you want to move a previously saved program from your library into your workspace. For example:

```
* load stat
```

clears your workspace, loads the previously saved program STAT, sets the current line to the last line in the workspace, and sets the workspace name to STAT.

There is a line-by-line syntax check as the program is being loaded, with a display of the line(s) in error and any associated error message(s). The workspace name is set to the name used in the LOAD command, and the current line is set to the last line in the workspace.

If a source file named STAT does not exist, BASIC displays an error message.

Storing and Fetching Your Programs

The STORE and FETCH commands should be used in conjunction with each other.

The STORE command is like SAVE, and FETCH is like LOAD. Their syntax and use are similar.

The STORE command writes the contents of your workspace to a file, including the source program (the same information saved by the SAVE command) and an internal representation of the program.

The FETCH command reads a file containing a previously stored program into your workspace.

Use the STORE and FETCH commands instead of the SAVE and LOAD commands when you want to improve performance when developing large programs. That is, you can avoid the time spent syntax checking during a LOAD.

One reason *not* to use STORE/FETCH is because the internal format of the file cannot be edited, compiled, or printed outside the BASIC environment.

How to Set and Use Programmable Function (PF) Keys

Programmable Function (PF) keys are special numbered keys on a 327x type display terminals whose meaning can be defined by the user. BASIC associates character strings with the numbered PF keys. To set a PF key, you use the SET PF command. For example:

```
* set pf2 delayed renumber
```

Now, when you press PF2, the character string RENUMBER will be displayed in your input area; any data already in the input area is discarded. You can leave the string as is, or change it. Note the special keyword DELAY. This "delays the execution" of the character string defined for the PF key, and causes it to be placed in the terminal input area. The RENUMBER command will be executed when ENTER is pressed. If you want, you may modify the line before pressing the ENTER key.

You can also choose to have the character string entered immediately, without your intervention. For example:

```
* set pf4 immed set log on
```

Now, when you press PF4, SET LOG ON will be executed immediately.

If you specify neither DELAYED nor IMMED, the default is DELAYED. For example:

```
* set pf2 renumber
```

will have the same effect as the SET PF2 command above.

If you simply enter SET PF with no trailing characters, that key has no associated action. For example:

```
* set pf7
```

When you press a PF key with no associated action, the input area is cleared and a message appears, but the screen does not scroll.

Notes for Using the PF Keys

- To use the PF keys, you must have a 327x type display terminal, and be in SCROLL mode. Otherwise (that is, you are using another type of terminal and/or SET TERM LINE is in effect) the BASIC key settings will be ignored.

- The keys defined by SET PF command apply only to the BASIC environment while you are in command mode (that is, when an asterisk prompt is displayed). They do not affect the PF keys used in HELP, those specified by a BASIC program while in execution, or those specified for the host operating system.

- Initially there is no action associated with any of the PF keys. To use PF keys, you must issue the SET PFn command, either directly from the asterisk prompt, or indirectly via a profile. See "Using Profiles."

Notes for Using RETRIEVE PF Keys

When you set a PF key to RETRIEVE, for example,

```
* set pf6 retrieve
```

the data you enter in response to the

```
*
 _
```

command prompt (that is, your commands, BASIC lines, and immediate statements) will be saved in a stack. Thereafter, when you press a RETRIEVE PF key, the last line put on the stack is recalled into the input area. You can modify this line before executing it; in any case, you execute it by pressing the ENTER key. If you press a RETRIEVE key again, the next to last line is recalled, replacing the one currently displayed. You can do this until the end of the stack is reached; at that point, the display starts over again at the top of the stack. (The lines are stored in a 500 character stack.)

- You can set more than one PF key as a RETRIEVE key.

- No stack of remembered input lines is maintained until a SET PFn RETRIEVE command is entered.

- If all the Retrieve Key definitions are removed, all entries are removed from the stack and no more are added to it.

Using Profiles

In BASIC, a profile is a file you can use to define some of the characteristics of your BASIC environment. A profile may contain commands, immediate statements, and BASIC lines, just as you would type them in at the terminal. You can use your profile to assign PF key settings, turn logging on, or set options. You can create different profiles for different tasks, such as a profile for debugging, for writing programs, for running programs, and so on.

PROFILE Option and PROFILE Command

You can specify a profile when you invoke BASIC, with the PROFILE option. For example:

```
basic (profile (basprof))
```

would invoke BASIC, using the profile BASPROF (see your IBM BASIC System Services manual for more information on naming a profile). Here is a sample profile BASPROF.

```
!   note: this is basprof
set profile noterm
set pf1 immed help
set pf2 immed run
set pf3 quit
set pf4 load
set pf5 renumber
set pf6 immed save
set pf7 immed list back
set pf8 immed list forward
set pf9 system "
set pf10 retrieve
set pf11 profile newprog
set pf12 immed print page
set option arithmetic native, lprec, base 1
100 option base 1
```

The above profile, BASPROF contains commands (SET PROFILE, SET PF, and SET OPTION), a program statement (100 option base 1), and an immediate statement (the immediate remark on the first line). When you execute this profile, none of the lines will appear on your screen. If, however, you changed the SET PROFILE NOTERM command to be SET PROFILE TERM, all of the other lines from this profile will appear on your screen. Your PF keys will be set up by the SET PF commands executed, as will the default arithmetic, precision, and base options for your session by the SET OPTION command. The last line puts the program statement,

```
100 option base 1
```

in your workspace. If you do a LIST command after your profile has executed, you will see it there.

Notice that this profile allows you to invoke another profile with PF key 11. You can even cause your profile to read another profile, with the PROFILE command. For example, the command:

```
profile myprog
```

in the profile above would invoke the profile MYPROG.

Note also, that invoking a profile can cause program lines to enter the workspace. For example, in the above profile the statement

```
100 option base 1
```

would be entered in the workspace.

You can also invoke BASIC *without* a profile. For example:

```
basic (noprof)
```

would invoke BASIC and specify that a profile is not to be used. If you specify neither PROFILE nor NOPROF, the default is as follows:

- PROFILE, if a default profile exists
- NOPROF, if a default profile does not exist

If you specify PROFILE when you invoke BASIC but not the name of a profile, the profile used is the profile set at installation time. See your system administrator.

When you are using the PROFILE *command*, you must specify the name of a profile.

SET PROFILE Command

You can use the SET PROFILE command to control the display of lines when a profile is read. For example:

```
set profile term
```

specifies that any profile lines read are to be displayed, as well as any Processor warning or information messages. In other words, the lines will be displayed just as if the profile were typed in at the terminal. If logging is on, these lines and messages are logged. You would typically use SET PROFILE TERM to debug a profile with errors, to check a profile's contents, or for a demonstration.

When BASIC is invoked, NOTERM is the default option. You can also specify NOTERM in the SET PROFILE command. For example:

```
set profile noterm
```

specifies that the profile lines read are not to be displayed, and that Processor warning and informational messages are to be suppressed. These lines and messages are not logged. However, if logging is on, lines with errors are logged along with the messages.

SET Commands

There are six versions of the SET command, each with a different function:

```
set log
set msg
set term
set option
set pf
set profile
```

SET PF is discussed under "How to Set and Use Programmable Function (PF) Keys" on page 27, and SET PROFILE is discussed under "Using Profiles" on page 28. SET LOG, SET MSG, SET OPTION, and SET TERM will be discussed here.

SET LOG Command

Use SET LOG to control logging at the terminal. When logging is on, a record of terminal input/output is sent to a file in the order in which it occurs. For example:

```
set log on
```

will activate logging and send the log record to the file BASLOG (see your IBM BASIC System Services manual for naming conventions for files).

```
set log out (first)
```

will activate logging and send the log record to the file FIRST.

Later, you may print your log file and obtain a complete record of everything that happened during your BASIC session at the terminal. This could be extremely helpful for debugging.

Whenever you SET LOG OUT to a file that already exists, the existing contents of that file are erased before logging resumes. For example, if the file FIRST above contains 100 lines of input and output, and you enter SET LOG OUT (FIRST), the file is now empty as a result of that command. It will contain only the lines which are subsequently logged to that file, until you turn logging off.

You can, however, add a log to the end of an existing file. For example:

```
set log out (first) append
```

will activate logging, and add the log record to the end of whatever already exists in FIRST.

Also, although the first SET LOG ON command (without the OUT clause) in a BASIC session erases the default file BASLOG, subsequent SET LOG ON commands will not; the log record will be added to the existing file.

```
set log off
```

terminates logging. SET LOG OFF when logging is not on will result in an error message. Likewise, SET LOG ON when logging is already on will result in an error message.

SET MSG Command

The SET MSG command controls how error messages are displayed at the terminal. With the SET MSG command, you can specify whether you want the entire message or just the text to be displayed. You would use this command, for example, if you wanted to suppress the message code for warning messages. The example below shows a warning message with both code and text:

```
 * 500 let a = 1024
 .
 .
 .
 * break off 500
             1
###1) BAS00016W BREAKPOINT NOT SET FOR LINE.  LINE NUMBER IGNORED.
```

Now using SET MSG, the code is suppressed:

```
 * set msg (w) text
SET MSG COMPLETED.
 * break off 500·
             1
###1) BREAKPOINT NOT SET FOR LINE.  LINE NUMBER IGNORED.
```

This command controls messages displayed at the terminal, not messages written to a listing file as a result of a COMPILE command.

See *IBM BASIC Language Reference* for more information.

SET OPTION Command

You can use the SET OPTION command to override the default options on your system (with the exception of the RD option). If, for example, you are used to working with array subscripts which begin with 1 rather than 0, you'd want to use:

```
set option base 1
```

to override the default of OPTION BASE 0.

The value set by the SET OPTION command will affect running a program, compiling a program, and executing immediate statements. However, this value will be overridden by any options explicitly stated in an OPTION statement within a program (see "Using the OPTION Statement to Change Options" on page 83).

See *IBM BASIC Language Reference* for more details about other default options in your system you can override with the SET OPTION command.

SET TERM Command

Use the SET TERM command to change your terminal mode. You can specify SET TERM LINE or SET TERM SCROLL. However, SCROLL mode is meaningful only for display terminals of the 327x family, because for other terminals (such as typewriter terminals), terminal interaction is always in LINE mode.

You may find that LINE mode improves your terminal response time particularly if you are using a remote terminal. (See note under "Terminology Note and Other Notes" on page 2.)

As a result of an option set at installation time, your terminal mode may already be set to LINE mode.

The HELP mode and the INPUT FIELDS and PRINT FIELDS statements are not affected by the SET TERM command.

You can also execute the SET TERM command from a program. See "Using the SET TERM Command from a BASIC Program" on page 147.

Full-screen editing and IBM BASIC's definition for PF keys are not available in LINE mode.

Interrupting Commands

You can interrupt certain BASIC commands by generating an attention interrupt. See your IBM BASIC System Services manual for information on how to generate an attention interrupt on the operating system you are using. The commands that acknowledge the attention interrupt and their resulting actions are listed below.

AUTO The attention interrupt terminates AUTO mode as though you had responded to an AUTO line number prompt with a null line.

CHANGE The attention interrupt terminates the operation of the CHANGE command at the next line to be searched. For example, if you have entered CHANGE 100 TO 500/ABLE/BAKER/ALL and change has searched lines 100 through 195 when the attention interrupt occurs, the operation of the command is terminated at the next line. If the next line is 200, the character string ABLE has been changed to BAKER in every statement in lines 100 through 195.

COMPILE If a COMPILE command causes a file to be loaded and the interrupt occurs while loading, the action is the same as for a LOAD command, and the compilation is not started. If the interrupt occurs during the compilation, it is ignored.

FIND The attention interrupt terminates the operation of the FIND command at the next line to be searched as in the CHANGE command.

GO Interrupting a GO command is equivalent to interrupting an executing program (discussed in "Interrupting Execution" on page 191).

LIST The attention interrupt terminates the operation of the LIST command at the next line to be listed.

LOAD The attention interrupt terminates the operation of the LOAD command at the next line to be loaded. Lines previously loaded are left in the workspace.

MERGE The attention interrupt terminates the operation of the MERGE command and the workspace is unchanged; that is, the workspace remains in the same state as it was prior to MERGE.

RUN Interrupting a RUN command is equivalent to interrupting an executing program (discussed in "Interrupting Execution" on page 191). If a RUN command causes a file to be loaded and the interrupt occurs while loading, the action is the same as for the LOAD command, and execution of the program is not started.

SAVE The attention interrupt terminates the SAVE command, but the file may be left in a partial state. The workspace is unchanged.

STORE The attention interrupt terminates the STORE command, but the file may be left in a partial state. The workspace is unchanged.

If a command is not explicitly identified above, you cannot interrupt its processing.

Getting Help With the HELP Command

Anytime you want some assistance in applying a specific command to a specific situation, you can just enter HELP followed by the name of the command. For example, if you enter

```
* help find
```

the system responds with the screen display shown in Figure 1.

```
*** BASIC HELP ***** FINDCOMMAND (BASHFIND) ********* PAGE 1 OF 4 ***

The FIND command locates character strings in BASIC statements in the workspace
and displays the lines containing them.

+-----------------------------------------------+
|    FIND <start-line <TO end-line>>            |
|         delim string <delim <A<LL>>>          |
+-----------------------------------------------+

'start-line'
     may be either a line number, the keyword F<IRST>, or the keyword L<AST>.

'end-line'
     may be either a line number or the keyword L<AST>.  End-line must be
     greater than or equal to start-line.

'delim'
     is a delimiter  of the string.  It must  be a special character (other than
     0-9 or A-Z), including the space character such that:

     *  delim is not contained in string.


===>                                            PRESS PF1 FOR HOW TO USE HELP.
1=HLP 2=     3=PRV  4=     5=PRT  6=     7=SCB 8=SCF 9=     10=     11=     12=CAN
```

Figure 1. HELP Command—Example of a Response

The bottom line of the screen is a reminder of what each PF key does while you're in HELP. The meaning of each abbreviation is shown in Figure 2 on page 35.

PF Key	Code	What it Does
1	HLP	Displays the first page of a BASIC HELP item explaining how to use HELP
3	PRV	Displays the page of the previous HELP item
5	PRT	Prints all the pages of the current HELP item (in this example prints all 5 pages of LIST)
7	SCB	Scrolls backward 1 page of the current HELP item
8	SCF	Scrolls forward 1 page to the next page of the current HELP item
12	CAN	Cancels HELP and returns to BASIC

Figure 2. Use of PF Keys and Codes while in HELP

HELP has many other uses too. You can use HELP to get tutorial assistance on just about any feature of BASIC: commands, statements, other language features, explanations of messages you get. In fact, you can use HELP for almost any problem you encounter while you're developing programs.

Ending an IBM BASIC Session — QUIT Command

When you want to leave BASIC and return to the operating system, you use the QUIT command:

```
* quit
```

Your BASIC session is ended and you are now communicating with the system.

If you enter QUIT without a SAVE or a STORE, you lose the program in your workspace. QUIT clears your workspace before returning control to the operating system; it is therefore essential that you save or store your program before entering the QUIT command.

To log off from the system, use the procedure defined by your organization. See your IBM BASIC System Services manual or check with your system administrator.

Designing BASIC Programs

Defining the task of your program in complete detail is often called writing a specification. This program specification should be in a natural language like English or pseudo-English. Pseudo-English is a kind of shorthand which outlines the program tasks and their interrelationships in a brief but concise way. Most programmers develop their own style, but if you use shorthand words that are closely related to BASIC keywords, the final step of coding is much easier.

Programming specifications consist of three major parts:

1. **Input** — what raw data goes into the program
2. **Processing** — what type of changes take place in processing this raw data
3. **Output** — what the program produces

But these subspecifications are not necessarily undertaken in the above order. You usually write the specifications in the following order: output, input, and processing.

The output specifications describe:

- The final output of the program
- The data needed to achieve this result
- The intermediate processing required to achieve it

The input specifications describe:

- The size and type of each input value
- How the data is entered (for example, input from a terminal, read from a tape or disk file, and so forth)
- The validity checks performed on this data

The processing specifications describe:

- The flow of the program
- The computational algorithms, if any
- The functions, subroutines, subprograms used, if any

- The dependencies on input and output
- The overall logic of the program

The more straightforward and structured your specifications are, the easier it is to code and test a program. Therefore, you should spend considerable time analyzing and designing your program and writing the specifications. Estimates vary on the amount of time to spend in overall design, coding, and testing; however, if you take the time to design and write detailed and clear specifications, then coding and testing is much simpler and faster.

BASIC has language facilities that can aid you in implementing a straightforward structured design. The structured programming constructs such as SELECT/CASE blocks, IF blocks, and DO/LOOP blocks all can help you construct programs that are easy to design, easy to code, and easy to maintain.

Writing BASIC Programs

Because you have determined the task or operation to be performed and designed the overall flow of your program, you can start entering statements into your workspace. Each statement is a programming instruction to the system to perform an operation.

Programs work with collecting, storing, processing, and printing of data. Before we get into the operations performed on data, it is appropriate to review the specific data types that are recognized by BASIC.

This chapter describes the different features of BASIC which will help you in the task of writing BASIC programs. It describes the kinds of data and the different data types you can use in BASIC. It discusses how to define and use arrays. Next it introduces you to intrinsic functions and then shows you how to write expressions followed by loops, branching, chaining, and comments. The chapter ends with a discussion of helpful hints for avoiding common programming errors.

Other topics which you need to know when writing BASIC programs are discussed in the following three chapters: "Performing Input/Output," "Handling Exceptions," and "Defining and Calling Subprograms."

Defining and Using Data Items

There are two major kinds of data you can use: constants and variables. A constant may not change its value during program operation and can therefore be explicitly represented by its value. A variable, however, must be able to assume more than one value, so it is represented by a name.

Constants and variables can further be divided into two types: numeric for quantitative values and character for string values, where a string is a sequence of characters.

Defining and Using Numeric Data

You can define and use four categories of numeric data: integer, decimal, real single, and real double. In BASIC they are referred to as numeric data types.

Use of Numeric Data Types

It is usually not necessary to choose a specific numeric data type but there are cases where considerations of precision, range, or performance can make it desirable to specify a type other than the default.

Decimal data allows the greatest range of any BASIC numeric data type and is more precise than real data. Decimal data is normally the default data type in BASIC and can be used in most cases. However, integer and real data provide advantages for special situations.

Integer data provides the smallest range of any numeric data type. However, integer data is more efficient for use as index values when the range of integers is sufficient, and may be used in loops when speed is important.

Computation using real data is more efficient than with decimal data because it uses the computer's floating-point format. If your program does lots of computation, use real data (or integer where appropriate) to allow your program to run faster.

When your BASIC program passes data to other language subprograms (like FORTRAN), you also save time by using real or integer data instead of decimal. Since decimal data is stored in a format that is usable only by BASIC, a decimal data item must be converted to real when it is passed to another language subprogram. The conversion step can be avoided if you define the data to be real or integer in your BASIC program.

Real single data should be used in preference to real double when storage space is a consideration.

Real double data should be used in preference to real single when greater precision is desired.

Integer Data

An integer is a positive or negative whole number. The integer data type is useful for index values (for arrays and loops) and other cases where whole numbers are normally used.

The range of integer data is:

```
-2147483648 to +2147483647
```

The above two numbers are the lowest and highest values that a BASIC integer constant or variable may have.

Integers are usually represented in integer notation. Integer notation consists of one or more digits with no decimal points, no commas, no spaces, and no exponent. Examples using integer notation are:

```
0
10
+25
-214
233378
```

Computations using integer data are faster than with other types of numeric data and require the least amount of storage.

See *IBM BASIC Language Reference* for information concerning the internal representation of integer data and the rules for data typing.

Decimal Data

The decimal data type can be used for all numeric data in BASIC. It is normally the default data type and the easiest to use.

Decimal numbers can be expressed in integer notation (see above), fixed-point notation, or floating-point notation.

Fixed-point notation consists of one or more digits with a decimal point but without an exponent. For example:

```
.05
-1.236
+3.8
4.
```

Floating-point notation allows you an easy method of entering very small or very large numbers. Floating-point notation consists of a number and an exponent. The number may be written with or without a decimal point and it is followed by the letter E (or e), followed by an integer which is the exponent. Using integer notation, you would write 5 million as follows:

```
5000000
```

With floating-point notation, you can also write 5 million in any of the following ways:

```
5E+6
5.00e+6
+5E+06
.05E+8
500E4
```

The positive exponent means that the decimal point should be moved to the right the number of places specified. (You may read the letter E as "times 10 to the power of.") In the last example above, 5 million is expressed as 500 times 10 to the power of 4.

Very small numbers are just as easily represented. For example:

```
5.0E-6
5e-6
+5E-06
.05E-4
500E-8
```

The negative exponent means that the decimal point should be moved to the left. All the above represent the value .000005, or 5 times 10 to the minus 6th power.

BASIC saves 17 significant digits of decimal data (an 18th digit is used for rounding).

The exponent can be in the range -999 to +999. Therefore, the largest absolute value is:

```
.99999999999999999E+999
```

and the smallest absolute value is:

```
.1E-999
```

Note: The largest absolute value will be rounded and displayed by the PRINT statement as:

```
1.E+999
```

The smallest absolute value will be displayed by the PRINT statement as:

```
1.E-1000
```

See *IBM BASIC Language Reference* for the internal representation of decimal data and the rules for data typing.

Real Data (Real Single and Real Double)

Real data is data that is stored in System 370 floating-point format, in hexadecimal. Because this is the floating-point format of the computer, computation using real data can be done more efficiently than computation using decimal. When you have a program that calls for lots of computation, using real data instead of decimal will allow your program to run faster.

If your program does *not* involve a large number of calculations, there is little reason to complicate things by using real data, with one exception. When your BASIC program passes data to other language subprograms or to GDDM, you may save time by using real data instead of decimal data. Since decimal data is stored in a format that is internal to BASIC, decimal data is converted to real if passed to another language subprogram. The conversion step can be avoided if you define the data as real in your BASIC program.

Real numbers can be represented in integer notation, fixed-point notation, or floating-point notation.

Real data may have single or double precision. Single takes half the space of double, but double is more precise.

Real data is not as accurate as decimal data and rounding errors may occur. For example, the value 0.1 cannot be exactly represented using real data. It may be displayed by a PRINT USING statement, differently for real and decimal data types:

```
0.100000023841857910   if type is REAL SINGLE
0.099999999999999992   if type is REAL DOUBLE
0.100000000000000000   if type is DECIMAL
```

Real Single: The real single data type is floating-point binary data with a precision of (approximately) 6 decimal digits and an exponent in the range of (approximately) -78 to +75.

Real Double: The real double data type is floating binary data with a precision of (approximately) 15 decimal digits and an exponent in the range of (approximately) -78 to +75.

See *IBM BASIC Language Reference* for the internal representations of real single and real double data types and the rules for data typing.

Numeric Constants

Numeric constants are written in BASIC in integer, fixed-point, or floating-point notation. For example:

```
743
7.43e2
-22.111
-2.2111E+01
.00671
314159265e-8
5878E109
```

The type assigned to a constant depends on the context and the options in effect. The type chosen determines the accuracy of its internal representation. See *IBM BASIC Language Reference* for details.

Numeric Variables

A numeric variable is a named numeric data item whose value is subject to change during program execution. The variable name must not duplicate a keyword and must start with a letter. (A keyword is a word which has a predefined meaning to BASIC and which BASIC uses to perform a specific function.) See *IBM BASIC Language Reference* for a list of keywords. Variable names entered in upper, lower, or mixed case are treated the same by BASIC.

Assigning a type to Variables: Normally, variables are typed by default or with typing statements: DECIMAL, INTEGER, REAL SINGLE, or REAL DOUBLE. However, the rightmost character of a numeric variable name may also be used to designate its type. The type designator character is provided for compatibility with other BASICs. A "%" indicates integer type. A "#" indicates decimal or a real double type, depending on the options in effect (see your *IBM BASIC Language Reference* manual for details). For example:

```
index_number%
```

is an integer variable and, if the IBM–supplied default is in effect:

```
tax_amount#
```

is a decimal variable.

When a variable name does not end with a type designating character, and the IBM-supplied default is in effect, BASIC assigns decimal type, unless that variable or its first letter has been explicitly mentioned in a REAL or INTEGER statement, described briefly in the following section.

Numeric Typing Statements

For the complete rules on specifying types, see *IBM BASIC Language Reference.*

The INTEGER statement declares any specific variable or any variable beginning with a specific letter as having an integer type. For example:

```
100 integer charley,(t,s)
```

declares CHARLEY as an integer variable, and it also declares that all variables whose first letter is T or S are integer types.

If the INTEGER statement does not contain a list of variables, it specifies that all variables not otherwise typed are integer types. For example:

```
400 integer
```

defines all numeric variables that are not typed with a final # character, or a DECIMAL or REAL statement, to be integer.

The DECIMAL statement declares any specific variable or any variable beginning with a specific letter as having a decimal type. For example:

```
460 decimal able,baker,(i,j,k)
```

declares ABLE and BAKER as decimal variables, and it also declares that all variables whose first letter is I, J, or K are decimal types.

If the DECIMAL statement does not contain a list of variables, it specifies that all variables not otherwise typed are decimal types. For example:

```
60 decimal
```

defines all numeric variables which are not typed by means of a final % or # character, or an INTEGER or REAL statement, to be decimal.

The REAL statement declares any specific variable or any variable beginning with a specific letter as having a real type. For example:

```
10 real david,(a,b)
```

declares DAVID as a real variable, and it also declares that all variables whose first letter is A or B are real types. If the REAL statement does not contain a list of variables, it specifies that all variables not otherwise typed are real types. For example:

```
20 real
```

defines all numeric variables that are not typed with a final % or #, or a DECIMAL or INTEGER statement to be real.

The precision (single or double) is determined by the options in effect. However, you can explicitly set the precision by including SINGLE or DOUBLE in the REAL statement. For example:

```
30 real single john, (q,v)
```

declares JOHN as a real variable with single precision, and it also declares that all variables whose first letter is Q or V, are real single types.

Numeric Operators and Expressions

Your BASIC program can perform addition, subtraction, multiplication, division, and exponentiation. These five operations are performed on operands. For example:

```
40 A=B+C
```

The operand (variable) B is added to the operand (variable) C, and their sum is assigned to variable A.

The plus symbol is the operator which determines what is to be done with the numeric values in the operands B and C.

Operands may consist of any valid numeric expressions, but in situations involving multiple operands with a variety of operator symbols there is a priority order in which the operations are performed. Figure 3 will clarify this for you:

Numeric Operator	Meaning	Priority
^ or ¬ or **	to the power of	highest
*	multiplication	middle
/	division	middle
+	addition	lowest
-	subtraction	lowest

Figure 3. Numeric Operators, Their Meanings and Priorities

BASIC processes the highest priority first and the lowest priority last. In the statement,

```
100 A=B**2 - C*N+X/Y
```

The order of operations is:

1. B**2 is calculated first.
2. Then C*N.
3. Then X/Y.
4. Then the result of C*N is subtracted from B**2.
5. Then the result of this is added to the result of X/Y.

In addition to the operator priorities, remember the left-to-right evaluation of expressions. A-B+C, for example, is evaluated as (A-B)+C and not as A-(B+C), as both + and - have the same priority and the calculation is based on the left-to-right evaluation. If you substitute numeric values for A, B, and C you can easily see the difference in the two calculations. For example:

```
(1-2)+3=2
1-(2+3)=-4
```

Using Parentheses: Use parentheses to clarify groups of expressions, not only to aid in readability, but also to ensure that the order of calculations is correct.

Parentheses receive top priority. When parentheses are nested within another set of parentheses, the operation in the innermost pair is performed first. You can see from the previous example that parentheses affect the order of calculations. If -4 was the required result, the expression 2+3 must be enclosed in parentheses.

You cannot follow an operator immediately with another operator. For example: B**-2. The -2 must be enclosed in parentheses as follows: B**(-2).

A more complex expression would be something like

```
((1+X)**Y-1)/(ABLE+((1+R)**(M-1)))
```

Look at the innermost parentheses first and notice that M-1 is an expression enclosed in parentheses to show that the result of adding 1+R is to be taken to the power of M minus 1. The expression (1+X)**Y-1 is different. Here (1+X) is taken to the Yth power and 1 is subtracted from the result of the exponentiation. Notice further that the placement of parentheses shows that the entire expression on the left of the division operator (/) is to be divided by the entire expression on the right of the division operator (/).

Programming rules for the evaluation of expressions are in *IBM BASIC Language Reference.*

Assigning Numeric Values — LET Statement

To assign values to numeric variables or array elements, use the LET statement (See "Defining and Using Arrays" on page 50 for a discussion of arrays and array elements). For example, to assign the value 14 to the numeric variable J, write:

```
20 let j = 14
```

or

```
20 j = 14
```

Either of these statements assigns the value 14 to the variable J. (Note that the keyword LET is optional.)

To assign the same value to a series of variables, you can write:

```
30 let j,k,l = 14
```

or

```
30 j,k,l = 14
```

The expression to the right of the equal sign can be any valid numeric expression. The following LET statements are all valid.

```
40 let a = b*3 + 2*n
50 a = b*3 + 2*n
60 c = 1+n
```

In each case, the expression to the right is evaluated to a single value, which then replaces the current value of the variable or array element on the left.

When the result type of the expression differs from the type of the receiving variable to the left of the equal sign, the result value is converted to the type of that variable. For example, if I is integer and A and B are decimal for

```
70 i= a/b
```

the decimal result of the division, rounded to the nearest integer, is placed in i, though the division operation is performed in decimal. If a is decimal and i and k are integer, for

```
80 a = i*k
```

a decimal value is placed in a, though the result of this multiplication is an integer.

Only the variable on the left-hand side is affected by the assignment. For example, in the program

```
10 A = 1
20 C = A + 2
30 A = 2
```

The execution of line 30 does *not* change the value of C (which will be 3 because of line 20) even though the value of A + 2 is now 4.

Defining and Using Character Data Items

Your BASIC program can define and use character strings either in constants or variables.

A character string is a sequence of characters that may include any character defined in the EBCDIC or ASCII character set. (See *IBM BASIC Language Reference* for the characters within each character set.) Every character string has a length which includes embedded spaces.

Character String	Length
S.S.N.	6
DOLLAR AMOUNT	13
FIELD1;FIELD2**	19

Character Constants

If you enclose a character string with either apostrophes or quotation marks, you are defining a literal character string or character constant. The quotation mark or apostrophe is a character string delimiter. For example:

```
"xyz456"
'Weather report number 7'
"true or false (y/n)?"
'    4 leading and two trailing spaces  '
```

each of the above is a character constant. All character constants have a length which is the total number of characters within the apostrophes or quotation marks. Each space within the apostrophes or quotation marks is counted as a character in the length.

Whichever character you choose as a delimiter (' or "), you must use it at the beginning of the character string *and* at the end.

Enclose character constants within apostrophes or quotation marks, except in response to INPUT statements or in DATA statements, where, under certain circumstances, you can omit the apostrophes or quotation marks.

If you want to define an apostrophe or a quotation mark within a character constant you must double it without intervening spaces. For an example, see Figure 4.

```
* print '"The Centurion shouted, ''Hail, Caesar!''"'

results in

"The Centurion shouted, 'Hail, Caesar!'"
```

Figure 4. Quotation Marks within Character Constants

Character Variables

You can define character variables as named data items of character data whose value is subject to change during the execution of the program. The variable name must end with a $, start with a letter, and must not duplicate a keyword. Also, variable names may be entered in upper, lower, or mixed case; characters entered in different case are treated alike by BASIC. For example:

```
result$
tenant_name$
yearend$
Date$
YEAR$
```

For numeric variables, you are concerned with type and precision. For character variables, you are concerned with length. Character variables or character array elements have two lengths: maximum length and current length.

Use the DIM statement or the COM statement to specify the maximum length of a character variable. For example:

```
100 dim name$*25
110 com name$*25
```

Both of these statements define the maximum length of NAME$ to be 25 characters. (For more detail, see "Explicit Dimensioning" on page 53.) The maximum length does not change during the execution of a program.

As distributed by IBM, the default length value is 18; however, your organization may establish a different default value. Check with your system administrator.

The current length of a character variable may change each time a different string is assigned to this variable. Therefore, the current length of character variables can change during program execution. The current length is the length of the expression being assigned. If the length of the constant or expression is greater than the maximum length of the variable, a string overflow (SOFLOW) occurs. SOFLOW is a type of exception condition that occurs when a character string is assigned a value that is longer than its maximum length. For example, SOFLOW would occur if you attempted to assign the character string "Jennifer" to A$, which had a maximum length of five characters, defined in the statement DIM A$*5.

Character Operators and Expressions

Suppose you are writing part of a program that deals with formatting a report printout and you want a heading which under certain circumstances says, "BID ESTIMATE WITH MARKUP", and under other circumstances says, "BID ESTIMATE WITHOUT MARKUP". You could merely write two conditional PRINT statements, one for each entire heading. But there is a way to use character assignment and the concatenation (&) operator to achieve the same result.

Note that the only difference between the two headers is the character string OUT. Now if you separate the headers into three character expressions and assign these to three variables, your printout task will be very easy, as follows:

```
10 let a$ = "BID ESTIMATE WITH"
20 let c$ = " MARKUP"
30 if flag = 1 then
40     b$ = "OUT"
50 else
60     b$ = ""
70 end if
80 print a$ & b$ & c$
```

prints BID ESTIMATE WITHOUT MARKUP if flag=1 and prints BID ESTIMATE WITH MARKUP otherwise.

Concatenation adds together the character expressions concatenated to one another by the ampersand (&) symbol. When there are several long expressions involved, you should be careful not to go past the page margin on a printout or cause a program exception condition by exceeding the maximum allowed length of the receiving character variable in an assignment statement.

Character Substrings

A character string is a sequence of characters. The substring capability allows you to extract, replace, or insert specified characters in a string.

The substring is specified by a qualifier of the form (m:n), where "m" represents the starting character number and "n" represents the ending character number.

For example, if the character variable *address$* consists of "street,city,state," then extracting various substrings has this result:

Statement	Result
10 a$=address$(1:6)	a$ equals "street"
20 a$=address$(8:11)	a$ equals "city"
30 a$=address$(13:17)	a$ equals "state"

In the appended qualifier (m:n), both m and n may be numeric constants, variables, or more complicated numeric expressions. The use of variables as substring qualifiers is convenient for cases requiring repetitive reference to different substrings such as the scanning of a character string to search for a matching substring. If numeric expressions are used as substring qualifiers, BASIC evaluates these expressions and rounds them to the nearest integer.

In the example above, the substring qualifiers are added to character names. But substring qualifiers can also be added to character constants or any other character expressions. For example, the following statements illustrate valid substrings:

```
10 p$ = "abcde"(2:3)                    !p$ = "bc"
20 a$ = "bc":b$ = "efghijklmn"
30 q$ = a$&b$(4:7)                      !q$ = "bchijk"
40 r$ = (a$&b$)(4:7)                    !r$ = "fghi"
50 s$ = ((a$&b$)(4:7))(2:3)             !s$ = "gh"
60 def user_def$(x$,y$) = x$&y$
70 t$ = user_def$('123','456')(3:4)     !t$ = "34"
```

A substring qualifier may not follow another substring qualifier unless the preceding expression is enclosed in parentheses.

```
10 a$ = b$(3:4)(4:4)    !this is illegal
20 a$ = (b$(3:4))(4:4)  !this is legal
```

Substring operations are useful when processing character strings or variables. Because the beginning and ending designators in a substring may both be numeric expressions, the substring operation can be very useful for scanning data. For example, if you wanted to scan the character string in address$ for the substring "state", you could use the following statements:

```
100 for i=1 to len(address$)-4
110 if address$ (i:i+4)='state' then found
120 next i
```

where FOUND is a line label.

Defining and Using Arrays

Suppose you are keeping statistics about the weather. You will probably have a lot of data, such as average temperatures, inches of rainfall, high temperatures, and so forth. If you write a program to keep all this data on record, it would be a good idea to arrange the data in some order; for example, you can keep the average temperature for each month together in one place and the rainfall for each month together in another place.

With BASIC, you can keep groups of similar data together by organizing them into arrays. Arrays group data into categories; within each array are individual members that compose the actual data for that category.

In the weather example, suppose we wanted to maintain three sets of data:

- The names of the months (MONTH$)
- The average temperature for each month, which is assumed to be integer for this example (TEMP)
- The total rainfall for each month, which is assumed to be decimal for this example (RAIN)

We can look at this data in one of two ways:

- Three separate arrays, as shown in Figure 5 on page 52.
- One array in two dimensions, which is assumed to be decimal for this example, as shown in Figure 6 on page 52.

The WEATHER table is really a combination of two of the lists (temperature and rainfall). Notice that temperature is carried as a decimal value rather than an integer as all elements in an array must be the same type. Because you can read in two directions, it is called a *two-dimensional array*. The MONTH$ array cannot be combined with temperature and rainfall as it is a character array.

It will be a lot easier to work with the weather data if we keep it together in three one-dimensional arrays, or one two-dimensional array and one one-dimensional array, than it would be if we considered it as 36 separate variables. This section shows you how to work with arrays in BASIC programs.

Defining An Array

When you want to work with an array, you must first tell the system that you are working with an array and not with ordinary variables. This is called *defining* your array.

Defining the array merely involves telling the system how big it is going to be (so the system can allocate room for it), and telling the system what kind of data will be in it. (Later on, you fill in the data, but this is not part of defining the array.) The data for your arrays can be made up of numeric data or character data. If the data is character data, each item must have the same maximum length. You can define an array to have either kind of data in it, but it must have only that kind of data in it (notice how the data is defined in Figure 6). You can not mix characters and numbers in a single array, nor can you mix decimal, real double, real single, and integer items.

Each variable within an array is called an element of the array. You can refer to a particular element by giving its position within the array. These position numbers are called subscripts.

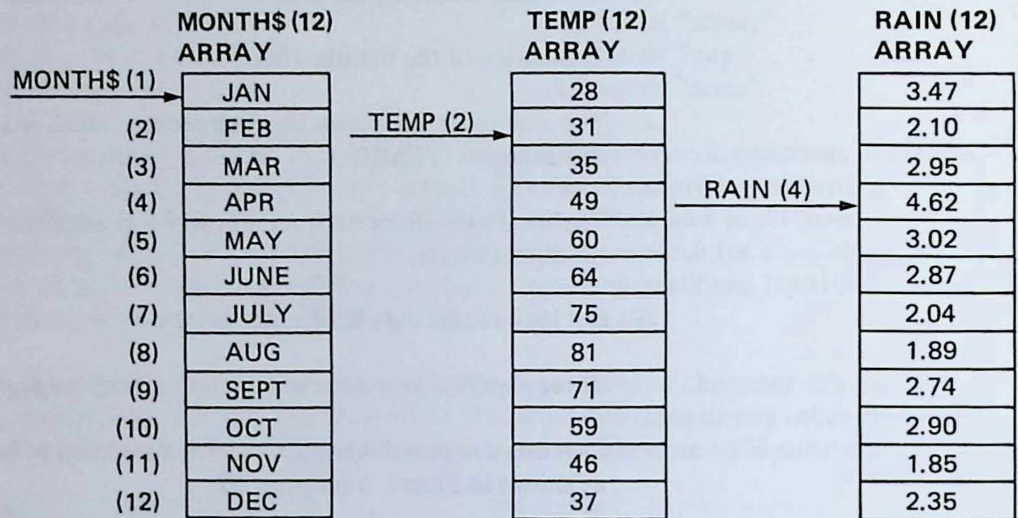


Figure 5. One-Dimensional Array Examples

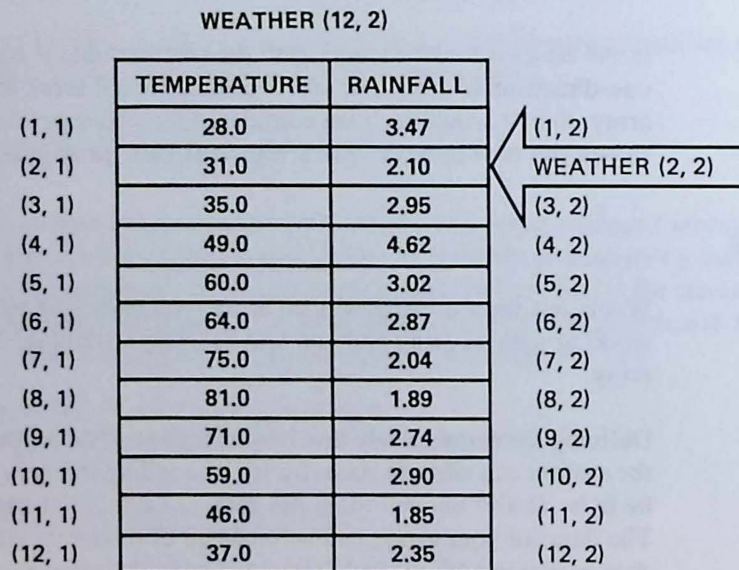


Figure 6. WEATHER Two-Dimensional Array Example

Dimensioning — OPTION And DIM Statements

An array can have one to seven dimensions, with each dimension having a specified size. Array subscripts can begin with zero (BASE 0 option), or one (BASE 1 option). If you omit the OPTION statement, then the lower bound will be the default value. It is usually zero; see *IBM BASIC Language Reference*. The total number of elements in any array is determined by the presence or absence of a BASE option in an OPTION statement. For example:

```
temp(5)
```

If the BASE 1 option is in effect, this refers to the fifth element of TEMP. If the BASE 0 option is in effect (the default), this refers to the sixth element of TEMP.

The size of a one-dimensional array is the total number of sequential elements. The size of a two-dimensional array is the number of elements in one dimension multiplied by the number of elements in the other dimension. For example:

```
100 option base 1
200 dim weather (12,2)
```

defines WEATHER as a 2-dimensional array, the first dimension containing 12 elements, the second dimension containing 2 elements. Therefore, the array contains 24 elements. If you omit the OPTION statement and BASE 0 is the default, the first dimension of WEATHER has 13 elements and the second has 3 elements, and the array has 39 elements. Each dimension is subject to the BASE option.

Explicit Dimensioning

Using the DIM statement, you can explicitly define the size and type of numeric arrays, and the size, type, and maximum length of character arrays. The COMMON (also spelled COM) statement allows you to explicitly define arrays as in the DIM statement. Furthermore, COMMON allocates these arrays and variables in a common area, which provides a means of passing values between program units (main programs and subprograms) and between programs during a CHAIN.

Both DIM and COMMON statements use integer constants within parentheses to define the upper bounds for each subscript in an array. If there is more than one dimension, the dimensions are separated by commas. For example (with BASE 0 option):

```
100 dim aray(20), name$(20,100)
150 com x(500)
```

ARAY is a one-dimensional numeric array containing 21 elements.

NAME$ is a two-dimensional character array containing 21 rows with each row containing 101 columns.

X is a one-dimensional numeric array containing 501 elements.

Typing is determined as described for "Character Variables" on page 48 and "Numeric Variables" on page 43.

The elements of a character array all have the same maximum length. This length can be declared in a DIM or a COM statement. For example:

```
100 dim name$(5)*20
```

declares that each of the 6 elements in the character array NAME$ has a maximum length of 20 characters.

If you don't specify a maximum length, the default value is assumed. See "Character Variables" on page 48.

Implicit Dimensioning

You can refer to an array in a program without declaring it in a DIM or COM statement. This is called implicit dimensioning.

Such an implicit array is automatically dimensioned to an upper limit of 10 for each dimension specified in the first usage of the array name. If the BASE 1 option is in effect, each implicit dimension has a range of 10 elements; if the BASE 0 option is in effect, the range of each dimension is 11 (0 through 10).

Implicit dimensioning can be convenient when your array requires dimensions that are 10 or 11 elements long. In other cases, you must use explicit dimensioning.

Using Subscripts To Access Arrays

If the BASE 0 option is in effect, you refer to the first element of array RAIN as RAIN(0), the second element as RAIN(1), and so forth. If you specify the BASE 1 option, then the first element is RAIN(1) the second element RAIN(2), and so forth.

You can also use variables and expressions as array subscripts. You must ensure, however, that the expression never evaluates to a number that exceeds the upper limit implicitly or explicitly defined or is less than the lower limit (one for BASE 1 or zero for BASE 0). BASIC generates an exception if the subscript is not within these limits. And, unless you include exception handling statements for this in your program, this exception halts execution of the program. See "Handling Exceptions" on page 129.

To access the 5th element of the 3rd row, with the reference WEATHER(K,L), and if the BASE 1 option is in effect, the subscripts K and L must evaluate to 3 and 5, respectively.

You can use expressions as array subscripts. For example, you could define a tax table as the array TAX(2,10,20). In this array, a particular rate could be found by the statement:

```
140 let rate = tax(status,dependents,log10(income))
```

where the variable STATUS is 1 for single and 2 for married, DEPENDENTS is the number of dependents, LOG10 is an intrinsic function, and the expression LOG10(INCOME) evaluates to the nearest rounded integer of a tax bracket. (You probably would find the LOG10 of income to be inappropriate for use in a practical tax situation.) Notice that LOG10 is a keyword, but its use in this example is correct, as it describes a function, and it is not the name of a variable.

As the above example shows, you can use functions or other complex expressions as subscripts. This is convenient, but you should be careful when you use them. When you use a subscript expression, make sure it will fall within the lower and upper bounds.

An array subscript may also be an element of another array, provided, of course, that this reference is to a numeric array. For example:

```
10 value = z(y(5,3))
```

where Y(5,3) is an element of a numeric array named Y.

Subscripting Character Arrays

A character array is subscripted in the same way as numeric arrays, and you can use character arrays in a variety of situations that require repeated or selective generation of character data.

For example, you could dimension a character array,

```
10 option base 1
20 dim day$(7)*10
```

where DAY$ is an array of seven elements, each with a maximum length of 10 characters. Then you could fill it with the names of the weekdays using either assignment statements or DATA and READ statements, and select any particular day by designating the appropriate subscript of the array. If you had started with Monday as the first day of the week, then;

```
30 print day$(1)
```

would print "Monday", and

```
40 print day$(2)
```

would print "Tuesday".

Using the numeric variable D to represent the number of the day that is calculated by some other part of the program,

```
50 print day$(d)
```

would print the name of the day based on the calculated number of the day.

You can use substring qualifiers with character array elements as with character variables. Referring to the previous example:

```
90 print day$(2)(1:4),day$(6)(1:3)
```

would print:

```
Tues             Sat
```

Using Array Expressions

Any programming task that entails moving groups of data is probably tedious if the language demands that each element of the group must be individually specified in the action statement.

BASIC's array operations provide an easy way to transfer data to and from arrays and also to and from files, without the necessity of writing many assignment or input/output statements.

The convenience of using array expressions is obvious in the following examples.

If every element of the array A(2,10) must be filled with a specific value X, you could use a nested loop:

```
410 for i = 0 to 2
420   for j = 0 to 10
430     let a (i,j) = x
440   next j
450 next i
```

However, the MAT statement

```
410 MAT a=(x)
```

does it all!

(See "FOR/NEXT Blocks" on page 70 for a discussion of FOR/NEXT loops.)

Assigning Array Values — MAT Statements

The MAT statement allows you to perform operations immediately on entire groups of data. This can simplify your programming task. For example:

```
410 mat a = (x)
```

performs the same operation as the five statements above.

The expression to the right of the equal sign may also be an array function. For example:

```
985 mat a = trn(a)
```

fills the array A with the transpose of itself. (You are allowed to use the same array name on both sides of an assignment.)

You can also perform arithmetic operations with arrays using either individual assignment statements or MAT.

Array Addition and Subtraction

Adding or subtracting two numeric arrays gives you a resultant array whose every element is the sum or the difference of the corresponding elements of the other two arrays. If these arrays are moderately large, you can see the distinct advantage of a MAT statement over individual assignment statements.

For example, assume that you want each element of array A to be the sum of the corresponding elements of B and C. You could write:

```
80 option base 0
90 dim a(4), b(4), c(4)
100 a(0) = b(0) + c(0)
110 a(1) = b(1) + c(1)
120 a(2) = b(2) + c(2)
130 a(3) = b(3) + c(3)
140 a(4) = b(4) + c(4)
```

or, much more efficiently:

```
80 option base 0
90 dim a(4), b(4), c(4)
100 mat a = b + c
```

Array Scalar Multiplication

Scalar multiplication merely multiplies each member of an array by the same number. For example:

```
100 dim x(20),y(20)
110 mat x = (3.2)*y
```

multiplies every element in array Y by 3.2 and stores the result in the corresponding elements of array X.

Array Matrix Multiplication

Matrix multiplication behaves according to row and column rules that you will remember only by using them often enough. Each element of the resulting array is the dot (inner) product of the row in the first array multiplied by the same column of the second. For example:

```
100 option base 1
110 dim a(3,3),b(3,2),c(2,3)
300 mat a = b * c
```

has the same effect as the following scalar assignment statements if B and C have the values shown in Figure 7 on page 58.

```
300 a(1,1) = b(1,1) * c(1,1) + b(1,2) * c(2,1)
310 a(1,2) = b(1,1) * c(1,2) + b(1,2) * c(2,2)
320 a(1,3) = b(1,1) * c(1,3) + b(1,2) * c(2,3)
330 a(2,1) = b(2,1) * c(1,1) + b(2,2) * c(2,1)
340 a(2,2) = b(2,1) * c(1,2) + b(2,2) * c(2,2)
350 a(2,3) = b(2,1) * c(1,3) + b(2,2) * c(2,3)
360 a(3,1) = b(3,1) * c(1,1) + b(3,2) * c(2,1)
370 a(3,2) = b(3,1) * c(1,2) + b(3,2) * c(2,2)
380 a(3,3) = b(3,1) * c(1,3) + b(3,2) * c(2,3)
```

After the operation is completed, array A contains the following values:

```
a(1,1) = 2 *  8 + 3 * 11 =  49
a(1,2) = 2 *  9 + 3 * 12 =  54
a(1,3) = 2 * 10 + 3 * 13 =  59
a(2,1) = 4 *  8 + 5 * 11 =  87
a(2,2) = 4 *  9 + 5 * 12 =  96
a(2,3) = 4 * 10 + 5 * 13 = 105
a(3,1) = 6 *  8 + 7 * 11 = 125
a(3,2) = 6 *  9 + 7 * 12 = 138
a(3,3) = 6 * 10 + 7 * 13 = 151
```

The results are shown graphically in Figure 7.

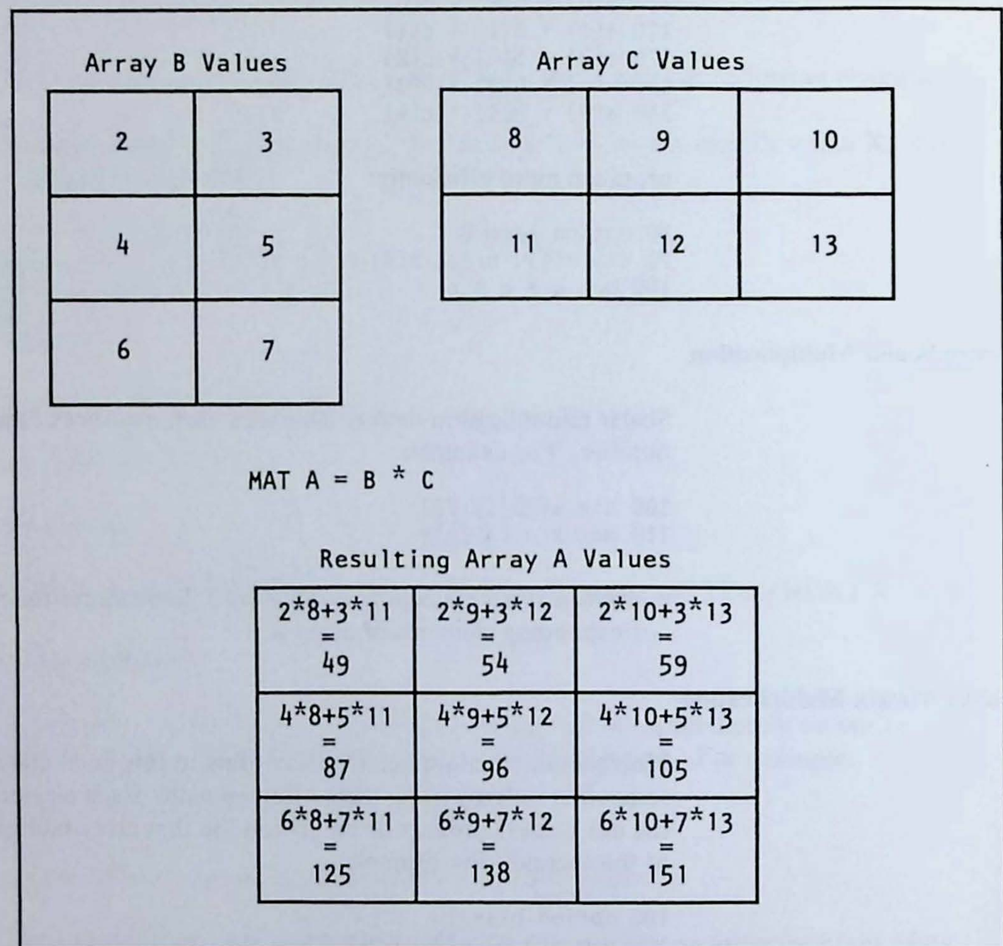


Figure 7. Matrix Multiplication Example

In all numeric operations, remember that the result of the rightmost expression is converted to the type of the left.

Redimensioning Arrays

You can redimension an array in either the size of each dimension or the number of dimensions as long as the total number of elements does not exceed the original total number of elements. (See *IBM BASIC Language Reference* for more detail.)

Use the MAT statement to restructure (redimension) numeric and character arrays. For example:

```
100 option base 1
110 dim alpha (2,4)
```

You can restructure the ordering of the elements of ALPHA using the MAT statement:

```
400 mat alpha = alpha (4,2)
```

changes ALPHA from a 2-by-4 array to a 4-by-2 array.

Figure 8 shows how restructuring works.

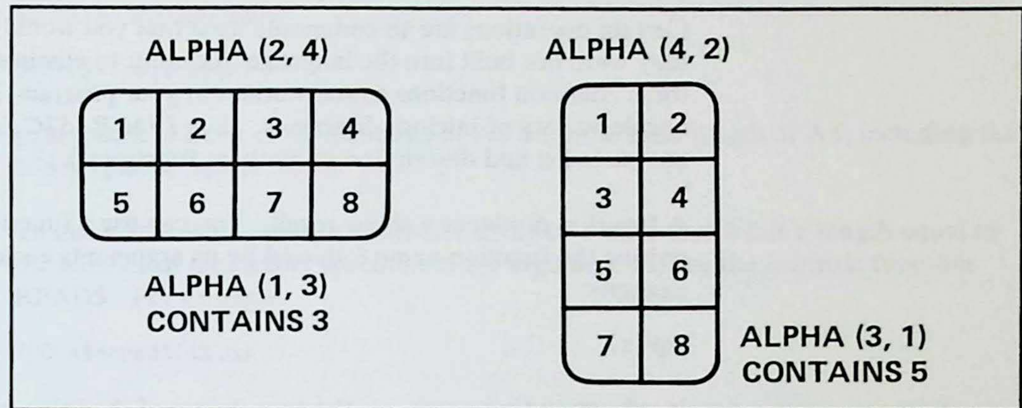


Figure 8. Array Redimensioning Example

You have not increased the total number of elements, but you have changed the size of each dimension and therefore each subscript range. As long as you don't exceed the original total number of elements, you can change the number of dimensions. For example:

```
500 mat alpha = alpha(2,2,2)
```

changes ALPHA from a two-dimensional array with eight elements to a three-dimensional array with eight elements.

When you redimension an array to a smaller number of elements than the original, the excess elements of the original are not accessible.

It is valid to combine certain array operations with redimensioning. Assume that BETA is a 4-by-2 numeric array, then:

```
550 mat beta = zer(2,2,2)
```

changes BETA from a 4-by-2 array to a three-dimensional 2-by-2-by-2 array and also sets all the elements in the BETA to 0, because ZER is an array function that sets each element of an array to 0.

Defining and Using Functions

Functions define a specific operation and produce a single result. You can use a function anywhere in your program that you can use constants, variables, or array element references. There are two types of functions: intrinsic and user defined.

Using Intrinsic Functions

Certain operations are so commonly used that you would be seriously affected if they were not built into the language. In order to eliminate the chore of writing these common functions as subroutines in your program, BASIC supplies you with a wide variety of intrinsic functions. (See *IBM BASIC Language Reference* for a complete list and description of all these functions.)

A function produces a single result. You can use a function in your program by writing the function name followed by its arguments enclosed in parentheses. For example:

```
sqr(x)
```

is a function that computes the square root of the value contained in X. X is referred to as the argument of the SQR function.

Some functions have more than one argument. You must always provide the required number and type of arguments when you use that function in your program. Multiple arguments are separated by commas. For example:

```
max(x,y,z)
```

MAX returns the largest (algebraically) of its arguments.

You can also use expressions or variables in the argument list of a function. For example:

```
sqr(2*4)
```

the expression 2*4 is the argument to the function SQR.

Functions can be used in expressions, for example:

```
5*sqr(16)
```

Functions can be used as arguments; for example:

```
cos(sqr(64)/8-1)
```

The function SQR is part of an expression which is the argument to the function COS.

Some functions that can help you manipulate character strings are described below. As an example, if r$= "help" the statement

```
100 F = len(r$)
```

places 4 in the variable F, which is the *current* length of the character string in R$ when the LEN function is executed.

Character fields are often easier to manage if they are standardized to some predetermined length. This means you will want to have a quick way to add spaces in front of or after character groups whose lengths are less than the predetermined length.

To insert spaces in front of a character string to ensure that it has a length equal to the number of characters specified in the argument M, use the intrinsic function LPAD$. For example:

```
340 a$=lpad$(a$,m)
```

puts spaces in front of the contents of A$ until the total length of A$, including the added spaces, equals M.

To insert spaces following a character string to ensure that it has a length equal to the number of characters specified in the argument M, use the intrinsic function RPAD$. For example:

```
700 a$=rpad$(a$,m)
```

puts spaces to the right of the contents of A$. In either of the above, if M is less than or equal to LEN(A$), spaces are not added.

To replace an embedded character string with another character string, use the function SREP$. This function can be used to change any misspelled words in a stream of text. For example:

```
430 a$=srep$(a$,m,b$,c$)
```

searches character strings in A$ and replaces one embedded group of characters with another. The function starts looking at position M in A$ for all nonoverlapping occurrences of the character group B$, and replaces it with the group C$. Search and replace is a basic requirement of text processing programs.

For example: the typographical error represented by a$="reclository."

```
620 a$=srep$(a$,3,"cl","p")
```

changes the "reclository" in A$ to "repository."

For clarity, the example shows only one replacement occurrence, but in a longer character string there could be more than one replacement occurrence. This would be the case if the same word were misspelled several times in a portion of text. (For a complete description of all the intrinsic functions, refer to *IBM BASIC Language Reference*.)

Most functions can also be used in the MAT statement to apply the function to every element of an array. For example, to call a FORTRAN routine it is a good idea to have all elements of a character array be of the same length. This can be easily accomplished using the RPAD$ function. For example:

```
780 mat a$ = rpad$(a$,10)
```

would pad every element of a$ with spaces to a length of 10. See *IBM BASIC Language Reference* for more details.

Defining and Using Your Own Functions

Intrinsic functions are compact and already available but they are necessarily limited to some reasonable collection of functions which will probably be frequently needed by the majority of users.

If you want to create a function in your program not included in the intrinsic group of functions, you may do so, provided that you completely define the function. There are two ways to accomplish this, and both use the keyword DEF.

Single-Statement Functions

A single-statement function completely defines the function in the space of one program statement by using an equal sign to assign the value of the function to the function name. For example:

```
200 def poly(x) = x**3+x**2+x
```

POLY is the name of the function. Function names may be any valid numeric or character names. The "X" inside the parentheses is the parameter for this function. If more than one parameter is needed to compute the function value, then the parameters must be separated by commas to form a parameter list; for example: (N,R,W).

To use the single-statement function defined above, you could write:

```
200 let y = poly(a)
```

The POLY(A) refers to the function named POLY. "A" is the argument that the function is to operate on. If A has the value 2 when the function is called, then Y would become A**3+A**2+A or 8+4+2 or 14.

Single-statement functions lend themselves well to formulas and expressions that fit easily on one line.

Multi-Statement Functions

Multi-statement functions also begin with the DEF statement, the name of the function, and a parameter list, but without an equal sign on the same line. Instead, the function is defined on one or more statements which are called the "body" of the function:

```
100 def    abc(x)
110 abc=1
120 if x<>1 then abc=4
130 fnend
```

When this function is executed, it returns a value of 1 if X equals 1; otherwise, it returns the value of 4.

To end a multi-statement function, you must write an FNEND statement.

Using Your Own Functions

When you use a function, the list of arguments following the function name must agree in number and order with the list of parameters in the DEF statement. In addition, a character argument must correspond to a character parameter and vice versa. Numeric arguments will be converted to the type of the corresponding numeric parameter. For example:

```
100 integer v,p,k,l : decimal t,j,fnmix
110 def fnmix(n$,t,v,p)=len(n$)+t-v/p
120 let name$ = "merlin"
130 let j = 54.3
140 let k = 4
150 let l = 20
160 print fnmix(name$,j,k,l)
```

The PRINT statement at line 160 uses the function FNMIX; the arguments NAME$, J, K, and L agree with the parameter list in the DEF statement at line 110. Statement 160 prints 60.1 (the length of "MERLIN" is 6, added to 54.3 is 60.3, which becomes 60.1 when 4/20 is subtracted).

For character functions in which the maximum length of the returned value is required, write an asterisk and a length immediately following the character function name. For example:

```
330 def fna$*10(a$,b$) = a$ & b$
```

restricts the result of the concatenation to a maximum length of 10 characters. If the maximum length is not specified, the default maximum length is assumed (see "Character Variables" on page 48.)

When invoking your function, omit the length restriction. For example:

```
 * list
100 q$ = "minne"
110 r$ = "sota"
120 def fna$*9(a$,b$) = a$ & b$
130 print fna$(q$,r$)
140 end
 * run
minnesota
END AT LINE 140.
```

Comparing and Combining Expressions

When you write a program with decision making capability, you will be writing statements that involve relational and logical expressions. Rudimentary programs need to compare and combine expressions, and most programs make extensive use of relational comparisons between values.

A relational expression compares the values of either two numeric expressions or two character expressions to determine if the statement is true or false.

Logical expressions allow the combination of relational expressions using AND, OR, and NOT to determine if the combined relational expressions are true or false.

Relational and logical expressions can be used in DO, EXIT IF, IF, and LOOP statements. CASE statements also allow an abbreviated form of relational expressions.

Relational Expressions

Two simple numeric or character expressions with a relational operator between them form a relational expression. (A relational expression using arrays is not valid.) Figure 9 lists the relational operators and describes their meanings.

Relational Operator	Meaning
= or EQ	equal to
<> or >< or NE	not equal to
>= or => or GE	greater than or equal to
<= or =< or LE	less than or equal to
> or GT	greater than
< or LT	less than

Figure 9. Relational Operators and Their Meanings

For example:

```
a = b   can be read as: a equals b
a NE b  can be read as: a is not equal to b
a <= b  can be read as: a is less than or equal to b
```

It is the values of the variables that are being compared in each case, not the names of the variables. The "truth" or "falsity" of each relation depends entirely on the values of the variables at the time your program makes the comparison. If you assign a value of 5.7 to A and a value of 5.7 to B then the relation A = B is true, and the relation A NE B is false.

The relational operators you use most often with character expressions are equal to and not equal to. However, this is not a restriction. You can use any relational operator in a comparison between two character expressions. For example:

```
"ab" = "ab"
"ab" < "abcd"
"xy" > "ab"
```

All these relations are true. (See *IBM BASIC Language Reference* for a discussion of characters and collating sequences.)

Be careful when comparing numeric values for equality or inequality. This is especially true of real data. For example,

```
.1 + .1 + .1 + .1 + .1
```

will not exactly equal .5, if computed using real data, because of limitations in accuracy.

Logical Expressions

Many practical situations impose a series of conditions that must be met before a certain action will be taken.

"To buy this house, you must have a down payment AND qualify for a mortgage loan." In this case, both conditions must be true before you can buy the house.

"I can accept your check after I see your driver's license OR a credit card." In this case, if either condition is true, your check will be accepted.

Application programming tasks must take into account many such real-life situations.

A payroll check, for example, is only to be printed when all of the following conditions are satisfied:

1. The employee is a valid employee.
2. You have correctly calculated the gross pay based on salary or wages.
3. You have calculated all the appropriate taxes and subtracted them from the gross amount.
4. You have sufficient money in the bank to cover the check.

In BASIC, you have the operators NOT, AND, and OR to combine relational expressions into logical expressions.

AND Logical Operator

AND specifies that two relational expressions must be true for the entire statement to be true. For example:

```
a = b AND c = d
```

A must equal B, and C must equal D, for the logical expression to be true.

OR Logical Operator

OR specifies that either of two relational expressions must be true for the logical expression to be true. For example:

```
a = b OR c = d
```

If either A equals B, and/or C equals D, the logical expression is true.

NOT Logical Operator

There will be times when you will need to test for the inverse of a situation:

"If I have NOT cheated, then I deserve the trophy." For me to take possession of the trophy, the supposition of cheating must NOT be true.

NOT can be used to reverse the meaning of a single relational expression or an entire logical expression.

If you use NOT you may want to use parentheses to ensure you are reversing the correct expression(s). For example:

```
not (a=b) and c=d
```

If A is not equal to B, and C equals D, the expression is true.

The following is equivalent to the above, but is more likely to be misinterpreted by someone looking at the expression:

```
not a = b and c = d
```

To reverse the meaning of an entire logical expression, you can enclose the expression in parentheses and place the NOT operator in front of the leftmost parenthesis. NOT then reverses every operator within the parentheses.

```
not (a=b and c=d)
```

has the same meaning as

```
a NE b  or  c NE d
```

the equal signs were reversed to not equal and the AND was reversed to OR. The above is not always true, however, because of the priority rules. See "Priority of Evaluation" below.

Priority of Evaluation

To predict the outcome of longer logical expressions you need to know the order in which the expressions are evaluated. The order is as follows:

Operation	Priority
Arithmetic or character expressions	highest
Relational expressions	|
NOT	|
AND	v
OR	lowest

If the logical operators are of equal priority, they are evaluated on a left-to-right basis. For example:

```
x = y AND b < c AND D > e
```

Is evaluated as follows:

1. The relational expression X = Y is evaluated.
2. The relational expression B < C is evaluated.
3. The relational expression D > E is evaluated.

4. All the relational expressions must be true for the logical expression to be true. If any relational expression is false, the logical expression is false.

However, parentheses override the priority of operations, just as they do in numeric expressions. For example:

```
(tally>limit or tally<xero) and x<>y
```

is evaluated as follows:

1. The relational expressions TALLY>LIMIT, and TALLY<ZERO are evaluated.

2. The OR operator is evaluated.

3. The relational expression X<>Y is evaluated.

4. The AND operator is evaluated.

If there were no parentheses, the AND operation would take higher priority than the OR.

If priority is done explicitly with parentheses then the rule for reversing AND and OR when applying NOT (mentioned in the previous section) is correct. For example:

```
not (a = b and c = d or e = f and g = h)
```

is not equivalent to

```
a NE b or c NE d and e NE f or g NE h
```

because parentheses were not added appropriately. This difference would be especially great if all were equal except G and H. The correct equivalent expression is

```
(a NE b or c NE d) and (e NE f or g NE h)
```

Controlling Program Flow

Application programs seldom are so short and simple as to require no program flow other than the default sequence of progressing straight down the list of statements from first to last.

The first programs a new programmer writes might be of this type, but you will soon see the advantages of being able to have the program automatically jump from one statement to another depending on conditions set up by some previous action or new data.

You could, of course, write a very long program with no flow control in it, but the program would not have much "intelligence," that is, it would have little decision making capability other than that provided by IF statements without branching. But equally important would be the difficulty of "structuring" such a program.

You can use structured programming to help you keep your program simple, easy to understand, and easy to maintain. BASIC provides statements such as the DO/LOOP block, the FOR/NEXT block, the IF/ELSE block, and the SELECT/CASE block which you will find helpful when preparing structured programs. These statements are described in the following sections.

Using Program Loops

Looping is one way of controlling program flow. Looping is the repetitive execution of the same group of statements until a particular condition is met. A variety of statements can be placed within a loop; what remains constant is the repetitive action. The DO and the LOOP, or the FOR and the NEXT, are demarcations between the remainder of the program and the block of statements to be repeated.

DO UNTIL/LOOP Block

To illustrate the use of a DO loop, assume you need a program module that will go through a long list of employee numbers (in character format) in an array until a certain given number (GIVEN$) is matched. A DO loop can structure this subtask as follows:

```
100 sbs = 0
110 do until given$=en$(sbs)
120    sbs=sbs+1
130 loop
```

GIVEN$ contains the employee number to be matched and EN$ is the array containing each employee number.

The statements between DO UNTIL and LOOP are repeated until the expression associated with the UNTIL clause is true. Control then passes to the next statement after LOOP.

If the condition is true the first time that GIVEN$ is tested, the loop is not executed.

If you want the test to take place *after* the loop is executed, you can let the LOOP statement do the controlling, by coding the UNTIL clause in the LOOP statement:

```
100 sbs=0
110 do
120    sbs=sbs+1
130 loop until given$=en$(sbs-1)
```

Note that in this case the loop will be executed one more time before the expression associated with the UNTIL clause is true.

You can actually code an UNTIL clause or a WHILE clause (discussed in the next section) in *both* the DO and LOOP statements, as shown in Figure 10 on page 69.

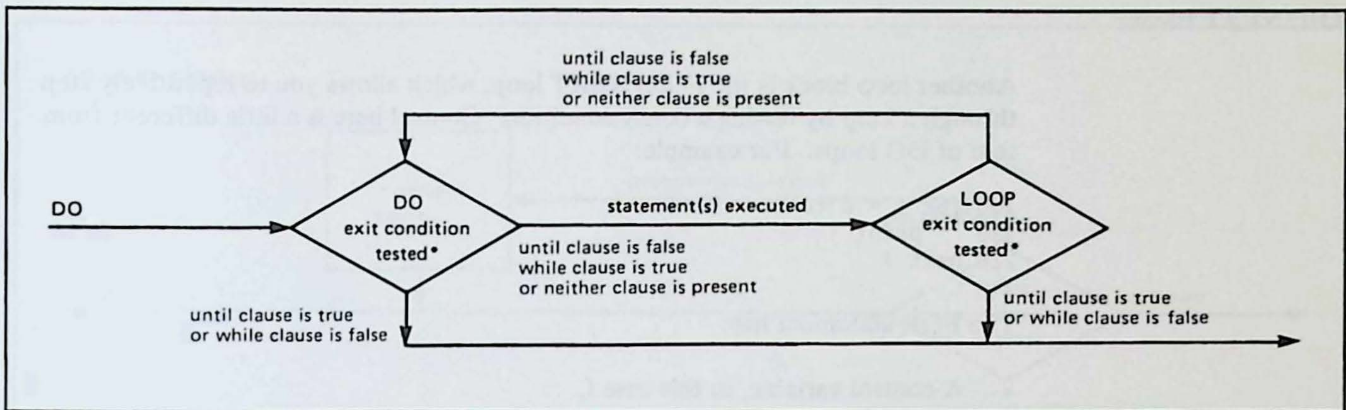


Figure 10. DO/LOOP Blocks

DO WHILE/LOOP Block

You may find it more logical in some circumstances to use WHILE instead of UNTIL. The statements between DO WHILE and LOOP are repeated as long as the logical expression following the WHILE clause is true. For example, in

```
100 do while a>10.5
110     a = a-1
120 loop
```

the statement A = A-1 is repeated as long as A is greater than 10.5. When A is less than or equal to 10.5, control is transferred to the statement following LOOP. If A is less than or equal to 10.5 when the DO statement is executed, control immediately passes to the statement following LOOP and the statement A = A-1 is not executed.

A variation is to let the LOOP statement do the controlling. If you code the WHILE clause on the LOOP statement rather than on the DO statement, the condition is tested after the statements within the loop are executed instead of before. Thus, the statements within the loop will be executed at least once. If a WHILE clause is coded on the DO statement, there is no guarantee that the loop will be executed once.

```
100 do
110    a=a-1
120 loop while a>10.5
```

When the DO loop is executed, the statement A=A-1 is executed regardless of whether A is less than or equal to 10.5.

In either case, the logical expression associated with the WHILE clause must be false before the exit condition is met.

You can code a WHILE clause in *both* the DO and LOOP statements, as shown in Figure 10 on page 68. This gives you flexible and powerful control of looping.

FOR/NEXT Blocks

Another loop block is the FOR/NEXT loop, which allows you to repetitively step through a loop by testing a count condition. Control here is a little different from that of DO loops. For example:

```
100 for i = a to b
110    print i
120 next i
```

The FOR statement has:

- A control variable, in this case I,
- An initial expression, in this case A, and
- A final expression, in this case B.

When the loop is entered, I is set equal to A.

Each time through the loop, I is increased by 1.

When I tests as greater than B, the loop is exited and the statement after NEXT I is executed.

Notice that the control variable I is used in both the FOR and the NEXT statements. The control variable must be a numeric variable.

The initial and final values may be numeric variables or any other valid numeric expressions. For example:

```
100 for j = c*3+d to (x+y)/z
```

The initial value is the expression *c*3+d* and the final value is the expression *(x+y)/z*.

The flow of control in a FOR/NEXT loop is shown in Figure 11 on page 71.

STEP Clause — FOR Statement: The optional STEP allows you to increase or decrease the control variable by values other than one. For example, if you wanted to print numbers from 100 to 0 in steps of 10 for the ordinate of a graph, you could code:

```
100 m = -10
110 for j = 100 to 0 step m
120    print j
130 next j
```

m, the expression immediately following the STEP clause, is the increment. It can be negative, as in this example, or it can be positive. However, to avoid unforeseen results, you must ensure that the positive or negative increment is valid for the range of the initial and final values (a positive increment in this example would not be meaningful).

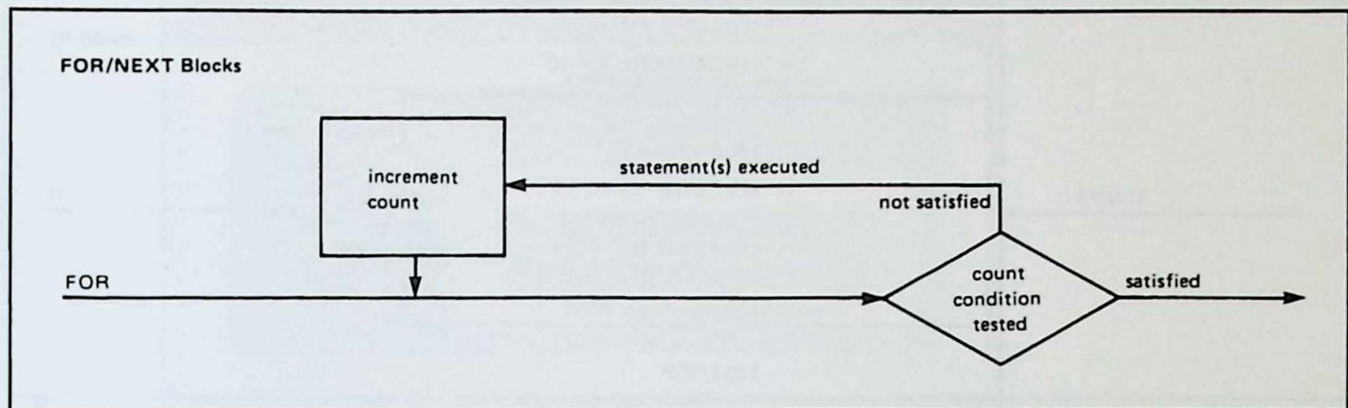


Figure 11. FOR/NEXT Blocks

Loop Considerations

Control can enter a loop in your program only at the DO or FOR statement.

You can transfer control out of a loop with a GOTO statement, but you cannot transfer control from outside a loop to a statement within the body of one of these loops. You can, however, use functions or you can call subroutines from within a loop, because control leaves the loop and returns without going anywhere else (except perhaps to another subroutine or function).

You can also use the EXIT IF statement to transfer control out of a loop. For example:

```
380 do while a<10
390     a=a+1
400 exit if a>b
410 loop
```

transfers control out of the loop when the value in A is greater than the value in B.

When control is transferred out of a FOR loop, the value of the FOR control variable remains at its current value.

Using Nested Loops

As long as you keep the DO and LOOP, or FOR and NEXT statements in matched pairs you can use loops within loops, as shown in Figure 12 on page 72.

The behavior of such "nested" loops is to satisfy the innermost before each successive outer loop is processed.

In this example, the inner B loop is processed 10 times each time the outer A loop is processed.

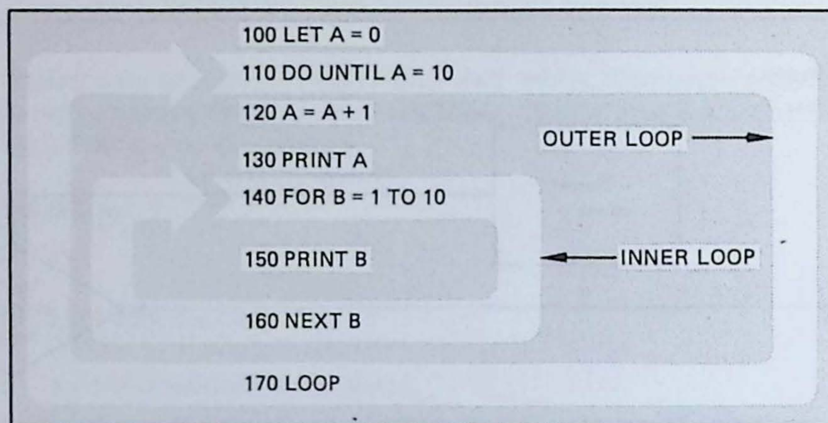


Figure 12. Nested Loops

IF/ELSE Blocks

An IF block allows you to enclose any group of statements in a single block to be processed when the IF condition is true; or in a pair of blocks, one or the other of which is processed depending on whether the IF statement is true or false, as shown in Figure 13 on page 73.

IF blocks are especially efficient when your program contains many statements to be executed when the condition is true or when the condition is false. For example:

```
100 if year = leapyear then
110     daynum = daynum+1
120     print "leapyear"
130 else
140     print "normal year"
150 end if
```

The statement at line 100 tests whether the value in YEAR is equal to the value in LEAPYEAR. If it is, the value in DAYNUM is increased by 1, the word "leapyear" is printed, and control transfers to statement 150. If YEAR is not equal to LEAPYEAR, the phrase "normal year" is printed and control transfers to the statement after line 150.

Depending on your application, there could have been a great many more statements between the IF line and the ELSE line, or between the ELSE line and the END IF line. The ELSE block can also be entirely omitted.

There is one important restriction. You can transfer control out of an IF block with a GOTO or GOSUB statement but you cannot use GOTO or GOSUB to transfer control from outside it to a statement within the body of one of these blocks. You can, however, use functions or call subroutines from within an IF block, because control leaves the block and returns without going anywhere else (except perhaps to another subroutine or function).

For IF blocks it is important to note that the THEN clause must be on the same line as the IF and the statement must be separate from the block of statements that follow the THEN. In this respect, there is an important difference between the block IF and the "simple" IF statement (see "Conditional Branching — IF

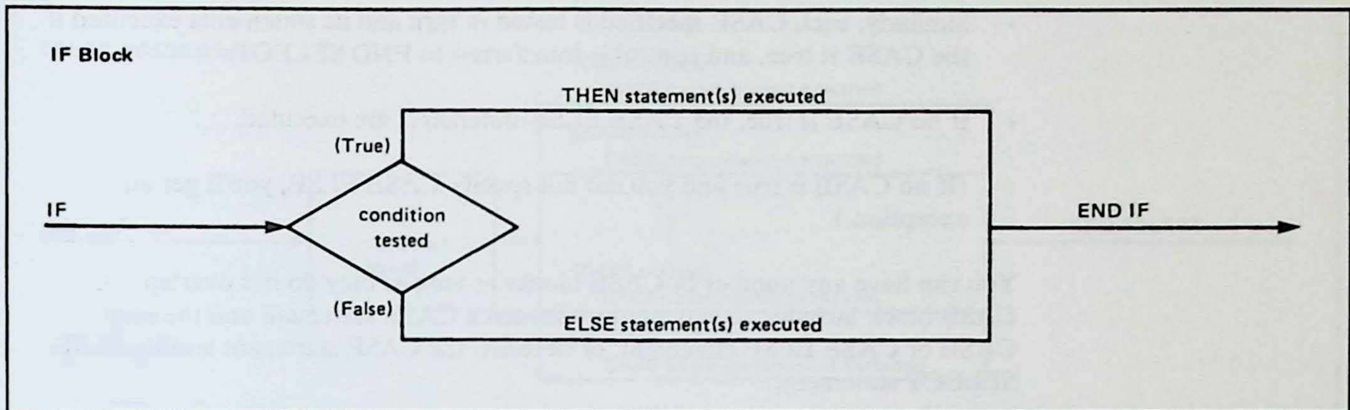


Figure 13. IF Block

Statement" on page 77). Unless you are careful, BASIC can easily assume that a block IF is a simple IF. For example:

```
100 if rc(1)=100 then print "*** END ***"
110    else
120         print "ERROR"
130         mat print rc
140 end if
```

The above ELSE statement will be an error because BASIC assumes that line 100 is a simple IF. The THEN block (but not the THEN keyword) must be separate statements.

SELECT/CASE Blocks

The SELECT block is more versatile as a structuring device than the IF block. The IF block allows you to choose two alternative paths of control; the SELECT block allows you many alternative paths of control.

The SELECT and END SELECT statements enclose a group of CASE statements; the entire group of statements is known as a SELECT block.

SELECT blocks allow the conditional processing of groups of statements. The way this is accomplished is by subdividing the SELECT block into sub-blocks, each of which is a CASE block.

A SELECT/CASE block is shown in Figure 14 on page 75. It is executed as follows:

- The SELECT expression is evaluated.
- The value of the SELECT expression is tested against the CASE-1 condition.
- If CASE-1 is true, the CASE-1 statements are executed, and control passes to END SELECT.
- The value of the SELECT expression is tested against the CASE-2 condition.
- If CASE-2 is true, the CASE-2 statements are executed, and control passes to END SELECT.

- Similarly, each CASE specified is tested in turn and its statements executed if the CASE is true, and control is transferred to END SELECT.
- If no CASE is true, the CASE ELSE statements are executed.

  (If no CASE is true and you did not specify CASE ELSE, you'll get an exception.)

You can have any number of CASE blocks as long as they do not overlap. A CASE block includes all statements between a CASE statement and the next CASE or CASE ELSE statement, or between the CASE statement and the END SELECT statement.

Starting with a simple example, let's say you need only two CASE blocks:

```
100 select n
110 case 1
120     let x = x**3 + x**2 + x
130     if x > 3.5 then x = 3.5
140 case 2
150     let x = x**2 + x
160     if x > 2.5 then x = 2.5
170 end select
```

In the above example, because there is no CASE ELSE block, the only valid values for N are 1 and 2. You would assign N=1 to select CASE 1 or N=2 to select CASE 2. When processing is completed for either of these cases, control transfers to the next statement following END SELECT.

If you have a CASE ELSE block in the SELECT block and none of the CASE constants match the evaluated SELECT expression, the statements included in the CASE ELSE block are executed. For example:

```
100 select (c-2)
110 case 1
120     m(1,2) = m(1,2) + x
130     m(2,3) = m(2,3) + y
140 case 2
150     m(1,2) = m(1,2) + x-y
160     m(2,3) = m(2,3) + y-x
170 case else
180     m(1,2) = 0
190     m(2,3) = 0
200 end select
```

Any evaluation of (C-2) which does not equal 1 or 2 causes the CASE ELSE block, statements 170, 180, and 190, to be executed. This would happen if the value of C was anything other than 3 or 4. If you have not coded a CASE ELSE block, the select expression must evaluate to exactly match one of the constants appearing in the CASE clauses, or an exception occurs.

You can supply a range of numbers for the case item:

```
200 case 8 to 15
```

This case block is selected whenever the associated SELECT expression value is between 8 and 15, *inclusive.*

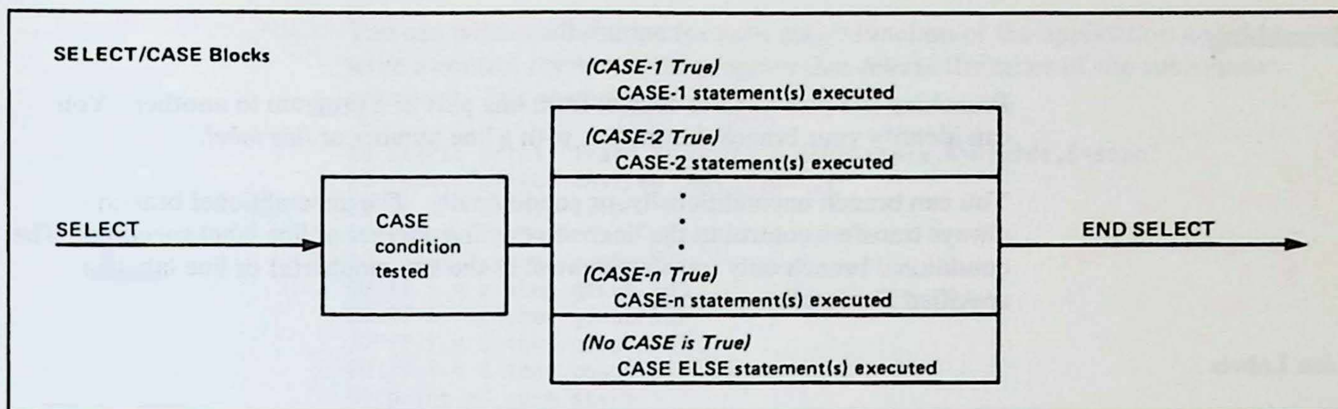


Figure 14. SELECT/CASE Blocks

You can also qualify the CASE item with a relational operator and/or a list of items. For example:

```
300 case < 1, 8, > 10
```

selects this case block when the associated SELECT expression equals 8, or evaluates to anything less than 1 or greater than 10. This could be done for the character case as well.

You can also use character expressions in the SELECT expression.

Again, there must be a matching character constant in one of the CASE clauses if there is no CASE ELSE block.

The example:

```
100 select month$
110 case "january"
120     days = 31
130     print "january hath"; days
140 case "february"
150     if year$ <> leapyear$ then days = 28 else days =29
160     print "february hath"; days
170 case "march"
180     days = 31
190     print "march hath"; days
200 end select
```

selects the CASE block whose associated character constant equals the value in the variable MONTH$.

If you write a similar example for yourself containing cases for all the months of the year, you would be assured that one of the cases would be selected, provided MONTH$ evaluated to the correct spelling.

You might want a CASE ELSE block to process any misspellings that occur.

Branching

Branching lets you transfer control from one part of a program to another. You can identify your branch destination with a line number or *line label.*

You can branch unconditionally, or conditionally. The unconditional branch always transfers control to the line with the line number or line label specified. The conditional branch only transfers control to the line number(s) or line label(s) specified if a condition is true.

Line Labels

A line label is a character string you can enter after a line number and then use to identify that line in lieu of the line number. Each line number can have only one line label, and the same line label can appear only once in a program unit. Line labels may be up to 40 characters long, the first of which must be alphabetic (A through Z). The remaining characters can be alphabetic, numeric, or the underline character. In doesn't matter whether you use upper, lower, or mixed case letters for your line labels. For example:

```
100 first_choice: let a = b
```

Line 100 can now also be identified as FIRST__CHOICE.

Certain identifiers may not be used as line labels (see "Reserved Word List" in *IBM BASIC Language Reference*) for a list of keywords, and your system administrator for those reserved words specific to your organization.

We strongly encourage you to use meaningful line labels because it significantly improves the readability of large programs.

GOTO — Unconditional Branch

The GOTO statement always transfers control to the line specified by line number or line label. For example:

```
900 goto 310
```

transfers control to line 310 and

```
200 goto process_next_item
```

transfers control to the line with the line label PROCESS__NEXT__ITEM.

GOSUB/RETURN — Unconditional Branch

The GOSUB statement also transfers control to the specified line. However, it also sets up a return path to the statement immediately following the GOSUB statement. The program transfers control via this return path when it executes a RETURN statement. One way, for example, to use GOSUB/RETURN statements is to invoke a subroutine and return from a subroutine. Subroutines are a convenient method of segmenting a program.

You can write a subroutine for each major function of the application and then write a control portion of the program that selects the order of the subroutines processed. For example:

```
10 start: print "1=add,2=subtract,3=multiply,4=divide,5=stop"
20 input "select desired operation":i
25 if i = 5 then stop
30 input "first number":j
40 input "second number":k
50 if i = 1 then gosub add
60 if i = 2 then gosub sub
70 if i = 3 then gosub mul
80 if i = 4 then gosub div
90 print r: goto start
100 add: Rem add subroutine
110 r=j+k : return
200 sub: Rem subtract subroutine
210 r=j-k : return
300 mul: Rem multiply subroutine
310 r=j*k : return
400 div: Rem divide subroutine
410 r=0
420 if k=0 then return
430 gosub dodiv : return
500 dodiv: r=j/k : return
```

invokes one of the subroutines or stops, depending on the value of I.

You can call another subroutine from within a subroutine (line 430 above) and you can have more than one RETURN statement in a subroutine. The RETURN statement transfers control back to the statement immediately following the GOSUB, whether on the same line or the next line. The short subroutine of line 500 is really unnecessary but is included to illustrate calling a subroutine from within a subroutine.

Conditional Branching — IF Statement

You can use the IF statement to conditionally transfer to one statement or another based on the value of an expression. For example:

```
185 if var < limit then goto less_than_limit
```

transfers control to the line labeled *less_than_limit* if VAR is less than LIMIT.

```
235 if a < b then add else subtract
```

either transfers control to the line labeled *add* if A is less than B, or transfers control to the line labeled *subtract* if A is greater than or equal to B.

```
490 if procedure$ = "done" then aa
```

transfers control to the line labeled *aa* if PROCEDURE$ equals "done".

To conditionally call a subroutine in a simple IF statement, you follow "THEN" with a GOSUB and a line number or line label, as follows:

```
765 if len(J$) = 0 then gosub process_null
```

This statement transfers control to the subroutine called *process_null* if the length of J$ is equal to zero.

Remember that any form of transfer to a nonexistent line number or line label causes an exception.

Conditional Branching — ON GOSUB Statement

Use the ON GOSUB statement to transfer control conditionally to one of several subroutines, based on a numeric expression. At execution time, the numeric expression is used to select one of the line numbers and/or line labels in the list following GOSUB. For example, in this statement:

```
50 on i gosub alpha, beta, gamma, delta
```

the value in I is evaluated. If it has a value of 1, control is transferred to the subroutine *alpha*. If it has a value of 2, control is transferred to the subroutine *beta*, etc. If I has a value of less than 1 or greater than 4, an exception occurs and the program is immediately terminated (unless you have previously executed an ON ERROR GOTO statement).

To avoid this exception condition, you can use either the ELSE clause or the NONE clause. For example:

```
90 on i gosub alpha, beta, gamma, delta else xyz=xyz+100
```

If I has a value other than 1, 2, 3, or 4, the variable XYZ is increased by 100.

```
90 on i gosub alpha, beta, gamma, delta none omega
```

transfers control to the line labeled *omega* if I has a value other than 1, 2, 3, or 4.

The ON GOSUB statement sets up a return path to the next statement immediately following the ON GOSUB statement.

Execution of the RETURN statement causes control to be returned to the statement immediately following the last active ON GOSUB statement.

Conditional Branching — ON GOTO Statement

Use the ON GOTO statement to transfer control conditionally to one of several lines, based on a numeric expression. At execution time, the numeric expression is used to select one of the line numbers and/or line labels in the list following GOTO. For example, in this statement:

```
50 on i goto alpha, beta, gamma, delta
```

the value of I is evaluated. If it has a value of 1, control is transferred to the line labeled *alpha*. If it has a value of 2, control is transferred to the line labeled *beta*, and so forth. If I has a value of less than 1 or greater than 4, an exception occurs and, if you are not handling exceptions with an ON ERROR GOTO statement, the program is immediately terminated.

To avoid this exception condition, you can use either the ELSE clause or the NONE clause. For example, either:

```
90 on i goto alpha, beta, gamma, delta else goto omega
```

or:

```
90 on i goto alpha, beta, gamma, delta none omega
```

transfers control to the line labeled *omega* if I has a value other than 1, 2, 3, or 4.

The following example shows the results of an ON GOTO statement.

```
 * list
100 input p
110 on p goto first, middle, last none nomore
120 first: print "first and ";
130 middle: print "middle and ";
140 last: print "last ";
150 print "one"
160 nomore: if p > 3 then print "none"
 * run
 ? 1
first and middle and last one
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 160.
 * run
 ? 2
middle and last one
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 160.
 * run
 ? 3
last one
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 160.
 * run
 ? 4
none
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 160.
```

Halting Program Execution

You can halt the execution of your program by using the END, STOP, or PAUSE statement.

Using the END Statement

Use the END statement to specify the physical end of a main program and to terminate program execution. END should be the last physical statement in a main program. (Only subprograms may follow this statement.) When processing reaches an END statement, all active files are closed by the END statement and control returns to the operating environment, which will be either the BASIC environment or, if you are running your program outside the BASIC environment, the operating system.

You may return a value from your program to the operating system when the program is executed outside the BASIC environment. If this is the case, you can return a value by coding a numeric expression as part of the END statement. For example:

```
9999 end i*2
```

returns the rounded integer result of the expression I*2 to the operating system. The value of the expression is ignored by operating systems that do not support this feature.

STOP Statement

The STOP statement has exactly the same effect as an END statement, except that a STOP statement may appear anywhere in a program. In fact, you can have STOP statements at several places in your program, that halt execution under the alternative conditions you choose.

The STOP statement, like the END statement, may contain a numeric expression that is returned to the operating system. For example:

```
410 stop A+B
```

returns the rounded integer result of the sum of A and B to the operating system.

PAUSE Statement

The PAUSE statement causes your program to stop temporarily until you press the ENTER key or issue a GO command. The PAUSE statement is executed only in the interactive mode (when you're running the program at a terminal); if your program is executing in the batch mode, it is ignored.

You can code a message with each PAUSE statement. For example:

```
110 pause "THIS IS END OF ROUTINE 4"
```

prints THIS IS END OF ROUTINE 4 and temporarily stops the execution of your program at line 110.

If you omit a PAUSE message, the line number of the PAUSE statement is displayed. For example, if you code:

```
110 pause
```

Then, during program execution, the following message is displayed:

```
PAUSE AT LINE:     110
```

You can code a PAUSE statement anywhere you want in your program.

Chaining BASIC Programs

The CHAIN statement allows you to execute separate programs one after the other automatically. Suppose you have an applications task that is so large it is cumbersome to run as one program. You can break such a task into several programs and "chain" them together, as shown in Figure 15. Also, you might have programs to migrate from other BASICs which are set up with CHAIN programs.

Generally speaking, program chaining is seldom needed with today's virtual storage operating systems. Called subprograms (see "Defining a Subprogram — SUB and END SUB Statements" on page 137) are usually used instead of chaining in modern programs.

The chained-to program is loaded in place of the chained-from program. Therefore, you reduce virtual storage requirements when you breakup a large program with CHAIN statements. Data can be passed to a chained program by storing the data in COMMON variables or by specifying in the CHAIN statement the variables that are to be passed. For further details, see your IBM BASIC System Services manual.

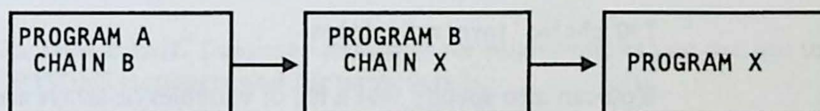


Figure 15. Chaining Programs

Starting Chained Program Execution — CHAIN Statement

A typical demonstration of chaining is a set of programs in which one member of the set is the controlling program and is the member from which the entire process is operated. You want to control a set of accounting programs, let us say, from one master menu program which chains each selected program and is itself chained to by all the other programs. You can easily add another function or delete a function from such a system merely by adding or deleting a menu item and supplying the additional program.

A program that reaches a CHAIN statement halts its own operation, transfers the chained program from auxiliary storage to your workspace, and starts the new program. The new program may itself chain to any other program, including the program that chained to it. Thus a master menu program may chain to any one of the programs on its roster. The selected program performs its function and then chains back to the master menu program. One of the options in the master program allows for halting execution entirely. For example:

```
100 rem Master Menu Program
110 start: print "1. Transaction entry"
120 print "2. Validate batch"
130 print "3. Post to General Ledger"
140 print "4. Print Financial Reports"
150 print "5. Halt"
160 input "Enter selection number": k
170 on k goto trns,valid,postg,frep,fin none invalid
180 trns:  chain "trnsctn"
190 valid: chain "validate"
200 postg: chain "postgl"
210 frep:  chain "freport"
220 invalid: print "Invalid selection." : goto start
230 fin:   end
```

This master program executes one of the CHAIN statements at lines 180 through 210 or halts the program based on the input from the user.

The simplest CHAIN statement is followed only by a character expression whose value is the name of the program to be chained. Lines 180 through 210 above are examples of simple CHAIN statements. Such statements cause all open files to be closed when the CHAIN statement is encountered.

If you want all files to remain as they are (opened or closed and at their current position), use the FILES option as follows:

```
290 chain "invntry", files
```

You can also specify that a list of variables or arrays are to be passed to the chained-to program. The contents of these variables are then available to the chained-to program. For example:

```
205 chain b$,files, m,y
```

specifies that the data in M and Y is to be passed to the program whose name is the value of B$, when that program is loaded as a result of the CHAIN statement. M and Y can be arrays or simple variables. All variables in the chained-to program are cleared on entry, except data in variables specified as above or in COMMON statements. The same variable or array cannot be specified in both a COMMON statement and a CHAIN statement. The COMMON statements pass variables across a chain independently.

Defining Chained Variables — USE Statement

To indicate in the receiving program which variables are to be received, you must use the USE statement. For example:

```
320 use m,y
```

establishes a one-to-one correspondence between the variables named in the CHAIN statement above and the chained-to program, provided of course, that these variables are also defined in the chained-to program.

The variables need not be listed in the same order in both the CHAIN and USE statements, but they must have the same name, type, maximum number of elements, maximum length, and precision, as appropriate for the type.

Using the OPTION Statement to Change Options

You can use the OPTION statement to explicitly specify certain options when you are running or compiling your program. Using this statement in your program enables you to override default options.

Certain options of the OPTION statement are discussed in different sections of this manual:

- We have already discussed how this statement can be used to change whether an array subscript will start at 0 or 1 by using the BASE option. (See "Dimensioning — OPTION And DIM Statements" on page 52.)

- In the section "Controlling Horizontal Spacing" on page 95, we describe how you can use the PRTZO option of the OPTION statement to alter the width of your print zone.

- In the section "Controlling Printing of Rounded Decimal Digits" on page 98, we describe how you can use the RD option to control the number of rounded digits that are displayed by the PRINT statement.

Some of the other options of the OPTION statement are discussed below.

See *IBM BASIC Language Reference* for other options you can use on the OPTIONS statement and for more details.

Printing Precision for Real Data (SPREC | LPREC)

By using SPREC or LPREC in an OPTION statement, you can specify how many digits of precision will be printed for data by the PRINT statement (without the USING clause). SPREC specifies single precision real data (6 digits) and LPREC specifies double precision real data (12 digits).

See "Direct Execution with SPREC or LPREC" on page 191 for a discussion of using SPREC and LPREC on a RUN command.

The use of SPREC and LPREC also controls whether the default for real data is real double or real single.

Collating Sequence for Comparing Character Data (COLLATE)

The COLLATE option can be used to determine which collating sequence BASIC will use for the comparison and conversion of character data. For example, if you specify:

```
400 option collate native
```

it means the collating sequence is Extended Binary Coded Decimal (EBCDIC). But if you say:

```
400 option collate standard
```

it means that the collating sequence is American National Code for Information Interchange (ASCII).

Remember that character data is always represented in EBCDIC. COLLATE only affects the *comparison* of character strings and the ORD and CHR$ intrinsic function results.

Inverting Print Edit Facility (INVP)

Normally when numeric values are printed, the period represents the decimal point and the comma is used to separate digits. This is known as the U.S. format. However, in European format it's just the opposite: the comma represents the decimal point and the period separates digits. To change from one format to the other you use the INVP option. For example, if you specify:

```
200 option invp on
```

you will get the European format; and if you specify

```
200 option invp off
```

the format will be U.S. Example:

```
U.S.          123,456.78
European      123.456,78
```

Levels of Messages (FLAG)

With this option you can control the level of BASIC messages displayed at the terminal. There are 4 levels of messages that BASIC can display at your terminal. The levels in increasing order of severity are:

I Informative Messages

W Warning Messages

E Error Messages

S Severe Error Messages

When you specify one of the above levels in an OPTION FLAG statement, it means that only messages of levels higher or equal to the indicated level will be displayed at the terminal. For example if you specify:

```
100 option flag (s)
```

only severe error messages will be reported; any informative messages, warning messages, and nonsevere error messages will not be displayed.

Including Comments in Your Program

It will be helpful to you and others looking at your program if you include some comments about the purpose of the program, what its environmental requirements are, and the applicable manuals.

What to Comment

You should insert blank lines and some type of clear identifying remark at the beginning of every major segment, subroutine, or subprogram belonging to your main task. It is helpful to enclose these segment titles in a box of asterisks or use some other graphic method to emphasize them.

```
400 rem
410 rem ****************************
420 rem * calculation subroutine *
430 rem ****************************
440 rem
```

If the title you have given a program segment is not self-explanatory, then make a few further comments about the program segment and what its role is.

You should give comments for most, if not all, of your variables. The comment should describe what the variable contains and how it is used. If a variable is used for many temporary or intermediate results, explain this (it shouldn't be necessary to describe it in more detail unless you are using it in a unique way).

You should also make clarifying comments within the body of your source code whenever you can see that the code itself may be somewhat obscure. This is particularly true if you are using some complicated mathematical formulas. You should comment each formula or group of formulas to specify its function.

There is no need to comment every line of code. Just try to make sure that you or any other programmer will be able to readily understand what the code is doing several years after it is written.

How to Comment

A comment is a character string which need not be enclosed in quotation marks.

```
100 rem this is a comment
110 ! this is also a comment
```

Precede a comment with the word REM or an exclamation mark (!) when it occupies a line of its own. Whatever follows a REM or ! on the same line is a comment and is ignored when the program is running. If you branch to a comment line, the comment is ignored and execution continues with the next line.

Comments may take up a whole line or follow a statement on the same line. When you follow a statement with a comment on the same line, use the exclamation mark to tell BASIC where the statement ends and the comment begins. A comment may follow with any BASIC statement except DATA and IMAGE.

```
100 let a=0! Initialize variable A
```

The comment "Initialize variable A" must be short enough to fit on the same line as the BASIC statement. It may not continue to the next line, unless you use REM, or include another exclamation point after the statement on the next line. For example:

```
100 let a=0! initialize
110 rem variable a
```

or

```
100 let a=0! initialize
110        ! variable a
```

No statement may follow a comment on the same line because it would be considered part of the comment.

Be careful when you are using the continuation symbol (&) with a statement. The continuation symbol immediately follows the comment, but applies to the statement, *not* to the comment. For example:

```
100 if a <> b and ! test stop condition &
  & c=b then stop
```

The continuation symbol continued the statement. The above line is equivalent to the following:

```
100 if a <> b and c=b then stop
```

Avoiding Common Programming Errors

Generally, programming errors are caused by lack of attention to detail. The following sections discuss the most common programming errors.

Not Saving Your Program

First, avoid the all too common and disastrous error of not saving your workspace through the SAVE or STORE command.

This is not a programming error *per se*, but it is a common enough occurrence to warrant mentioning it.

You will lose your program in the workspace if BASIC terminates abnormally or the computer goes down unexpectedly. Even though these occurrences are rare, it is recommended that you SAVE or STORE your program periodically while it is being entered or modified.

You will lose the program that you have just entered if you issue the QUIT command without using the SAVE or STORE command first. The QUIT command clears your workspace just before it transfers control to the host system. Therefore, if you have not explicitly saved your program, you have lost it when you leave BASIC.

Mechanical Errors

Mechanical errors generally affect a single statement and do not affect the flow of your program. Here are some tips you should keep in mind when you are entering a program or when you are testing it.

- Review the word you just typed before continuing with the next. Whatever you miss after that can usually be found by a careful visual inspection of the source listings.
- Check for the extremes in the expected range of a variable.
- Keep a list of all variables and arrays you use so that you won't inadvertently reuse the same variable or array when you need to preserve its value. The LIST command with the XREF clause is useful in maintaining this list.
- Be careful to initialize variables; don't assume you did it somewhere else without checking that you really did.
- Be sure when you delete portions of a program or make line changes, that you are not destroying a label or a line number that is the destination of a transfer statement somewhere else in the program.
- Remember that statements for array BASE options take effect throughout the program unit regardless of where they are placed. If you define arrays that have subscripts including zero and then forget that there is a BASE 1 option in an OPTION statement somewhere in the program unit, you will get erroneous results.
- For each assignment or input statement, verify that the variable has the correct size and type.

  In particular, don't initialize more array elements than an array contains. Make sure that all subscripts are within the bounds of the array's dimensions.
- Check for invalid data references to items of differing types (for example, integer and decimal). Verify that, if you store data into an integer, the rounded result isn't zero, unless this is acceptable.
- Provide exception handling routines (ON condition).

Logical Errors

Logical errors are the result of mistaken thinking, and often involve the use of logical operators and the use of decision-making statements. The proper use of these elements to achieve the desired results requires careful analytical thinking. Don't be tempted to go too fast or to be more clever than necessary. The most elegant programs are those that achieve the desired result in the simplest and most straightforward manner. These are the programs that probably will work correctly all the time. Some logical errors are:

- Transferring control from outside a DO loop into an intermediate point in a DO loop. This also pertains to FOR loops, IF blocks, and CASE blocks.

- Attempting to process input records after a file has been opened for output, or vice versa.
- Defining arguments and parameters that do not agree in type and/or length; for example, integer arguments with decimal parameters or vice versa.
- In a subprogram, assigning a new value to a parameter when the value of the argument is still needed.
- Referring to a function with other than the defined number of arguments.
- Performing an absolute comparison (that is, a comparison using an absolute value) with transcendental functions, such as trigonometric or hyperbolic functions. Such functions use approximation techniques, and therefore all such comparisons should include a small contingency.

  For example, when comparing for the size of 3 radians, a comparison such as the following is appropriate:

  ```
  200 if SIN(X)>.14112 and SIN(X)<.14113 then ...
  ```

  (The precision can be increased or decreased to that which is appropriate for your program.)
- Performing an absolute comparison with real data. Real data is an approximation to decimal values and therefore may not be exactly equal to decimal values.

Performing Input/Output

Application programs manipulate data and may perform calculations on this data. Often, this data is stored on files external to the program, or you enter it from a terminal.

Input statements provide your program with this data.

Output statements provide your program with a method of saving or displaying any data that is the result of its operations.

BASIC has a variety of input/output (I/O) statements. These statements allow your program the capability of communicating directly with your terminal or with files.

The capabilities of most of the I/O statements are discussed below.

(See *IBM BASIC Language Reference* for a discussion of all I/O statements.)

Using Program-Defined Data

You can define data within your program that can be used to initialize the values of various variables. This defined data could also be used by your program when you are testing it. This is particularly helpful if you are testing subprograms independently that you will merge with your main program after the subprograms have been thoroughly tested.

Specifying Program-Defined Data — DATA Statements

Use the nonexecutable DATA statement to store data lists inside your program to be accessed by the READ statement. You can use this to transfer values to any variable or array for initialization or for testing.

The values in the data list are accessed sequentially, starting with the first item in the DATA statement and ending with the last item. The items can be numeric or character constants (expressions are not allowed). If a character constant contains leading or trailing spaces or embedded, leading, or trailing commas, or begins with a number followed by an asterisk, you must enclose the string in quotation marks or apostrophes.

For example, the items in the following statement:

```
250 data "john smith ","priscilla alden "
```

contain trailing spaces, and therefore must be enclosed in apostrophes (') or quotation marks (").

The DATA statements may be placed anywhere in the program, but it's good programming practice to place them consecutively near the beginning of the program.

If the items in the DATA statement are exhausted before the variables in the READ statement, the items in the next DATA statement are used for the remaining variables in the READ statement. If the items in all the DATA statements are exhausted before all the variables in a READ statement have been processed, an exception occurs and execution of your program is terminated (unless you trap the exception).

If you want to use the same value in several variables that are specified in consecutive order in a READ statement, you can use a replication factor in a DATA statement. For example:

```
 * list
100 data 4*5,0
110 rem 5 is replicated four times
120 read a,b,c,d,e
130 print "a=";a,"b=";b
140 print "c=";c,"d=";d
150 print "e=";e
160 print "end"
170 end
 * run
a= 5                    b= 5
c= 5                    d= 5
e= 0
end
END AT LINE 170.
```

Reading Program Defined Data — READ Statements

Use the READ statement to retrieve values from DATA statements and to assign them to the variables specified by the item list in the READ statement.

It is important that the data types in the DATA statement be compatible with the variable types in the READ statement. If they are not compatible, an exception will occur when the READ statement is processed.

The first time a READ statement is processed, the first value in the first DATA statement is assigned to the first variable, the second value is assigned to the second variable, and so forth. This continues until all the items in all the DATA statements have been used, or until all the variables in the READ statement have been used. For example:

```
* list
90 ! blackjack
100 data 17,18,20,16,18,18
110 data 18,19,0,0
120 print "blackjack"
130 print "house","player"
140 readab: read a,b
150 if b=0 then goto print_end
160 if a > b then print a,b,"lose"
170 if a < b then print a,b,"win"
180 if a = b then print a,b,"draw"
190 goto readab
200 print_end: print "end of game"
210 end
* RUN
blackjack
house                player
 17                   18                   win
 20                   16                   lose
 18                   18                   draw
 18                   19                   win
end of game
END AT LINE 210.
```

The value of zero read into variable B is used to indicate the end of the list for this example.

If the READ statement transfers values from the DATA statement to arrays, the rightmost subscript is transferred first. (That is, if you have a two-dimensional array, the values are read as follows: all the columns for row one are read first, all the columns for row two are read next, all the columns for row three are read next, and so forth). For example:

```
* list
100 data 5,4,3,9,8,7,2*20,99
110 option base 1
120 dim a(3,3)
130 mat read a
140 print "result"
150 print a(1,1),a(1,2),a(1,3)
160 print a(2,1),a(2,2),a(2,3)
170 print a(3,1),a(3,2),a(3,3)
180 end
* run
result
 5                    4                    3
 9                    8                    7
 20                   20                   99
END AT LINE 180.
```

You may include in the READ statement the capability of regaining control of the program if an exception is encountered while the data is being assigned to the variables. For example:

```
100 dim a$*5
110 data "this is too long"
120 read a$ soflow 900
```

transfers control to statement 900, because the character string is longer than 5 characters.

Reusing Program-Defined Data — RESTORE Statement

You can use the RESTORE statement to cause the next READ statement processed to start over with the first item in a DATA statement or with a particular DATA statement. In this way, your program can reuse the same DATA statements.

RESTORE causes the next READ statement to start over with the first item in the DATA statement referred to by line number or line label. If no line number or line label is specified, the RESTORE repositions the file pointer to the first item in the first DATA statement. If the RESTORE statement specifies a line number or line label, the DATA statement must be the first and only statement on that line.

For example:

```
100 data 123
110 data 456
120 data 789
130 more: data abc
140 read a
150 restore 120
160 read b
170 restore
180 read c
190 restore more
200 read x$
```

The value assigned to A will be 123. The value assigned to B will be 789, because RESTORE 120 in line 150 causes the READ B statement in line 160 to read DATA 789 in line 120. The value assigned to C will be 123, because the RESTORE in line 170 repositioned the data file to its beginning. X$ will contain ABC, since RESTORE MORE in line 190 repositioned the data file to the value specified in line 130.

Be careful not to create an endless loop when using the RESTORE statement. If you had used it immediately following statement 140 in the program BLACKJACK on page 91, an endless loop would have occurred because your program would reposition the data pointer to the first entry, and A and B would always be set to 17 and 18, which are the first values in the first DATA statement.

A RESTORE statement without a line number or line label is ignored if there are no DATA statements in the program unit.

Terminal Input/Output

The terminal is a convenient way of communicating between you and your program. When you run your program, it can execute an I/O statement that requests you to enter data from the terminal. After you have entered this data, your program can read and verify it. If you have entered any incorrect data, your program can display the data it doesn't understand. Your program can also print the results of any operation on your terminal. Programs that operate this way are generally called interactive programs.

Retrieving Formatted Terminal Input — INPUT Statement

Use the INPUT statement to provide values to your program from the terminal. An item list specifies a list of numeric or character string variables to which the input values are assigned when this statement is executed. The variables in the list are separated by commas. For example:

```
100 input a$, x1, b$, x2
```

In this example, the program expects a character string, a number, a character string, and another number. When the program prompts for input, the values entered must be the same type (character or numeric) as the corresponding variable in the item list. However, a number may be input to either a character variable or a numeric variable.

Character strings are normally enclosed with quotation marks. This is not required, however, if the character string does not:

- Begin or end with a space or a quote
- Contain a comma
- Begin with a number immediately followed by an asterisk

During the operation of your program when the INPUT statement is executed, a question mark preceded and followed by a space is displayed, and execution is suspended until the required data is entered. The input values can be entered on a single line or on multiple lines, but they must be separated by commas. For example:

```
100 input a$, x1, b$, x2
```

The input in response to this could be one of the following:

```
? able,5,this is the time,199
```

or

```
? able,
? 5,
? this is the time,
? 199
```

or

```
? able,5,this is the time,
? 199
```

After you have entered the value 199, your program variables contain:

Variable	Value
A$	able
X1	5
B$	this is the time
X2	199

If you do not want to change the value in a variable when the INPUT statement is processed, just type consecutive commas. You can also enter a slash at the end of the line to prevent the values of the remaining variables from being changed.

For example, assume the variables contain the above values and the next time you process the INPUT statement at 100, you enter:

```
? baker,,change/
```

After you have entered this line, your program variables contain:

Variable	Value
A$	baker
X1	5
B$	change
X2	199

If you inadvertently enter an incorrect type of value for the variable specified in the INPUT statement, an exception message is displayed and you are given another chance to enter the data starting with the first item in the INPUT statement.

Retrieving Unformatted Terminal Input — LINE INPUT Statement

Use the LINE INPUT statement to transmit unformatted character strings from the terminal to the character variables specified in an item list. Each variable in the list is assigned an entire line of input.

You can also spell this statement LINPUT.

Using the LINE INPUT statement, you can include commas, spaces, apostrophes, and quotation marks in the character strings. For example:

```
 * initialize
 * 150 line input c$,d$,e$
 * run
 ? JOHN, PAUL & MARY
 ? JOHN SMITH, JR
 ? ALEX'S HOBBY SHOP
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 150.
```

Using PROMPT Messages

You can replace the question marks on input lines with more meaningful messages. To do this, you enter the message you want printed immediately preceding the item list in the INPUT, LINE INPUT, or LINPUT statements. You must enclose the message in quotation marks and follow it with a colon. When the statement is executed, the message (character string) is displayed instead of the question mark, and the requested information is entered on the same line. For example :

```
 * list
100 input "NAME ":a$
110 linput "STREET ADDRESS ":b$
120 linput "CITY,STATE,ZIP ":c$
130 end
 * RUN
NAME  JOAN JONES
STREET ADDRESS  123 VIA DE LAPAZ
CITY,STATE,ZIP  SAN JOSE,CA,92107
END AT LINE 130.
```

If you want to use a character expression (for example, a character variable) instead of a character string, you enter the keyword PROMPT, followed by the character expression and a colon, immediately preceding the item list in the INPUT, LINE INPUT, or LINPUT statement. When the statement is executed, the value of the character expression is printed on the screen instead of the question mark, and the requested information is entered on the same line. For example :

```
 * list
100 e$="MARRIED NAME "
110 linput prompt e$:a$
120 print a$
130 end
 * RUN
MARRIED NAME  JOAN JONES SMYTHE
JOAN JONES SMYTHE
END AT LINE 130.
```

Displaying Items at the Terminal — PRINT Statement

Use the PRINT statement to display a single item or a list of items at the terminal. The items may be character constants (messages enclosed in quotation marks), numeric constants, variables, or expressions (numeric or character).

In the PRINT statement, you can separate the items by commas or semicolons.

Controlling Horizontal Spacing: If you use commas to separate two items, the next item is printed in the next print zone.

A print zone is defined as the number of print positions allocated for printing an item. The default print zone size is 20 characters, and the number of print zones on a line depends on the output device and the margin settings. You can override the default print zone size of 20 by using the OPTION statement with the PRTZO option (see examples below).

The PRINT statement, using a print zone of 20 characters and four zones on a line, gives the following results:

```
 * list
100 let a,b,c,d = 123
110 let e,f = 456
120 print a,b,c,d,e,f
130 end
 * RUN
 123                123                 123                 123
 456                456
END AT LINE 130.
```

Variables E and F are printed on the next line, because variable D was printed in the last print zone on line one.

If you are printing a character string that is longer than a print zone, it will extend into the next print zone(s). For example, using a print zone of 15 characters gives the following results:

```
 * list
100 option prtzo 15
110 print "this is longer than one zone","zone 3"
120 end
 * RUN
this is longer than one zone  zone 3
END AT LINE 120.
```

If you are printing a character string that extends beyond the end of a line, the characters are printed on the next line starting in column 1. Here is an example, using a print zone of 20 characters:

```
 * list
100 print,,, "this is longer than one zone"
110 end
 * RUN

this is longer than one zone
END AT LINE 110.
```

Each time you specify a comma in the PRINT statement item list, the next print zone is selected. For example , using a print zone of 15 characters, you get the following:

```
 * list
100 option prtzo 15
110 print "zone 1",,"zone 3"
120 end
 * RUN
zone 1                        zone 3
END AT LINE 120.
```

If you use a semicolon, it overrides the next print zone selection. Therefore, the next item is printed where the last one stopped. Numeric items are always printed with a leading space if they are positive (a minus sign for negative values), and one trailing space (this space is omitted if no items follow on the line). For example:

```
 * list
100 let a = 5
110 print "a=";a
120 end
 * run
a= 5
END AT LINE 120.
```

If you had used a comma instead of the semicolon, the line would print as follows, using a print zone of 15 characters:

```
a=              5
```

You can also control horizontal spacing by including the keyword TAB in the item list. TAB(X) allows you to specify the column number where the next item in the list is to be printed. X must be a positive numeric constant, variable, or expression. For example:

```
 * list
100 let a = 7
110 let b = 3
120 print tab(5);"tab to 5";tab(a*b);"tab to 21"
130 end
 * RUN
    tab to 5        tab to 21
END AT LINE 130.
```

If the value you specify in a TAB clause is less than the print column already reached on the current line, the item following the TAB clause is printed on the next line. For example:

```
 * list
100 print tab(5);"56789ab";tab(8);"next line"
110 end
 * RUN
    56789ab
       next line
END AT LINE 110.
```

Ordinarily, it is not a good practice to use commas as separators with PRINT statements that also contain TAB clauses. The output may not be what you expect. For example (using a print zone of 20 characters):

```
 * list
100 print tab (10),"col21"     !col21 because of comma
110 print tab(10);"col10"      !semicolon
120 print -123,tab(10); "next line"
130 end
 * run
                    col21
         col10
-123
         next line
END AT LINE 130.
```

The comma in line 100 forces the character string "col21" to print in the next zone, which is column 21. In line 120, the comma after -123 causes a skip to the next zone. Then, since TAB column 10 is in a lower zone, the value "next line" is printed in column 10 of the *next* line.

If you set margins with the MARGIN statement, the values you specify in the TAB clauses are relative to these margins.

For example, if the left margin is set to column 11 and the value of the TAB clause is 5, the item is printed in column 15, which is the fifth column relative to column 11 (column 11, 12, 13, 14, 15).

```
 * list
100 margin left 1
110 print "12345678901234567890"
120 margin left 11 right 60
130 print tab(5);"col15"
 * RUN
12345678901234567890
              col15
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 130.
```

Controlling Pages: To eject to a new page, include the keyword NEWPAGE in the item list. NEWPAGE ejects to top of form, if the output channel is a hard-copy terminal.

NEWPAGE clears the screen for a display terminal. In this case, the keyword should precede all items that are to be printed, since the statement

```
100 print a,b,newpage,c
```

would result in only the value C being displayed on a display terminal. On a hard-copy terminal, A and B would be printed on one page, and C would be printed at the top of the next page.

Controlling Printing of Rounded Decimal Digits: You can control the number of rounded decimal digits that are displayed when the Print statement is executed. You do it by using the RD option of the OPTION statement. For example, if you specify RD 03, a value of 4.5678 is printed as 4.568, and if RD 05 is specified, 4.5678 is printed as 4.56780.

Reading and Writing Selected Fields

You can format a terminal screen with information appearing in specific columns and rows on the screen. You can then request data to be entered so that it appears within this predefined format.

You might use this BASIC feature to display a form which would then be filled in, much as you would fill out a printed form.

The statements that let you do this are PRINT FIELDS and INPUT FIELDS. In both of these statements, you can specify:

1. A field definition for each field to be processed, specifying:

   a. The row and column where each field is to begin

   b. The form of the input or output data

   c. Display attributes for the field; that is, whether the field should be highlighted (display attribute H), should display at normal intensity

(display attribute N), or not be displayed at all (display attribute I, for invisible)

d. Control attributes for the field, which can request automatic field advancing (control attribute A) or initial cursor positioning (control attribute C)

2. The input or output list that identifies the data to be transferred

The following example demonstrates how to display two fields and read two fields using the PRINT FIELDS and the INPUT FIELDS statements.

```
100 option base 1
110 dim a$(2), b$(2), id$*15, pswd$*12
120 data "1,1,c9,h", "2,1,c9,h"
130 mat read a$
140 data "1,11,c15,nac,n", "2,11,c12,i,n"
150 mat read b$
160 print newpage
170 print fields mat a$: "enter id", "password"
180 input fields mat b$: id$, pswd$
```

This series of statements is processed as follows:

1. Statements 100 through 150 set up two arrays of field definitions, A$ and B$, each with two elements.

2. The PRINT NEWPAGE statement clears the screen.

   (If you don't do this, you can get line-by-line data intermixed with the full-screen data you want to process.)

3. The PRINT FIELDS statement displays:

   a. *enter id* in row 1 of the screen, beginning at column 1 (from the first field definition in array A$), followed by a space (from the C9 specification).

   Because the display attribute is H, the characters are highlighted. We specified nine spaces (C9), even though ENTER ID requires only eight, because we wanted 2 spaces between ID and the cursor (see below).

   b. *password* in row 2 of the screen, beginning at column 1 (from the second field definition in array A$), followed by a space (from the C9 specification).

   Because the display attribute is H, the characters are highlighted.

   The screen now shows the following:

```
enter id  _
password
```

4. The INPUT FIELDS statement executes as follows:

   a. The cursor has been positioned at screen row 1, column 11 (from the first field definition in array B$), as the program waits for you to enter up to 15 characters of data.

When you enter the data, it is displayed on the screen at normal intensity because N was specified as the display attribute.

If you enter 15 characters of data, the cursor automatically advances to the next field (A is specified as a leading control attribute).

If you enter fewer than 15 characters of data, you can position the cursor to the beginning of the next field, using the cursor positioning keys (such as the "next field" key).

b. When the cursor is positioned at screen row 2 column 11 (from the second field definition in array B$), the program waits for you to enter up to 12 characters of data.

   When you enter the data, it is not displayed on the screen because I was specified as the display attribute.

c. When you press the enter key:

   1) Up to 15 characters of data contained in screen row 1 beginning at column 11 are transferred to the character variable ID$.

   2) Up to 12 characters of data contained in screen row 2 beginning at column 11 are transferred to the character variable PSWD$.

For example, if you entered TWILIGHT as your user id and ZONE as your password, the screen would show the following:

```
enter id  TWILIGHT
password
```

Requesting Formatted Output — IMAGE and FORM Statements

When you want to format your output — for example, for terminal screen display or for a printed report — use the IMAGE or FORM statement, together with the PRINT USING statement.

Both the IMAGE and FORM statements provide a method of showing the positioning of fields on a line and of specifying the kind of editing that should be performed upon the fields within the line.

You can use the IMAGE statement with PRINT statements and with PRINT File statements.

With IMAGE, you can lay out all the fields on one line, exactly as they will be produced by the program.

The FORM statement is more flexible; you can use it with both input and output statements: PRINT, READ, REREAD, WRITE, and REWRITE. FORM is more useful when you want to skip a page in the middle of a line, when you want a line longer than your screen is wide, etc.

In both IMAGE statements and FORM statements, you can specify *character strings* that will be printed exactly as you specify. For example:

```
100 image:enter a 2-digit number
```

or

```
110 form "enter a 2-digit number"
```

then, when you use either of these specifications to display the line, you'll get the following actual display:

```
enter a 2-digit number
```

In both IMAGE statements and PIC conversion specifications within FORM statements, you can use *specification characters* to display numeric items in the format you want.

Some of the specification characters for each statement are different, some are the same; results are similar but not completely identical.

Specification Characters: Some of the most-used specification characters for these statements are shown in Figure 16.

Specification Character	Usage
#	Specifies position of each digit in a numeric field
Z	Specifies a digit position in a numeric field that is printed only if the digit is a significant digit (FORM statement only)
.	Specifies position of an actual decimal point in a numeric field
,	Specifies position of a comma in a numeric field
*	Specifies a digit position in a numeric field that is filled with an asterisk (*) if the digit is a nonsignificant zero
$	Specifies a dollar sign in a numeric field. A series of $ specifiers at the beginning of the field represents a floating $ character
+	Specifies for numeric data that a sign will be printed
-	Specifies for numeric data that, if the value is negative, a sign will be printed

Figure 16. IMAGE and FORM Statements — Most-Used Specification Characters

When you want to show the position of each digit in a numeric field, you can specify the # specification character. For example:

```
980 image:###
990 form pic(###)
```

During execution, when the field is displayed, you'll get:

Output Data	Displayed IMAGE Output	Displayed FORM Output
0001	1	001
0012	12	012
0123	123	123

Note that the IMAGE statement suppresses leading zeros when the specification character (#) is used, whereas the FORM statement does not. To suppress leading zeros in a FORM statement, you use the Z specification character, as follows:

```
300 form pic(zzz)
```

During execution, when the field is displayed, you'll get:

Output Data	Displayed FORM Output
0001	1
0012	12
0123	123

If you want to use a decimal point with a numeric field, you can specify the decimal point specification character. For example:

```
200 image:###.##
240 form pic(###.##)
```

During execution, when the field is displayed, you'll get:

Output Data	Displayed IMAGE Output	Displayed FORM Output
123.45	123.45	123.45
12.345	12.35	012.35
234.5	234.50	234.50

(Notice that the displayed field is justified on the decimal point.)

If you want to use the comma within a numeric field, you can specify the comma specification character. For example:

```
100 image:##,###
110 form pic(##,###)
```

During execution, when the field is displayed, you'll get:

Output Data	Displayed IMAGE Output	Displayed FORM Output
0003	3	00,003
0100	100	00,100
9875	9,875	09,875

If you want to use the asterisk to replace leading zeros within a numeric field, you can specify the asterisk specification character. For example:

```
300 image:****
310 form pic(***#)
```

During execution, when the field is displayed, you'll get:

Output Data	Displayed IMAGE Output	Displayed FORM Output
0003	***3	***3
0100	*100	*100
5000	5000	5000

If you want to use the dollar sign within a numeric field, you can specify the dollar sign specification character. For example:

```
425 image:$$$#
450 form pic($$$$)
```

During execution, when the field is displayed, you'll get:

Output Data	Displayed IMAGE Output	Displayed FORM Output
0012	$12	$12
0800	$800	$800
0003	$3	$3

Character strings may also be displayed using numeric specifiers. For example, if you have the statement

```
200 image:#####
```

and the character string WRONG, you will get

```
WRONG
```

printed in your field.

The specification character conversions in IMAGE and FORM are not exactly the same, as some of the previous examples show. See *IBM BASIC Language Reference* for the exact specification characters and positioning control for IMAGE and FORM statements.

Printing Formatted Output — PRINT USING Statement

Use the PRINT USING statement to print the data in the item list in the exact format specified in the IMAGE or FORM statement. For example:

```
120 print using xyz : A,B,C
.
.
.
180 xyz: image:   ****.##  $$$$#.##  $$$#.##
```

In the PRINT USING statement, the 180 specifies the line label of the IMAGE statement that specifies the format to be used in printing items A, B, and C.

You may use this form of PRINT if you want to print something requiring a special format, such as a check or a financial report.

The following examples demonstrate some of the capabilities of the IMAGE statement or the FORM statement referred to by the PRINT USING statement. If the low order data is longer than the image, the data is rounded when printed.

```
 * list
100 abc: image :##.##
110 input x
120 print using abc : x
 * RUN
 ? 10.107
10.11
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 120.
```

Note that the number was rounded to two decimal places.

Spaces in the conversion specification are included in the print line. For example:

```
 * list
100 imx: image : $$## *******.## $#### #,###,###.##
110 print using imx : 8,100,20,1785.59
 * RUN
  $ 8 ****100.00 $  20     1,785.59
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 110.
```

You may also use the IMAGE conversion specification to print character strings. These are used only for position information, *not* conversion of character strings.

```
 * list
100 abc: image : ####
110 def: image : >####
120 ghi: image : <####
130 jkl: image : >$$$$###
140 print using abc : "abc"
150 print using def : "def"
160 print using ghi : "ghi"
170 print using jkl : "jkl"
 * run
 abc
   def
 ghi
      jkl
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 170.
```

Notice that the keyword IMAGE is optional.

The following program demonstrates one possible use of the PRINT USING statement.

```
 * list
090 decimal p
100 jkl: image :$***,***,***.## dollars
110 input "first name:":a$
120 input "middle initial:":b$
130 input "last name :":c$
140 input "amount due:":p
150 print "pay to the order of ";a$;" ";b$;". ";c$
160 print using jkl : p
 * run
first name: john
middle initial: p
last name : jones
amount due: 1798.50
pay to the order of john p. jones
$******1,798.50 dollars
END OF MAIN PROGRAM ENCOUNTERED AFTER LINE 160.
```

If you want to use the FORM statement rather than the IMAGE statement, you can change statement 100 to:

```
100 form pic($***,***,***.##)," dollars"
```

Handling Exceptions — INPUT and PRINT

You can handle exceptions during input or printing with the exception handling facilities (see "Handling Exceptions" on page 129), or you can handle them with the EXIT, CONV, or SOFLOW clauses in the INPUT, LINE INPUT (LINPUT), or PRINT statements themselves. (CONV and SOFLOW are exception conditions that occur when the data is erroneous). For more information on CONV and SOFLOW, see "Exception Recovery" on page 123. For example:

```
 * list
100 dim a$*3
110 start: input x,a$ conv numb, soflow toolong
120 stop
130 numb: print "must be number"
140 goto start
150 toolong: print "string too long"
160 goto start
 * RUN
 ? abc,100
must be number
 ? 100,abcdef
string too long
 ? 100,a
STOP AT LINE 120.
```

The first value input, ABC, is not a number; therefore, the exception handling routine at 130 is executed. The second time data is entered, the string ABCDEF is too long; therefore, the exception routine at 150 is executed.

You can use an EXIT clause in the statements, instead of CONV and SOFLOW:

```
100 dim a$*3
110 start: input x,a$ exit this
120 stop
125 this: exit conv numb, soflow toolong
130 numb: print "must be number"
140 goto start
150 toolong: print "string too long"
160 goto start
```

and, with the same input, you will get the same results as before.

Setting Page Margins — MARGIN Statement

To set the left, right, top, and bottom margins of a page for output, use the MARGIN statement. Use the keywords LEFT, RIGHT, TOP, and BOTTOM to establish the settings.

Follow each keyword with a numeric expression; the value in the expression is rounded to a positive integer:

- LEFT n specifies the leftmost column position.
- RIGHT n specifies the rightmost column position.
- TOP n specifies where the first line of a page is printed.
- BOTTOM n specifies where the last line of a page is printed.

To specify all four margins for a page, you can enter:

```
100 margin top 5 left 10 right 100 bottom 55
```

In this example, the page margins would be: top margin is at line 5, left margin is at 10, right margin is at 100, bottom margin is at 55, and there are to be 51 lines on the page.

You can set margins for files and for terminals. When you set the margins for files, you specify the file reference number immediately after the keyword MARGIN. (This value is a numeric expression in the range from 0 to 255 preceded by a "#" character.) For example:

```
100 margin #3 top 5 left 10 right 100 bottom 55
```

If you do not specify a file number (or specify 0 as the file reference number), the margin statement applies to your terminal.

When the MARGIN statement is executed, the settings specified take immediate effect and remain in effect until changed by the execution of a MARGIN statement.

Using Files

Files provide you with a method of saving and retrieving data for long periods of time. In BASIC, files are activated and defined with the OPEN statement. This section describes the OPEN statement and how to use sequential, relative, keyed, and stream files. If you want to print a report that was prepared using BASIC, see "Accessing Display Sequential Files" on page 110. If you want to read data that is already there, see "Accessing Native Sequential Files" on page 109.

BASIC supports record files and stream files. Record files are a collection of records containing related data; each record is a collection of related fields. Record files can be sequential, relative, or keyed.

A stream file is a continuous stream of data, such as textual data, in which each field may be unrelated to any other field.

The sequence of operations for all files is:

- OPEN—activates the file.
- Operate on the file using the appropriate input/output statements.
- CLOSE—deactivate the file.

If you attempt to open a file that is already open, or to close a file that is already closed, you'll get an exception. Also, if you attempt any input/output before your program executes an OPEN statement, you'll get an exception. (Exceptions are discussed in more detail in the section "Exception Recovery" on page 123.)

Opening and Closing Files

Before you can use a file, you must activate or open it with an OPEN statement. You can open files for input (INPUT), output (OUTPUT), or for both input and output (OUTIN). For example:

```
100 open #3:name "empl", outin, internal, sequential
```

opens the sequential file named EMPL as file reference number 3 for both input and output processing.

In an OPEN statement for a file, you must specify

- A file reference number — which identifies the file within the program
- A file specification — which identifies the file to the system. See your IBM BASIC System Services manual for details on file specifications.

In addition, the following file attributes may be specified:

- The file access mode — input, output, or input/output
- The file type — display, internal, or native

- The file organization — sequential, relative, keyed, or stream
- The file pointer — begin or append/end
- The file record type — variable or fixed

(All the attributes above are defined and explained in the following sections.)

Note that a default is supplied if a file attribute is not specified. See *IBM BASIC Language Reference* for the defaults. Also, certain combinations of file attributes may not be allowed under some host operating systems or with some file access methods. See your IBM BASIC System Services manual for details.

Your program can deactivate or close a file by executing an END, STOP, CHAIN, or CLOSE statement. For example:

```
100 close #3:
100 end
100 stop
```

If you want to switch an input file to an output file (or vice versa) and continue operation of the same program, you must explicitly deactivate the file by using the CLOSE statement before reopening it. If you attempt to read an output file or write an input file, you'll get an exception.

The CLOSE statement deactivates the file; any subsequent OPEN statement reactivates the file and may also reposition it.

Record Files

Record files contain fields that are grouped together in records. An example of a record is all the data about a single employee. The file would contain records for all the employees in an organization.

Each record in a file may be of fixed- or variable-length. If every record is the same length, you use the keyword FIXED in the OPEN statement. Use the keyword VARIABLE if records have different lengths.

You would choose VARIABLE on your employee file if you were carrying hours worked for each job. Because employees can be assigned to a varying number of jobs, the data you're recording for each employee would vary.

A record file can have one of three types of organization based on the arrangement of the records within the file: sequential, relative, or keyed.

Using Sequential Files

Sequential files are organized serially. The position of a record depends upon the order in which you wrote it. The first record you write occupies the first record position in the file, the second record occupies the second position, and so forth. You can only read these records in the order in which you wrote them (sequentially). You would have to start from the beginning of the file and read records 1, 2, 3, 4, and 5 before you retrieved the data in the 6th record.

Typically, you should use a sequential file organization only if you plan to access all the records in a file. You might select this file organization for a transaction file in which you are recording all the transactions that occurred over a specific period of time. You could then produce reports in various sort sequences. You would probably never need to retrieve only one particular transaction.

Sequential files may have display, internal, or native format. The only differences between these three formats is the way in which the data is carried in the fields in each record and the I/O statements that you use. (See Figure 19 on page 121 for the I/O statements that you can use with each.)

Accessing Internal Sequential Files

The fields in the record are in BASIC internal format and may contain a series of numeric and string values, with each value preceded by an identification that describes the type and length of the value that follows.

The following example writes the number of years, principal, and interest as records in a sequential file with internal format:

```
100 integer yr,years : decimal intr,prin,rate
110 open #3: name "interest",output,internal,sequential, begin
120 input "principal,rate,years": prin,rate,yr
130 for years = 1 to yr
140    intr = prin*rate: prin=prin+intr
150    write #3: years,prin,intr
160 next years
170 close #3:
180 end
```

The WRITE statement at line 150 causes the variables YEARS, PRIN, and INTR to be written as a record to file #3.

If you want to write the file with new data (for example, you want to calculate the data based on a different interest rate), you can do this by merely running the program again. The OPEN statement deletes the previous data because it is an OUTPUT file with pointer BEGIN.

Accessing Native Sequential Files

Sequential files with native format are formatted to conform to the associated FORM statement.

To write the previous interest example as a sequential file with native format, you would change statements 110 and 150, and add a FORM statement (line 180) that describes the format of each field in the record.

```
100 integer yr,years : decimal intr,prin,rate
110 open #3: name "interest",output,native,sequential
120 input "principal,rate,years": prin,rate,yr
130 for years = 1 to yr
140    intr = prin*rate: prin=prin+intr
150    write #3,using 180: years,prin,intr
160 next years
170 close #3:
180 form ni, pd 8.2, nd
190 end
```

The FORM statement causes YEARS to be written to the first field in the BASIC internal integer format, PRIN to be written to the second field in packed decimal format with two fractional digits, and INT to be written to the third field in the BASIC internal floating decimal format.

Assuming you entered 10000, 0.05, and 4 as input values, the records would contain:

```
1  10,500.00  500.00
2  11,025.00  525.00
3  11,576.25  551.25
4  12,155.06  578.81
```

in the format described above.

The next time you access this file for either input or output, you should use the same FORM statement.

If you want to add more records to this file (for example, you want to calculate the data based on a different interest rate), you can do this by merely running the program again. The OPEN statement positions the file at its end, because it is an OUTPUT file. The additional records are entered at the end of the file.

Accessing Display Sequential Files

Display files have a format similar to the format used for printing at your terminal or on a printer.

You can use display files if you want to transfer data to a printer or to another system.

Display files can be used to prepare and print reports. Here, for example, are two programs: the first (DISPLAY1) creates a report as a display file:

```
 * load display1
 * list
100 rem DISPLAY1:
110 rem BASIC program to write a display file
120 integer i
130 open #2: "displ", display, output, begin
140 for i=1 to 10
150  print #2: " This is number "; i; "of 10"
160 next i
170 stop
180 end
 * run
STOP AT LINE 170.
```

The second program (DISPLAY2) prints the report on your screen:

```
* load display2
* list
100 rem DISPLAY2:
110 rem BASIC program to print a display file
120 dim line$*80
130 ! Can use file DISP1 which was written by DISPLAY1
140 input "Enter the name of the file:" : name$
150 open #2: name$, display, input
160 do
170  linput #2: line$ eof stp
180  print line$
190 loop
200 stp: print "  End of input file"
210 end
* run
Enter the name of the file: displ
This is number 1 of 10
This is number 2 of 10
This is number 3 of 10
This is number 4 of 10
This is number 5 of 10
This is number 6 of 10
This is number 7 of 10
This is number 8 of 10
This is number 9 of 10
This is number 10 of 10
 End of input file
END AT LINE 210.
```

The following example reads the data from the internal sequential file created above (see "Accessing Internal Sequential Files" on page 109) and writes the data both to your terminal and to a display file.

```
* list
85 decimal
90 dim chr1$*40
100 open #3: name "interest",input, sequential,begin,internal
110 open #4: name "disp",output,display,sequential
120 chr1$ = "years          amount          interest"
130 print #4: chr1$          !Writes heading to file
140 print chr1$              !Prints heading on your terminal
150 read #3: yrs,summ,intr eof 200
160 print #4,using 190: yrs,summ,intr
170 print using 190: yrs,summ,intr
180 goto 150
190 form pic(zzzz),pos 13,pic($$$,$$#.#) , &
 & pos 30,pic($zz,###.##)
200 ed$ = "end of calculation"
210 print #4: ed$
220 print ed$
230 close #3:
240 close #4:
250 end
* run
years          amount          interest
  1          $10,500.0         $  500.00
  2          $11,025.0         $  525.00
  3          $11,576.3         $  551.25
  4          $12,155.1         $  578.81
end of calculation
END AT LINE 250.
```

SEQUENTIAL in the OPEN statement at line 110 is redundant, as DISPLAY files may only have this organization; BEGIN at statement line 100 is also redundant, as INPUT positions the file to the beginning by default.

Positioning a Sequential File

The OPEN or RESET (RESTORE) statement allows you to position a file at its beginning or end by using the keyword BEGIN or END. The OPEN statement automatically positions a file to its beginning if you open it for input and to its end if you open it for output if positioning is not specified.

A way to write a program so that you can use it to both create a new file and add records is:

```
40 input "NEW/ADD": inp$
50 if uprc$(inp$) <> "ADD" then
60 open #3: name "interest",output,display,sequential,begin
70 else
100 open #3: name "interest",output,display,sequential
110 end if
.
.
.
190 end
```

If you enter the word ADD, the program opens the file at the end of all the records. If you enter the word NEW (or any word except ADD), it opens the file at the beginning of the file. Then the next time you write to the file, you will start at the beginning, as though it were a new file.

Another way to do this is to use the SCRATCH statement, as follows:

```
40 input "NEW/ADD": inp$
50 open #3: name "interest",output,display,sequential
60 if uprc$(inp$) <> "ADD" then scratch #3
.
.
.
190 end
```

In the example above, line 60 sets up the file so that the next operation works on the beginning of the file. Instead of two OPEN statements, this example uses SCRATCH for the case when the file is overwritten. (Note: SCRATCH erases the file.)

Using Relative Files

Relative files are record files in which record numbers determine their position in the file. Regardless of the order in which you create the records, they are stored according to the record number used in the write statement. For example, if you create records in the following order: record 2, record 1, record 9, record 4, record 10, the records are stored in the file as shown in Figure 17 on page 113.

The fields in each record of a relative file must have a NATIVE format, which means that you use a FORM statement whenever you are accessing them. You must *write* to a file with the record number (directly). You can *read* from a relative

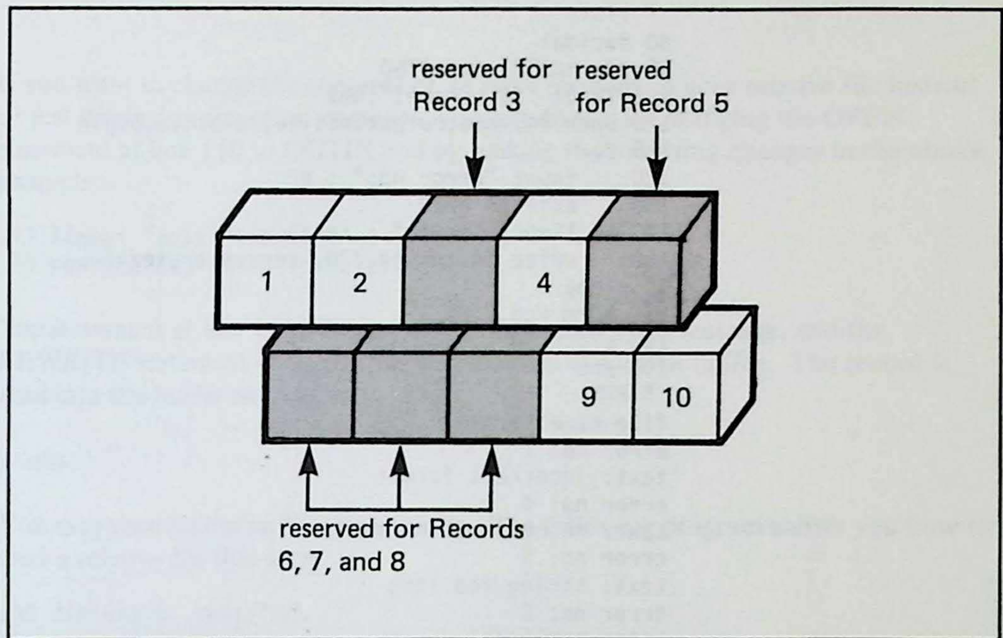


Figure 17. Relative Files — Record Sequence

file either sequentially or directly. When you read a relative file sequentially, "empty" or "null" records are skipped over.

You might use this organization for an error message file in which the error messages are numbered consecutively and you want to retrieve a message by number. You could retrieve the record with one access based on the error number (that is, record number). However, if there are large gaps in the numbering sequence, you probably shouldn't select this organization as space is allocated for all the missing numbers.

Accessing Relative Files

When you open a relative file, you use the keyword RELATIVE in your OPEN statement. To access a record in a relative file, you include the keyword RECORD (REC) with the associated record number in the input/output statement. (See Figure 19 on page 121 for the I/O statements you can use with relative files.)

The following program creates a message file whose error number corresponds to the record number. BEGIN causes any previous file with the same name to be scratched first and gives you an empty file to work with.

```
 * list
80 decimal
90 dim nm$*8, text$*80
100 input "file name:":nm$
110 open #4: nm$,output,native,relative,begin
115 do
120    input "error no:" : er
130    exit if er=0
140    linput "text:" : text$
150    write #4, using 170, rec=er:er,text$
160 loop
170 form nc5,c 80
180 close #4:
190 end
 * RUN
file name: ermsg
error no: 1
text: incorrect format
error no: 4
text: record not on file
error no: 5
text: string too long
error no: 2
text: pgm loop
error no: 0
END AT LINE 190.
```

If you wanted to retrieve and display any error message by number from the relative message file, you could use the following program:

```
 * list
90 dim nm$*8, text$*80
100 input "file name":nm$
110 open #4: nm$,input,relative,native
120 do
130    input "error no:":er
140    exit if er=0
150    read #4,using form1, rec=er:er,text$ norec prntno
160    if er <> 0 then print using form2 : er,text$
170 loop
180 close #4:
190 stop
200 prntno: print using form3 :er
210    er=0
220 continue
230 form1:  form nc5, c 80
240 form2:  form pic(zzzz#),x 1,c 80
250 form3:  form "record",pic(zzzz)," not on file"
260 end
 * RUN
file name ermsg
error no: 5
    5 string too long

error no: 3
record   3 not on file
error no: 0
STOP AT LINE 190.
```

During execution, the program retrieves error message number 5 and prints "5 string too long". When you request error number 3, there isn't any corresponding record, so the message "record 3 not on file" is printed.

Updating a Relative Record

If you want to change the contents of an error message in your relative file instead of just displaying the error messages, you can do this by changing the OPEN statement at line 110 to OUTIN and by making the following changes in the above example:

```
162 linput "text:":text$
165 rewrite #4, using forml:er, text$
```

The statement at line 162 allows you to enter a new error message, and the REWRITE statement at line 165 updates the record in the buffer. The record is read into the buffer at statement 150.

Reading Relative Files Sequentially

You may read a relative file sequentially. The following program shows you how to read a relative file this way:

```
100 dim nm$*8, text$*80
110 open #4:"ermsg", input, relative, begin, native
120 goread: read #4, using al:er, text$ eof endl
130 al:form nc5, c 80
.
.
.
200 goto goread
.
.
.
300 endl:end
```

The OPEN statement at line 110 positions the file to the first record, because it is INPUT. The READ statement at line 120 reads the file records sequentially, because it does not use the keyword RECORD or REC.

When you read a relative file sequentially, the records are read in ordered sequence. Empty or null records are skipped.

Positioning Relative Files

When you are accessing files sequentially, you can position relative files at the beginning of the file with the BEGIN clause on an OPEN or RESET statement. You can also position a relative file to a specific record number by using the RECORD or REC clause in the RESET statement. For example:

```
500 reset #8, record=20:norec 1000
```

positions the file to record number 20. If no such record exists, the program transfers to statement 1000.

If you open the file for OUTPUT with BEGIN, the file will be scratched first. If you do not want the file to be scratched, do *not* specify a pointer clause when opening for OUTPUT. If positioning is not specified for an INPUT or OUTIN relative file, the OPEN statement automatically positions a file to its beginning.

Using Keyed Files

Keyed files are organized by the key of each record. The key for each record is a string of characters contained at a specific location within the record. The location and length of the key are the same for all records in the file, but the values must be different.

Keyed files must have NATIVE format, which means that you use a FORM statement whenever you are accessing them.

You can access any record directly by using the record key, or you can access all of them sequentially (that is, in the order in which the keys collate). For example, if the record key is EMPNO, and you place the records into the file in the following order: EMPNO=5001, EMPNO=3246, EMPNO=4666, the records are ordered as shown in Figure 18.

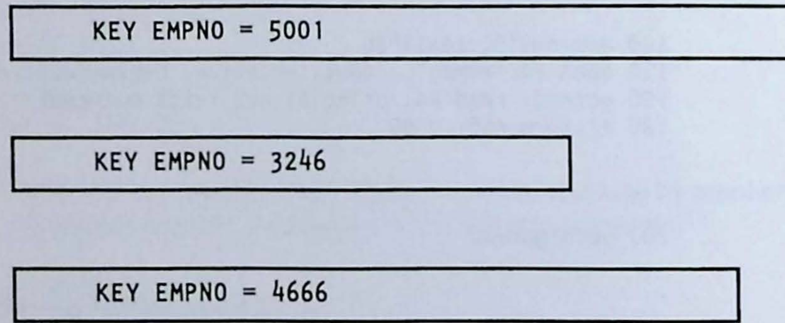


Figure 18. Keyed Files — Record Sequence

A keyed file lends itself to any data from which you want to retrieve a record based on some known piece of information. For example, in an employee file, you could key the file on employee number. If you then wanted to display the data about that employee, you could retrieve that record with one access based on the employee number. You might want to key the information on employee name, but this could present a problem, as the name might not be unique (for example, John Smith).

Keyed files can only be used under VSAM (Virtual Storage Access Method). Further, the attributes of your keyed file must be specified to the host system before you invoke BASIC to execute your program. See "Using VSAM" on page 125 for a discussion of VSAM, or see your IBM BASIC System Services manual for a complete description of the way in which you define VSAM files.

Accessing Keyed Files

When you open a keyed file, you use the keyword KEYED. (See Figure 19 on page 121 for the I/O statements that you can use with keyed files.)

The following program can be used to create a keyed file named EMPLOYEE. The records are keyed on employee number, which is a character field in positions 1 to 5 in each record. The program is designed to recognize an employee number of "last" as the last of the input.

```
100 dim emp$*5,nm$*8,addr$*65,dayt$*8,name$*20
105 decimal s,h
110 input "file name": nm$
120 open #9: nm$,outin,keyed,native
130 start: input "employee number": emp$
140 if emp$ = "last" then goto done
150 line input "name": name$
160 line input "address": addr$
170 input "date,wage,hours": dayt$,sal,hour
180 write #9,using a1: emp$,name$,addr$,&
  & hour,dayt$,sal dupkey b1
190 a1: form c 5, c 20,c 65,n6.2,c 8,n6.2
200 goto start
210 b1: print "employee already on file"
220 goto start
230 done: close #9:
240 end
```

The following is an example of the operation of the program, with sample input.

```
 * run
file name employee
employee number 100
name jone,j.p.
address 123 45 st, city
date,wages,hours 10/01/80,20,0
employee number 101
name smith,p
address 123 45 st, city
date,wages,hours 10/01/80,30,0
employee number  200
name butler,j
address 876 53 st, city
date,wages,hours 10/01/80,25.30,0
employee number 150
name cook,b
address 85 12st, city
date,wages,hours 10/01/80,27.62,0
employee number last
END AT LINE 240.
```

Additional records can be added to this file at any time. You can use the same program.

If you wanted to retrieve a single employee record to print the name at your terminal, you could use the next program. In this example, it is the KEY clause in line 150 that specifies the key value of the right record.

```
 * list
100 dim emp$*5,nm$*8,name$*20
110 input "file name": nm$
120 open #9: nm$,outin,keyed,native
130 start: input "employee number": emp$
140 if emp$ = "last" then stop
150 read #9,using a1,key=emp$: emp$,name$ nokey b1
160 a1: form c 5,c 20
170 print name$
180 goto start
190 b1: print "employee not on file"
200 goto start
210 end
 * RUN
file name employee
employee number 150
cook,b
employee number 201
employee not on file
employee number last
STOP AT LINE 140.
```

Updating a Keyed Record

If you wanted to enter the hours an employee worked instead of just displaying the employee's name, you could do this by inserting the following four statements after line 170 in the above example.

```
174 decimal hours
175 input "hours": hours
176 rewrite #9,using c1: hours
177 c1: form pos 91,n6.2
```

REWRITE in line 176 changes only the value HOURS, which is a numeric value starting in position 91 of the record. The READ statement at line 150 places all the values of the record into a buffer. Then REWRITE moves HOURS into position 91 of the buffer and rewrites the entire record. This REWRITE statement does not specify the key. It just changes the data in the last record read. One can also specify the key of the record to be rewritten.

The following is an example of the operation of this changed program:

```
 * run
file name employee
employee number 150
cook,b
hours 72.5
employee number last
stop at line 140
```

In order not to inadvertently change the wrong employee record, you should insert three other statements between lines 170 and 174, to verify that the name of the employee is the one you want.

```
171 check: input "correct employee? y/n": na$
172 if lwrc$(na$(1:1)) = "n" then goto start
173 if lwrc$(na$(1:1)) <> "y" then goto check
```

Reading Keyed Files Sequentially

When you read a keyed file sequentially, the records are read in ordered sequence, regardless of whether they were created in order. The order of a keyed file is the sequence in which the keys collate. The following program shows you how to read a keyed file this way.

```
100 dim emp$*5,nm$*8,name$*20,addr$*65
110 input "file name": nm$
120 open #9: nm$,input,keyed,begin,native
130 do
140     read #9,using a1: emp$,name$,addr$ &
    &   eof end1
150 a1: form c5, c20, c65
160     print name$;tab(30); emp$
170     print addr$
180 loop
190 end1: print "end of file"
200 close #9:
210 end
```

The OPEN statement (line 120) positions the file to the first record because it is INPUT (BEGIN is redundant). The READ statement (line 140) reads the next record on the file, because it does not use the keyword KEY; thus, the records are read in consecutive order. The output from the above program is in employee number sequence:

```
jone,j.p                    100
123 45 st,city
smith,p                     101
123 45 st,city
cook,b                      150
85 12 st,city
butler,j                    200
876 53 st,city
end of file
END AT LINE 210.
```

Positioning Keyed Files

The OPEN or RESET statement allows you to position a file at its beginning by using the keyword BEGIN. If positioning is not specified, the OPEN statement automatically positions a file to its beginning if you open it for INPUT or OUTIN. If you specify BEGIN and OUTPUT, the file will be scratched first. If you do *not* want the file scratched, do not specify a pointer clause when opening for OUTPUT.

You can position keyed files to the record whose key value satisfies the expression specified in the KEY or SEARCH clause. For example:

```
100 reset #10,key = "abc":nokey 200
```

positions the keyed file #10 to the record whose KEY is ABC. If no such record exists, it transfers to statement 200.

Using Stream Files

Stream files are organized sequentially as a continuous stream of numeric and/or character data.

The file is organized sequentially, so if you want to read the fifth value on the file, you must start at the beginning of the file and read all the intervening values.

Each data item has an internal format. This means that the data is carried in the BASIC internal format, with an indicator preceding the field that describes the type and length of the field. The variables in your item list in any input statement must have the same data type as the values in the file. For example, if you write a value as a character string, you must retrieve it as a character string, and if you write a numeric value, you must retrieve it as a numeric value. Numeric values are handled exactly as they are in the LET statement (that is, decimal can be converted to integer, and vice versa).

See Figure 19 on page 121 for the I/O statements that you can use with stream files.

Accessing Stream Files

STREAM files can be accessed only sequentially. (You can write or read them only in sequential order.) To read them, you can use READ, GET, or INPUT; to write them, PUT or WRITE.

The following example writes the year, the principal, and the interest as a stream of data to file #3:

```
 * list
100 integer yr,years : decimal prin,intr,rate
110 open #3: name "interest",output,internal,stream
120 input "principal,rate,years?":prin,rate,yr
130 for years=1 to yr
140    intr=prin*rate: prin=prin+intr
150 put #3: years,prin,intr
160 next years
170 close #3:
180 end
 * RUN
principal,rate,years? 10000, .05, 4
END AT LINE 180.
```

The PUT statement at line 150 causes the variables YEARS, PRIN, and INTR to be written to file #3, each as a stream of characters.

Type	Organization	Statements	Use
DISPLAY	SEQUENTIAL	INPUT LINE INPUT PRINT RESET BEGIN RESET END SCRATCH	You would use this file type when you wanted to save data that is in a printer format. You might create a file which could be spooled to a printer or output to a tape.
INTERNAL	STREAM	INPUT GET PUT WRITE RESET BEGIN RESET END SCRATCH	You would use this file type when each record has a single value. The length of each record may vary, as the value in the record is described in the record. You could use this as a text file in which each word is a separate field.
INTERNAL	SEQUENTIAL	INPUT READ WRITE GET RESET BEGIN RESET END SCRATCH	The only difference between a SEQUENTIAL, NATIVE and a SEQUENTIAL, INTERNAL file is the format of the records themselves.
NATIVE	SEQUENTIAL	READ USING WRITE USING REWRITE USING REREAD USING RESET BEGIN RESET END SCRATCH	NATIVE or INTERNAL sequential files can be used the same way. You would use one of these file types when you are saving data in consecutive sequence and you only want to "report" or read the data from the beginning. For example, a transaction register that maintains everything that is entered in the exact order it was entered.
NATIVE	RELATIVE	READ USING WRITE USING DELETE REC RESET REC RESET BEGIN REREAD USING REWRITE USING SCRATCH	You would use this file type when you know that the different records can be numbered. You might use this for error messages, or if you're keeping track of the occurrence of a particular number.
NATIVE	KEYED	READ USING DELETE KEY WRITE USING REREAD USING RESET KEY RESET BEGIN REWRITE USING SCRATCH	You would use this file type when a particular piece of data that you're using is always unique (for example, employee number) and you want to access individual records rapidly. You could use this file organization for Inventory of parts, Employee data, Bank account data, and so forth. The file must be a VSAM file.

Figure 19. I/O Statements Valid for Each File Organization and Type

This example retrieves the values from file #3 and prints the values at your terminal:

```
 * list
100 integer aa: decimal bb,cc
110 open #3: name "interest",input, stream, internal
120 print "years      amount      interest"
130 do
140    get #3: aa, bb, cc eof 180
150    print using 170: aa, bb, cc
160 loop
170 form pic(zzzz#),pos 9,pic($zz,###.##) , &
  & pos 24,pic($zz,###.##)
180 end
 * RUN
years       amount        interest
    1   $10,500.00     $   500.00
    2   $11,025.00     $   525.00
    3   $11,576.25     $   551.25
    4   $12,155.06     $   578.81
END AT LINE 180.
```

It's not necessary to use the same variable names you used in creating the file, but character data must correspond to character variables and numeric data must correspond to numeric variables. BASIC will automatically convert between numeric data and numeric variables of different type.

Positioning STREAM Files: The OPEN or RESET statement allows you to position a stream file at its beginning or end by using the keyword BEGIN or END. If positioning is not specified, the OPEN statement automatically positions a stream file to its beginning if you open it for input, and to the end if you open it for output. If you open a file for OUTPUT and BEGIN, the file will be scratched.

Using Array I/O Statements

You can use arrays to retrieve data from files or to write data to files. You can specify that the I/O statement is only to process arrays, by inserting the keyword MAT at the beginning of the statement. Or you can include arrays in the item list for the I/O statement, preceding each with the keyword MAT.

When you read or write arrays, all the values associated with the rightmost subscript are transferred, then the next subscript to the left is incremented and again the values associated with the rightmost subscript are transferred. This continues until all the dimensions have attained their maximum value.

For example, if you have an array A(2,3), with base 1, with the following values:

```
a(1,1) = 5         a(1,2) = 6          a(1,3) = 7
a(2,1) = 10        a(2,2) = 20         a(2,3) = 30
```

the data is transferred in this order: 5, 6, 7, 10, 20, 30.

If you are reading character strings from a record and transferring them to an array, the length of the character string in the file cannot be longer than the maximum allowed for each array element. If it is, you'll get a string overflow exception condition (SOFLOW).

If you are reading numeric values, the same rules apply for converting between numeric data types as apply to the LET statement.

If you are writing data from arrays to a file, the length of the character string is the length of an element of the array if the file has an INTERNAL format, or it is the length as specified in the FORM statement if the file has a NATIVE format. Numeric data follows the same rules (that is, length of element or length as defined in the FORM statement).

Examples of I/O statements that access arrays (BASE 0 option is in effect):

```
100 decimal
110 rem read a decimal array - internal format
120 dim a(3,2)
130 open #3: "example1",internal,input
140 mat read #3: a

100 real single
110 rem read a real single array - native format
120 dim a(3,2)
130 open #3: "example2",native,input
140 mat read #4,using a1: a
150 a1: form 12*n5.2

100 rem write a character string array - internal format
110 dim chr1$(2,3,4)
120 open #3: "example3",internal,output
130 mat write #3: chr1$

100 rem write a character string array - native format
110 dim chr1$(2,3,4)
120 open #3: "example4",native,output
130 mat write #3,using a1: chr1$
140 a1: form 60*c20
```

Other than the array, you can transfer other values to or from the file. In fact, you can carry in the file the actual number of entries in each dimension. This could be useful, if the length of arrays to be read varies from one record to another.

For example, assume you have a two-dimensional array that varies in size; you can write the size of each dimension as shown here:

```
100 write #3:udim(a,1),udim(a,2),mat a
```

The first value on the record is the size of the first dimension, the second is the size of the second dimension, followed by all the values in the array named A.

On input you can use the first two values to redimension an array:

```
200 input #3: i,j,mat a(i,j)
```

Exception Recovery

While processing I/O statements, you can provide exception recovery from some I/O exceptions by including exception recovery clauses in your I/O statements. The exception recovery clause contains a keyword describing the exception, and a line number or line label to which control is transferred if an exception occurs. Figure 20 on page 124 describes the keywords with their meanings. Figure 21 on page 125 describes which keywords may be used with each I/O statement.

Keyword	Description
CONV	The value cannot be converted from or to the format of the field, or the record length has been exceeded.
DUPKEY	The key you requested for a new record already exists.
DUPREC	The record number you requested for a new record already exists.
ENDPAGE	The last line written equals the line count for a page, where a page is defined as the number of lines specified in the BOTTOM clause of the MARGIN statement that currently is in effect for the associated file or the terminal.
EOF	The last record was already read, or there is no more room on the file to accommodate the new record.
IOERR	A hardware exception has occurred. This may prevent other exception conditions from being recognized.
NOKEY	The key you requested doesn't exist.
NOREC	The record number you requested doesn't exist.
SOFLOW	The character string being processed is too long.
EXIT	An exception of some kind has occurred. *Note:* If you use EXIT, you may not use any of the other keywords. Furthermore, the statement that you are referring to must be an EXIT statement.

Figure 20. Exception Recovery Clauses

If you include multiple exception recovery clauses, only the first exception encountered is honored. For example:

If the record was written in the NATIVE format with the first field an 8-character string and the second field a 20-character string,

```
100 dim b$*10
110 read #2,using a1 : a,b$ conv 180, soflow 190
120 a1: form c8,c20
```

control would transfer to line 180, because the conversion exception occurred on the first field. You are trying to read a character value into the numeric variable A. You will only discover the character string overflow condition, after you've changed the program (changed a to a$):

```
100 dim b$*10
110 read #2,using a1 : a$,b$ conv 180,soflow 190
120 a1: form c8,c20
```

When you run these statements, you will discover that you have a string overflow, because B$ is only 10 bytes long, and the data in the record is 20 bytes long.

	CONV	DUPKEY	DUPREC	ENDPAGE	EOF	IOERR	NOKEY	NOREC	SOFLOW	EXIT
OPEN						X				X
CLOSE					X	X				X
RESET						X	X	X		X
GET	X				X	X			X	X
PUT					X	X				X
READ	X				X				X	X
READ File	X				X	X	X	X	X	X
REREAD	X					X			X	X
WRITE	X	X	X		X	X			X	X
REWRITE	X				X	X	X	X	X	X
INPUT	X					X			X	X
INPUT FIELDS	X					X			X	X
INPUT File	X				X	X			X	X
LINPUT						X			X	X
LINPUT File					X	X			X	X
MARGIN										
MARGIN File										
PRINT	X					X			X	X
PRINT FIELDS	X					X			X	X
PRINT File	X			X	X	X			X	X
SCRATCH						X				X
DELETE						X	X	X		X

Figure 21. Exception Recovery Clauses for I/O Statements

When more than one statement has the same list of exception recovery clauses, it may be more convenient to place the clauses in an EXIT statement and then refer to the EXIT statement. For example:

```
100 dim b$*10
110 read #2 using 120 : a$,b$ exit 190
120 form c8,c20
190 exit soflow 200,conv 300
200 rem handle soflow
300 rem handle conv exception
```

Note that an EXIT clause cannot appear with other exception recovery clauses. For example:

```
* 100 rem the following statement is illegal,
* 110 rem because exit and conv are mutually exclusive.
* 120 read #2 : a$,b conv 180, exit 190
                                1
###1) BAS30116S 'EXIT' NOT ALLOWED IN A STATEMENT WITH OTHER ERROR CLAUSES.
```

Using VSAM

VSAM (Virtual Storage Access Method) provides a powerful and flexible method of manipulating files.

VSAM files can have the following organizations: sequential, relative, keyed, or stream.

If your program processes VSAM files, you must first predefine the attributes of the file to the host system, and these attributes must correspond to the attributes of your OPEN statement. To guarantee that there is a match between the host system and your BASIC program, you must provide the proper documentation describing the BASIC program requirements so that you or your user can provide the correct data when the program is executed.

If the program uses keyed files, you must use VSAM, as it is the only access method that supports this organization. Further, you must ensure that the key for the record starts and ends in the location you specified to the host system. For example, if the program expects the key to start at position 10 in the record and to be 5 characters long, you must specify this exactly to the host system. If you specify a different key position or length, the program will not be able to retrieve a record on key. The program will only be able to read the file sequentially.

There are two intrinsic functions provided in BASIC that can be used to find the starting position of the key and the length that were specified to the host system. They are:

KPS(M) provides the starting byte position of the key for file #M.

KLN(M) provides the number of positions the key occupies for file #M.

These functions can be helpful, if you are reading a foreign file (that is, one created by another program or system) and you know the format of each record, but are unsure of the key position and length, or if you want to verify that the attributes assigned to the host system are the same as those of your program. You could provide an exception recovery procedure in case they aren't, so that the user doesn't attempt to create records incorrectly, or fail in an attempt to read any record.

See your IBM BASIC System Services manual for a description of the way you define VSAM files to the host system.

Displaying and Purging Your Files

You should list all your program and data files on a regular basis. You should review them, determine which ones you no longer need, and then purge these from the system. This provides you with space that can be used by new programs and data, in addition to keeping the number of your files to a manageable size.

Be careful when you purge a file — check what you have typed *before* you press the ENTER key. There is nothing more disconcerting than realizing you're purging the wrong program or file just as you've hit the ENTER key.

Displaying Your Files — QUERY Command

Use QUERY when you want to examine the files you have saved, stored, or written by your program(s).

```
query
```

or

```
query all
```

lists all your files.

```
query program
```

lists all the programs you have saved with the SAVE command. (Note that this command lists all BASIC files that are source files containing your source programs.)

```
query file
```

lists all the data files you have created for, or by, running your programs.

```
query workspace
```

lists all the programs you have stored with the STORE command.

```
query aa
```

displays information about the file(s) whose name is AA.

```
query aa*
```

displays information about the file(s) whose name begins with AA.

The results of issuing the QUERY command will be displayed differently depending on which operating system BASIC is running under. Note that some operating systems do not support more than one file type; on such systems, the keywords PROGRAM, FILE, and WORKSPACE are ignored. See your IBM BASIC System Services manual for details and restrictions on the QUERY command.

Normally, QUERY displays the data at your terminal; however, you can direct the output to a printer file by including the keyword OUT. For example:

```
query all out (xyz)
```

writes a listing of all your files to a listing file named XYZ.

Deleting Your Files — PURGE Command

Use PURGE when you want to delete files. PURGE deletes all the files with the name specified and makes the space available for use. Remember that your operating system may allow several different source, workspace, object, and data files all with the same name but with different file types, and PURGE will delete them *all*. For example:

```
purge stat
```

deletes the file or files named STAT and makes the space available. If there are multiple file types all with the name STAT, *all* the files (source,workspace, object, and data files) with the name STAT will be deleted.

PURGE does not affect your workspace or change the current line number.

If you attempt to PURGE a file that isn't there, you will receive an error message.

Note that some operating systems do not support more than one type of file. See your IBM BASIC System Services manual for details and restrictions.

Handling Exceptions

An unexplained program halt or a confusing message can leave us wondering where to begin solving our problem. You can prevent this type of experience by providing a means for the program to regain control after an exception has occurred.

An exception usually occurs when the system encounters something illegal. A division by zero is a typical example. Other commonly experienced exception conditions are: attempting to open a nonexistent disk file, trying to write beyond the end of a file, referring to an array element outside the bounds of the array, and so forth.

For every line of code you write, be aware of the error potential. If there is any possible circumstance that could cause an exception, be assured that it will happen.

BASIC provides four primary exception handling statements: ON condition, CAUSE, CONTINUE, and RETRY.

Exceptions — ON Condition Statement

The ON condition statement determines the action to be taken when an exception occurs. After an ON condition statement is executed, it remains in effect until a new ON condition statement for that condition is executed.

You are allowed three action options when an exception occurs:

- Transfer control to a specified line number or label (GOTO).
- Ignore the exception (IGNORE).
- Permit the BASIC system-defined action to occur (SYSTEM).

There are two parts to an ON condition statement. The first part, called the condition, immediately follows the word ON. Following the condition is the action part, describing what action to take under the given condition. For example:

```
100 on error goto error_routine
```

transfers control to line label error__routine on an exception condition.

At error__routine you could print a message concerning the exception and ask the user whether to ignore the exception and continue, to halt the program, or to perform some other operation. That is your decision. You have "trapped" the exception, forced a transfer to a specific line, and your program remains in control.

"ERROR" is a general condition that applies to those exceptions not covered by the list of more specific conditions, shown in Figure 22 and Figure 20 on page 124.

Condition	Applies to
ATTN	"attention" interrupt from the terminal
CONV	improper conversion of numeric data
ENDPAGE	attempt to print a new line beyond the bottom margin of the current page (see "Setting Page Margins — MARGIN Statement" on page 106.)
OFLOW	numeric overflow
SKEY	pressing one of the "PF" keys on a 327x terminal while running a program in SCROLL mode
SOFLOW	string overflow
UFLOW	numeric underflow
ZDIV	division by zero

Figure 22. Specific ON Conditions

Using Specific ON Conditions

You can use the ON condition statements for all the conditions described in Figure 22. You code the condition following the keyword ON and code the action you want your program to take when the exception condition occurs. To show you the actions of an ON condition statement, the division by zero condition with all possible actions is described in detail.

```
100 on zdiv ignore
110 let c = a/b
```

causes the value in C to be replaced with the largest number that can be represented with the same sign as A without interrupting the program when the variable B has a value of zero.

```
100 on zdiv system
110 let c=a/b
```

prints a message and causes the value in C to be replaced with the largest number that can be represented with the same sign as A.

```
100 on zdiv goto message
110 let c=a/b
```

causes program control to be transferred to the line labeled MESSAGE when the variable B has a value of zero (C is unchanged).

The three action choices, IGNORE, SYSTEM, and GOTO, are valid for any ON condition statement except CONV and ERROR. The CONV and ERROR

conditions do not allow the action IGNORE. For a description of the nine conditions and their actions, see *IBM BASIC Language Reference*.

Exceptions — I/O Statements

You can include exception handling routines in your I/O statements. For a discussion of this capability, see "Exception Recovery" on page 123.

When writing I/O statements that handle exceptions, you should be aware that there are three conditions that are the same as in the ON statement. These are CONV, SOFLOW, and ENDPAGE. If you have written these clauses in the I/O statements themselves or in an associated EXIT statement, they override the action of the ON condition statement. For example:

```
100 on conv goto 900
.
.
.
200 input #2 : a,b conv 350
```

Control transfers to statement 350 rather than 900 if the data being input to A is not numeric.

Certain exception conditions that may only be specified in an ON statement can occur while an I/O statement is being executed. If this occurs and you have an active ON condition statement, the action specified in the ON condition takes effect.

If an SKEY exception occurs during the execution of an INPUT or INPUT FIELDS statement, you can use the GOTO action to determine which key was pressed.

```
100 on skey goto 980
110 input a$
.
.
.
980 xyz=keynum
990 on xyz gosub 1000,2000,3000,4000 none 5000
```

transfers control to line 980 if a PF key is pressed during the execution of the INPUT statement at line 110. The subroutine that is executed depends on which PF key was pressed.

Notice that line 990 is an ON GOSUB statement and not another ON Condition statement. The result of this type of exception handling is to transfer control to a different subroutine for each PF key pressed. Frequently repeated subtasks can thus be invoked with a single key stroke, which is a major convenience for users of your program.

Displaying Exception Codes

When an exception occurs, you can use the intrinsic functions ERR, LINE, and CODE to display information about this exception. ERR gives the BASIC exception code, LINE gives the line number at which the exception occurred, and CODE gives the host operating system exception code. You can also display a message that refers the user to a HELP sequence and/or to Appendix A of *IBM BASIC Language Reference*, which lists the exception codes and their meanings. For example:

```
100  on error goto to_err
200  gsscls = 202113793
900  call gddm (gsscls)
1000 to_err :  ! error handling routine
1100   print 'line of error=';line;' err=';err;' code= ';code
1200 end
```

Here is another example:

```
700  on error goto to_err
800  dim a$(10),b$(2)*5,c$(10)
900  mat a$=b$
1000 do                ! The loop for reading
1100  for i=1 to 10
1200      read a$(i)
1300  next i
1400  data '1','2','3','4','5','6','7','8','9','10'
1500  for i = 1 to 10 ! The loop for printing
1600      print a$(i);' ';
1700  next i
1800  goto fini         ! end
1900  to_err :          ! error handling routine
2000     Print 'line of error=';line;' err=';err;' code= ';code
2200  select err
2300      case 2001   ! Subscript out of bounds
2400          mat a$=c$
2500      case 8001   ! No more data to read
2600          RESTORE
2700      case else
2800          a=b
2900  end select
3000 loop
3100 fini :            ! the end
3200 end
```

Creating Exception Codes — CAUSE Statement

You are not limited to the exception codes provided by BASIC if you want to intentionally "cause" an exception to occur under some condition peculiar to your program. The CAUSE statement may be followed by any numeric expression. The rounded integer value of this expression *is* the exception code, which may or may not be a BASIC exception code. CAUSE 10, for instance, produces the exception code 10, which is available through the ERR function as above. Ten is not a BASIC exception code, so you can assign it any meaning you wish.

Reprocessing After an Exception — RETRY Statement

There may be some circumstances in which you want a statement to be executed again if it causes an exception. The statement RETRY reexecutes the statement at which the exception occurred. For example:

```
100 dim aa$*2
110 on soflow goto 500
120 input "-character code" :aa$
.
.
.
500 print "too long"
510 retry
```

If you responded to the prompt at 120 with 3 or more characters, the retry at 510 would cause the input statement at 120 to be re-executed.

```
400 input a$ soflow 2000
.
.
.
550 input b$ soflow 2000
.
.
.
2000 print "that string is too long. please try again."
2010 retry
```

If the value entered at either of the INPUT statements is too long, statement 2000 prints a message and the RETRY statement at 2010 causes the statement for which the exception occurred (400 or 550) to be re-executed.

Continuing After an Exception — CONTINUE Statement

CONTINUE, however, causes control to be given to the statement immediately *following* the statement causing the exception. For example:

```
100 on zdiv goto e10
110 a=a/b
120 input b
.
.
.
500 e10:print "zero divide error at line";line
510 continue
```

If a division by zero exception occurs, control is transferred to an exception recovery routine at line 500, where the message "zero divide error at line 110" is printed. The CONTINUE statement causes control to be returned to line 120.

Exception Handling Considerations

Because exceptions can occur almost anywhere in your program, you must carefully consider how your program is going to handle them. The following describes areas where problems can occur if an exception condition occurs in your program.

RETRY and CONTINUE Without an Exception

If your program executes a RETRY or CONTINUE statement and an exception has not previously been intercepted by the program, an exception is generated by the RETRY or CONTINUE statement itself. For this reason, you should keep these statements within your exception handling routines. Then they cannot be encountered unless an exception exists.

Exception Loops

After an ON condition statement is executed, the indicated action is taken whenever the exception condition occurs. The ON condition statement continues in effect until a new ON statement with the same condition is executed within the same program unit (that is, main or subprogram). The ON is not deactivated when the condition occurs. Thus, infinite loops can be a danger, as shown in the following example:

```
100 on zdiv goto 200
110 i=0
120 k=j/i
.
.
.
200 print "division by zero at line ";line
210 x=y/i
.
.
.
```

This program will loop indefinitely between lines 200 and 210, because the statement at line 210 causes a zero divide exception, which transfers control to 200.

Because the program is operating interactively, the terminal user can issue an attention interrupt to regain control. The capability of interrupting a loop in a program operating in batch mode is dependent on the host system.

You can avoid this situation by writing your exception handlers in such a way that they do not contain exception-causing statements, or they deactivate exceptions as soon as they get control (ON...IGNORE or ON...SYSTEM) and reactivate (ON...GOTO) upon exit.

RETRY may also cause an endless loop if the problem is not adequately corrected. For example:

```
100 on zdiv goto 200
110 i=0:j=10
120 k=j/i
.
.
.
200 rem handle z-div
210 j=2
220 retry
```

will cause an endless loop since the wrong variable is changed.

Losing the Location of an Exception

After an exception occurs and causes a transfer of control, be careful in the handling of subsequent exceptions. If another exception occurs that you have not specified in an ON statement with the IGNORE action, all knowledge of the first exception (LINE, ERR, CODE) is lost. Therefore, a RETRY or CONTINUE statement may cause an exception to occur. For example:

```
100 dim a$*1
110 on soflow goto 200
120 on zdiv goto 300
130 let i=0
140 let a$='12'
.
.
.
190 rem routine to handle soflow
200 let k=j/i
210 continue
.
.
.
290 rem routine to handle zdiv
300 continue
```

Because the variable A$ can contain only one character, line 140 causes a string overflow exception to transfer to line 200. The division by zero exception at line 200, which is within the string overflow exception handling routine, causes the string overflow condition to be lost. Therefore, when the CONTINUE statement at line 300 transfers control to the CONTINUE statement at line 210, another exception occurs, because an exception must have been intercepted or active when a CONTINUE statement is executed. (The exception caused by line 210 is ERR=-10008, "RETRY or CONTINUE with no active exception.")

Again, the best approach is to make sure exceptions cannot occur within your exception handler routines, possibly by temporarily ignoring exceptions.

For information on exceptions within subprograms, see "Exceptions in Subprograms" on page 140.

Exceptions in Functions

When you are writing your own functions, you should be aware that the rules for handling exceptions in functions are not the same as they are in the main program or in a subprogram.

See *IBM BASIC Language Reference* for a discussion of exception handling within functions.

Defining and Calling Subprograms

Subprograms give you an easy way to split a large program into manageable programming units. Your main program can call one or more subprograms (each containing a logical body of code) at various points during the processing. Subprograms can call other subprograms, and also call themselves; recursive calls are allowed.

The calling program passes control to a subprogram through the CALL statement. When a CALL statement is executed, control is immediately transferred to the subprogram.

Calling programs can share data with subprograms, either through variables common to both programs or through an argument list in the CALL statement.

The CALL statement calls subprograms written in BASIC, Assembler, COBOL, FORTRAN, or PL/I. Programming in BASIC may require you to make a distinction between the parameter-passing conventions of subprograms written in these languages. BASIC supplies "interface routines" that enable you to call subprograms written in COBOL, FORTRAN, or PL/I. Other interface routines allow you to execute the BASIC SET TERM command or SYSTEM command or to perform Graphical Display Data Manager operations or SQL operations. No such interface routines are required for subprograms written in BASIC or Assembler.

Defining a Subprogram — SUB and END SUB Statements

A BASIC subprogram begins with the keyword SUB, followed by its unique name and an optional list of parameters. The subprogram is ended by the END SUB statement.

When the subprogram exits via the END SUB or a SUBEXIT statement, control is transferred to the statement immediately following the CALL to the subprogram.

In the following example, the subprogram NEWTON successively approximates X to determine where the graph of the equation of a curved line crosses the X-axis. The CALL statement at line 110 passes seven arguments to the subprogram NEWTON and the new value of X is returned. The argument list holds the initial value of X, the value by which X is increased, and the coefficients of each term in the equation.

```
100 x = 1      ! Main program
110 call newton (x,0.05,-1.0,1.0,2.0,2.0,1.0)
120 print x
130 end

240 sub newton (x, delta, a, b, c, d, e)    ! Subprogram
250 rem: the equation of a curved line
260 start: y = a + b*x + c*(x**2) + d*(x**3) + e*(x**4)
270 rem: its derivative
280 z = b + 2*c*(x) + 3*d*(x**2) + 4*e*(x**3)
290 del = y/z
300 x = x - del
310 if abs(del) > delta then goto start
320 end sub
```

The subprogram begins with the SUB statement at line 240. The END SUB statement at line 320 serves both to mark the textual end of the subprogram and to transfer the control back to the statement immediately following the CALL statement.

A subprogram must have a name that is different from any other subprogram named in the same program. A subprogram name is used in a CALL statement to transfer control to that particular subprogram.

Note: the following names are predefined and may *not* be used as the name of a BASIC subprogram: BASIC, CLINK, COBOL, FLINK, FORTRAN, GDDM, PLINK, PLI, SQL, and SYSTEM.

Invoking a Subprogram—CALL Statement

A CALL statement can include an argument list. This list must match the number and type of the parameters in the SUB statement exactly, but not necessarily the names themselves. For example:

```
100 dim sales (100)
110 integer rnum
150 call report (date$,rnum,sales())
      .
      .
      .
210 end
220 sub report (d$,n,a())
230 integer n
240 print "sales report for ";d$,"r#";n
250 for i=0 to size(a)
260   if a(i) <>0 then print a(i)
270 next i
280 end sub
```

The CALL statement at line 150 passes three arguments to the subprogram REPORT. The first two arguments are simple variables; the third argument is an array. This subprogram prints the nonzero elements of an array designated by SALES() for a date and report number passed to it by DATE$ and RNUM.

Notice that parameter D$ is a character variable, like DATE$. N is not only numeric, like RNUM, but is an integer. Because there is not automatic conversion of numeric data between a BASIC program unit and a BASIC subprogram, types

must match *exactly*. If either line 110 or 230 were missing, an exception would occur.

Notice also how arrays are indicated in argument and parameter lists: SALES() and A(). Dimension is not given, but is passed internally to the subprogram. If the third argument had been written as SALES(100), it would refer to a simple value, the 100th element of the SALES array, and the corresponding parameter of the subprogram would have had to be a simple variable rather than an array.

Transferring Control from a Subprogram—SUBEXIT Statement

The SUB END statement always causes a transfer of control back to the calling program. In addition, you can also transfer control out of a subprogram by using a SUBEXIT statement. More than one SUBEXIT statement can be present within a subprogram.

For example, in the following subprogram, control is transferred out of the subprogram at line 210 or 240, depending on the value of TRANCODE:

```
100 sub exitpro (trancode,i)
.
.
.
180 on trancode goto 190, 220
190 ! Transaction one
.
.
.
210 subexit
220 ! Transaction two
.
.
.
240 subexit
.
.
.
300 end sub
```

As soon as any SUBEXIT statement is executed, the subprogram returns control to the line following the CALL statement.

Using the STOP Statement in a Subprogram

You can use a STOP statement within a subprogram. Just as anywhere else in a program, a STOP statement halts the entire program, not just the subprogram.

Using the CHAIN Statement in Subprograms

The effect of a CHAIN statement within a subprogram is the same as that of a CHAIN statement anywhere else in a program (see "Chaining BASIC Programs" on page 81).

Program Data in Subprograms

You may use variables and arrays within a subprogram that are not declared in COMMON and are not passed as arguments. Such variables and arrays are unique to the subprogram and are initialized each time the subprogram is called.

Exceptions in Subprograms

Subprograms should contain EXIT or ON condition statements to perform exception handling for any exceptions that may occur during the operation of the subprogram. When a subprogram is called, all previously executed exception handling statements are deactivated. When the subprogram exits, all previously executed exception handling statements for the calling program unit are reactivated, including values for the ERR, CODE, and LINE intrinsic functions. For example:

```
100 com c
110 on zdiv goto 170
120 x=1.e50
130 y=0
140 call aa(x,y)
150 z=x/c
160 stop
170 print "zdiv at line ";line
180 stop
.
.
.
300 sub aa(p1,p2)
310 on zdiv goto 340
320 c=p1/p2
330 subexit
340 print "zdiv in aa at ";line
350 c=0
360 end sub
```

The ON ZDIV condition statement at line 110 is active during the execution of statements 120 and 130, is deactivated when subprogram AA is called, and reactivated when subprogram AA returns control to statement 150.

During the execution of AA, the ON ZDIV at line 310 is active. It is deactivated at line 330 or 360.

If an exception occurs in a subprogram *and* the SYSTEM action for that exception is to stop program execution *and* the action currently specified for that exception (either explicitly or implicitly) is SYSTEM, then the exception stops the whole program.

Using Subprograms Written In Other Languages

To call a subprogram written in COBOL, FORTRAN, or PL/I, use the name of the language where you would ordinarily put the subprogram name. Then start your argument list with a character expression whose value is the name of the subprogram to be called. For example:

```
200 call cobol("bal",frd,curr,final)
  . . .
350 call fortran("stat",var1,var2,var3,var4)
  . . .
470 call pli(sub$,j,k)
```

In this example, the name of the called COBOL program is BAL, and the name of the called FORTRAN program is STAT. In the CALL PLI statement, the variable name SUB$ must contain a character string that names the PL/I subprogram being called.

The arguments being passed to a subprogram written in other languages must follow the same rules as those for passing arguments to BASIC subprograms. That is, the actual arguments in the CALL statement must be the same number and type as in the parameter list in the subprogram.

You should be aware, however, that the way data items are passed to the COBOL, FORTRAN, or PL/I subprogram differs from the way data items are passed in BASIC subprograms. Data items are passed to COBOL, FORTRAN, and PL/I subprograms as shown in Figure 23.

BASIC	COBOL	FORTRAN	PL / I
INTEGER	PIC S9(9) USAGE COMP-4	INTEGER	FIXED BIN(31)
REAL SINGLE	USAGE COMP-1	REAL*4	FLOAT BIN(21)
REAL DOUBLE	USAGE COMP-2	REAL*8	FLOAT BIN(53)
DECIMAL	USAGE COMP-2	REAL*8	FLOAT BIN(53)
CHARACTER	PIC X(n) USAGE DISPLAY	CHARACTER*(*) CHARACTER*n	CHAR(n) VARYING CHAR(*) VARYING

Figure 23. Type Conversions for Interlanguage Calls

Here are some further points about interlanguage calling:

- Calling FORTRAN with character arguments requires Release 3 or Release 4 of VS FORTRAN.
- The called subprogram cannot return decimal values. It may return integer, real single, real double, and character values, although care must be taken with character string lengths.
- Decimal arrays cannot be used as arguments in interlanguage calls.
- BASIC stores arrays with the rightmost subscript varying most rapidly.

When calling FORTRAN, BASIC does not rearrange an array argument to be stored in FORTRAN order. If you want to have the array stored in FORTRAN order, you should use the TRN array function in a MAT LET statement before the call. After the call, the array may be transposed back if desired. Note: this is unnecessary for one-dimensional arrays.

In many cases, programs containing interlanguage calls are first compiled and then link-edited with the called subprograms. Some operating systems allow called subprograms to be loaded dynamically; others do not. (See your IBM BASIC System Services manual for more information.) If you load a subprogram dynamically, your operating system may require calls to interface routines to get the called programs loaded before they are actually called.

For example, if a FORTRAN subroutine named BESSEL were to be dynamically loaded and called by a BASIC program running under CMS, the BASIC program would have to first execute the statement

```
100 call flink ("bessel")
```

which causes BESSEL to be loaded. And then make the actual call:

```
200 call fortran ("bessel", argument,...)
```

Details concerning both link-editing and dynamic loading can be found in your IBM BASIC System Services manual.

Using Subprograms Written in Assembler Language

An Assembler language subprogram is assembled outside of the BASIC environment and is given control by a BASIC CALL statement. BASIC handles calls to Assembler language subprograms as if they were compiled BASIC subprograms. In general, BASIC uses normal linkage conventions.

```
Main Program

100 x = 1
110 call newton(x,0.05,-1.0,1.0,2.0,2.0,1.0)
120 print x
130 end
```

Using the same arguments that you passed to the BASIC version of subprogram NEWTON, you can expand the following Assembler language subprogram fragment:

SUBPROGRAM

```
NEWTON  CSECT
* ENTRY CODE
        STM   14,12,12(13) SAVE REGS
        BALR  12,0         GET BASE ADDRESS
        USING *,12         ESTABLISH ADDRESSABILITY
        ST    13,SAVE+4    BACK CHAIN TO CALLER'S SAVE AREA
        LA    15,SAVE
        ST    15,8(13)     FORWARD CHAIN TO NEW SAVE AREA
        LR    13,15        OWN SAVE AREA ADDR INTO STD REG
*
* COMPUTATION PORTION OF NEWTON
*
* EXIT CODE
        L     13,SAVE+4    RESTORE OLD SAVE AREA ADDR
        LM    14,12,12(13) RESTORE REGISTERS
        SR    15,15        CLEAR REGISTER 15
        BR    14           RETURN
*
SAVE    DC    18F'0'       STANDARD SAVE AREA
        END
```

This Assembler language subprogram could give results equivalent to those returned by the BASIC subprogram NEWTON discussed under "Defining a Subprogram — SUB and END SUB Statements" on page 137. The Assembler language version, just as the BASIC example, receives control from the main program at the time of the call, and executes the subprogram instructions. It then returns control to the main program at line 120, the next executable statement in the main program.

Argument List for Assembler Language Subprograms

The argument list for an Assembler language subprogram contains an argument count, argument codes, and addresses of variables and arrays used as arguments.

The calling program places the address of the argument list in general register 1.

The argument list is set up as follows:

argument count
argument codes . . .
argument addresses . . .

The argument count is the number of arguments and occupies a fullword. If there are no arguments, the argument count will be zero.

The argument codes consist of four bits per argument and describe the type of the argument and occupy as many fullwords as required. If the last fullword of argument codes is not filled, the codes are left-justified in the fullword. The upper bit of the 4-bit code for an argument designates whether the argument is an array or a scalar, with B'1' implying an array. The low-order three bits designate the type:

B'000' integer

B'001' real single

B'010' real double

B'011' decimal

B'100' character

An argument address is the address of the corresponding argument or, in the case of arguments which are constants or expressions, the address of the value of the constant or expression. The form of the argument pointed to by the argument address depends upon the type and whether the argument is an array or a scalar as designated by the corresponding argument code.

Integer arguments are fullword (aligned) signed binary integers.

Real single arguments are fullwords (aligned). Real double arguments are two fullwords (aligned).

Decimal arguments are three fullwords (aligned). Of these 12 bytes, the low-order 2 bytes contain a signed binary exponent value. The low-order 4 bits of the 10th byte contain the sign, and the remaining high-order 19 hexadecimal digits contain the left normalized, packed-decimal argument value. Note that left normalization means that the first significant (nonzero) digit of the argument value must be left-adjusted in the field.

The sign value may be any valid decimal sign code:

B'1010' positive

B'1011' negative

B'1100' positive (preferred)

B'1101' negative (preferred)

B'1110' positive

B'1111' positive

Note that a decimal zero may also be represented as 3 fullwords of zeros, including binary zeros in the sign field.

If the value of the decimal argument is changed by the Assembler language subprogram, the value must conform to the above format, the last two digits of the

19 hexadecimal digits must be zero, and the exponent value must be between -999 and +999, inclusive.

Character arguments appear as fullword aligned byte sequences of EBCDIC characters preceded by two 2-byte length fields (the argument address points to the first length field). The first 2-byte length field gives the maximum length of the character string and should not be changed by the Assembler language subprogram. The second 2-byte length field gives the current length of the character string and may range from 0 to the maximum length. The current length field may be changed by the Assembler language subprogram as long as it does not exceed the limits.

Arrays appear as follows:

element span	type
maximum number of elements	
current dimension 1	
. . .	
current dimension 7	
array elements	

Element span (2 bytes) is the number of bytes per element: that is, 4 for integer and real single, 8 for real double, 12 for decimal, 4+ the maximum element length (rounded up to the nearest multiple of four) for characters. The element span must *not* be changed by the Assembler language subprogram.

Type (2 bytes) is the same as described for argument codes above. The three bit type code occupies the rightmost three bits of the two byte field. The type must *not* be changed by the Assembler language subprogram.

Maximum number of elements is a fullword. The maximum number of elements must not be changed by the Assembler language subprogram.

Each current dimension is a fullword and gives the current number of elements for the corresponding dimension. (Note that this is not the upper bound for the dimension.) If a current dimension is zero, the following current dimensions must also be zero. An array has as many dimensions as there are current dimensions not equal to zero. The current dimensions of the array may be changed by the Assembler language subprogram as long as the maximum number of elements is not exceeded and the first current dimension is greater than zero.

The array elements will appear in the form shown for scalars above. The elements of the array are stored with the rightmost subscript varying most rapidly.

Note that character array elements are placed on fullword boundaries. The maximum length of character array elements may be determined by examining the maximum length field of the first element.

The following considerations should also be taken into account:

1. Arguments that are numeric variables or character variables (without substring qualifiers) are passed by reference, as follows:

   - Any reference to such a parameter by the Assembler language subprogram is a reference to the corresponding argument in the calling BASIC program.

   - Any assignment to such a parameter in the Assembler language subprogram is an assignment to the corresponding argument in the calling BASIC program.

2. If an argument is an array element, its subscripts are evaluated once, when the CALL statement is executed. The previous stated rules apply.

3. If an argument is a constant or an expression that involves numeric or character operators, it is evaluated once, when the CALL statement is executed. (Note that a character substring is considered such an expression.) The resulting value is assigned to a temporary location available to the called subprogram. In any reference to the corresponding parameter by the Assembler language subprogram, this temporary value is used; in any assignment to the corresponding parameter by the Assembler language subprogram, this temporary location is used.

   Note that a constant or expression with a null value may have a maximum length equal to 0.

4. The number and type of arguments used in the CALL statement must agree with the number and type of arguments expected by the Assembler language subprogram.

Calling sequence

BASIC generates a calling sequence to transfer control to the subprogram, placing the following addresses in the following registers:

- Register 1—the address of the argument list.

- Register 13—the address of the BASIC calling program unit's save area.

- Register 14—the return address in the calling program.

- Register 15—the address of the Assembler language subprogram.

The program then branches to the address in general register 15.

When the Assembler language subprogram completes, the registers should be restored to their values on entry (Assembler language subprograms may use the save area, designated by register 13 on entry, for saving the caller's registers) except for register 15 which should be returned containing zero.

Linking Assembler Language Subprograms

When coding an Assembler language subprogram, you must include the following linkage instructions:

1. Instructions to save any of the general registers that this Assembler language subprogram uses in the save area reserved by the calling BASIC program unit.

2. Instructions to restore the saved registers before returning control to the calling program unit.

3. An instruction to set register 15 to zero before returning.

In addition to these instructions, when arguments are passed, the Assembler language subprogram may obtain the value of arguments and may set the return values for arguments, using the addresses passed in the argument list pointed to by general register 1.

Note that the following names are predefined and may *not* be used as the name of an Assembler language language subprogram: BASIC, CLINK, COBOL, FLINK, FORTRAN, GDDM, PLINK, PLI, SQL, and SYSTEM.

For details on link-editing and dynamic loading, see your IBM BASIC System Services manual.

Calling System Commands from a Program

The CALL SYSTEM statement causes an operating system command to be performed. For example:

```
125 call system (a$)
```

invokes an operating system command that is the character value of A$.

```
700 call system ('filedef sample tapl calc3 fortran')
```

executes a CMS FILEDEF command. (See your IBM BASIC System Services manual for a detailed list of the commands you can use.)

Using the SET TERM Command from a BASIC Program

The SET TERM command can be executed from a BASIC program using the CALL BASIC statement. This statement anywhere in your program causes the mode to change to whatever you specify (line or scroll). For example:

```
100 call basic ("set term line")
```

when executed, would cause LINE mode to be in effect for terminal I/O.

```
300 call basic ("set term scroll")
```

when executed, would cause SCROLL mode to be in effect, if the terminal is 327x type.

The following two statements would allow you to specify SCROLL or LINE mode when the program is executed:

```
400 input "scroll or line mode?": c$
410 call basic ("set term " & c$)
```

By entering SCROLL in response to the prompt, SCROLL mode would go into effect if the terminal is a 327x type. If LINE is entered, LINE mode would go into effect. Of course, a better program would verify your response (LINE or SCROLL) before executing the CALL BASIC statement.

CALL BASIC is the only way to control SCROLL or LINE mode when you use the BASICRUN command to execute a compiled BASIC program. BASICRUN will always use SCROLL mode unless your program specifies:

```
400 CALL BASIC ("set term line")
```

When executing inside the BASIC environment, however, the SCROLL/LINE setting may be specified by the SET TERM command, profile, or installation option.

Calling Graphical Display Data Manager (GDDM) Operations

You can perform graphic operations within a BASIC program by calling the Graphical Data Display Manager (GDDM). GDDM is an IBM program product that can take words or graphics and put them in a suitable form for transmission to display terminals and printers. It is designed for use with IBM display terminals and printers that are capable of presenting graphic images, as opposed to those that can only present text characters. For a summary of the devices that can be used, see *Graphical Data Display Manager Base Programming Reference.*

Kinds of Graphics You Can Produce with GDDM

Figure 24 on page 149 and Figure 25 on page 150 are some examples of the graphics you can produce when you call GDDM from a BASIC program.

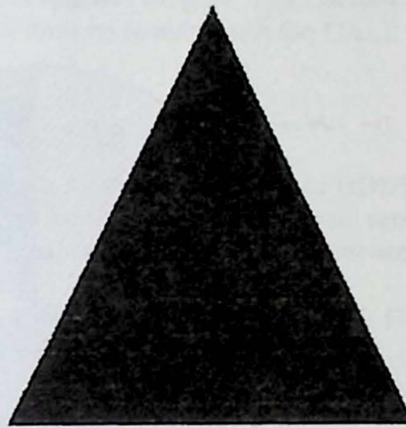
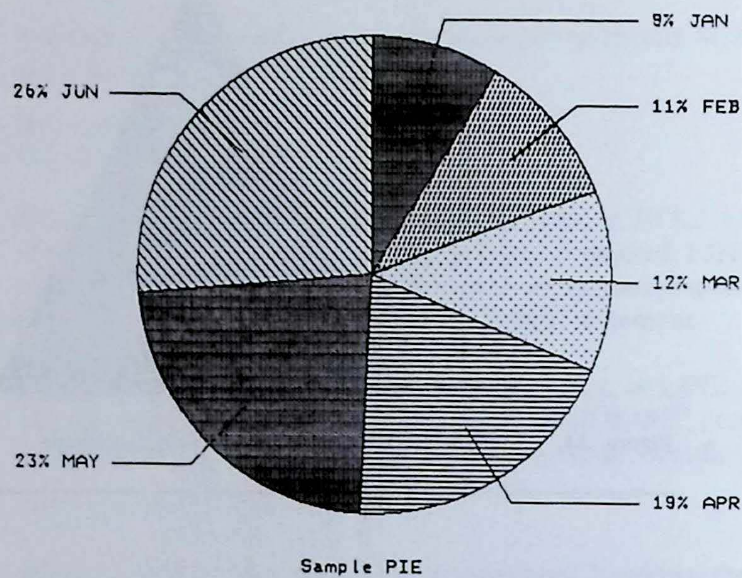


Figure 24. Example of GDDM Graphics: Triangular Shape



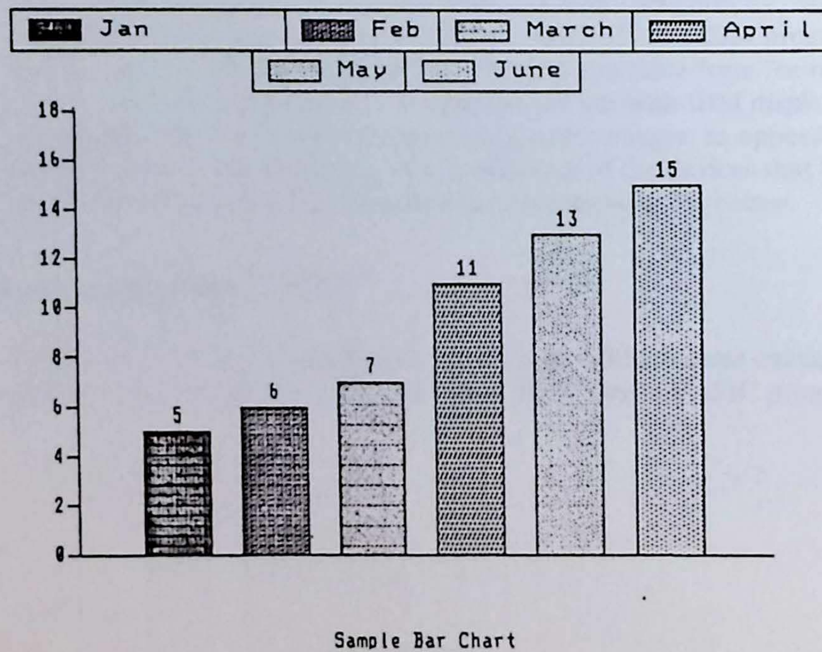


Figure 25. Examples of GDDM Graphics: Pie and Bar Charts

The programs which produced these pictures are shown in "Calling GDDM" and "The Pie and Bar Chart Program" on page 153. Before we explain those programs, however, you must be familiar with the CALL GDDM statement.

Calling GDDM

Release 3 or 4 of GDDM is required for all calls to GDDM. Before you can use GDDM you must have access to the GDDM program product. See your IBM BASIC System Services manual for information about accessing GDDM.

To call GDDM from a BASIC program, use the CALL GDDM statement. For example:

```
* 4200 CALL GDDM ("ASREAD",attype,atmod,count)
```

where *ASREAD* is the name of a GDDM function, and *attype,atmod,count* are BASIC variables containing the arguments for this function.

The first argument must be a character expression (you can also use RCP codes—see *IBM BASIC Language Reference*); the value of this expression must be the name of a GDDM or Programmable Graphics Feature (PGF) function.

The number and type of arguments in the call vary; they depend upon the operation to be performed as specified by the function name or RCP code. See the *Graphical Data Display Manager Base Programming Reference* for a list of the function names and RCP codes and for an explanation of the argument requirements for graphic operations.

Initialization and completion functions are handled automatically by the BASIC interface routine to GDDM. Therefore, the following functions cannot be used: FSINIT, SPINIT, FSEXIT, FSRNIT, and FSTERM.

The following is a simple program that shows how BASIC can be used to call GDDM. It uses GDDM functions to draw the triangle shown in Figure 24 on page 149.

```
100 !   This BASIC program uses GDDM
110 !     to draw a solid red triangle
120 CALL GDDM ("GSCLR")                       ! Clear the Graphics Field
130 CALL GDDM ("GSPS",.98,1.0)                ! Set-up Picture Size
140 CALL GDDM ("GSWIN",0.0,100.0,0.0,100.0)   ! Coordinates Range
150 CALL GDDM ("GSPAT",16)                    ! Pattern --> Solid
160 CALL GDDM ("GSCOL",2)                     ! Color --> RED
170 CALL GDDM ("GSAREA",1)                    ! Start Area for Triangle
180 CALL GDDM ("GSMOVE",30.0,30.0)            ! Starting Point
190 CALL GDDM ("GSLINE",50.0,70.0)            ! Draw Left Side
200 CALL GDDM ("GSLINE",70.0,30.0)            ! Draw Right Side
210 CALL GDDM ("GSLINE",30.0,30.0)            ! Draw Bottom
220 CALL GDDM ("GSENDA")                      ! ...and Send to Page
230 CALL GDDM ("ASREAD",attype,atmod,count)   ! Send Page to Terminal
240 CALL GDDM ("GSCLR")                       ! Clear the Graphics Field
250 CALL GDDM ("CHTERM")                      ! Terminate PG Routines
260 END                                       ! End IBM BASIC Program
```

The number of arguments and argument types are checked for validity using GDDM's Call Descriptor Table. Arguments which are valid according to this table

but are still incorrect due to some unchecked condition, may cause unrecoverable errors and the loss of the current program in the workspace.

Numeric arguments are converted to the numeric type required by GDDM for all functions (except CHART).

A character input argument must have at least as many characters as needed by GDDM (BASIC will not automatically pad with spaces). Excess characters are ignored.

For example, when using the GSLSS function to load a symbol-set:

```
1200 CALL GDDM ("GSLSS",2,"ADMDVSS ",65)
```

The third argument ("ADMDVSS ") is the name of the symbol set to be loaded. Because GDDM requires an 8 character name for the symbol-set, you must pad the argument with a space.

A character output argument must have a maximum length which is equal to or greater than the length needed by GDDM; on return, GDDM updates the argument's current length according to the output length. GDDM determines the output length from preceding arguments in the call statement.

For example, when using the ASCGET function to retrieve the screen contents of a previously defined GDDM "field":

```
1700 CALL GDDM ("ASCGET", field_id, 20, field_contents$)
```

The character variable FIELD__CONTENTS$ is passed as the argument receiving the returned data. The preceding argument (20) specifies the length of the data to be returned. Therefore, FIELD__CONTENTS$ must have a maximum length of at least 20. This could be accomplished via the following statement:

```
110 DIM field_contents$*20          ! Maximum length of 20
```

When using a GDDM function which specifies a character array parameter, use a BASIC character expression containing substrings that corresponds to the elements of the GDDM array parameter. A BASIC character array cannot be used, because its format is not compatible with GDDM.

This means that you must place the data into a character expression, where the first n characters correspond to the first element of the array, the second n characters correspond to the second element of the array, etc.

For example, when calling GDDM to establish labels on a graph, one uses the GDDM function CHKEY, (See *Graphical Data Display Manager Presentation Graphics Feature Programming Reference*).

CHKEY has 3 parameters associated with it:

- The first is an integer count of how many labels there are being passed.
- The second is an integer specifying the number of characters in each label being passed.
- The third is expected to be a character array, where each element of the array is of fixed length and corresponds to a graph label.

To illustrate, suppose we wish to call GDDM, and define 3 labels for our chart ("Orange", "Fig", and "Bananas"). We shall use a character expression named LABELS$ to contain our substrings.

Each substring of LABELS$ must be padded with spaces to fill to the length of the longest label being passed, in this case, "Bananas".

Therefore, the character expression LABELS$ should be dimensioned to a length of 21, with each of the 3 substrings padded to a length of 7 characters. The substrings should contain the following data:

```
labels$ (1:7)  = "Orange "
labels$ (8:14) = "Fig    "
labels$(15:21) = "Bananas"
```

With the character expression built this way, we can then use the call to GDDM:

```
CALL GDDM ("CHKEY", 3, 7, labels$)
```

Note: See the BASIC subprogram named "CHRTOPT" (included in the CHRTCUP program on page 240.) for an example of a general purpose utility to convert BASIC character arrays to the fixed length string format required by GDDM.

The Pie and Bar Chart Program

The following is a more advanced example of a BASIC program calling GDDM. It uses the PGF feature to draw the pie or bar chart shown in Figure 25 on page 150. The program prompts the user for data input (numeric value and character label) for up to seven items, and then displays the resulting graph in either pie or bar chart format, according to the user's choice.

```
100 Rem Sample program to illustrate use of GDDM in producing PIE or BAR Charts
110 Option base 1 : Integer :  Real single yarray
120 Dim prompt$*80,yarray(7),items$(7),colors(7),key_att(4),label_att(4),item$*80
130 Data 1,2,3,4,5,6,7
140 Mat read colors                         ! Read Colors Array
150 Data 2,0,0,150
160 Mat read key_att                        ! Read Legend Array
170 Data 3,0,0,200
180 Mat read label_att                      ! Read Labels Array
190 Print "For each item, enter data value, label (e.g., 214, PRUNES)"
200 Print "Maximum of 7 items - slash when done   (e.g.,  /)"
210 Print
220 Do                  ! Loop to obtain value,label until max of 7 or /
230    i=i+1: prompt$ = "Value, label for item " & STR$(i) & "-->"
240    Input  prompt prompt$: yarray(i), items$(i)
250    If LEN(items$(i)) > length then length = LEN(items$(i))
260 Loop while (yarray(i) > 0  and i< 7)  ! End condition for loop
270 If yarray(i)=0 then i=i-1             ! If value=0, dec index
280 Mat yarray = yarray(i)                ! Re-DIM yarray
290 For j=1 to i                          ! For-loop to blank pad labels
300   item$=item$ & RPAD$(items$(j),length)
310 Next j
320 Input 'What is the title of your chart?' : title$
330 If title$="" then title$=" "          ! If no title given, blank it out
340 ttl_len=LEN(title$)                   ! Length of title is min of 1
350 Input 'What type of chart ? (PIE or BAR)': type$
360 GRAPH_IT:                             ! Start of GDDM calls
370 Call GDDM ("CHRNIT")                  ! Reset all Defaults
380 Call GDDM ("GSCLR")                   ! Clear Graphics Field
390 Call GDDM ("CHCOL",i,colors())        ! Set up the colors
400 Call GDDM ("CHSET",'ABPIE')           ! Use the data as absolute
410 Call GDDM ("CHSET",'KBOX')            ! Draw a box around the legend
420 Call GDDM ("CHSET",'SPIDER')          ! Draw spiders on PIE sectors
430 Call GDDM ("CHSET",'VALUES')          ! Put values on top of BAR's
440 Call GDDM ("CHXLAB",1,ttl_len,title$) ! Give it a title(if used)
450 Call GDDM ("CHKEYP", "H","T","C")     ! Orient the Legend(horz,top,center)
460 Call GDDM ("CHKATT",4,key_att())      ! Give it Legend attributes
470 Call GDDM ("CHLATT",4,label_att())    ! Give it Label attributes
480 Call GDDM ("CHKEY",i,length,item$)    ! Set up the Legend
490 If UPRC$(type$(1:1))='B' then         ! What kind of CHART???
500   type$='BAR': other_type$='PIE'
510   call GDDM ("CHBAR",i,1,yarray())    ! Plot the BAR chart
520 Else
530   type$='PIE': other_type$='BAR'
540   call GDDM ("CHPIE",1,i,yarray())    ! Plot the PIE chart
550 End if
560 Call GDDM ("ASREAD",attype,atmod,count)! Put it on screen
570 Call GDDM ("GSCLR")                   ! Clear Graphics Field
580 Call GDDM ("CHTERM")                  ! Terminate PG Routines
590 type$=other_type$
600 prompt$ = "Do you wish to replot the data as a " & type$ & " graph?(Y|N)"
610 Input prompt prompt$: y$              ! Prompt for replotting...
620 If UPRC$(y$(1:1))='Y' then goto GRAPH_IT ! If answer yes, doit again
630 End
```

Here is an explanation of what the program is doing.

- Line 110 - the origin for array subscripts is set to 1, the default numeric data type is integer, and one variable, YARRAY, is real single. (These data types are directly compatible with GDDM. The program would give the same results if all variables were of decimal type; however, BASIC would then have to perform data conversions.)

- Line 120 - Various character variables and arrays are dimensioned. The variable containing the prompt message (PROMPT$ - used in statement 610) is declared to have a maximum length of 80 characters. Since the program is designed to graph a maximum of 7 values, YARRAY, which contains these numeric values, and ITEMS$ and COLORS, which contain the character labels and color codes associated with the values, are all dimensioned as 7 element arrays. KEY__ATT, and LABEL__ATT are 4 element arrays containing format information required by GDDM.

- Line 130– The GDDM color codes are as follows: 1=Blue, 2=Red, 3=Pink, 4=Green, 5=Turquoise, 6=Yellow, 7=White.

- Lines 150-180 establish the formatting arrays used by GDDM for legend, and label attributes. For an explanation of the CHKATT and CHLATT function parameters, see *Graphical Data Display Manager Presentation Graphics Feature Programming Reference.*

- Lines 190-260 prompt the user for input, and builds the YARRAY and ITEMS$ arrays. In order to exit the input loop, the user can either complete the data entry for all 7 items, enter a zero value, or enter the slash character ("/"). (The slash technique is a feature of BASIC's INPUT statement and does not change the value of the variables in the input list. Hence YARRAY(i) and ITEMS$(i) will have values of zero and "" if the user exits with a slash.) If the loop is exited with fewer than 7 values, then YARRAY(i) will be zero, and the loop counter i must be adjusted (line 270). Line 280 then re-dimensions YARRAY to the actual number of items.

- Lines 290-310 process the ITEMS$ array, and iteratively concatenates each element to the scalar character variable ITEM$. The RPAD$ intrinsic function is used to "pad" blanks to the right of each element. (This is necessary because GDDM can't process BASIC's variable length character arrays—character arrays must be passed as scalars with each array element in a fixed position.)

- Lines 370-480 are the GDDM calls that will produce the graphs. The comments identify the various GDDM services that are used.

- Lines 490-550 determine whether the user specified a pie or a bar chart, and then calls GDDM to plot the appropriate graph.

- Line 560 causes the graph to be displayed on the screen.

- Lines 570-580 terminates the GDDM processing. These lines are executed after the user has viewed the graph and has pressed the enter key.

- Lines 600-620 prompt the user as to whether the data should be re-graphed as a different type of chart. If so, the program branches back to the GRAPH__IT label at line 360.

Calling the Interactive Chart Utility (ICU)

Associated with GDDM is the Presentation Graphics Feature (PGF). The PGF provides a special interface for producing business charts and includes the Interactive Chart Utility (ICU), which is a general purpose program for the production of such charts at the terminal.

You can call the ICU from a BASIC program with a CALL GDDM statement specifying the CHART function. For example:

```
200 CALL GDDM ("CHART",A(),B(),C$,D(),X(),Y(),K$,L$,H$)
```

where
- A is an integer array
- B is a real single array
- C$ is a character expression
- D is an integer array
- X is a real single array
- Y is a real single array
- K$,L$,H$ are character expressions

The arguments A,B,C$ are used to construct the CHART_CONTROL parameter for calling the CHART function. The D argument is used to describe "free data", and the the X and Y arguments contain the x and y coordinate values for the data to be graphed. The K$, L$, and H$ arguments are used to specify the keys, labels, and heading information, respectively. For more details on specifying the arguments, see *IBM BASIC Language Reference.*

See Appendix B, "Sample Program for Calling the GDDM Interactive Chart Utility" on page 237 for an example of using the ICU.

Using SQL to Access Relational Data Bases

BASIC provides access to data maintained by IBM's relational data base systems: IBM DATABASE 2 (DB2) in the MVS environment, and Structured Query Language/Data System (SQL/DS) in VM. Access to the data is specified using the Structured Query Language (SQL).

SQL is a high-level language that uses the relational data model. A *relation* in the relational data model can be thought of as a simple two-dimensional table (similar to a BASIC two-dimensional array—except that the columns may be of different data types). The columns of the table have names, and the rows contain data. In DB2, the intersection of a row and column is called a value, and in SQL/DS it is called a field. In BASIC terms, each column may be defined for either numeric or character data, and each row in the column may contain data or it may be empty.

These tables are accessed through SQL statements. The structure, content, and integrity of the tables are usually managed by a data base administrator, who grants authorization for certain kinds of access. This discussion of SQL assumes you are authorized for data base access.

As its full name implies, SQL is a query language. However, it is also a data manipulation, data definition, and authorization language.

The users of SQL and the type of SQL statements they typically use are:

- *Application programmers* use query statements to retrieve data from the data base. They also use data manipulation statements for inserting, deleting, and updating rows in a table. A single SQL statement can process multiple rows of the table.

- *Data base administrators* use data definition statements to create new tables or change the structure of existing tables.

- *System administrators* use authorization statements to grant and revoke authorization to use tables (and other privileges) to all users.

For more information on the responsibilities, privileges, and authority of SQL users, see *IBM DATABASE 2 General Information* for DB2, *SQL/Data System Concepts and Facilities* for SQL/DS.

This discussion of SQL operations is confined mainly to retrieving and manipulating data. However, both DB2 and SQL/DS permit many more data base operations. You should refer to the appropriate publications listed in " Related Documentation" on page v for more information.

As a BASIC user, why would you want to use relational data bases rather than conventional file systems? For one thing, if your organization already has relational data bases, you can now write simple BASIC programs to access and manipulate that data, utilizing the power and flexibility of relational data base management systems. Also, SQL is easy to use; it enables you to work with data without having to learn a complex storage format. You don't need to be dependent on the internal details of the data base's physical organization. Thus, as a SQL user, you can retrieve and manipulate data without knowing how it is represented in storage. Using SQL statements, the data base management system (DB2 or SQL/DS) finds its own way to the data. You access the data you want by specifying *what* data you want, rather than *how* to access it.

Structure of a Relational Table

Each table is identified by a table-name; each column within the table is given a column-name. Figure 26 on page 158 shows a sample SQL table called SUPPLIERS. This table has 4 columns (SUPPLNO, NAME, ADDRESS, and REGION) and 11 rows.

Table: SUPPLIERS

SUPPLNO	NAME	ADDRESS	REGION
51	DEFECTO PARTS	16 BUM ST., BROKEN HAND, WY	WESTERN
57	ATLANTIS CO.	8 OCEAN AVE., WASHINGTON, DC	EASTERN
71	TITANIC TOOLS	45 SINKING ST., ICEBERG, WISC	MIDWEST
108	EAGLE HARDWARE	153 TRANQUILITY PLACE, APOLLO, MN	MIDWEST
198	XYZ DISTILLERY	90 PROOF BLVD, WHISKEYTOWN, CA	WESTERN
202	SKY PARTS	128 ORBIT ST., SYDNEY, AUSTRALIA	FOREIGN
88	VAN GOGH PAINTS	1 EAR ST, NYC, NY	EASTERN
222	KNIGHT LTD	456 ARTHUR COURT, CAMELOT, ENGLAND	FOREIGN
233	COMPUTER PARTS	512 BYTE BLOCK, PALO ALTO, CA	WESTERN
52	VESUVIUS, INC.	79 LAVA ST., POMPEII, NY	EASTERN
145	GOLD MEDAL CO	39 OLYMPIC BLVD., MARATHON, OHIO	EASTERN

Figure 26. Sample SQL Table: SUPPLIERS. Note that an actual SQL table is likely to contain many more rows.

If such a table exists, SQL SELECT statements can be written to retrieve data from it by specifying what data is wanted; the specifications include a column list and optional *search-conditions* contained in a WHERE clause. For example, the following SQL statement selects the name and address of all suppliers located in the eastern region:

```
SELECT NAME,ADDRESS
FROM SUPPLIERS
WHERE REGION = 'EASTERN'
```

SQL statements consist of clauses. The above SQL statement consists of three clauses. The SELECT clause tells what columns to use, by name. The FROM clause indicates the name of the table to use. The WHERE clause tells what rows to use by giving a condition that is satisfied (all rows where the REGION column is EASTERN).

The resulting table is shown in Figure 27. Note that only 2 columns were selected (NAME and ADDRESS) and that 4 rows satisfied the search condition.

NAME	ADDRESS
ATLANTIS CO.	8 OCEAN AVE., WASHINGTON, DC
VAN GOGH PAINTS	1 EAR ST, NYC, NY
VESUVIUS, INC.	79 LAVA ST., POMPEII, NY
GOLD MEDAL CO	39 OLYMPIC BLVD., MARATHON, OHIO

Figure 27. Result of SQL SELECT Statement

To keep things simple, we'll only talk about tables in this discussion of SQL. Actually, in DB2 and SQL/DS there are many other data objects besides tables (things like views, indexes, data spaces, groups, and so forth). If you want more

information about these, see the DB2 or SQL/DS publications listed in " Related Documentation" on page v.

How to Use the CALL SQL Statement

BASIC enables you to communicate with relational data bases by means of the CALL SQL statement. You use the CALL SQL statement to pass a SQL statement to the data base system from within your BASIC program. For a detailed description of the syntax of the CALL SQL statement, see *IBM BASIC Language Reference.*

The data base system executes the SQL statement, performing the requested operation on the data base. If your SQL request requires data to be retrieved from the data base, the appropriate data will be returned to your BASIC program. Figure 28 on page 160 illustrates how BASIC and the data base system handle your CALL SQL statement and Figure 29 on page 160 illustrates the different parameters of a CALL SQL that retrieves data from a table named Q.STAFF. (The Q.STAFF table is shown in Appendix D, "The SQL Sample Tables" on page 249.) The SQL query is:

```
SELECT NAME, SALARY, DEPT
FROM   Q.STAFF
WHERE  JOB = 'CLERK'
```

Note that even though each clause is written on a separate line (a common practice when writing SQL statements), it's all one statement. So if you stored your SQL statement in a BASIC variable, QUERY$:

```
300 QUERY$ = "SELECT NAME,SALARY,DEPT FROM Q.STAFF "
310 QUERY$ = QUERY$ & "WHERE JOB = 'CLERK'"
```

your CALL SQL statement would look like this:

```
400 CALL SQL (QUERY$,SQLRC(),MSG$,EMP$(),SAL(),DEPT())
```

Note: The text of the SQL statement is shown in uppercase. Strictly speaking, this is not necessary because the data base system automatically treats all SQL statements as if they were entered in uppercase, except for the contents of quoted character constants, which are evaluated as entered. To avoid confusion, however, it's good programming practice to enter all SQL statements in uppercase.

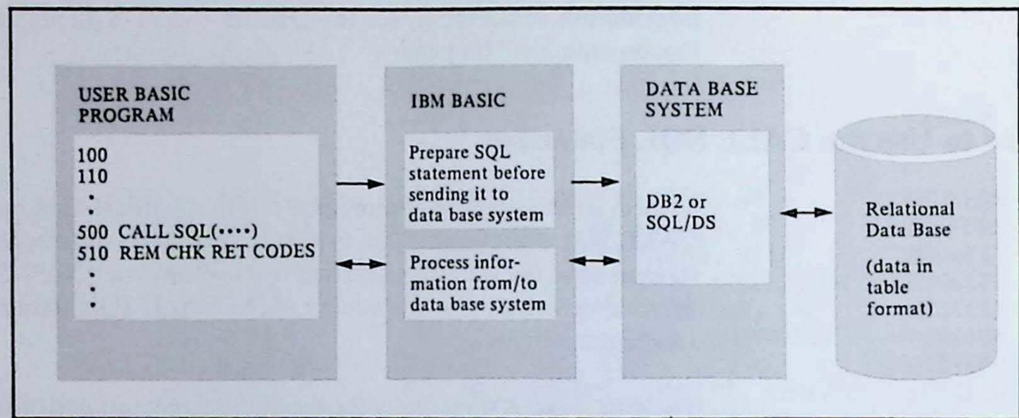


Figure 28. Using SQL to Access Relational Data Bases

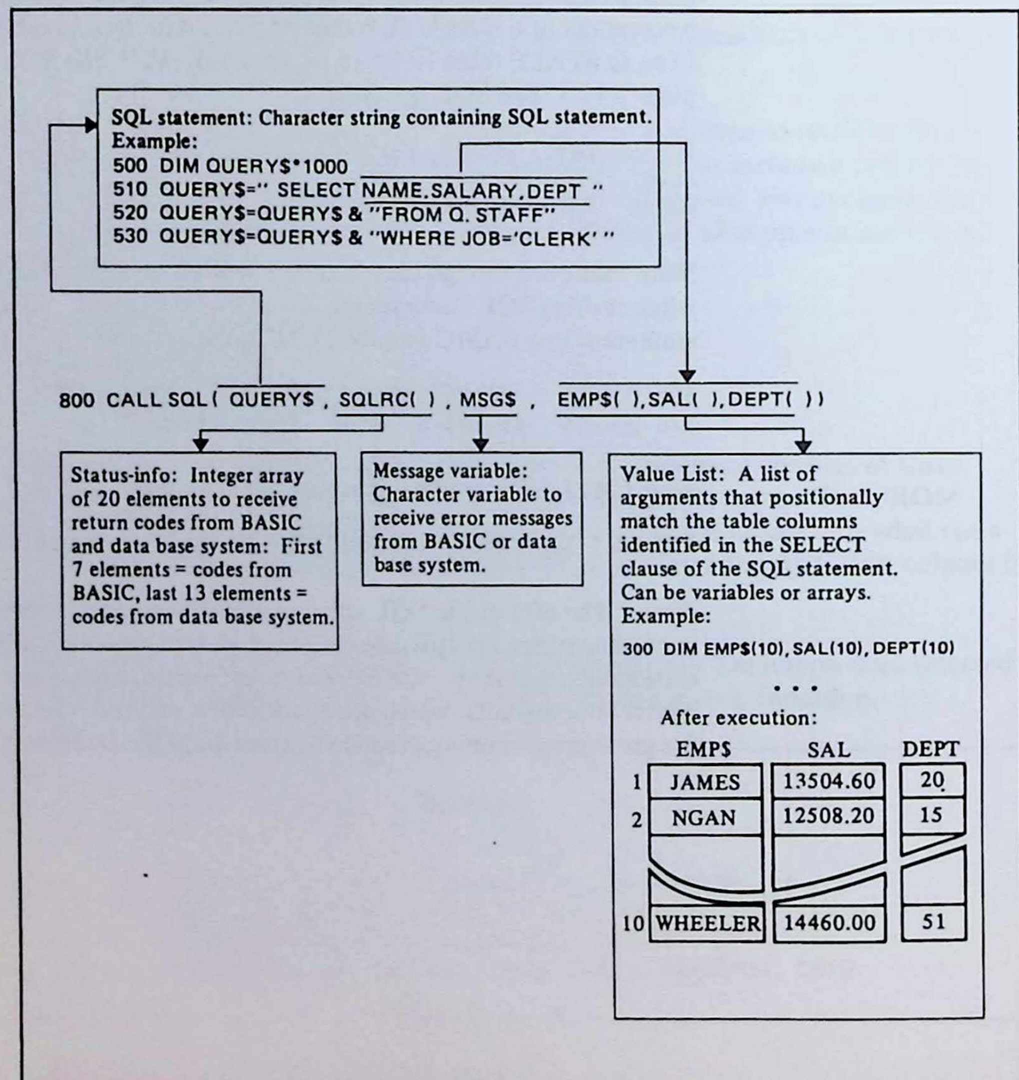


Figure 29. A CALL SQL Statement to Retrieve Data from a Table

Retrieving Multiple Rows Versus Single Rows

Information can be retrieved from a table many rows at a time or a single row at a time. The CALL SQL statement shown in "How to Use the CALL SQL Statement" on page 159 and in Figure 29 on page 160 is an example of getting back multiple rows with one execution of a CALL SQL statement. Notice that the three arguments of the value list, EMP$(), SAL(), and DEPT(), are specified as *one-dimensional arrays.* The DIM statement for these arrays looks like this:

```
800 DIM EMP$(10), SAL(10), DEPT(10)
```

The number of rows to be retrieved is indicated by the dimension of the arrays. In the above example, the number of rows to be retrieved at a time is 10. (OPTION BASE 1 is assumed.) It's mandatory that, if any argument in such a value list is an array, all arguments be arrays and all have the same number of elements. It would be an error if you said:

```
800 DIM EMP$(10), SAL(15), DEPT(17)
```

because BASIC would not be able to tell how many rows you really want.

BASIC will "fill up" the value list arrays with as much data as will fit. Thus in our example, after the CALL SQL statement is executed, you would find the following

EMP$	SAL	DEPT
JAMES	13504.60	20
NGAN	12508.20	15
NAUGHTON	12954.75	38
YAMAGUCHI	10505.90	42
KERMISCH	12258.50	15
ABRAHAMS	12009.75	38
SNEIDER	14252.75	20
SCOUTTEN	11508.60	42
LUNDQUIST	13369.80	51
WHEELER	14460.00	51

after the first execution of the CALL SQL statement. Since only 12 rows satisfy the search condition (only 12 clerks are in the table), the next time the statement is executed, you'd find:

EMP$	SAL	DEPT
BURKE	10988.00	66
GAFNEY	13030.50	84

The remaining eight elements of EMP$ would contain empty strings and the remaining eight elements of SAL and DEPT would contain zeros.

You can retrieve *fewer* rows than that indicated by the dimension of the arrays of the value-list. Thus in the example above, if you wanted to retrieve seven rows at a time, you'd place a value of seven in the third element of the SQLRC array before executing the CALL SQL statement. This is explained in more detail in "Checking Return Codes" on page 162.

Retrieving multiple rows at a time can result in better performance. It can also be a convenient way to write programs if the number of rows to be retrieved

corresponds to a natural unit or division of work (for example, a page of a printed report or a screen's worth of data). See page 169 for an example of CALL SQL retrieving multiple rows at a time.

If, on the other hand, you want each execution of your CALL SQL statement to retrieve a single row, you'd define the variables in the value list as *scalars*. In other words, to retrieve a single row, your BASIC statements would be:

```
800 DIM EMP$*30
810 DECIMAL SAL
820 INTEGER DEPT
```

and your CALL would be:

```
400 CALL SQL (QUERY$, SQLRC(), MSG$, EMP$, SAL, DEPT)
```

See page 171 for an example of a CALL SQL statement retrieving single rows.

Deactivating a SQL Request

Each execution of CALL SQL with a SELECT request will retrieve a row (or a group of rows) until an end-of-table condition is encountered. Before the end-of-table is reached, the SQL statement is considered "active". You can have up to 20 active requests in your program. After the end-of-table condition is encountered, that SQL statement is no longer active and execution of the same CALL SQL request will be treated as a new query.

You can, however, explicitly deactivate an active SQL retrieval request by one of two methods:

- By issuing a COMMIT or ROLLBACK. Use this method with care, as it will deactivate *all* currently active SQL requests in your program.

- By placing a negative integer value in the third element of the status-info (SQLRC) array and executing the CALL SQL with the last SELECT statement unchanged. This tells the data base system that you don't want any more rows retrieved from the table. This technique is illustrated in "Example: Retrieval with User-Specified WHERE Clause" on page 243.

Checking Return Codes

Upon return from the CALL SQL you would normally want to test certain return codes to determine whether or not your call was successfully executed and whether or not certain error or exception conditions were encountered.

BASIC expects you to define a 20-element integer array which it will use to store return codes. This is the second argument (status-info) of a CALL SQL statement and is named SQLRC in the examples. In order to simplify references to the different elements of the return code array, let's assume that the BASE 1 option is in effect.

After execution, the first seven elements of SQLRC will contain codes from BASIC. The remaining 13 elements contain codes from the data base system.

The return codes you probably will be using quite frequently are those stored in the first 3 elements of SQLRC. A zero value in SQLRC(1) indicates that your SQL call was successfully executed. If your request involves retrieving data from a table, an end-of-table condition is indicated by: SQLRC(1) = 8 and SQLRC(2) = 100.

When retrieving multiple rows at a time, an end-of-table condition will be indicated when the last row is returned. When this happens it means that each array is probably only partially filled. The empty elements will be filled with zeros if the data type is numeric, or spaces if data type is character. In our example, we've defined our arrays to receive 10 rows at a time. Since 12 rows satisfy the search condition, an end-of-table return code would only be returned after the second execution of the CALL SQL statement.

When retrieving data a single row at a time, the end-of-table condition is indicated when no more rows can be retrieved. In our example, if the CALL SQL statement was within a loop, the next-to-the-last CALL SQL executed would retrieve the last row (GAFNEY 13030.50 84). And on the last execution of the CALL SQL, you'd get an end-of-table condition, but no data.

To determine the count of the number of rows retrieved by a multiple row request, you look in SQLRC(3) after the CALL SQL is executed. In our example, a value of 10 would be returned in SQLRC(3) after the first execution, and 2 after the second execution. (For single row requests, SQLRC(3) always contains a value of 1 after successful retrieval, and 0 after end-of-data.)

You can also use SQLRC(3) to pass information to BASIC prior to execution of a multiple row CALL SQL retrieval request (SELECT statement). The value set in this field may be zero, positive, or negative:

- *Zero*: means that you want the number of rows retrieved to be determined by the dimension of the value-list arrays.
- *Positive*: means that you want this value to be used for your table segment size. This value must be equal to or less than the dimension of the value-list arrays. If it's greater, you'll get an error code in SQLRC(2).
- *Negative*: tells the data base system you don't want any more rows retrieved for this active SQL statement. You wish to deactivate the request.

SQLRC(3) has yet another use: for multiple row INSERT operations (using a VALUES clause), you may set this field to how many rows of data you want to insert into a table in the data base. The value you set in this field may be zero or positive (a negative value will give you an error code in SQLRC(2)):

- *Zero*: means the number of rows to be inserted will be determined by the dimension of the value-list arrays.
- *Positive*: means that you want the first *n* elements of the value-list arrays to be inserted as rows into the table (where *n* is the value you set in SQLRC(3)). This value must be equal to or less than the dimension of the value-list arrays. If it's greater, you'll get an error code in SQLRC(2).

See pages 181 and 183 for examples of inserting rows into a table.

For a complete description of the 20 elements in the SQL *status-info* array, look under "CALL SQL Statement" in the *IBM BASIC Language Reference*. Also, you can get more information about SQL return codes from *IBM DATABASE 2 Messages and Codes* or *SQL/Data System Messages and Codes*.

Messages

In Figure 29 on page 160 notice that the third argument of the CALL SQL statement, the *message variable (MSG$)* identifies an area of your program where BASIC stores messages after the CALL SQL statement is executed. It is a character variable. And typically it will contain a single message. The length of the area you reserve for it should be at least 256, since this is the longest possible message that can be returned by the data base system. BASIC will truncate the message if you don't define an area large enough. The examples in "CALL SQL Examples" starting on page 168 show how messages can be printed.

The Value-List: Receiving Data from the Data Base

So far we've discussed the first three arguments of the CALL SQL statement (SQL-stmt, status-info, and message-var). Any arguments following the first three arguments constitute the *value-list.* The value-list is required if your CALL SQL statement requires areas in your program in which data retrieved from the data base will be stored. In other words, you need it whenever you use a SELECT statement. (It is also required for INSERT, UPDATE, or DELETE statements which pass data to the data base; this is discussed in the next section.)

Figure 29 on page 160 illustrates how the value-list is used for retrieving data and Figure 30 on page 165 shows the relationship between your CALL SQL statement, your SQL statement, and the relational table. Also, the retrieval examples on pages 169 and 171 illustrate how data is retrieved from a table.

Notice in the figures and in the examples that the arguments of the value-list positionally match the column names specified in the SELECT statement. Thus in the figure EMP$() matches NAME, which means that employee names retrieved from the NAME column of the Q.STAFF table will be stored in the EMP$ array.

In the SQL SELECT statement the column names may be in any order; they don't have to be listed in the order in which they are arranged in the table in the data base. Be careful, therefore, when coding a SQL statement like:

```
SELECT * FROM Q.STAFF
```

The * means "return all columns", and the data base system will attempt to return the columns *in the order in which they appear in the table*. This means you need to know the order of the columns because the variables you specify in your value-list must be arranged in that order.

The Value-List: Passing Data to the Data Base System

You can write CALL SQL statements to insert, update, or delete rows in a table without using a value-list, but it may become cumbersome when many rows of data are involved. For example, the following SQL statement will insert a new row for an employee named Smith:

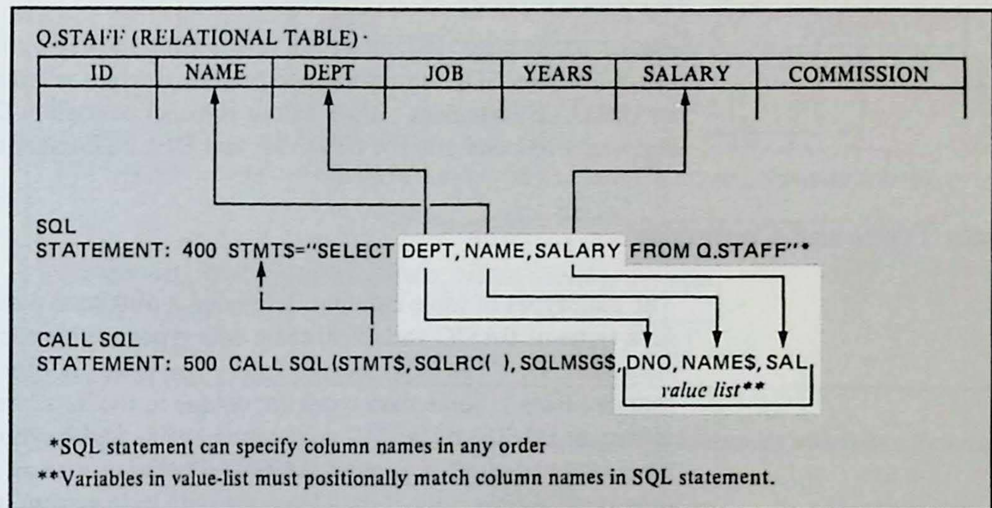


Figure 30. Relationalship Between CALL SQL Value-list, SQL Statement, and Relational Table

```
INSERT INTO Q.STAFF
VALUES (360,'SMITH',38,SALES,1,17233.00,500)
```

Notice, however, that the values to be inserted are "hard-coded". You could only insert one row at a time and the statement would need to be changed for new values for inserting subsequent rows. Also, if the values to be inserted happen to be stored in variables in your program, getting them into the SQL statement would be a clumsy operation. A much simpler and more efficient technique is to use *index values* in the SQL statement, thus:

```
INSERT INTO Q.STAFF
VALUES (:1,:2,:3,:4,:5,:6,:7)
```

The CALL SQL statement would look like:

```
CALL SQL (STMT$,SQLRC(),MSG$,IDNO,EMP$,DEP,JB$,YR,SAL,COM)
```

where the variables IDNO, EMP$, DEP, etc. contain data for the row. The index values act as "place holders" in the SQL statement. By the time the data base system receives the SQL statement index values in the SQL statement will have been replaced by the appropriate values from the variables of the value-list. Thus the contents of IDNO (360) will replace :1, the contents of EMP$ (SMITH) will replace :2, and so on. The actual SQL statement sent to the data base system would be equivalent to:

```
INSERT INTO Q.STAFF
VALUES (360, 'SMITH',38,'SALES', 1,17233.00,500)
```

The above example inserts a single row for each execution of the CALL SQL statement. The variables in the value-list are scalars. In order to insert multiple rows at a time, you would define the variables in the value-list as arrays. You tell SQL how many rows to insert by placing the appropriate integer value in the third element of the SQL-codes array. Thus, to insert 10 rows at a time, you'd put either 10 in SQLRC(3) or put 0 in SQLRC(3) and dimension the value-list arrays to 10.

Prior to executing the CALL, you would, of course store in the variables the appropriate table row data you intend to insert. See pages 181 and 183 for examples of inserting rows into a table using index values.

Figure 31 on page 167 shows the relationship between your CALL SQL value-list, your SQL statement, and relational tables when index values are used in an UPDATE statement. Index values can also be used in DELETE statements. See pages 184 and 186 for UPDATE and DELETE examples.

Data Types and Conversions

The data types of table columns defined in a relational data base are similar to the data types of BASIC variables; some data types are identical, some are not. (See Appendix D, "The SQL Sample Tables" on page 249 for the data types of the sample tables.) Some data types are unique to the data base system while some are unique to BASIC. If BASIC encounters a data type in your program that the data base system doesn't recognize, BASIC will change it to an appropriate data base data type. Conversely, if data from the data base system is in a form not recognized by BASIC, it will change it to a BASIC data type. For more information about data types and data conversion, see "CALL SQL Statement" in *IBM BASIC Language Reference*.

Handling NULL Values

In the SQL language a null value in a table is interpreted to mean "unknown" or "missing"; it does not mean, nor should it be confused with zero, or blank. SQL has a special way of recognizing null values; it uses the NULL keyword to check for null values in a table. In BASIC you would use this keyword in the WHERE clause of your SQL statement. However, in BASIC, variables are used to receive data from or transmit data to the data base system; and you can't use the keyword NULL in a variable. So what you need is a previously agreed to value which BASIC will use to represent a null value. This previously agreed to value may be established by you by means of a special variation of the CALL SQL statement. For example, if you wanted the value of zero to represent nulls, you'd code the following:

```
CALL SQL ("SETOPT NULL 0", SQLRC(), SQLMSG$)
```

The number you choose must be an integer between -2147483648 and +2147483647. If you don't establish your own value for nulls, BASIC uses a default value of -2147483648 (the smallest integer in BASIC).

The above discussion applies only for *numeric* data types. For non-numeric data types, the default value representing null is always the empty string (`''`). It cannot be changed.

The example on page 174 illustrates retrieving rows which contain null data.

Note: BASIC does not pass control to the data base system when it is executing this CALL SQL statement.

It's possible that the value you establish for nulls may conflict with an actual value in the table. For example, in the CALL SQL example above, we've told BASIC

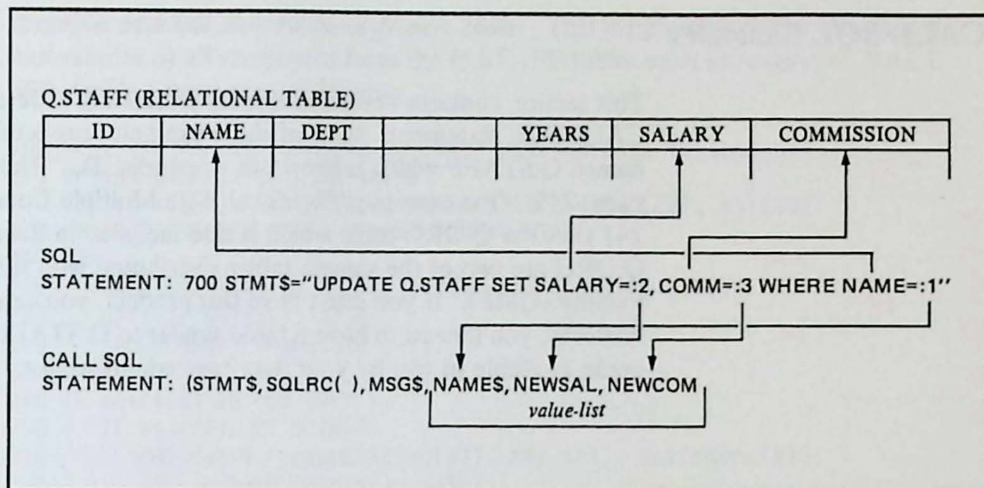


Figure 31. Relationship Between CALL SQL Value-list, SQL Statement with Index Values, and a Relational Table

that null values retrieved from the data base should be treated as zeros. Now if a column contains both nulls and zeros, and if a zero value is retrieved, how can you tell if it's a null or really a zero? BASIC can detect such a conflict. The information returned in the fourth and fifth elements of the status-info array (SQLRC(4) and SQLRC(5))will help you locate and resolve such conflicts. The example on page 175 shows how you can distinguish between null values and actual data.

Some Things to Keep in Mind...

SQL is a high-level language with great power and flexibility. There is a wide range of statements you can use and you can write some statements in many different ways to produce the the same results. The SELECT statement, for example, can often be written in several different ways to retrieve the same data. However, the performance and the resources used by different forms of the SELECT statement vary considerably. As an application programmer, you should be aware of the implications of the SQL statements you use, and try not to tie up data base resources needlessly. For a discussion of this subject, DB2 users should see "Appendix E. Performance Guidelines for SQL Statements" in *IBM DATABASE 2 Application Programming Guide for TSO Users* and SQL/DS users should consult "Chapter 12. Performance Management" in *SQL/DS Planning and Administration - VM/SP.*

When your main program ends, either normally or abnormally, BASIC will do an automatic ROLLBACK WORK for any uncommitted work left at the end of the main program. Therefore, if you've done a lot of INSERTs, DELETEs, UPDATEs, or CREATEs, all your work will be lost if you don't make the changes to the data base permanent by issuing a COMMIT statement.

In a long running program which includes statements like INSERT, DELETE, UPDATE, or CREATE, if you wait till the end of the program to issue COMMIT, you'll probably tie up resources longer than necessary. It's good practice to issue COMMIT as soon as it is appropriate to do so.

CALL SQL Examples

This section contains several examples illustrating different ways of using the CALL SQL statement. Most of these examples use a relational data base table named Q.STAFF which is shown in Appendix D, "The SQL Sample Tables" on page 249. One example, "Retrieval With Multiple Concurrent Selection," on page 241 uses the Q.ORG table which is also included in the appendix. Q.STAFF and Q.ORG are two of the sample tables distributed with the Query Management Facility (QMF). If you don't have this product, you can still run these examples; however, you'll need to have a table similar to Q.STAFF and Q.ORG created and made available to you by your data base administrator.

Example: Multiple Row Retrieval from a Table: This program retrieves the names and salaries of all employees from the Q.STAFF table; each execution of CALL SQL yields a maximum of 20 rows.

```
100 REM: Program to retrieve multiple rows of data in segments
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256, name$(20), sal(20)
130 stmt$ = "SELECT NAME, SALARY FROM Q.STAFF"
140 DO
150    CALL SQL (stmt$, sqlrc(), sqlmsg$, name$(), sal())
160    EXIT IF sqlrc(1) GT 4
170    CALL report (name$(), sal())
180 LOOP
190 IF sqlrc(2) EQ 100 THEN
200    IF sqlrc(3) GT 0 THEN
210       MAT name$ = name$(sqlrc(3)): MAT sal = sal(sqlrc(3))
220       CALL report (name$(), sal())
230    END IF
240    PRINT "*** End of Table ***"
250 ELSE
260    PRINT "*** Data Base or SQL Statement Error:"
270    PRINT "       BASIC/SQL Return Code:  ";sqlrc(2)
280    PRINT "       Data Base Return Code:  ";sqlrc(8)
290    PRINT "       Error Message Text: ";sqlmsg$
300 END IF
310 END
.
.
.
400 SUB report (a$(), x())
410 REM:  Subprogram to format & print salary report
420 OPTION BASE 1
430 PRINT NEWPAGE
440 PRINT "NAME           SALARY"
450 PRINT
460 FOR i = 1 to SIZE(a$)
470    PRINT USING "<######## ######.##": a$(i), x(i)
480 NEXT i
500 END SUB
```

Notes:

- Line 110: Because we will be dealing with an array for the status-info (SQLRC) argument, it's a good idea to explicitly specify the origin for array subscripts with the OPTION statement. Also, note that SQLRC *must* be INTEGER data type.

- Line 120 explicitly dimensions the NAME$ and SAL arrays to have 20 elements each. This size was chosen to be compatible with the number of data rows that can be displayed on a 24-line, full-screen terminal.

  In keeping with the discussion in "Checking Return Codes" on page 162, the status-info array is coded as a single dimensional matrix with 20 elements. Also, since the length of the SQL-stmt character argument will usually be greater than the BASIC default maximum for character strings, it should be explicitly set to its maximum likely value. In this case, we named this variable STMT$, and arbitrarily picked a maximum length of 1000. Finally, the message-text argument (named SQLMSG$ in the examples) should have a maximum length of 256, since this is the largest possible message that will be returned.

We did not explicitly specify a length for the NAME$ character variable; this is because the NAME column's maximum length of 9 is smaller than the IBM supplied installation default of 18. However, in general, it is good practice to dimension character variables to match their data base counterparts. In this way you will avoid any possibility of data truncation.

- Lines 140-180: The program loops, repeatedly issuing the CALL SQL statement requesting the data base system to retrieve a group of 20 rows and then printing the 20 rows. Notice that the value list of statement 150 (NAME$, SAL) positionally matches the order of the selected columns (NAME, SALARY). The loop is exited when a nonzero value is found in SQLRC(1).

  Note: The first element of the return code array reflects indications of errors, warnings, or success detected by BASIC or the data base system. Therefore, we only need check SQLRC(1) to determine whether or not to proceed. Since we are concerned only with error or exception conditions, and wish to ignore warnings (return code = 4), we exit when the return code is greater than 4.

- Lines 190-240: If the SQLRC(2) value is 100 then all of the rows have been retrieved, and the last segment retrieved may be partial (that is, contain fewer rows than the segment size). Since the REPORT subprogram simply prints everything that it is passed, we must redimension the NAME$ and SAL arrays to match the number of rows actually retrieved. This value is specified in SQLRC(3).

- Lines 400-490: This subprogram will print as many lines as will fit on a screen ("a screen's worth" of data).

- If you were to execute this program a second time, it's possible that the sequence of the rows retrieved would not be the same as the first time. The data base system doesn't guarantee the order of the rows for SELECT operations unless your SQL statement has an ORDER BY clause. Regardless of the sequence, however, you should have the same row data. Use an ORDER BY clause if the sequence of rows retrieved is important to you.

Example: Single Row Retrieval from a Table: This program is similar to the previous program except that it only retrieves the names and salaries of employees in the Q.STAFF table who have annual salaries greater than $20,000. Retrieval is a row at a time. Although both of these columns can potentially contain null entries (that is, data base table definition does not specify NOT NULL attribute), we aren't concerned with this because we know that both of these table columns happen to be fully populated with data. Later examples will show how to deal with nulls (see pages 174 and 175).

```
100 REM: Program to retrieve data a row at a time
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256
130 stmt$ = "SELECT NAME, SALARY FROM Q.STAFF WHERE SALARY > 20000"
140 PRINT "NAME         SALARY"
150 PRINT
160 DO
170    CALL SQL (stmt$, sqlrc(), sqlmsg$, name$, sal)
180    EXIT IF sqlrc(1) GT 4
190    PRINT USING "<######## ######.##": name$, sal
200 LOOP
210 IF sqlrc(2) EQ 100 THEN
220    PRINT "*** End of Table ***"
230 ELSE
240    PRINT "*** Data Base or SQL Statement Error:"
250    PRINT "    BASIC/SQL Return Code:  ";sqlrc(2)
260    PRINT "    Data Base Return Code:  ";sqlrc(8)
270    PRINT "    Error Message Text: ";sqlmsg$
280 END IF
290 END
```

Notes

- Lines 160-200: The program loops, repeatedly retrieving a row from the data base system and then printing the data. Notice again, that the value list of statement 170 (NAME$ and SAL) positionally matches the order of the selected columns (NAME and SALARY). The loop is exited (line 180) only when a return code greater than 4 is reported in SQLRC(1).

- Lines 210-280: A SQL return code of 100 in SQLRC(2) indicates that there are no more rows left to retrieve. (This is like an "end of file" condition.) If this is the case, we print an end of table message and terminate. However, if we exit the loop with a SQLRC(2) value other than 100, we have an error condition, in which case appropriate diagnostic information is printed.

Example: Multiple Row Retrieval for an Entire Table: Using multiple row retrieval you can retrieve an entire result table into the value list arrays. This can be a useful programming technique if your result table does not contain too many rows and if your program can take advantage of the array form. The following example illustrates how to retrieve the entire salary column from the Q.STAFF table with a single CALL SQL statement. It then passes the array to a subprogram named STATS which computes and prints some summary statistics.

```
100 REM: Program to retrieve entire SALARY column
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256, sal(100)
130 stmt$ = "SELECT SALARY FROM Q.STAFF"
140 CALL SQL (stmt$, sqlrc(), sqlmsg$, sal())
150 SELECT sqlrc(1)
160    CASE LE 4
170       PRINT "*** More Rows in Data Base Than Expected"
180       PRINT "    Number of Rows Retrieved: ";sqlrc(3)
190    CASE 8
200       IF sqlrc(2) EQ 100 THEN
210          MAT sal = sal(sqlrc(3))  ! Redim. to match rows returned
220          PRINT "Analysis of Salaries in Q.STAFF"
230          CALL stats (sal())
240       ELSE
250          PRINT "*** BASIC/SQL Error:"
260          PRINT "    BASIC/SQL Return Code:  ";sqlrc(2)
270          PRINT "       Error Message Text:  ";sqlmsg$
280       END IF
290    CASE ELSE
300          PRINT "*** Data Base Error:"
310          PRINT "    Data Base Return Code:  ";sqlrc(8)
320          PRINT "       Error Message Text:  ";sqlmsg$
330 END SELECT
340 END
  .
  .
  .
400 SUB stats(x())
410 REM: Mean, median, standard deviation,
420 REM  variance, range, max and min
  .
  .
  .
790 SUB END
```

Notes

- Line 120 explicitly dimensions the SAL array to have 100 elements. This example requires that this dimension be greater than the number of rows retrieved, in order to obtain correct results.

- Lines 160-180: If the SQLRC(1) return code is 0 or 4, then either

  - Not all of the rows were returned, or

  - The number of rows returned exactly matches the size of the SAL array argument.

  Either case is considered an error—we're getting back more rows than we planned for. We print an appropriate message followed by the value of SQLRC(3) which specifies the number of row values actually retrieved.

- Line 200: If the SQLRC(1) return code is 100 we know that the SQL statement executed as expected. In other words, the entire set of salaries is returned, the number of rows returned is less than the dimension of the SAL array, and there were no errors.

- Line 210: The SAL array is redimensioned to match the number of rows actually returned. This is required by the subprogram. (If we didn't perform the redimensioning, then additional zero values in the array would skew the statistical parameters.)

- Lines 220-230: A report heading is printed and then the STATS subprogram is called to calculate and print statistics.

- Lines 240-270: If SQLRC(2) is neither 100 nor zero, it means BASIC has found a general error condition. The appropriate diagnostic message is printed.

- Lines 290-320: If the general return code is not 0, 4, or 8, the only other possibility is 12 (error detected by the data base system). The specific error condition is found in SQLRC(8).

- Lines 400-790: A subprogram is used for the calculations rather than in-line code in order to illustrate both the convenience of multiple row retrieval, and the use of a generalized subprogram. A listing of this subprogram is found in Appendix E, "The STATS Subprogram" on page 253.

Example: Single Row Retrieval With Null Values: This program is a modification of a previous example which retrieved single rows. The difference is that the commission column (COMM) is retrieved in addition to SALARY and NAME for those employees having salaries greater than $20,000. Since the commission column is known to contain nulls, our program must be sensitive to this situation and print nulls in a meaningful way. For this example, nulls will be converted to zeros.

```
100 REM: Program to retrieve data a row a time and recognize nulls
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256
130 stmt$ = "SETOPT NULL 0" ! Assign zero to numeric nulls
140 CALL SQL (stmt$, sqlrc(), sqlmsg$)
150 IF sqlrc(1) NE 0 THEN GOTO sql_error
160 stmt$ = "SELECT NAME, SALARY, COMM FROM Q.STAFF "
165 stmt$ = stmt$ & "WHERE SALARY > 20000"
170 PRINT "NAME          SALARY     COMMISSION"
180 PRINT
190 DO
200    CALL SQL (stmt$, sqlrc(), sqlmsg$, name$, sal, comm)
210    EXIT IF sqlrc(1) GT 4
220    PRINT USING printmask: name$, sal, comm
230 LOOP
240 IF sqlrc(2) EQ 100 THEN
250    PRINT "*** End of Table ***"
260 ELSE
270    GOTO sql_error
280 END IF
290 STOP
300 printmask:  IMAGE :<######## ######.## ######.##
310 sql_error:  ! Processing for unexpected SQL return code
320    PRINT "*** Data Base or SQL Statement Error:"
330    PRINT "       SQL Statement Text:  ";stmt$
340    PRINT "    BASIC/SQL Return Code:  ";sqlrc(2)
350    PRINT "    Data Base Return Code:  ";sqlrc(8)
360    PRINT "    Error Message Text: ";sqlmsg$
370 END
```

Notes

- Lines 130-150: The CALL SQL statement is issued to cause all numeric nulls to be assigned a zero value. Although SETOPT is a special command to BASIC, and is not "passed through" to the data base system, it's still a good idea to test the SQLRC(1) return code to ensure that all is well.

  Note: If we omitted these statements, BASIC would, by default, assign nulls the value of the most negative integer number, (that is, -2147483648). Zero is the most meaningful choice for data of this program.

- Lines 310-360: Since we have multiple CALL SQL statements, these lines provide a single generalized error reporting routine which may be referenced using the label SQL__ERROR. Also, the STMT$ variable is printed to determine which CALL SQL caused the error.

- Since we know that the COMM column contains no zero values, we omit testing SQLRC(4) for null conflicts here.

Example: Single Row Retrieval With NULL Conflicts: This program illustrates how a program can recognize nulls even when their representation in a BASIC variable conflicts with a valid data value.

The Q.STAFF table is not appropriate for this example so the table used for this example is one called STOCKMARKET which is shown below. This table isn't available to you, so just follow the example "on paper".

```
COMPANY                SHARES  DIVIDEND  EARNINGS  PRICE
-------                ------  --------  --------  -----
ABC Inc.              2000000  0.95      19000000   9.50
Fast Buck Associates   100000  0.00     -52000000   1.00
New Age Financial      500000  0.00             0   1.00
United Paper-Clips Co 1000000  0.50          NULL  15.00
  .
  .
  .
```

Figure 32. STOCKMARKET Table

Notice that the EARNINGS column can contain NULL entries. This means that no earnings information is available for the company. It does not mean that the earnings are zero.

The object is to write a BASIC program that will print, for each company in the table, the name of the company, the dividend rate, and the earnings per share. This task is straightforward enough except for the fact that EARNINGS can contain any positive, negative, or zero value. If we find that EARNINGS is null, then we want to print "N/A" for earnings per share. The key question is how to distinguish between null entries and valid data.

```
100 REM: Retrieve data a row a time and resolve NULL conflicts
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256, c$*24
130 stmt$ = "SETOPT NULL 0" ! Assign zero to all nulls
140 CALL SQL (stmt$, sqlrc(), sqlmsg$)
150 IF sqlrc(1) NE 0 THEN GOTO sql_error
160 stmt$ = "SELECT COMPANY, SHARES, DIVIDEND, EARNINGS, PRICE "
170 stmt$ = stmt$ & "FROM STOCKMARKET"
180 PRINT "COMPANY                           DIV%  EARN/SHR"
190 PRINT
200 DO
210    CALL SQL (stmt$, sqlrc(), sqlmsg$, c$, s, d, e, p)
220    EXIT IF sqlrc(1) GT 4
230    PRINT USING printmask: c$, 100*d/p;
240    IF e EQ 0 AND sqlrc(5) EQ 0 THEN
250       PRINT "   N/A"
260    ELSE
270       PRINT USING "######.##": e/s
280    END IF
290 LOOP
300 IF sqlrc(2) EQ 100 THEN
310    PRINT "*** End of Table ***"
320 ELSE
330    GOTO sql_error
340 END IF
350 STOP
360 printmask:  IMAGE :<##################### ######.##
370 sql_error:  ! Processing for unexpected SQL return code
380    PRINT "*** Data Base or SQL Statement Error:"
390    PRINT "        SQL Statement Text:  ";stmt$
400    PRINT "     BASIC/SQL Return Code:  ";sqlrc(2)
410    PRINT "     Data Base Return Code:  ";sqlrc(8)
420    PRINT "     Error Message Text: ";sqlmsg$
430 END
```

Notes:

- Line 230: Notice that the last item in the print list (100*D/P) is followed by a semicolon. This causes the next printed value (from line 250 or 270) to appear immediately afterwards on the same line.

- Lines 240-280: If E is zero, we have an ambiguity since it may be a legitimate zero, or a zero used to represent null. We can distinguish between these two cases by testing SQLRC(5)—the "null column conflict indicator". If there are no null conflicts with the retrieved data, then E is zero, and SQLRC(5) will also be zero. However, if there is a null conflict, then SQLRC(5) indicates which column contains the null conflict. (For example, 1 = conflict in COMPANY, 2 = conflict in DIVIDEND). Since we only expect nulls in column 3, we don't need to test for anything other than a zero value.

  Note: SQLRC(5) actually specifies the first occurrence of a null conflict. If you expect null conflicts to occur simultaneously in multiple columns of the same row, you will not be able to detect them beyond the first conflict. Therefore, you should carefully choose the representation of nulls via the SETOPT NULL command to minimize this situation. If you are using segmented (multi-row) retrieval and need to detect null conflicts, you should also test SQLRC(4)—the "null conflict row indicator". It identifies the row number in your value list array of the first null conflict.

Example: Table Creation: Creating (that is, defining) a new table in the data base is a relatively easy task; however, not all users have the requisite authority. Therefore, before actually trying to run the remaining programs, check with your data base administrator to be sure you have the proper authority.

Table creation involves three steps:

1. Specifying a CREATE TABLE SQL statement
2. Executing the statement via the CALL SQL statement
3. Making the new table definition permanent by executing a a CALL SQL in which the SQL statement is a COMMIT command.

In the following example a table named PAYROLL is created. It has 3 columns -- NAME, SALARY, and COMM. The names and attributes of the columns (e.g., data types) are identical with the corresponding columns in Q.STAFF.

```
100 REM: Program to create a new table in the data base
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256
130 stmt$ = "CREATE TABLE PAYROLL "
140 stmt$ = stmt$ & "(NAME   VARCHAR(9),"
150 stmt$ = stmt$ & "SALARY DECIMAL(7,2),"
160 stmt$ = stmt$ & "COMM   DECIMAL(6,2))"
170 CALL SQL (stmt$, sqlrc(), sqlmsg$)
180 IF sqlrc(1) LE 4 THEN
190    stmt$ = "COMMIT WORK"
200    CALL SQL (stmt$, sqlrc(), sqlmsg$)
210    IF sqlrc(1) LE 4 THEN
220       PRINT "*** Table Created"
230       STOP
240    END IF
250 END IF
260 REM: Processing for unexpected SQL return code
270 PRINT "*** Data Base or SQL Statement Error:"
280 PRINT "       SQL Statement Text:  ";stmt$
290 PRINT "    BASIC/SQL Return Code:  ";sqlrc(2)
300 PRINT "    Data Base Return Code:  ";sqlrc(8)
310 PRINT "    Error Message Text: ";sqlmsg$
320 END
```

Notes:

- Lines 130-160 construct the CREATE TABLE SQL statement. Notice that because the statement is too long to place on a single line multiple lines are used to "build up" STMT$ with a series of character catenation operations. Also, don't forget the parentheses around the list of column names and data types. They're required and their omission will generate a negative SQL return code.

- Line 160: You may wish to add an "IN clause" to your SQL statement in order to specify the name of the "space" in the data base where your new table is to reside. This can be done by adding the following statement to your program: (Assume that "MYSPACE" is the name of the target space.)

  ```
  165 stmt$ = stmt$ & " IN MYSPACE"
  ```

  Note: An "IN clause" is required for the VM environment (when using SQL/DS VM), and may be required or recommended by your installation for MVS (DB2). Check with your system administrator to be sure.

A point regarding terminology—in DB2 the space is referred to as a "TABLESPACE", and in SQL/DS it is called a "DBSPACE". Also, in DB2, there are additional options available for the IN clause.

- Lines 190-210: A COMMIT WORK is needed in order to make the table definition permanent. And, as for all other SQL operations, return codes are tested to ensure that the statement executed correctly.

  Note: If the COMMIT WORK failed, the table definition would be automatically "undone" when the program terminated. This is because an automatic ROLLBACK WORK is performed by BASIC whenever uncommitted work is left at the end of the main program. You could also explicitly perform a ROLLBACK WORK operation via CALL SQL, for example:

  ```
  CALL SQL ("ROLLBACK WORK", sqlrc(), sqlmsg$)
  ```

  However, this is seldom necessary since it is the default action.

Example: Filling a Table by Copying Rows From Another Table: The easiest method of filling a table with data is to copy rows from another table. This example copies the names, salaries, and commissions of all employees in the Q.STAFF table who have salaries greater than $20,000, and inserts the result into the newly created (and empty) PAYROLL table.

```
100 REM: General program to execute SQL statements that
110 REM: don't use value-lists or host-variables.
120 REM: SQL statement is entered by user when prompted.
130 OPTION BASE 1: INTEGER sqlrc
140 DIM sqlrc(20), stmt$*1000, sqlmsg$*256, a$*156
150 PRINT "Enter SQL statement lines - null line when finished"
160 DO
170    LINE INPUT "  SQL ===>": a$
180    stmt$ = stmt$ & a$ & " "        ! add blank between lines
190 LOOP UNTIL a$ = ""
200 CALL SQL (stmt$, sqlrc(), sqlmsg$)
210 row_count = sqlrc(9)
220 IF sqlrc(1) EQ 0 THEN
230     stmt$ = "COMMIT WORK"
240     CALL SQL (stmt$, sqlrc(), sqlmsg$)
250     IF sqlrc(1) EQ 0 THEN
260         PRINT "SQL Statement Successfully Processed"
270         PRINT row_count;" Rows Inserted/Updated/Deleted"
280         STOP
290     END IF
300 END IF
310 REM: Processing for unexpected SQL return code
320 PRINT "*** Data Base or SQL Statement Error:"
330 PRINT "       SQL Statement Text:  ";stmt$
340 PRINT "    BASIC/SQL Return Code:  ";sqlrc(1)
350 PRINT "    Data Base Return Code:  ";sqlrc(8)
360 PRINT "    Error Message Text: ";sqlmsg$
370 END
```

When the program is run, you will receive the prompt message:

```
Enter SQL statement lines - null line when finished
  SQL ===>
```

You will receive the "SQL ===>" prompt each time you press ENTER after keying in a line of SQL. When you have completed the SQL statement press ENTER once again to enter a null line which terminates the prompting.

To perform the copy operation, respond to the prompts with the following SQL lines:

```
  SQL ===> INSERT INTO PAYROLL
  SQL ===> SELECT NAME, SALARY, COMM
  SQL ===> FROM Q.STAFF
  SQL ===> WHERE SALARY > 20000
  SQL ===>
```

If everything is correct you will receive the following confirmation messages:

```
SQL Statement Successfully Processed
  6  Rows Inserted/Updated/Deleted
STOP AT LINE 280.
```

Notes:

- The structure of this program is similar to that of the table creation example shown previously. The chief difference is that the SQL statement is not built from constants in the program. Instead, a more generalized approach is used where the program prompts the user for the SQL statement and then executes the user supplied SQL. In fact, this form of BASIC programming can be used for all SQL statements that do not involve value lists. See "Other SQL Examples" on page 187 for the different SQL statements you can use.

- It's not necessary to type each SQL clause on a separate line but most SQL users feel it's more readable that way. You can type the entire statement on a single line (if it fits); however, you should not press ENTER in the middle of a word, number, or quoted character constant.

- Line 170: Notice that a LINE INPUT statement is used, rather than the usual INPUT. This is needed because SQL statements frequently contain commas and/or quotes, which are treated as delimiters by INPUT.

- Line 210: SQLRC(9) is a count from the data base system of the number of rows affected by the SQL statement (that is, the number of rows inserted, updated, or deleted).

  Note: SQLRC(9) will have a zero value except for insert, update, and delete SQL statements. For example, a successful CREATE TABLE SQL statement will return a zero value for SQLRC(9).

Example: Inserting Single Rows Using Index Values: The following example inserts rows into the PAYROLL table by accepting input data for employee name, salary, and commission. The data is stored in variables which are passed as arguments to the SQL statement. The SQL statement uses index values which act as "place holders". The variables in the value-list are substituted into the SQL statement. The user is prompted for a row of data which is then inserted into the table. The following example illustrates this technique by prompting the user for data, and then inserting a row into the PAYROLL table,.

```
100 REM: Insert of single rows using index values
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256
130 PRINT "Input for PAYROLL Table"
140 PRINT "Enter QUIT for Employee Name to Exit"
150 DO
160    PRINT
170    INPUT "    Employee Name ===> ": n$
180    EXIT IF UPRC$(n$) = "QUIT"
190    INPUT "     Salary, Comm ===> ": s,c
200    stmt$ = "INSERT INTO PAYROLL VALUES (:1, :2, :3)"
210    CALL SQL (stmt$, sqlrc(), sqlmsg$, n$, s, c)
220    EXIT IF sqlrc(1) GT 4
230    stmt$ = "COMMIT WORK": CALL SQL(stmt$, sqlrc(), sqlmsg$)
240    EXIT IF sqlrc(1) GT 4
250    PRINT "*** Row added to table"
260 LOOP
270 IF sqlrc(1) GT 4 THEN
280    PRINT "*** Data Base or SQL Statement Error:"
300    PRINT "    BASIC/SQL Return Code:  ";sqlrc(2)
310    PRINT "    Data Base Return Code:  ";sqlrc(8)
320    PRINT "    Error Message Text: ";sqlmsg$
330 END IF
340 END
```

Notes:

- Line 200: ":1, :2, :3" are index values, which represent the 1st, 2nd, and 3rd arguments in the CALL SQL value-list.
- Line 210: Note that N$, S, and C are variables in the value-list. They are arguments which match the index values in the SQL statement.
- General

  If your insert conflicts with the table definition or other rows in the table, then the insert will fail and SQLRC(1) will be set to 12 and a negative SQL code will be returned in SQLRC(8).

  - You must be certain that the number of column values being inserted agrees with the table definition. If you are unsure of the order of the columns, you can specify them directly in your SQL statement, like this:

    ```
    INSERT INTO PAYROLL (NAME, SALARY, COMM) VALUES (:1, :2:, :3)
    ```

    The above SQL statement explicitly states that :1 is the NAME column, :2 is SALARY, etc.

  - You must also be careful to insert compatible data types. Although BASIC generally handles this problem for you, you must still match numeric

columns with numeric variables and character columns with character variables.

- Another type of conflict can occur when your table has a unique index and the row you are inserting duplicates the index's "key columns".

- Finally, your insert will conflict with the table definition if you attempt to insert a row with a null for a column defined with the NOT NULL attribute.

Example: Inserting Multiple Rows Using Segments: Just as we can retrieve multiple rows using arrays in the CALL SQL value list, we can insert multiple rows by specifying arrays as variables in the value-list. The following example (a variation of the previous one), allows you to insert up to 5 rows at a time into the PAYROLL table. This approach can result in improved performance because there are fewer executions of the CALL SQL statement. Also, you may appreciate the segmented data entry technique since there is frequently a noticeable delay and wait while the data base system is processing. Thus, entering data in batches results in fewer pauses and more predictable performance.

```
100 REM: Insert of multiple rows using segments
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256, n$(5), s(5), c(5)
130 PRINT "Input for PAYROLL Table"
140 PRINT "Enter QUIT for Employee Name to Exit"
150 DO
160   FOR i = 1 TO 5
170     PRINT
180     INPUT "    Employee Name ===> ": n$(i)
190     EXIT IF UPRC$(n$(i)) = "QUIT"
200     INPUT "     Salary, Comm ===> ": s(i) ,c(i)
210   NEXT i
220   i = i - 1          ! Adjust to actual number input
230   EXIT IF i = 0
240   sqlrc(3) = i       ! Specify number of rows to insert
250   PRINT
260   PRINT "*** Data Base Processing - Please Wait"
270   stmt$ = "INSERT INTO PAYROLL VALUES (:1, :2, :3)"
280   CALL SQL (stmt$, sqlrc(), sqlmsg$, n$(), s(), c())
290   EXIT IF sqlrc(1) GT 4
300   stmt$="COMMIT WORK": CALL SQL (stmt$, sqlrc(), sqlmsg$)
310   EXIT IF sqlrc(1) GT 4
320   PRINT "*** "; i; " rows added to table"
330 LOOP WHILE i = 5
340 IF sqlrc(1) GT 4 THEN
350   PRINT "*** Data Base or SQL Statement Error:"
360   PRINT "        SQL Statement Text:  ";stmt$
370   PRINT "     BASIC/SQL Return Code:  ";sqlrc(2)
380   PRINT "     Data Base Return Code:  ";sqlrc(8)
390   PRINT "     Error Message Text: ";sqlmsg$
400 END IF
410 END
```

Notes:

- Line 120: N$, S, and C are value-list variables dimensioned as arrays of 5 elements. This size was arbitrarily chosen—you may wish to use larger segments.

- Line 240: If we have not "filled" the arrays, we must tell BASIC how many rows to insert. This can be done in two ways:

  1. By specifying the number of rows in SQLRC(3), as is done in the example. This technique is similar to that for determining the number of rows actually retrieved - see the example on page 169

  2. By leaving SQLRC(3)=0 and redimensioning the arrays to their current size.

     ```
     240 MAT n$ = n$ (i): MAT s = s(i): MAT c = c(i)
     ```

This works because BASIC will insert all elements if SQLRC(3) is zero.

Example: Table Update - Changing the Values of Rows: This example updates the salaries and commissions of employees in the PAYROLL table. The user enters the employee name and new values for SALARY and COMM. Null values are entered as -1.

```
100 REM: Update of a single row using index values and value-list arguments
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, updt$*1000, sqlmsg$*256
130 updt$ = "UPDATE PAYROLL SET SALARY = :2, "
140 updt$ = updt$ & "COMM = :3  WHERE NAME = :1"
150 PRINT "Changes to PAYROLL Table"
160 PRINT "Enter QUIT for Employee Name to Exit"
170 PRINT "Enter -1 for NULL Salary or Comm"
180 stmt$="SETOPT NULL -1": CALL SQL (stmt$, sqlrc(), sqlmsg$)
190 DO WHILE sqlrc(1) LE 4
200   PRINT
210   INPUT "    Employee Name ===> ": n$
220   EXIT IF UPRC$(n$) = "QUIT"
230   INPUT " New Salary, Comm ===> ": s,c
240   stmt$=updt$: CALL SQL (stmt$, sqlrc(), sqlmsg$, n$, s, c)
250   IF sqlrc(1) LE 4 THEN
255     IF sqlrc(8) NE 100 THEN            ! If name found
260       row_count = sqlrc(9)
270       stmt$="COMMIT WORK": CALL SQL (stmt$, sqlrc(), sqlmsg$)
280       PRINT "***"; row_count; " Row(s) updated"
290     ELSE                                ! If name not found
300       PRINT "*** No employee named "; n$
305     END IF
308   ELSE                                  ! Error detected
310   END IF
320 LOOP
330 IF sqlrc(1) GT 4 AND sqlrc(8) NE 100 THEN
340   PRINT "*** Data Base or SQL Statement Error:"
350   PRINT "        SQL Statement Text:  ";stmt$
360   PRINT "   BASIC/SQL Return Code:  ";sqlrc(2)
370   PRINT "   Data Base Return Code:  ";sqlrc(8)
380   PRINT "   Error Message Text: ";sqlmsg$
390 END IF
400 END
```

Notes:

- Lines 170-180: The SETOPT NULL command is issued to translate all -1 input values to null. (This command works for both retrieval and insert.) To specify a numeric null, you enter a -1. Note that if the SETOPT statement fails, SQLRC(1) will be set to 8 and no updates will be made, because the update loop is protected by its WHILE SQLRC(1) LE 4 condition.

- Lines 250-255: A value of 100 in SQLRC(8) means that the data base system couldn't find the employee name you entered. SQLRC(1) is set to 4, indicating a warning which some programs (like this one) may find of interest, while others may not. Suppose, for example, that you were uneasy about maybe having entered CLARK instead CLERK a few times into the table. Why search for it? All you need do is execute the query:

```
UPDATE TABLE SET JOB = 'CLERK' WHERE JOB = 'CLARK'
```

If CLARK is found anywhere, it gets corrected, if not, no harm is done. In such a case a return code of 100 in SQLRC(8) would be of no concern. Contrast this with the SQLRC(8) of 100 that results from an end-of-table condition on SELECT. Since an end-of-table condition can't normally be ignored, SQLRC(2) is therefore set to the less ignorable value of 8.

- Line 260 and 280: SQLRC(9) will show the number of rows actually updated. In this example, we're assuming it'll always be 1. However, in the "real world" it is possible that several employees may have the same last name, so more than one row can be updated. It's good practice to examine SQLRC(9) to ensure that the number of rows updated matches your expectation.

Example: Deleting Rows from a Table: This example allows you to delete a row from the PAYROLL table by entering the name of the employee you want deleted from the table.

```
100 REM: Delete rows using index values and value-list arguments
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256
130 PRINT "Deletes Rows from PAYROLL Table"
140 PRINT "Enter QUIT for Employee Name to Exit"
150 DO WHILE sqlrc(1) LE 4
160   PRINT
170   INPUT "    Employee Name ===> ": n$
180   EXIT IF UPRC$(n$) = "QUIT"
190   stmt$ = "DELETE FROM PAYROLL WHERE NAME = :1"
200   CALL SQL (stmt$, sqlrc(), sqlmsg$, n$)
210   IF sqlrc(1) LE 4 THEN       ! If OK (or only warning)
215     IF sqlrc(8) NE 100 THEN  ! If name found
220       PRINT "***"; sqlrc(9); " Row(s) will be deleted. ";
230       INPUT "Do you wish to commit?": a$
240       IF UPRC$(a$(1:1)) = "Y" THEN
250         stmt$="COMMIT WORK": CALL SQL (stmt$,sqlrc(),sqlmsg$)
260         IF sqlrc(1) LE 4 THEN PRINT "*** Done"
270       ELSE
280         stmt$="ROLLBACK WORK": CALL SQL (smt$,sqlrc(),sqlmsg$)
290         IF sqlrc(1) LE 4 THEN PRINT "*** Deletes Cancelled"
300       END IF
310     ELSE                     ! Warning -- name not found
320       PRINT "*** No employee named "; n$
325     END IF
328   ELSE                        ! Error detected
330   END IF
340 LOOP
350 IF sqlrc(1) GT 4 AND sqlrc(8) NE 100 THEN
360   PRINT "*** Data Base or SQL Statement Error:"
370   PRINT "      SQL Statement Text:  ";stmt$
380   PRINT "   BASIC/SQL Return Code:  ";sqlrc(2)
390   PRINT "   Data Base Return Code:  ";sqlrc(8)
400   PRINT "   Error Message Text: ";sqlmsg$
410 END IF
420 END
```

Notes:

- Line 220: The number of rows (tentatively) deleted is returned in SQLRC(9)—the table modification indicator. Since it's possible to have multiple row entries with the same name, a SQL DELETE statement could delete more than one row. Therefore, it's advisable to test this indicator before committing the delete.
- Line 280: A SQL ROLLBACK is performed in case you change your mind about the delete. The ROLLBACK has the same form as the COMMIT and also requires that the SQL return code be tested.

Example: Dropping a Table from the Data Base: This operation can be most easily performed by running the "generalized" program shown on "Example: Filling a Table by Copying Rows From Another Table" on page 179.

When this program is RUN, respond to the prompt message:

```
Enter SQL statement lines - null line when finished
  SQL ===>
```

with the following SQL statement

```
SQL ===> DROP TABLE PAYROLL
SQL ===>
```

Note: Use the DROP TABLE statement with care; both the table's contents and its definition are destroyed, as well as any dependent data base objects such as views and indexes.

Other SQL Examples

The SQL examples shown in this section can be modified to illustrate many other ways you can use BASIC to access relational bases. For example, you can use the generalized program on "Example: Filling a Table by Copying Rows From Another Table" on page 179 and respond to the prompt with the appropriate SQL to execute any SQL statement that does not require a value list. This includes such statements as:

- ALTER TABLE
- CREATE INDEX
- CREATE VIEW
- DROP VIEW
- GRANT
- LOCK TABLE
- REVOKE

Additional examples dealing with more advanced topics can be found in Appendix C, "SQL Examples—Advanced Techniques" on page 241.

Running Your Programs

You can run BASIC programs either from inside the BASIC environment using the RUN command, or as compiled programs invoked from outside the BASIC environment.

Direct Execution Versus Compiled Execution

Examples in this document assume that you are using BASIC inside the BASIC environment and the programs you want to run are entered into the workspace. These programs are in source form: a series of program lines in exactly the form in which you entered them.

Running a source program in your workspace is called *direct execution* of the program. This is possible only when you're operating inside the BASIC environment (the workspace only exists inside the BASIC environment); direct execution requires that the BASIC Processor must first translate each BASIC statement to machine instructions and then execute these instructions.

When you *compile* a program, either with the COMPILE command inside the BASIC environment or by use of the compiler outside the BASIC environment, the translation from source statements to machine instructions is performed for the whole program, and the resulting machine instructions are saved in an object file. Object files can be loaded and run either from inside the BASIC environment or from the host operating system outside the BASIC environment. In fact, loading and running an object file is the only method of executing a BASIC program from the host operating system. Remember that an object file contains only machine instructions. The original source statements are retained only in the source file. So, if you run a compiled program from inside the BASIC environment (see "Running Your Programs — Compiled Execution" on page 194), you cannot list it, use immediate statements to examine or change its variables, or edit the program. You can only edit the source program and then recompile it.

Running Your Programs — Using the RUN Command

When you want to execute your source program, you use the RUN command to execute it directly, without first compiling it. All variables are set to zero or null, and execution begins at the lowest line number in your program and continues in line number sequence.

Use the RUN command when you want to execute your source program without a prior compilation or you want to debug the program.

Direct Execution of the Program in Your Workspace

You can run the source program that is already in your workspace or you can load a different source program into your workspace and run it.

The first operation of the RUN command is to perform a global semantic check for such errors as:

- Branching to a nonexistent line
- Branching into loops
- Unpaired DEF/FNEND, DO/LOOP, FOR/NEXT, IF/END IF, SELECT/END SELECT, SUB/END SUB statements

If errors are found during this global check or if your program has syntax errors, you are asked by BASIC if you want to run the program anyway. If you answer YES, BASIC starts executing your program. For example,

```
* run
```

starts running the source program already in your workspace. If your workspace is empty, you will get an error message.

Direct Execution of Filed Programs

If you want to execute a source program that is not currently in your workspace, you can specify the name of the program you want to run. For example:

```
* run aa (source)
```

loads the source program (file) AA into your workspace.

See your IBM BASIC System Services manual for file naming conventions. As the source file is loaded into your workspace, each line is syntax checked, errors are displayed for each incorrect line, and global semantic checks are made.

Actually, unless told otherwise, BASIC assumes RUN refers to a source file. So the "(SOURCE)" is unnecessary. The above command could be stated:

```
* run aa
```

After the program is loaded, BASIC starts running the program if there are no errors. If errors are found, BASIC prompts with a message asking if you want to run the program anyway. If you answer YES, BASIC starts executing your program.

Direct Execution with SPREC or LPREC

You may also include SPREC or LPREC on the above RUN commands. For example:

```
* run (lprec)
* run aa (sprec)
```

When the option is specified in the RUN command, the value specified is used for program units that do *not* explicitly specify the option in an OPTION statement.

These options determine the default precision that will be used when your program executes a PRINT statement that does not contain a USING clause and prints numeric values: 6 digits (SPREC) for short precision, or 12 digits (LPREC) for long precision.

LPREC and SPREC also specify the precision (single or double) for real data (except for variables that are explicitly typed in the program unit). SPREC specifies single precision real data, and LPREC specifies double precision real data.

If not specified, the default is as specified in a SET OPTION command. If not specified there, the default is as defined at installation. The IBM-supplied default for the installation option is SPREC.

Direct Execution with a Parameter (PARM)

A character string may be passed to a BASIC program when the program is invoked. For example:

```
* run test (parm ' 312 ')
```

where the character string is 312. The program can get the value of the character string passed by using the parameterless intrinsic function PARM$. For example:

```
10 print parm$
```

would print 312 (from the example above).

Interrupting Execution

You can interrupt the execution of your program in one of two ways.

The first way assumes you have determined prior to running your program where you want it to stop. In this case you either use the BREAK command to set breakpoints before execution is initiated, or code into your program PAUSE or BREAK statements at the points at which you want the program to stop.

The second way to interrupt execution is to generate an attention interrupt while the program is running. This causes the program to stop at the next statement to be executed. See your IBM BASIC System Services manual for details of how to generate an attention interrupt for your system.

When your program stops, BASIC tells you the line number at which it stopped, and you can use immediate statements to print or set variables in your program.

To resume execution of your program, you enter either GO or a null entry. The null entry (merely pressing ENTER) only works if you do nothing else. If you use any immediate statements or commands while the program is suspended, you must use the GO command to resume execution.

If you modify the program in any way, you cannot resume execution with a GO command; you must start over again at the beginning with the RUN command. (Modification could be caused by any line number editing, full-screen editing, or using one of the following commands: CHANGE, COMPILE, COPY, DELETE, DROP, EXTRACT, FETCH, INIT, LOAD, MERGE, RENUMBER, or RUN.)

If you are in an endless execution loop, or if you merely want to stop the execution of your program, generate an attention interrupt.

When you interrupt execution, your program stops execution at a statement boundary. This means it will honor the action you have coded in an ON ATTN statement when the current statement has finished processing.

For example, if the interrupt occurs when the following statement is being executed:

```
100 a=b+c
```

the interrupt is acknowledged when A has been assigned the sum of B and C.

During execution of array processing statements, the attention interrupt is acknowledged after an entire array has been processed. For example:

```
800 dim a(1000), b(1000)
810 mat a = (5)*b
```

If an attention interrupt is issued during execution of line 810, the interrupt is acknowledged after all 1000 entries of B have been multiplied by 5 and stored into all 1000 entries of A.

If an attention interrupt is issued while a PRINT statement is being executed, the interrupt is acknowledged after this statement has been executed. For example:

```
240 dim a(1000)
250 mat print a
```

If the attention interrupt is issued during execution of line 250, the interrupt is acknowledged after all 1000 entries of A have been printed.

You must decide how your program is to handle attention interrupts. You have three alternatives:

```
200 on attn ignore
```

ignores the interrupt, continues operation, and does not print a message.

```
210 on attn system
```

causes one of two results, depending on whether the program is run in interactive (that is, with a terminal) or in batch mode.

- In batch mode, the interrupt is ignored (that is, the same as ON ATTN IGNORE).
- In interactive mode (either inside or outside the BASIC environment), execution of the program is suspended after the current statement is finished executing, and BASIC prints an informational message. This is the default action, which occurs if you have no ON ATTN statement.

The ON ATTN GOTO statement causes control to be transferred to the line with the specified line number or label. For example:

```
220 on attn goto attn_interrupt
```

You must decide how your program is to handle this type of interrupt. You could code a sequence of steps to write a message, stop the program, or any appropriate operation. Just remember that control is transferred to the line with the line number or label you specify, regardless of the part of the program that is operating when you signal attention.

After an attention interrupt, you can enter commands and immediate statements. When you are ready to resume execution, use the RETRY and CONTINUE statements or the GO command. Execution will resume at the statement that was to be processed when the interrupt was acknowledged.

The first statement of the routine should be an ON ATTN IGNORE so that processing cannot be interrupted again thereby losing the location of the statement from which to RETRY or CONTINUE. The statement prior to the RETRY or CONTINUE should be an ON ATTN GOTO to reset the ON condition.

Performance Considerations

If you want your compiled programs to be smaller and to run faster, use the ON ATTN IGNORE statement.

You can eliminate one machine instruction (4 bytes) for each statement in the program by:

- Making ON ATTN IGNORE the **first** executable statement in your program unit, and
- Making ON ATTN IGNORE the **only** ON ATTN action statement in your program unit.

For more information, see your IBM BASIC System Services manual.

Running Your Programs — Compiled Execution

If the host system you are using supports interactive mode operation, you can compile your program after you have thoroughly tested it and just before you release it for distribution to the user community (that is, after it becomes a production program).

Execution of a program that has been compiled is faster than direct execution.

Execution of a program outside the BASIC environment requires less storage than executing a program inside the BASIC environment.

If the host system you are using supports only batch mode, you must always compile your program.

Compiled Execution — Inside the BASIC Environment

Compiled programs may be executed inside the BASIC environment. This may be desirable when faster execution and most of the features of the BASIC environment are desired.

Requesting Compilation — COMPILE Command

You should compile a program after it has been tested to ensure that it is error free, and when you are ready to use it in production. Then you can execute the compiled version of your program each time you use it, rather than compiling it first each time and then executing it. Running a compiled program saves computer time.

Object files generated by the COMPILE command are the same as generated by the compiler (see "Requesting Compilation — Compiler" on page 196). They may be executed using the RUN command (see "Requesting Compiled Execution — RUN Command" on page 195) or from the host operating system (see "Requesting Execution — Outside the BASIC Environment" on page 197 and "Requesting Batch Execution" on page 198).

Use the COMPILE command to compile the program in your workspace. (You may load a file into your workspace before compiling it.) The COMPILE command compiles the program in your workspace and produces two files, an object program file and a listing file. Normally and by default, these files are given the same name as your workspace. (Remember that the names of these files are system-dependent. See your IBM BASIC System Services manual for more details.) However, you may specify different names for either or both of them by use of the OUT and/or OBJECT clauses. Thus, assuming your workspace is named OLDPROG,

```
* compile
```

(without any clauses) will name both the object and listing files OLDPROG.

```
* compile out (newprog)
```

will name the listing file, but not the object file, NEWPROG.

```
* compile object (newprog)
```

will name the object file, but not the listing file, NEWPROG.

```
* compile out (newprog) object (newprog)
```

will name both files NEWPROG.

If your program has any errors that the compiler recognizes, error messages are displayed on your terminal and included in the listing file.

You may include options in your COMPILE command (for example, XREF). These are described in "Compiler Options" on page 198. The list of options is preceded by a left parenthesis and the options are separated by spaces or commas. A right parenthesis is required following the option if an OBJECT or OUT clause follows; otherwise it is optional. For example, the XREF and MAP options may be entered as:

```
* compile (xref, map)
```

If you wish to compile a program other than one in the workspace, you can specify a name in the COMPILE command and the named program will then be loaded into the workspace and compiled. Of course if you do this, the program currently in the workspace will be deleted. For example:

```
* compile test (map)
```

will load the program TEST into the workspace and compile it with the MAP option.

Requesting Compiled Execution — RUN Command

Use the RUN command with the OBJECT option to execute your compiled program. The object program file created previously is executed. (An object program file is a file containing the linkable or executable results of the compilation.)

```
* run xyz (object)
```

reads the previously compiled program XYZ and starts running it. If program XYZ has not been compiled, you will receive an error message.

The options SPREC and LPREC cannot be used in the RUN command if the program has been compiled. You can, however, specify the STEP and PARM options when running a compiled program. The STEP option will not take effect until the program chains to a source program.

If an object program is executed, the workspace is cleared.

Note: Running object inside the BASIC environment may require more storage than running a source program. If you encounter insufficient storage problems while running object, consider "Requesting Execution — Outside the BASIC Environment" on page 197. (Source programs execute in the workspace while object programs execute in non-workspace storage. When BASIC is invoked, a large portion of the available storage is allocated for the workspace with a smaller amount of storage left for running object programs, for dynamically loading the

Library, for buffers, and for other storage requirements. Outside the BASIC environment, a workspace is not allocated and, therefore, more storage is available for executing object programs.)

Compiled Execution — PAUSE Statement

Your program identifies the PAUSE statement and execution is halted just as it does during direct execution. You may press ENTER to continue execution or enter BASIC commands.

When the PAUSE statement is executed, any associated message is displayed at your terminal. If the PAUSE statement does not contain a message, BASIC displays "PAUSE AT LINE line-number."

If a compiled program is executed outside the BASIC environment but at a terminal, a PAUSE statement will display the message and wait for the user to press ENTER before continuing, but commands will not be accepted.

Note that, if a compiled program is executed in batch (that is, without a terminal), PAUSE statements are ignored.

Compiled Execution — Outside the BASIC Environment

Compiled programs may be executed outside the BASIC environment. This may be desirable when the full features of the BASIC environment are not needed. It is required if the program must be run in batch (that is, without a terminal); see "Requesting Batch Execution" on page 198. To run a program outside the BASIC environment, the program must be compiled. The execution of compiled programs outside the BASIC environment requires the use of the IBM BASIC Library that is dependent on the host system. (See your IBM BASIC System Services manual for details.)

Requesting Compilation — Compiler

To compile your program outside the BASIC environment, the compiler expects its input in the BASIC source file format. If you entered your program using BASIC inside the BASIC environment, you must use the SAVE command to save your program in the BASIC source file format. Files stored using the STORE command cannot be compiled by the compiler. If the host system you are using does not support interactive mode, refer to your IBM BASIC System Services manual to determine how to create a BASIC source file.

Object files generated by the compiler are the same as generated by the COMPILE command (see "Requesting Compilation — COMPILE Command" on page 194). They may be executed using the RUN command (see "Requesting Compiled Execution — RUN Command" on page 195) or from the host operating system (see "Requesting Execution — Outside the BASIC Environment" on page 197 and "Requesting Batch Execution" on page 198).

If you are logged on to the system, you can invoke the compiler by issuing the BASIC command designating a file. For example:

```
BASIC TEST
```

would compile the source program TEST. For more information on invoking the compiler while logged on to the system and how to invoke the compiler in batch, see your IBM BASIC System Services manual. Your organization may have another invocation procedure; check with your system administrator.

The compiler produces two files, an object program file and a listing file.

You may include options (for example, FIPS). These are described in "Compiler Options" on page 198. For example:

```
BASIC TEST (FIPS NOPROF)
```

See "Performance Considerations" on page 193 for information on making compiled programs more efficient.

Requesting Execution — Outside the BASIC Environment

A compiled program can be executed from the operating system outside the BASIC environment.

The execution of your program outside the BASIC environment requires a run-time environment dependent on the host system. If you are logged on to the system, you can execute an object program using the BASICRUN command. For example:

```
BASICRUN TEST
```

would execute the compiled program TEST. For more information on executing a program while logged on to the system or in batch mode, see your IBM BASIC System Services manual. Your organization may have another invocation procedure; check with your system administrator.

The program may output to the terminal and request input from the terminal.

You may also specify a parameter string to be passed to the program. For example:

```
BASICRUN TEST  THIS IS A PARM
```

The program TEST can use the PARM$ function. For the above example, PARM$ would return the value, "THIS IS A PARM".

A compiled program can also be linked with the required BASIC Library routines to create a self-contained application load module. The linked module can then be executed. For example, the linked module TEST could be executed by entering:

```
TEST
```

For more information, see your IBM BASIC System Services manual.

Requesting Batch Execution

Use the batch mode of running your program if the host system only supports the batch mode, or if speed of execution, rather than interfacing with your user in an interactive manner is important, or you do not want to tie up your terminal while your program is running. When you are running in batch, your program does not have access to a terminal.

The types of programs that work well in batch operations are report programs that are processing large files and reporting the results of these operations. These batch programs rarely need information supplied by the user, except for options that can be supplied through control files at the beginning of the execution of the program.

If you determine that your program is to execute in batch mode, you must also compile it to object program format before executing it.

The execution of your program in batch mode requires the use of the IBM BASIC Library that is dependent on the host system. See your IBM BASIC System Services manual for this and for how to invoke a program with a parameter.

In the batch mode of operation, if you have coded a PAUSE statement in your program, it is ignored.

Furthermore, any statements that you have used in your programs that communicate with the terminal (such as INPUT or PRINT) interface with a file. (See your IBM BASIC System Services manual for details.) Statements which require the use of special terminal characteristics such as INPUT FIELDS and PRINT FIELDS may not be used in batch mode. See your IBM BASIC System Services manual for a discussion of the batch run-time environment requirements.

Compiler Options

These options allow you to:

- Control the output from the compiler
- Indicate to your program the maximum number of significant digits to print when it is processing a PRINT statement (without the USING clause)
- Define the precision of real data

Specification of the options for the compiler is system-dependent. (See your IBM BASIC System Services manual for details.) Specification of the options for the COMPILE command is explained in "Requesting Compilation — COMPILE Command" on page 194.

The compiler options you can specify are listed below. The vertical bar (|) separates two mutually exclusive compiler options; this means you can select one or the other, but not both. The IBM-supplied defaults are underlined. Check with your system administrator about the defaults used in your organization. (Note: the sample compiler listings shown in Figure 33 on page 200 and Figure 34 on page 202 are for the CALC sample program which is included in Appendix A, "Sample Programs" on page 219.)

The following options are available:

FIPS | <u>NOFIPS</u>

FIPS produces an information message for any statement that doesn't conform to the Federal Information Processing Standard for BASIC syntax. This option is not affected by the FLAG option.

NOFIPS does not produce the diagnostic warnings.

<u>FLAG (I)</u>

produces all levels of messages (informational, warning, error, or severe).

FLAG (W)

produces warning, error, and severe messages, only.

FLAG (E)

produces error and severe messages, only.

FLAG (S)

produces severe error messages, only.

LIST | <u>NOLIST</u>

LIST produces a listing of the machine instructions in assembler code which are generated for each source statement. Figure 33 on page 200 shows part of a sample LIST output. (This is the result of compiling the CALC program shown in "Sample Program 3: CALC" on page 227.)

The machine instructions are printed directly after the source statement lines to which they apply. The LIST option generates a listing which is separate from the listing generated by the SOURCE option.

You should use this option only when your program has errors you can't find and correct through debugging at the source language level. The generated machine instructions can help your system administrator in locating the error.

NOLIST does not produce a machine instruction listing.

```
1160    if base < 2 or base > 36 then <-------- MAIN PROGRAM source statement
   42         EX     0,10(C)
   46         L      4,150(D)        BASE  )
   4A         C      4,D4(B)               |
   4E         L      6,C8(B)               |
   52         BC     2,0(6,8)              |
   TAG#D0 EQU *                            |
   56         L      4,150(D)        BASE  |
   5A         C      4,5C(B)               > generated instructions
   5E         L      6,C8(B)               |
   62         BC     4,0(6,8)              |
   TAG#CC EQU *                            |
   66         L      F,D8(B)               |
   6A         BC     F,0(F,8)              )
   TAG#C8 EQU *
              .
              .
              .
1610 end <------------------------------------ MAIN PROGRAM source statement
   678        EX     0,10(C)   )
   67C        SR     4,4       > generated instructions
   67E        LA     F,518(0)  |
   682        BALR   E,C       )
   684        DC     X'00001000' )
   688        DC     X'00002000' |
   68C        DC     X'000000B0' |
   690        DC     X'000000AC' |
   694        DC     X'000002B8' > generated instructions
   698        DC     X'00000550' |
   69C        DC     X'00000000' |
   6A0        DC     X'00000000' |
   6A4        DC     X'FF14884A' )
              .
              .
              .
02020 sub factor(factor_answer) <------------- SUBPROGRAM source statement
   0          STM    E,C,C(D)    )
   4          BC     F,C(F)      |
   8          DC     X'00000000' |
   C          LR     8,F         > generated instructions
   E          L      B,8(F)      |
   12         AR     B,F         |
   14         LA     F,494(0)    |
   18         BALR   E,C         )
2025 integer factor_answer, in_base, scan_index, character_class, temp
2030    option collate standard
2040    common input_line$,In_base,scan_index,character$,character_class
2050    call nxtchr
   1A         EX     0,10(C)
   1E         MVC    C0(4,D),30(B)
   24         LA     7,74(B)
   28         LA     1,C0(D)
   2C         BAL    E,4(C)
              .
              .
              .
2240    subexit <----------------------------- SUBPROGRAM source statement
   1D4        EX     0,10(C)   )
   1D8        LA     F,454(0)  > generated instructions
   1DC        BALR   E,C       )
2250 end sub <-------------------------------- SUBPROGRAM source statement
   1DE        EX     0,10(C)   )
   1E2        LA     F,454(0)  > generated instructions
   1E6        BALR   E,C       )
   1E8        DC     X'00001000' )
   1EC        DC     X'00002000' > generated constants
   1EF        DC     X'0000006C' )
              .
              .
              .
```

Figure 33. LIST Compiler Option Output

|

MAP | <u>NOMAP</u>

MAP causes the compiler to produce an allocation map of all the variables and subprograms used by your program. Separate maps are produced for the main program and for each subprogram.

The main program map contains:

1. Common variable allocation in location order with type, number of dimensions (if applicable), and location.

2. Variable allocation in alphabetic order with type, number of dimensions (if applicable), and location.

3. User-defined functions in alphabetic order with their type and associated parameters (if any). The type and order of the required parameters immediately follow the name of the function.

4. An alphabetic list of called subprograms (if any).

5. An alphabetic list of intrinsic functions used (if any).

6. Library routine list.

In addition to the above, the subprogram map contains an alphabetic list of the parameters with their types.

Figure 34 on page 202 shows an allocation map. (This figure is a result of compiling the CALC program shown in "Sample Program 3: CALC" on page 227.)

You would request the MAP option only if you wanted a separate listing of your variables, or if you needed to gain some space by deleting some variables and needed to know where the variables were located.

NOMAP does not produce an allocation map.

```
MAIN PROGRAM

***** COMMON ALLOCATION *****

 NAME                          TYPE             ELEMENTS    R2 OFFSET

 INPUT_LINE$                   CHARACTER*80                     0
 IN_BASE                       INTEGER                         54
 SCAN_INDEX                    INTEGER                         58
 CHARACTER$                    CHARACTER*18                    5C
 CHARACTER_CLASS               INTEGER                         74

 ***** VARIABLE ALLOCATION *****

 NAME                          TYPE             ELEMENTS   R13 OFFSET

 BASE                          INTEGER                        14C
 C                             INTEGER                        148
 C$                            CHARACTER*18                    E8
 EXP_VALUE                     INTEGER                        140
 OUT_BASE                      INTEGER                        13C
 X                             INTEGER                        144

 ***** USER-DEFINED FUNCTIONS (DEFS) *****

 NAME                          TYPE

 BASE_CHECK                    DECIMAL
   BASE                        PARAMETER, INTEGER

 ***** SUBPROGRAMS REFERENCED *****

 EXPR       NXTCHR

 ***** INTRINSIC FUNCTIONS REFERENCED *****

 UPRC$      ABS       REM       CHR$      ORD       INT

 ***** LIBRARY ROUTINES REQUIRED *****

  BLICCAS   BLICCHR   BLIIEND   BLIIIN   BLIIINT   BLIIIOL   BLIITAB...
```

Figure 34 (Part 1 of 2). MAP Compiler Option Output

```
SUBPROGRAM

 ***** SUBPROGRAM PARAMETERS *****

 NAME                                TYPE

 FACTOR_ANSWER                       INTEGER

 ***** COMMON ALLOCATION *****

 NAME                                TYPE              ELEMENTS     R2 OFFSET

 INPUT_LINE$                         CHARACTER*80                        0
 IN_BASE                             INTEGER                            54
 SCAN_INDEX                          INTEGER                            58
 CHARACTER$                          CHARACTER*18                       5C
 CHARACTER_CLASS                     INTEGER                            74


 ***** VARIABLE ALLOCATION *****

 NAME                                TYPE              ELEMENTS    R13 OFFSET

 TEMP                                INTEGER                            E8


 ***** SUBPROGRAMS REFERENCED *****

 EXPR     NXTCHR


 ***** INTRINSIC FUNCTIONS REFERENCED *****

 ORD


 ***** LIBRARY ROUTINES REQUIRED ***

 BLINORD   BLISCTL   BLISINT
```

Figure 34 (Part 2 of 2). MAP Compiler Option Output

OBJECT | NOOBJECT

OBJECT creates an object program file; that is, a compiled program that you can execute.

If you select NOOBJECT, you do not have a program you can execute.

PROFILE | PROFILE (file-spec) | NOPROF

PROFILE causes a default profile to be read. PROFILE (file-spec) causes the specified profile to be read. NOPROF suppresses reading a profile.

The default is to read the default profile if it exists.

These options can only be used in the compiler mode. They may *not* be used on the COMPILE command inside the BASIC environment.

See "Using Profiles" on page 28 and *IBM BASIC Language Reference* for more details on profile. See also your IBM BASIC System Services manual for specifying a file-spec for a profile.

SOURCE | NOSOURCE

SOURCE produces a listing of your program including diagnostic error messages. Most errors are shown in the listing by an underscore of the erroneous statement or character, followed by an error message.

NOSOURCE does not produce a listing of your source program if there are no syntax diagnostics. If there are syntax diagnostics, lines with syntax diagnostics are listed

SPREC | LPREC

SPREC specifies that 6 is the maximum number of significant digits that will be printed by your program when it is processing an unformatted PRINT statement (a PRINT statement without a USING clause). It also specifies that the default precision for real data will be single.

LPREC specifies that 12 is the maximum number of significant digits that will be printed. It also specifies that the default precision for real data will be double.

XREF | NOXREF

XREF produces a cross-reference listing of the variables, line numbers, and line labels in your program.

The listing is in three parts: line numbers in numeric order, line labels in alphabetic order, and variables in alphabetic order.

When a reference to a line is made using a line label, the associated line number is followed by a colon in the cross-reference listing.

Listed with each line number, line label, or variable are the line numbers of the lines that refer to it. References that may change the value in a variable are marked with an asterisk (*). References that define a variable or array (such as in a DIM, COM, or DECIMAL statement) are marked with a slash (/).

Use the cross-reference listing to determine where variables or statements are used. The cross-reference listing is particularly helpful in finding common programming errors. Figure 35 on page 206 shows a cross-reference listing. (This listing was obtained as a result of compiling the CALC program shown in "Sample Program 3: CALC" on page 227.)

NOXREF does not produce a cross-reference listing.

```
MAIN PROGRAM

 LINE    LINE LABEL        REFERENCES    (:=LABEL)  (SORT BY LINE NUMBER)

 1220      TITLE          1210:
 1340                     1350
 1360                     1370


 LINE    LINE LABEL        REFERENCES    (:=LABEL)  (SORT BY LINE NUMBER)

 1220      TITLE          1210:


 NAME                        REFERENCES    (/=DEFINED, *=MODIFIED)

 ABS                       1450
 BASE                      1135/     1140/     1160      1160
 BASE_CHECK                1140/     1150*     1180*     1350      1370
 C                         1135/     1470      1480*     1480      1480     1490
 C$                        1135/     1440*     1490      1490*     1520*    1520       1530
 CHARACTER$                1130/
 CHARACTER_CLASS           1130/     1430      1430
 CHR$                      1490
 EXP_VALUE                 1135/     1410*     1420*     1450      1520
 EXPR                      1420
 IN_BASE                   1130/     1135/     1230*     1250      1340*    1350
 INPUT_LINE$               1130/     1280*     1290      1290*     1300
 INT                       1500
 NXTCHR                    1400
 ORD                       1490
 OUT_BASE                  1135/     1230*     1250      1360*     1370       1470     1500
 REM                       1470
 SCAN_INDEX                1130/     1135/     1390*     1550
 UPRC$                     1290
 X                         1135/     1450*     1470      1500*     1500       1510

SUBPROGRAM

 NAME                        REFERENCES    (/=DEFINED, *=MODIFIED)

 CHARACTER$                1640/
 CHARACTER_CLASS           1635/     1640/     1650      1650      1710
 EXP_ANSWER%               1630/     1635/     1670*     1740      1740*      1770*    1770
 EXPR                      1630/
 IN_BASE                   1635/     1640/
 INPUT_LINE$               1640/
 SCAN_INDEX                1635/     1640/     1660      1660*
 TEMP                      1635/     1700*     1730*     1740      1760*    1770
 TERM                      1670      1730      1760
```

Figure 35. XREF Compiler Option Cross-Reference Listing

Debugging Your Programs

After you have created your program, you should next test it to ensure that it operates successfully. If the program is short and straightforward, it can be easily tested. If it is long and/or complex, the testing can be very time consuming.

In either case, it is extremely critical that your testing or debugging be thorough and complete.

There are several statements and commands you will find useful in locating and correcting errors in your program:

- You can use the BREAK command to specify lines at which you want program execution to stop.
- You can use BREAK statements in your program itself; then if you include the DEBUG ON statement in your program, program execution stops at these statements.
- You can use the RUN STEP and GO STEP command to step through the program, or any portion, one statement at a time.
- You can use immediate statements to inspect and modify any variables when the program is stopped.
- You can use TRACE ON/OFF statements to trace the program flow.
- You can use the HELP command for clarification of error messages.

There are three approaches you can use in testing a program:

- You can use the debugging commands that are available in BASIC. The combination of debugging commands and immediate statements should be all you need if the program is short, straightforward, and/or clearly written.
- You can insert statements into your program to cause it to stop or display information as it is executing. These are the BREAK, PAUSE, and PRINT statements. Normally you use this technique if you are having difficulty locating an error, such as occurs when the flow of the program is not what you expected.
- You can use the TRACE statements to display or print both the line number of the statements executed and the values assigned to any variable by the statements. You should use the TRACE statements if you can't find the problem with approach 1 or 2.

Debugging a program when you're operating inside the BASIC environment is easier and quicker than outside the BASIC environment.

Using Debugging Commands

You can use debugging commands without changing your program. You can enter these commands before entering RUN, or you can enter them anytime your program is stopped. There is a big advantage to this, if you can debug the program without adding or modifying source statements. There is less work for you and less chance for error after the program is debugged, because there are no added source program statements to remove.

Suspending Execution — BREAK Command

Use the BREAK ON command to specify a list of line numbers at which you want program execution to stop. This is useful when you need to inspect a particular area of your program to discover what's wrong during execution. For example:

```
* break on 100, 150, 199, 301, 425
```

stops program execution at each of the above statements. When execution stops (just before the line you specify), you can use immediate statements to examine any variables or to set different values for any variables. To continue execution of the program, you can use the GO command. (See "Resuming Execution — GO Command" on page 209.)

Use the BREAK OFF command to remove breakpoints from your program.

You can remove them all by specifying:

```
* break off
```

or you can remove only those at specified lines:

```
* break off 150, 425
```

removes only those breakpoints at lines 150 and 425, but does not deactivate those at lines 100, 199, and 301.

If you are unsure of the line numbers at which you have set breakpoints, you can specify:

```
* break?
   100      199      301
```

This form of BREAK displays all the line numbers that have breakpoints.

Resuming Execution — GO Command

To resume execution of your program after it has stopped at a breakpoint, or a PAUSE statement, you use the GO command. If you use the GO command by itself, your program will continue execution without stopping until it reaches another breakpoint, or a PAUSE, STOP, or END statement.

STEP Mode — RUN and GO Commands

You can step through your program one statement at a time by issuing the RUN STEP command. For example:

```
* run step
```

starts execution one statement at a time.

When running named source file programs, you can also specify STEP mode:

```
* run aa STEP
```

loads the source program (file) AA into your workspace, syntax checks each line as it is read, performs a global semantic check, and starts stepping through it statement by statement.

You can also enter STEP mode by using the GO STEP command, when the program halts at a breakpoint or a PAUSE statement. Breakpoints remain in effect during the GO STEP mode of operation.

In STEP mode, BASIC halts before each statement (including the first), and displays the line number. You can use immediate statements to display any variables or to set new values in any variable. You then respond by pressing the ENTER key or by entering GO STEP if you want to continue step execution. Use this mode of operation if you are having difficulty finding a particular problem, or if you want to set some variable before continuing the operation.

When you no longer need to execute your program one line at a time, enter GO, and it will resume execution without stopping until it reaches another breakpoint or a PAUSE, STOP, or END statement.

STEP may be specified when executing object programs but does not take effect until a CHAIN to a source program occurs.

Using Debugging Statements

You can insert debugging statements into your program instead of using debugging commands. This changes your program and may require you to change some of the line numbers of your statements. You can do this by issuing a RENUMBER command, or by changing line numbers yourself. If you do need to edit your program in this way, you should get a new source listing so you can more easily follow the operation of your renumbered program.

Activating Debugging — DEBUG Statement

Use the DEBUG ON statement to turn the debugging system on; use the DEBUG OFF statement to turn debugging off. However, these statements activate and deactivate debugging only in the program unit in which they occur.

The DEBUG ON statement must be executed prior to any other debugging statements, such as BREAK or TRACE. Otherwise, the other debugging statements act as null statements. (That is, the BREAK statement doesn't stop the program, and the TRACE statement doesn't display any data.)

Suspending Execution — BREAK statement

You can insert one or more BREAK statements into your program wherever you want it to stop.

During program execution, if a DEBUG ON statement is executed before a BREAK statement, your program stops when it executes the BREAK statement. You can then use immediate statements to examine any variables or to set them to different values.

To resume execution, you can use the GO command. (See "Resuming Execution — GO Command" on page 209.)

Suspending Execution — PAUSE statement

You can use the PAUSE statement instead of the BREAK statement in your programs. The effect is the same, except that the PAUSE statement stops the execution even if DEBUG OFF is in effect. You can also code a message on the PAUSE statement; the message is displayed when the PAUSE statement is executed. Thus, you can identify in descriptive terms where or why each pause is occurring.

For example, if you code the following PAUSE statement in your program:

```
300 pause "no cases selected"
```

Then during execution, when execution is suspended at line 300, the following message appears at the terminal:

```
no cases selected
```

If you do not code a message on the PAUSE statement, BASIC displays the line number of the PAUSE statement. For example:

```
PAUSE AT LINE:     300
```

To resume execution, the terminal user can either press the ENTER key or issue a GO command. (See the GO command.)

Using Immediate Statement

When you're inside the BASIC environment, you can enter a statement from the terminal without a line number. This is called an immediate statement and causes BASIC to execute the statement immediately rather than storing it for later execution.

You can execute an immediate statement anytime BASIC is in the "command mode," signified by an asterisk prompt on your terminal. BASIC is in the command mode before you start executing a program, after a program has executed, or at any point when a program has stopped because of a PAUSE or BREAK statement, a breakpoint, an attention interrupt, or an error.

Not all statements can be executed immediately and those that can may have special semantics or restricted syntax.

You can use immediate statements to:

- Test the validity of an expression
- Test your formatted output
- Inspect or modify program variables while operation of your program is suspended
- Inspect program variables in the main program after the program has been executed

Main program variables continue to exist after the program has completed execution; they can be inspected using immediate statements anytime until the execution of a command or an editing operation changes the program. (Note that subprogram variables cease to exist after the subprogram has exited, and can only be inspected or modified when execution is suspended within the subprogram.)

Using the Immediate PRINT Statement

Immediate PRINT statements allow you to print the contents of variables and arrays in your program. For example:

```
* print alpha
```

prints the contents of ALPHA.

```
* mat print xyz
```

prints the contents of array XYZ.

If you want to include a USING clause in the print statement, you cannot have the clause refer to a line label or line number. The USING clause must be a character expression within the PRINT statement itself. For example:

```
* print using "##.### >####":a,b$
```

prints the values of A and B$ using the IMAGE format enclosed in quotes. (See "Requesting Formatted Output — IMAGE and FORM Statements" on page 100.)

When the program is stopped, you can print any variable in the containing program unit (that is, the main program or the subprogram). For example, assume variable AA is defined in the main program and another variable AA is defined in a subprogram. The PRINT AA immediate statement prints the value of a different variable depending on whether your program is suspended in the main program or in the subprogram.

Using the Immediate LET or MAT Statement

You can assign new values to the variables or arrays and, optionally, redimension arrays in your program or subprogram when your program is suspended. (You might need to do this if the value in a variable is incorrect and you want to set it so you can continue testing the remainder of your program.)

The only variables you can access at a breakpoint are the variables belonging to the program unit that is the currently suspended, main program or subprogram.

```
* let x=10
```

assigns 10 to variable X.

```
* mat a = (10)* a
```

multiplies all the values in array A by 10.

You must use the keyword LET or MAT in any immediate statement if the variable on the left of the assignment is a command keyword. For example:

```
* let list=4
```

Further, you cannot use a function that is defined within your program. For example:

```
* let y = func(x)
```

is illegal if FUNC is defined in your program.

Using the Immediate STOP Statement

The immediate STOP statement, when entered while your program is at a breakpoint, works as a normal STOP statement. It halts execution of your program and closes all files.

Using Immediate Variables

You can use an immediate variable if you want to test some arithmetic statement without using one of your program variables. For example:

```
* option base 1
* integer a
* dim a(20)
```

creates the immediate integer array A with 20 elements, which you can use in calculations instead of one of your program variables.

Use immediate declarations to define the attributes of immediate variables with their dimensions, if applicable. An immediate variable is created according to the immediate declarations and immediate options you specify. An immediate variable does not affect the variables in your program.

If you create an immediate variable during a main program breakpoint or before you run the program, that immediate variable is valid for the main program.

If you create an immediate variable during a subprogram breakpoint, that immediate variable is valid for that subprogram.

If you execute any command or editing operation that changes the program in the workspace, all immediate and program variables are dropped (cease to exist). In addition, if you execute a COMPILE, INITIALIZE, or RUN command, the immediate and program variables are dropped. (However, RUN creates a new generation of program variables.)

You can selectively remove immediate variables with the DROP command. This may be necessary if you want to attach new attributes to an existing variable, because the old attribute must be detached before you can enter a new declaration. For example, after the three immediate statements at the start of this section:

```
* drop a
```

drops the attributes of variable A. It is no longer an integer array. If you want, you can now redefine A to have decimal type, as follows:

```
* decimal a
```

Using the Immediate Typing Statements

Use the immediate DECIMAL, REAL, and INTEGER statements to set the type of an immediate variable.

The immediate DECIMAL, REAL, and INTEGER statements do not "create" variables. They merely record that the specified identifier is assigned a particular type. The immediate variable is created when a subsequent immediate statement (for example, LET) uses the identifier. For example:

```
* integer alpha
        .
        .
* let alpha=20
```

The immediate variable ALPHA is an integer and is "created" with the LET statement.

Using The Immediate DIM Statement

Use the immediate DIM statement to dimension an immediate array and to establish the maximum string length for immediate character variables.

An immediate DIM statement creates an immediate array just as a LET statement creates a simple variable. Therefore, the immediate type declaration must precede the DIM statement. For example:

```
* option base 1
* integer abc
* dim abc(20,10)
```

creates the immediate, 20-by-10, integer array ABC.

TRACE Statement

For simple debugging problems, you can use the BREAK statement or command to set and reset breakpoints and the immediate statements to set or examine any variables.

If, however, the problem is sufficiently complex or the bug remains difficult to find, the number of possibilities may be too large for investigation by anything but the TRACE function. You can use TRACE to display line numbers and variable values over any portion of a running program. In order for TRACE to produce this information, debugging mode must be on. For example:

```
100 debug on
      .
      .
      .
150 trace on
      .
      .
      .
200 trace off
```

turns the trace on at line 150 and off at line 200.

By selectively using TRACE ON and TRACE OFF, in this way, you can bracket the area or areas in your program that need to be traced.

During program tracing, you get the following information:

- If a statement changes the sequence of execution, the line number of the statement and the line number of the next statement to be executed

- If a statement changes the value of a variable or array, the line number of the statement and the new value assigned

With this information, you can get a detailed picture of how execution is proceeding. Figure 36 on page 216 shows a portion of the sample program CALC that includes TRACE ON and TRACE OFF, and a sample of the trace output that results during execution.

The data output by TRACE is displayed on your terminal.

If the areas of your program that are bracketed by TRACE ON and TRACE OFF are large, you can direct the TRACE output to a file for later reference. For example:

```
* trace on to #3
```

sets the TRACE mode on and directs the report to the file indicated by #3. Your program must have opened this file previously, just as any BASIC data file (the file should have OUTPUT, DISPLAY, and SEQUENTIAL attributes).

You can use TRACE reports of this type to find the most obscure program errors.

```
 * list 1870 to 1990
1870 integer temp, character_class, term_answer
1880       temp = 0
1885 debug on
1886 trace on
1890       select character_class
1900          case  = 3                    ! asterisk
1910             call factor(temp)
1920             term_answer = term_answer * temp
1930          case  = 4                    ! slash
1940             call factor(temp)
1950             if temp <> 0 then term_answer = ip(term_answer / temp)
1960          case else
1970             subexit
1980       end select
1985 trace off
1990    loop
 * run
                     MULTIBASE INTEGER CALCULATOR

Input base is  10   Output base is  10
Type expression, or X to change base, or Q to quit.

 8/2*3
TRACE BRANCH FROM LINE 1900 TO LINE  1930
TRACE BRANCH FROM LINE 1940 TO LINE  2020
TRACE BRANCH FROM LINE 2240 TO LINE  1940
TRACE ASSIGNMENT AT LINE 1950 VALUE:  4
TRACE BRANCH FROM LINE 1960 TO LINE  1985
TRACE BRANCH FROM LINE 1910 TO LINE  2020
TRACE BRANCH FROM LINE 2240 TO LINE  1910
TRACE ASSIGNMENT AT LINE 1920 VALUE:  12
TRACE BRANCH FROM LINE 1930 TO LINE  1985
TRACE BRANCH FROM LINE 1900 TO LINE  1930
TRACE BRANCH FROM LINE 1930 TO LINE  1960
Answer is 12

Input base is  10   Output base is  10
Type expression, or X to change base, or Q to quit.

 q
STOP AT LINE 1320.
 *
 _
```

Figure 36. Example of a Program TRACE

Using the Diagnostic HELP Command

When you're inside the BASIC environment, you can respond to any error or exception message by merely typing HELP, or HELP followed by the 8-character message code for the error or exception. BASIC responds by displaying more detailed information concerning the exception, with indications of where to find additional information. Figure 37 on page 217 shows an example of the type of additional information you are given.

```
 * list
90 decimal A
100 input A
110 print using "###.##": A
120 end
 * run
 ? 1234.5
##### BLI-8328W LINE 110. NUMERIC VALUE WILL NOT FIT IN IMAGE SPECIFICATION.
      REPLACED WITH ASTERISKS. PROGRAM EXECUTION CONTINUES.
******
END AT LINE 120.
 * help BLI-8328
```

```
 ***  BASIC HELP  *****  BLIN8328  **********************  PAGE  1 OF  1  ***

BLI-8328W LINE n.  NUMERIC VALUE WILL NOT FIT IN IMAGE SPECIFICATION.  REPLACED
WITH ASTERISKS. PROGRAM EXECUTION CONTINUES.


EXPLANATION: A conversion  specification was  not large  enough to  contain the
numeric value  it attempted to  convert so  the entire specification  was filled
with asterisks.

SYSTEM ACTION: Displays warning message and proceeds.

USER RESPONSE: Change either  the IMAGE specification or the  numeric data item
to fit each other, or ignore if not important.

EXCEPTION CODE: -8328
ON CONDITION: CONV
EXIT CONDITION: CONV

HELP REFERENCES: IMAGE
```

Figure 37. Diagnostic HELP Command Example

The HELP command can also be used to get reference and tutorial information (see "Getting Help With the HELP Command" on page 34).

Appendix A. Sample Programs

Sample Program 1: EDITOR

Substring replacement is a common editing problem. In this program, first a key string is requested from the terminal. Then change commands are accepted and applied to the key string. Each change command must have the form

```
<delimiter><old string><delimiter><new string><delimiter>
```

where <delimiter> may be any character.

The program demonstrates the use of character strings; in particular, several intrinsic character functions and substring qualifiers.

```
 * load editor
38 LINES LOADED.  'EDITOR' IS THE CURRENT PROGRAM NAME.
 * list
100 !  EDITOR
110 !
120 DIM SOURCE$*80, CHANGE$*80 : integer K
130 !
140 DO
150   !
160   ! Get string to be changed.  If 'END', terminate the program.
170   !
180   LINE INPUT 'KEY STRING (OR END)': SOURCE$
190   EXIT IF UPRC$(SOURCE$) = 'END'
200   DO
210     !
220     ! Get change request.  If 'NEXT', get new string to be changed.
230     ! Otherwise, the change request must have the form:
240     ! <delimiter> <old string> <delimiter> <new string> <delimiter>
250     !
260     LINE INPUT 'CHANGES (OR NEXT)': CHANGE$
270     EXIT IF UPRC$(CHANGE$) = 'NEXT'
280     !
290     ! Find position of second occurrence of <delimiter> in CHANGE$.
300     !
310     LET K = POS( CHANGE$(1:LEN(CHANGE$)-1), CHANGE$(1:1), 2)
320     !
330     ! Ensure only three occurrences of <delimiter> and that
340     ! the last character in CHANGE$ is <delimiter>.
350     !
360     IF K<>0 AND POS(CHANGE$,CHANGE$(1:1),K+1)=LEN(CHANGE$) THEN
370       !
380       ! Make the change and display the new string.
390       !
400       SOURCE$=SREP$(SOURCE$,                    &
  &                     1,                          &
  &                     CHANGE$(2:K-1),             &
  &                     CHANGE$(K+1:LEN(CHANGE$)-1))
410       PRINT SOURCE$
420     ELSE
430       PRINT 'ILLEGAL'
440     END IF
450   LOOP
460 LOOP
470 END
 * run
KEY STRING (OR END) now is the time for all gold men
CHANGES (OR NEXT) /gold/good/
now is the time for all good men
CHANGES (OR NEXT) .men.women.
now is the time for all good women
CHANGES (OR NEXT) next
KEY STRING (OR END) when is the course of human events
CHANGES (OR NEXT) /is/in/
when in the course of human events
CHANGES (OR NEXT) next
KEY STRING (OR END) end
END AT LINE 470.
```

Sample Program 2: LOANS

This program computes any one of several parameters associated with a loan, based upon the values of the other parameters.

All calculations are done with decimal arithmetic so "pennies" are not lost. Line labels are used to identify local subroutines (targets of GOSUB statements) and a separate CASE block is used for each of the values to be computed.

```
 * load loans
162 LINES LOADED.  'LOANS' IS THE CURRENT PROGRAM NAME.
 * list
100 REM LOANS
110 REM THIS PROGRAM CALCULATES DIFFERENT LENDING INFORMATION
120 REM BASED ON COMMON LENDING PARAMETERS
130 REM
140 REM  VALUE                    PARAMETERS
150 REM
160 REM  PRINCIPAL           PAYMENT AMOUNT,INTEREST,#PAYMENTS PER YEAR,
170 REM                      # YEARS LOAN IS OUTSTANDING
180 REM  PAYMENT AMOUNT      PRINCIPAL,INTEREST,#PAYMENTS PER YEAR,
190 REM                      # YEARS LOAN IS OUTSTANDING
200 REM  TERM IN YEARS       PRINCIPAL,INTEREST,#PAYMENTS PER YEAR,
210 REM                      PAYMENT AMOUNT
220 REM  BALANCE DUE         PRINCIPAL,PAYMENT AMOUNT,INTEREST,
230 REM                      #PAYMENTS PER YEAR, PAYMENT # (IN YEAR),
240 REM                      YEAR #
250 REM  LAST PAYMENT        PRINCIPAL,PAYMENT AMOUNT,INTEREST,
260 REM                      # PAYMENTS PER YEAR,#YEARS LOAN IS OUTSTANDING
270 REM  INTEREST RATE       PRINCIPAL,PAYMENT AMOUNT,# PAYMENTS PER YEAR,
280 REM                      # YEARS LOAN IS OUTSTANDING
290 REM
300 DO
310    PRINT NEWPAGE
320    PRINT"THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:"
330    PRINT
340    PRINT"    1. PRINCIPAL ON A LOAN"
350    PRINT"    2. REGULAR PAYMENT"
360    PRINT"    3. TERM IN YEARS"
370    PRINT"    4. BALANCE REMAINING AT A CERTAIN PAYMENT"
380    PRINT"    5. LAST PAYMENT AMOUNT"
390    PRINT"    6. ANNUAL INTEREST RATE"
400    PRINT
410    INPUT 'SELECT A NUMBER (0=DONE): ': SELECTION%
420    PRINT
430    SELECT SELECTION%
440       CASE 0 !**************************** FINISHED
450          STOP
460       CASE 1 !**************************** PRINCIPAL
470          GOSUB GET_INTEREST
480          GOSUB GET_PAYMENT
490          GOSUB GET_PAYMENTS_PER_YEAR
500          GOSUB GET_YEARS
510          PRINCIPAL=PAYMENT*PAYMENTS_PER_YEAR*(1-1/(INTEREST/       &
  &                    PAYMENTS_PER_YEAR+1)**(PAYMENTS_PER_YEAR*       &
  &                    YEARS))/INTEREST
520          PRINT TAB(20); 'PRINCIPAL IS ';
530          PRINT USING DOLLARS: PRINCIPAL
540       CASE 2 !**************************** PAYMENT
550          GOSUB GET_PRINCIPAL
560          GOSUB GET_INTEREST
```

```
570        GOSUB GET_PAYMENTS_PER_YEAR
580        GOSUB GET_YEARS
590        PAYMENT=(INTEREST*PRINCIPAL/PAYMENTS_PER_YEAR)/             &
 &                 (1-1/(INTEREST/PAYMENTS_PER_YEAR+1)**               &
 &                 (PAYMENTS_PER_YEAR*YEARS))
600        PRINT TAB(20); 'PAYMENT IS ';
610        PRINT USING DOLLARS: PAYMENT
620     CASE 3 !**************************** NUMBER OF YEARS
630        GOSUB GET_PRINCIPAL
640        GOSUB GET_INTEREST
650        GOSUB GET_PAYMENT
660        GOSUB GET_PAYMENTS_PER_YEAR
670        YEARS=-(LOG(1-(PRINCIPAL*INTEREST)/(PAYMENTS_PER_YEAR*      &
 &               PAYMENT))/(LOG(1+INTEREST/PAYMENTS_PER_YEAR)*         &
 &               PAYMENTS_PER_YEAR))
680        PRINT USING "FORM POS 20,'TERM IS ',PIC(ZZ),' YEARS'": YEARS
690     CASE 4 !**************************** BALANCE REMAINING
700        GOSUB GET_PRINCIPAL
710        GOSUB GET_INTEREST
720        GOSUB GET_PAYMENT
730        GOSUB GET_PAYMENTS_PER_YEAR
740       GET_LASTPAY: PRINT 'ENTER PAYMENT NUMBER (WITHIN YEAR) AFTER ';&
 &         'WHICH TO CALCULATE BALANCE:';
750        INPUT ' ': LASTPAY
760        IF LASTPAY < 0 THEN PRINT 'INVALID RESPONSE': GOTO GET_LASTPAY
770        IF LASTPAY > PAYMENTS_PER_YEAR THEN
780           PRINT 'NUMBER OF PAYMENTS CANNOT BE GREATER THAN ';      &
 &                  'NUMBER PER YEAR '; PAYMENTS_PER_YEAR
790           GOTO GET_LASTPAY
800        END IF
810     GET_LASTYEAR: INPUT 'ENTER YEAR OF LAST PAYMENT: ': LASTYEAR
820       IF LASTYEAR < 0 THEN PRINT 'INVALID RESPONSE': GOTO GET_LASTYEAR
830       IF LASTYEAR > 50 THEN
840          PRINT LASTYEAR;'IS TOO LARGE, MAXIUM IS 50'
850          GOTO GET_LASTYEAR
860       END IF
870        FOR J = 1 TO PAYMENTS_PER_YEAR*(LASTYEAR-1)+LASTPAY
880           I = ROUND(PRINCIPAL*INTEREST/PAYMENTS_PER_YEAR, 2)
890           PRINCIPAL = PRINCIPAL - (PAYMENT - I)
900        NEXT J
910        BALANCE = ROUND(PRINCIPAL, 2)                 ! ROUND TO CENTS
920        IF BALANCE < 0 THEN BALANCE = 0
930        PRINT TAB(20); 'BALANCE DUE ';
940        PRINT USING DOLLARS: BALANCE
950     CASE 5 !**************************** LAST PAYMENT
960        GOSUB GET_PRINCIPAL
970        GOSUB GET_INTEREST
980        GOSUB GET_PAYMENT
990        GOSUB GET_PAYMENTS_PER_YEAR
1000        GOSUB GET_YEARS
1010        FOR J = 1 TO PAYMENTS_PER_YEAR * YEARS
1020           I = ROUND(PRINCIPAL*INTEREST/PAYMENTS_PER_YEAR, 2)
1030           IF PRINCIPAL + I < PAYMENT THEN
1040              LASTPAY = ROUND(PRINCIPAL+I, 2)
1050              GOTO PRINT_LASTPAY
1060           END IF
1070           PRINCIPAL = PRINCIPAL - (PAYMENT - I)
1080        NEXT J
1090        LASTPAY = ROUND(PAYMENT + PRINCIPAL, 2)
1100       PRINT_LASTPAY: PRINT TAB(20); 'LAST PAYMENT IS ';
1110        PRINT USING DOLLARS: LASTPAY
```

```
1120        CASE 6 !**************************** ANNUAL INTEREST RATE
1130           GOSUB GET_PRINCIPAL
1140           GOSUB GET_PAYMENT
1150           GOSUB GET_PAYMENTS_PER_YEAR
1160           GOSUB GET_YEARS
1170           INTEREST=.5
1180           INT2 = 0
1190           DO
1200              R1=(INTEREST*PRINCIPAL/PAYMENTS_PER_YEAR)/(1-1/((INTEREST/   &
   &              PAYMENTS_PER_YEAR + 1)**(PAYMENTS_PER_YEAR * YEARS)))
1210              R1=ROUND(R1, 2)
1220              INT3=ABS(INTEREST-INT2)/2
1230              INT2=INTEREST
1240              EXIT IF R1 = PAYMENT
1250              IF R1 > PAYMENT THEN
1260                 INTEREST = INTEREST - INT3
1270              ELSE
1280                 INTEREST = INTEREST + INT3
1290              END IF
1300           LOOP
1310           PRINT TAB(20);"INTEREST IS ";ROUND(100*INTEREST,3);"%"
1320        CASE ELSE!************************
1330           PRINT 'ILLEGAL CHOICE.  MUST BE BETWEEN 0 AND 6.'
1340     END SELECT
1350     LINE INPUT 'PRESS ENTER TO CONTINUE': T$
1360 LOOP
1370 GET_PRINCIPAL:  INPUT 'ENTER PRINCIPAL: ': PRINCIPAL
1380                 IF PRINCIPAL < 0 THEN
1390                    PRINT 'INVALID RESPONSE'
1400                    GOTO GET_PRINCIPAL
1410                 END IF
1420                 IF PRINCIPAL > 1E6 THEN
1430                    PRINT 'TOO LARGE.  MAXIMUM IS $1,000,000.'
1440                    GOTO GET_PRINCIPAL
1450                 END IF
1460                 RETURN
1470 GET_INTEREST:  INPUT 'ENTER ANNUAL INTEREST RATE: ': INTEREST
1480                IF INTEREST < 0 OR INTEREST >= 100 THEN
1490                   PRINT 'INVALID RESPONSE. RATE MUST BE BETWEEN 0 AND 100%'
1500                   GOTO GET_INTEREST
1510                END IF
1520                INTEREST = INTEREST / 100
1530                RETURN
1540 GET_PAYMENT: INPUT 'ENTER REGULAR PAYMENT AMOUNT: ': PAYMENT
1550              IF PAYMENT < 0 THEN PRINT 'INVALID RESPONSE': GOTO GET_PAYMENT
1560              RETURN
1570 GET_PAYMENTS_PER_YEAR:  INPUT 'ENTER NUMBER OF PAYMENTS PER YEAR: ': &
   &                               PAYMENTS_PER_YEAR
1580                         IF PAYMENTS_PER_YEAR < 1 OR               &
   &                            PAYMENTS_PER_YEAR > 365 THEN
1590                            PRINT 'INVALID RESPONSE.  NUMBER MUST BE ';&
   &                                  'BETWEEN 1 AND 365.'
1600                            GOTO GET_PAYMENTS_PER_YEAR
1610                         END IF
1620                         RETURN
1630 GET_YEARS:  INPUT 'ENTER TERM (LENGTH) OF LOAN IN YEARS: ': YEARS
1640             IF YEARS < 1 OR YEARS > 50 THEN
1650                PRINT 'INVALID RESPONSE.  TERM MUST BE BETWEEN 1 AND 50'
1660                GOTO GET_YEARS
1670             END IF
1680             YEARS = INT(YEARS)
1690             RETURN
1700 DOLLARS: IMAGE : $$,$$$,$$$.##
1710 END
 * run
```

```
THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

    1. PRINCIPAL ON A LOAN
    2. REGULAR PAYMENT
    3. TERM IN YEARS
    4. BALANCE REMAINING AT A CERTAIN PAYMENT
    5. LAST PAYMENT AMOUNT
    6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  1

ENTER ANNUAL INTEREST RATE:  15
ENTER REGULAR PAYMENT AMOUNT:  12644.44
ENTER NUMBER OF PAYMENTS PER YEAR:  12
ENTER TERM (LENGTH) OF LOAN IN YEARS:  30
                PRINCIPAL IS     $999,999.98
PRESS ENTER TO CONTINUE
THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

    1. PRINCIPAL ON A LOAN
    2. REGULAR PAYMENT
    3. TERM IN YEARS
    4. BALANCE REMAINING AT A CERTAIN PAYMENT
    5. LAST PAYMENT AMOUNT
    6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  2

ENTER PRINCIPAL:  1000000.00
ENTER ANNUAL INTEREST RATE:  15
ENTER NUMBER OF PAYMENTS PER YEAR:  12
ENTER TERM (LENGTH) OF LOAN IN YEARS:  30
                PAYMENT IS      $12,644.44
PRESS ENTER TO CONTINUE

THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

    1. PRINCIPAL ON A LOAN
    2. REGULAR PAYMENT
    3. TERM IN YEARS
    4. BALANCE REMAINING AT A CERTAIN PAYMENT
    5. LAST PAYMENT AMOUNT
    6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  3

ENTER PRINCIPAL:  1000000.00
ENTER ANNUAL INTEREST RATE:  15
ENTER REGULAR PAYMENT AMOUNT:  12644.44
ENTER NUMBER OF PAYMENTS PER YEAR:  12
                TERM IS 30 YEARS
PRESS ENTER TO CONTINUE

THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

    1. PRINCIPAL ON A LOAN
    2. REGULAR PAYMENT
    3. TERM IN YEARS
    4. BALANCE REMAINING AT A CERTAIN PAYMENT
    5. LAST PAYMENT AMOUNT
    6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  4
```

```
ENTER PRINCIPAL:  1000000.00
ENTER ANNUAL INTEREST RATE:  15
ENTER REGULAR PAYMENT AMOUNT:  12644.44
ENTER NUMBER OF PAYMENTS PER YEAR:  12
ENTER PAYMENT NUMBER (WITHIN YEAR) AFTER WHICH TO CALCULATE BALANCE:  12
ENTER YEAR OF LAST PAYMENT:  30
                    BALANCE DUE           $.25
PRESS ENTER TO CONTINUE
THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

    1. PRINCIPAL ON A LOAN
    2. REGULAR PAYMENT
    3. TERM IN YEARS
    4. BALANCE REMAINING AT A CERTAIN PAYMENT
    5. LAST PAYMENT AMOUNT
    6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  5

ENTER PRINCIPAL:  1000000.00
ENTER ANNUAL INTEREST RATE:  15
ENTER REGULAR PAYMENT AMOUNT:  12644.44
ENTER NUMBER OF PAYMENTS PER YEAR:  12
ENTER TERM (LENGTH) OF LOAN IN YEARS:  30
                    LAST PAYMENT IS     $12,644.69
PRESS ENTER TO CONTINUE

THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

    1. PRINCIPAL ON A LOAN
    2. REGULAR PAYMENT
    3. TERM IN YEARS
    4. BALANCE REMAINING AT A CERTAIN PAYMENT
    5. LAST PAYMENT AMOUNT
    6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  6

ENTER PRINCIPAL:  1000000.00
ENTER REGULAR PAYMENT AMOUNT:  12644.44
ENTER NUMBER OF PAYMENTS PER YEAR:  12
ENTER TERM (LENGTH) OF LOAN IN YEARS:  30
                    INTEREST IS  15 %
PRESS ENTER TO CONTINUE

THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

    1. PRINCIPAL ON A LOAN
    2. REGULAR PAYMENT
    3. TERM IN YEARS
    4. BALANCE REMAINING AT A CERTAIN PAYMENT
    5. LAST PAYMENT AMOUNT
    6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  7

ILLEGAL CHOICE.  MUST BE BETWEEN 0 AND 6.
PRESS ENTER TO CONTINUE
```

```
THIS PROGRAM CALCULATES ANY OF THE FOLLOWING:

     1. PRINCIPAL ON A LOAN
     2. REGULAR PAYMENT
     3. TERM IN YEARS
     4. BALANCE REMAINING AT A CERTAIN PAYMENT
     5. LAST PAYMENT AMOUNT
     6. ANNUAL INTEREST RATE

SELECT A NUMBER (0=DONE):  0

STOP AT LINE 450.
```

Sample Program 3: CALC

This program uses integer arithmetic extensively. It calculates integer expressions using a base (or radix) of the user's choice. The answer is then printed according to another user-specified base.

Recursive subprogram calls, and COMMON variables, are used.

```
 * load calc
158 LINES LOADED.  'CALC' IS THE CURRENT PROGRAM NAME.
 * list
1000 rem  CALC
1010 rem
1020 rem  This program is a multibase integer calculator.  It evaluates
1030 rem  expressions involving the operations addition (+), subtraction
1040 rem  (-), multiplication (*), and division (/).  The precedence of
1050 rem  operators is the same as for BASIC numeric expressions.
1060 rem  Parentheses can be used to alter the normal precedence of operators.
1070 rem
1080 rem  Both the input base (for expression operands) and the output base
1090 rem  (for printing of answers) can be set independently.  Alphabetic
1100 rem  characters are used for digits greater than 9.
1110 rem
1120 option collate standard
1130 common input_line$*80,In_base,scan_index,character$,character_class
1135 dim c$*80 : integer in_base, scan_index, character_class, basenum,  &
  &                     out_base, exp_value, c, x
1140 def base_check(basenum)
1150    base_check = 0
1160    if basenum < 2 or basenum > 36 then
1170       print 'Illegal base.  Must be between 2 and 36.'
1180       base_check = 1
1190    end if
1200 fnend
1210 print using title:
1220 title: form page, pos 20, 'MULTIBASE INTEGER CALCULATOR', skip 4
1230 in_base,out_base=10
```

```
1240 do
1250    print 'Input base is '; in_base, 'Output base is '; out_base
1260    print 'Type expression, or X to change base, or Q to quit.'
1270    print
1280    line input '': input_line$
1290    input_line$ = uprc$(input_line$)    ! Guarantee upper case
1300    select input_line$
1310       case = 'Q'                       ! Quit
1320          stop
1330       case = 'X'                       ! Change bases
1340          input 'Type desired input base: ': in_base
1350          if base_check(in_base) = 1 then 1340
1360          input 'Type desired output base: ': out_base
1370          if base_check(out_base) = 1 then 1360
1380       case else                        ! Evaluate expression
1390          scan_index = 0
1400          call nxtchr
1410          exp_value = 0
1420          call expr(exp_value)
1430          if character_class = 8 then        ! Check for end of line
1440             c$ = ''
1450             x = abs(exp_value)
1460             do
1470                c = rem(x,out_base)
1480                if c > 9 then c = c + 7
1490                c$ = chr$(c+ord('0')) & c$
1500                x = int(x/out_base)
1510             loop until x = 0
1520             if exp_value < 0 then c$ = '-' & c$
1530             print 'Answer is '; c$
1540          else
1550             print tab(scan_index + 1); '$'
1560             print '***** SYNTAX ERROR *****'
1570          end if
1580    end select
1590    print
1600 loop
1610 end
1620 !*****************************************************************
1630 sub expr(exp_answer)
1635    integer exp_answer, in_base, scan_index, character_class, temp
1640    common input_line$*80,in_base,scan_index,character$,character_class
1650    if not(character_class=1 or character_class=2) then !'+' or '-'
1660       scan_index = scan_index - 1     !back up one character
1670       call term(exp_answer)
1680    end if
1690    do
1700       temp = 0
1710       select character_class
1720          case  = 1                ! plus
1730             call term(temp)
1740             exp_answer = exp_answer + temp
1750          case  = 2                ! minus
1760             call term(temp)
1770             exp_answer = exp_answer - temp
1780          case else
1790             subexit
1800       end select
1810    loop
1820 end sub
```

```
1830 !******************************************************************
1840 sub term(term_answer)
1845    integer term_answer, in_base, scan_index, character_class, temp
1850    common input_line$*80,In_base,scan_index,character$,character_class
1860    call factor(term_answer)
1870    do
1880       temp = 0
1890       select character_class
1900          case  = 3                ! asterisk
1910             call factor(temp)
1920             term_answer = term_answer * temp
1930          case  = 4                ! slash
1940             call factor(temp)
1950             if temp <> 0 then term_answer = ip(term_answer / temp)
1960          case else
1970             subexit
1980       end select
1990    loop
2000 end sub
2010 !*************************************************************
2020 sub factor(factor_answer)
2025    integer factor_answer, in_base, scan_index, character_class, tem
2030    option collate standard
2040    common input_line$*80,In_base,scan_index,character$,character_class
2050    call nxtchr
2060    if character_class = 5 then               ! left paren
2070       call nxtchr
2080       call expr(factor_answer)
2090       if character_class = 6 then
2100          call nxtchr                   ! right paren - continue scanning
2110       else
2120          character_class = 7         ! not right paren - illegal char
2130       end if
2140       subexit
2150    end if
2160    if character_class <> 0 then character_class = 7
2170    do while character_class = 0                ! alphanumeric
2180       temp = ord(character$) - ord('0')
2190       if temp > 9 then temp = temp - 7  ! letter
2200       exit if temp >= in_base
2210       factor_answer = factor_answer * in_base + temp
2220       call nxtchr
2230    loop
2240    subexit
2250 end sub
2260 !******************************************************
2270 sub nxtchr
2280    rem  this routine scans the next character in input_line$.
2290    rem  character$ is reset to contain the next character.
2300    rem  character_class is set to:  0 = alphanumeric
2310    rem                              1 = +
2320    rem                              2 = -
2330    rem                              3 = *
2340    rem                              4 = /
2350    rem                              5 = (
2360    rem                              6 = )
2370    rem                              7 = illegal character
2380    rem                              8 = end of line
2390    option collate standard
2395    integer character_class, in_base, scan_index
2400    common input_line$*80,In_base,scan_index,character$,character_class
2410    do
2420       scan_index = scan_index + 1
2430       if scan_index > len(input_line$) then
2440          character_class = 8              ! eol
2450          subexit
```

```
2460        end if
2470        character$ = input_line$(scan_index : scan_index)
2480     loop until character$ <> ' '
2490     character_class = pos( '+-*/()', character$)
2500     if character_class <> 0 then subexit
2510     if (character$ >= '0' and character$ <= '9') or &
  &        (character$ >= 'A' and character$ <= 'Z') then
2520        character_class = 0                   ! alphanumeric
2530     else
2540        character_class = 7                   ! illegal character
2550     end if
2560     subexit
2570 end sub
 * run
                    MULTIBASE INTEGER CALCULATOR

Input base is  10   Output base is  10
Type expression, or X to change base, or Q to quit.

  ((1+2)*3+4*5/6)-7
Answer is 5

Input base is  10   Output base is  10
Type expression, or X to change base, or Q to quit.

  x
Type desired input base:  16
Type desired output base:  10

Input base is  16   Output base is  10
Type expression, or X to change base, or Q to quit.

  7FFFFFFF
Answer is 2147483647
Input base is  16   Output base is  10
Type expression, or X to change base, or Q to quit.

  x
Type desired input base:  16
Type desired output base:  16

Input base is  16   Output base is  16
Type expression, or X to change base, or Q to quit.

  3 + 1fffffff * 4
Answer is 7FFFFFFF

Input base is  16   Output base is  16
Type expression, or X to change base, or Q to quit.

  x
Type desired input base:  16
Type desired output base:  8

Input base is  16   Output base is  8
Type expression, or X to change base, or Q to quit.

  3 + 1fffffff * 4
Answer is 17777777777

Input base is  16   Output base is  8
Type expression, or X to change base, or Q to quit.
```

```
  x
Type desired input base:  2
Type desired output base:  36

Input base is  2   Output base is  36
Type expression, or X to change base, or Q to quit.

  100100001100110011101 * 10000 + 100
Answer is BASIC

Input base is  2   Output base is  36
Type expression, or X to change base, or Q to quit.

  10 * 2 + 1
       $
***** SYNTAX ERROR *****

Input base is  2   Output base is  36
Type expression, or X to change base, or Q to quit.

  q
STOP AT LINE 1320.
```

Sample Program 4: CUST

This program creates, changes, displays, or deletes records in a keyed file. Each record has a 15-byte key based on a unique customer number, and contains the following information: last name, first name, middle initial, street number, direction, street name, apartment number, city, state, and address.

Prior to running this program and external to BASIC, a VSAM file must be defined. See your IBM BASIC System Services manual for details on how to do this.

```
 * load CUST
141 LINES LOADED.  'CUST' IS THE WORKSPACE NAME.
 * list
1000 rem CUST
1010 rem This program creates,adds,changes a file containing:
1020 rem          customer number
1030 rem          last name,first name,initial
1040 rem          street address (including direction & apt#)
1050 rem          city,state,address code
1060 dim operation$*1                       !operation
1070 dim cnum$*15                           !customer number
1080 dim last$*15,first$*12,mid$*1          !customer name
1090 dim st_no$*7,dir$*2,st_name$*15,apt$*4 !customer street address
1100 dim city$*15,state$*2,addrcode$*6
1110 integer added,changed,deleted          !# records added,changed,deleted
1120 on soflow goto too_long   !Enable exception handler for string overflow
1130 print
1140 input "ENTER FILENAME:": fn$
1150 open #1:fn$,outin,keyed,native
1160 do
1170    start1:  print
1180             print tab(13);"A(DD)"
1190             print tab(13);"D(ELETE)"
1200             print tab(13);"I(NQUIRE)"
1210             print tab(13);"C(HANGE)"
1220             print tab(13);"S(TOP)"
1230             input   "SELECT ONE: ":operation$
1240    select uprc$(operation$)
1250       case "A"
1260          rem  *******************************************************
1270          rem  *   ADD a new record                                   *
1280          rem  *******************************************************
1290          do
1300             print
1310             print "ADD FUNCTION"
1320             gosub get_cnum
1330             exit if cnum$ = "DONE"
1340             gosub get_new_data
1350             write #1,using record_format:cnum$,last$,first$,mid$,&
   &             st_no$,dir$,st_name$,apt$,city$,state$,addrcode$ dupkey dup
1360             print "*** RECORD ADDED ***"
1370             added = added +1             !number of records added
1380             goto add_end
1390             dup: read #1, using record_format,key=cnum$: cnum$,last$,&
   &                  first$,mid$,st_no$,dir$,st_name$,apt$,city$,state$,addrcod
1400                  print "CUSTOMER # ALREADY ASSIGNED TO:";
1410                  print last$;", ";first$;" ";mid$
1420          add_end: loop
```

```
1430       case "D"
1440          rem  ************************************************************
1450          rem  *  DELETE a record                                         *
1460          rem  ************************************************************
1470          do
1480             print
1490             print "DELETE FUNCTION"
1500             gosub display_record
1510             exit if cnum$ = "DONE"
1520             delete #1, key=cnum$:
1530             print "*** RECORD DELETED ***"
1540             deleted = deleted + 1   !Number of records deleted
1550          loop
1560       case "I"
1570          rem  ************************************************************
1580          rem  *  INQUIRE about a record                                  *
1590          rem  ************************************************************
1600          do
1610             print
1620             print "INQUIRE FUNCTION"
1630             gosub display_record
1640          loop until cnum$ = "DONE"
1650       case "C"
1660          rem  ************************************************************
1670          rem  *  CHANGE a record                                         *
1680          rem  ************************************************************
1690          do
1700             print
1710             print "CHANGE FUNCTION"
1720             gosub display_record
1730             exit if uprc$(cnum$)="DONE"
1740             gosub get_new_data
1750             rewrite #1,using record_format,key=cnum$: CNUM$,LAST$,&
   &              first$,mid$,st_no$,dir$,st_name$,apt$,city$,state$,addrcode$
1760             print "*** RECORD CHANGED ***"
1770             changed = changed + 1   !Number of records changed
1780          loop
1790       case "S"
1800          rem  ************************************************************
1810          rem  *   STOP                                                   *
1820          rem  ************************************************************
1830          print
1840          if changed <> 0 then print "RECORDS CHANGED: "; changed
1850          if added <> 0 then print "RECORDS ADDED: "; added
1860          if deleted <> 0 then print "RECORDS DELETED: "; deleted
1870          close #1:
1880          stop
1890       case else              !Illegal operation character
1900          print "*** ILLEGAL RESPONSE ***"
1910    end select
1920 loop
1930 rem  ****************************************************************
1940 rem  *  Subroutine to:  1. Get a customer number from the terminal. *
1950 rem  *                  2. Read the record corresponding to the     *
1960 rem  *                     customer number and display its contents.*
1970 rem  ****************************************************************
1980 display_record:  gosub get_cnum     !Get the customer number
1990                  if cnum$ = "DONE" then return
2000                  read #1, using record_format,key=cnum$:cnum$,last$,&
   &                   first$,mid$,st_no$,dir$,st_name$,apt$,city$,state$,&
   &                   addrcode$  nokey cannot_find
2010                  print last$;', ';first$;' ';mid$
2020                  print st_no$;' ';
2030                  if dir$<>'' then print dir$;' ';
2040                  print st_name$;
2050                  if apt$<>'' then print ' apt. ';
```

```
2060                    print apt$
2070                    print city$;', ';state$;' ';addrcode$
2080                    return
2090      cannot_find: print "*** CUSTOMER NOT ON FILE ***"
2100                    goto display_record
2110 rem  ***************************************************************
2120 rem  *  Subroutine to get customer number from user at the terminal. *
2130 rem  ***************************************************************
2140 get_cnum:  input "ENTER CUSTOMER NUMBER (OR 'DONE'): ": cnum$
2150            cnum$ = uprc$(cnum$)   !Convert to upper case
2160            if cnum$ = "DONE" then return
2170            cnum$ = rpad$(cnum$,15)  !Pad to 15 characters
2180            return
2190 rem  **************************************************************
2200 rem  *  Subroutine to get the data for a new record from the user.  *
2210 rem  **************************************************************
2220 get_new_data:  print "ENTER DATA FOR NEW RECORD:"
2230                input "   LAST NAME: ": last$
2240                input "   FIRST NAME: ": first$
2250                input "   MIDDLE INITIAL: ": mid$
2260                input "   STREET NUMBER: ": st_no$
2270                input "   DIRECTION: ": dir$
2280                input "   STREET NAME: ": st_name$
2290                input "   APT NUMBER: ": apt$
2300                input "   CITY: ": city$
2310                input "   STATE: ": state$
2320                input "   CODE: ": addrcode$
2330                return
2340 rem  **************************************************************
2350 rem  *  Exception handler for string overflow                       *
2360 rem  **************************************************************
2370 too_long: print "*** STRING IS TOO LONG ***"
2380            retry
2390 record_format: form c15,v15,v12,v1,v7,v2,v15,v4,v15,v2,v6
2400 end
 * run

ENTER FILENAME: customr

            A(DD)
            D(ELETE)
            I(NQUIRE)
            C(HANGE)
            S(TOP)
SELECT ONE:  a
ADD FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  123
ENTER DATA FOR NEW RECORD:
   LAST NAME:  jones
   FIRST NAME:  john
   MIDDLE INITIAL:  p
   STREET NUMBER:  1000
   DIRECTION:
   STREET NAME:  14th
   APT NUMBER:
   CITY:  new york
   STATE:  ny
   CODE:  10022
*** RECORD ADDED ***
```

```
ADD FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  done

            A(DD)
            D(ELETE)
            I(NQUIRE)
            C(HANGE)
            S(TOP)
SELECT ONE:  i
INQUIRE FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  123
jones, john p
1000 14th
new york, ny 10022
INQUIRE FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):

            A(DD)
            D(ELETE)
            I(NQUIRE)
            C(HANGE)
            S(TOP)
SELECT ONE:  c

CHANGE FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  123
jones, john p
1000 14th
new york, ny 10022
ENTER DATA FOR NEW RECORD:
   LAST NAME:  jones
   FIRST NAME:  john
   MIDDLE INITIAL:  p
   STREET NUMBER:  900
   DIRECTION:  sw
   STREET NAME:  14th
   APT NUMBER:  ab
   CITY:  new york
   STATE:  ny
   CODE:  10027
*** RECORD CHANGED ***

CHANGE FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  done

            A(DD)
            D(ELETE)
            I(NQUIRE)
            C(HANGE)
            S(TOP)
SELECT ONE:  d

DELETE FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  123
jones, john p
900 sw 14th apt. ab
new york, ny 10027
*** RECORD DELETED ***
```

```
DELETE FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  done

            A(DD)
            D(ELETE)
            I(NQUIRE)
            C(HANGE)
            S(TOP)
SELECT ONE:  i

INQUIRE FUNCTION
ENTER CUSTOMER NUMBER (OR 'DONE'):  123
*** CUSTOMER NOT ON FILE ***
ENTER CUSTOMER NUMBER (OR 'DONE'):  done

            A(DD)
            D(ELETE)
            I(NQUIRE)
            C(HANGE)
            S(TOP)
SELECT ONE:  s

RECORDS CHANGED:  1
RECORDS ADDED:  1
RECORDS DELETED:  1
STOP AT LINE 1880.
```

Appendix B. Sample Program for Calling the GDDM Interactive Chart Utility

This section contains sample programs that show how to invoke the Graphical Data Display Manager's (GDDM) Interactive Chart Utility (ICU). The ICU is a general purpose graphing program that is included with GDDM's Presentation Graphics Feature (PGF). It allows you to interactively construct and format charts and uses a series of menu panels to request both data and format information. With the ICU, you can specify and change the graph types (for example, line, bar, pie, tower), legends, headings, and other formatting information. Once a chart has been created, you can have the ICU save the data and format definition to disk files, so that they can be recalled for later use.

From a BASIC program, you can invoke the ICU by specifying the "CHART" function argument with the CALL GDDM statement. You also pass arguments for data arrays and formatting information. Once the ICU is invoked, your terminal is controlled by GDDM. You communicate with the ICU via menu dialog and program function keys. When you exit the ICU, control is transferred back to the BASIC application.

The ICU can be a very powerful tool when used in combination with a BASIC application. For example, a BASIC program can obtain historical data from files, analyze the data statistically, compute a trend line, and then invoke the ICU to plot the results. Although the ICU is very powerful and easy to use, its invocation requires a number of specialized arguments. Its use can be simplified by writing a generalized BASIC subprogram to provide suitable defaults and appropriate argument formats. An example of such a subprogram (named EZICU) is included at the end of this section. This subprogram uses fewer and simpler arguments than those required by an equivalent CALL GDDM statement.

Using EZICU

EZICU causes your data to be presented using ICU defaults of a line graph, auto-scaled axes, legend centered in the right margin, and a one line heading centered at the top. EZICU also determines if the data is free or tied, and determines the number of elements per group and component.

Before you can use the EZICU subprogram, you must

- Have access to the GDDM program product with the Presentation Graphics Feature (PGF) and a graphics display terminal supported by GDDM.
- Construct a main application program with the appropriate data arrays, variables, and the CALL statement for the EZICU subprogram (see example below).

- Code the EZICU subprogram (or equivalent) as listed later in this section. The subprogram can be included in your BASIC workspace, or else compiled separately.

The format of the CALL to EZICU is:

```
CALL EZICU(x(),y(),keys$(),labels$(),heading$)
```

where:

x
: is a one dimensional, real single array containing the x axis data points for "free data" graphs. (If you are graphing "tied data", this array should contain all zeros.)

y
: is a one dimensional, real single array containing the y axis data points (usually the main data to be graphed). The number of elements in y must match those in x. For free data, the values of y positionally correspond to the values in x. For tied data, the data points should be ordered by groups within components. For example, the first point represents the first group of the first component, the second point is the second group of the first component, and so forth.

keys$
: is a one dimensional character array containing the key legend text for each group. The first element is the text for the first group. *keys$* must be dimensioned as an array, even if contains zero elements.

labels$
: is a one dimensional character array containing the x axis label text for each component. The first element is the text for the first component. *labels$* must be dimensioned as an array, even if contains zero elements.

heading$
: is a character variable or expression containing the heading text for the graph.

The following is an example of an application program with a call to the EZICU subprogram:

```
100  option base 0
110  dim keys$(3),labels$(6),heading$*132,x(27),y(27)
120  integer: real single x,y
130  data 10,20,30,40,50,60,70,20,30,40,50,60,70,10,30
140  data 40,50,60,70,10,20,40,50,60,70,10,20,30
150  mat read y
160  data "1983","1984","1985","1986"
170  mat read keys$
180  data "APPLES","ORANGES","KIWIS","LEMONS","CHERRIES"
190  data "GUAVAS","GRAPES"
200  mat read labels$
210  heading$="Sales of Various Fruits, by Year"
220  call EZICU(x(),y(),keys$(),labels$(),heading$)
230  end
```

Notes:

- Line 110: All arguments to EZICU are explicitly dimensioned.
- Line 120: The x and y data arrays are of the real single data type.
- Lines 130-150: The y array contains 28 data points. Based on the *labels$* and *keys$* arrays, dimensioned in line 110, this represents 4 groups of 7 components.
- Lines 160-170: The *keys$* array contains 4 elements (1983, 1984, and so forth), representing the 4 data groups. These will appear as a legend in the graph's right margin.
- Lines 180-200: The *labels$* array consists of 7 elements (APPLES, ORANGES, etc.) representing the 7 data components. These will appear equally spaced along the x axis of the graph.
- Line 210: The *heading$* variable contains the title of the graph ("Sales of Various Fruits, by Year"), which will appear centered at the top of the screen.
- Line 220: Since the x array is not assigned in the program, it contains all zeros (initial value). This signals EZICU to use tied data, rather than free data.

```
300 sub EZICU (x(),y(),keys$(),labels$(),heading$)
310  dim key$*400,label$*400,a(18),b(3),c$*40,d(50)
320  option base 0 : integer : real single b,x,y
330  ! Set defaults for elements of ICU Arrays
340  LEVEL   =1   : DISPLY   =2 : HELP     =1 : ISOLATE =0 : BINDING =0
350  COMPS   =0   : ELEMENTS=0 : KEYL     =0 : LABELL  =0 : HEADINGL=0
360  PRTDEP  =0   : PRTWID   =0 : PRTCOPY=1 : PRTHEAD =0 : PRTUNIT =0
370  DRYTYPE=0   : DRYTYPEQ=0 : EXPLVL  =0 : PRTVOFF=-1 : PRTHOFF =-1
380  !   FORMNAME     DATANAME     PRTNAME      DRYNAME      DRYLIB
390  c$='*       ' & '*       ' & '*       ' & '        ' & '        '
400  if dot(x,x)=0 then                      ! All elements of x=0?
410    BINDING =0                            ! Tied data
420    for n=0 to size(x)-1                  ! Initialize x with
430      x(n)=n                              ! incrementing numbers
440    next n
450  else
460    BINDING =1                            ! Free data
470    s=(size(x)-2)                         ! Determine number of
480    e=-1 : g=1 : hold=0                   ! elements in each group
490    for f=0 to s
500        if x(f) <> x(f+1) then            ! If x values not equal,
510          d(e+1)=g : e=e+1                ! store elements/group
520          if d(e) > hold then hold=d(e)! Store highest value group
530          g=0 : 1st_one=x(f+1)
540          if f=s then d(e+1)=1
550        end if
560        g=g+1
570    next f
580  end if
590  ! Convert keys$() array to string with fixed length/element
600  call chrtopt (keys$(),numkey ,keyl ,key$)
610  if binding=1 then COMPS=1st_one else COMPS=numkey
620  if COMPS=0 then COMPS=1                 ! If no keys, use 1 group
630  ! Convert labels$() array to string with fixed length/element
640  call chrtopt (labels$(),labnum ,labell ,label$)
650  if BINDING=1 then ELEMENTS=hold else ELEMENTS=labnum
660  if ELEMENTS=0  then ELEMENTS=size(x)
670  If labell <>0 then ELEMENTS=-ELEMENTS ! If Labels, then elements shown
680  headingl =len(heading$)                 ! Length of heading
690  ! Assign values to arrays and call the ICU
700  a(0) =LEVEL    : a(1) =DISPLY    : a(2) =HELP     : a(3) =ISOLATE
710  a(4) =BINDING  : a(5) =COMPS     : a(6) =ELEMENTS: a(7) =KEYL
720  a(8) =LABELL   : a(9) =HEADINGL : a(10)=PRTDEP   : a(11)=PRTWID
730  a(12)=PRTCOPY  : a(13)=PRTHEAD  : a(14)=PRTUNIT  : a(16)=DRYTYPE
740  a(17)=DRYTYPEQ : a(18)=EXPLVL
750  b(0) =PRTDEP   : b(1) =PRTWID    : b(2) =PRTVOFF : b(3) =PRTHOFF
760  call gddm ("CHART",a(),b(),c$,d(),x(),y(),key$,label$,heading$)
770 end sub
780 sub chrtopt(items$(),i ,length ,item$)
790  ! General purpose subprogram to convert items$() input array to
800  ! linear string item$ with fixed position for each element.
810  ! Number of elements (i) and max element length also returned.
820  option base 0 : integer : length = 0
830  for i=0 to udim(items$,1)        ! Find max items$() elements
840     if len(items$(i)) > length  then length = len(items$(i))
850  next i
860  if length=0 then i=0: subexit  ! Escape if all elements empty
870  for j =0 to i-1                  ! Pad each element with blanks
880    item$=item$ & rpad$(items$(j),length )
890  next j
900 end sub
```

Appendix C. SQL Examples—Advanced Techniques

This appendix illustrates some of the more complex and sophisticated ways in which you can use BASIC to obtain data from a relational data base.

Example: Retrieval with Multiple Concurrent Selections

This program prints an alphabetic directory of department names and employees in each department. In addition, an asterisk is printed next to the names of department managers.

Two concurrent selections are used, one to retrieve rows from the Q.ORG table, the other to retrieve from Q.STAFF (see Appendix D, "The SQL Sample Tables" on page 249). The program operates as follows:

First, selection is made against Q.ORG, with the department-number, department-name, and manager-id columns being returned. An ORDER BY clause causes the rows to be retrieved in department-name sequence.

For each row returned by this selection, the department name is printed and a second selection is made against the Q.STAFF table. This selection returns the ID and NAME columns for the department number returned by the first selection. The program repeatedly selects and prints the name of each employee found in the department.

When the second selection reaches the end of table, it is automatically closed. Then, another row is returned from the first selection, and the process is repeated.

When the first selection reaches end of table, the program terminates.

```
100 REM: Program that uses two concurrent selections
110 OPTION BASE 1: INTEGER SQLRC
120 DIM sqlrc(20), stmt$*1000, stmt2$*1000, sqlmsg$*256
130 PRINT NEWPAGE
140 PRINT "Department Directory from Q.STAFF Table"
150 PRINT "   (asterisk denotes department manager)"
160 PRINT
170 PRINT "Dept Name"; TAB(16); "Employee Name"
180 PRINT
190 stmt$ = "SELECT DEPTNUMB, DEPTNAME, MANAGER "
200 stmt$ = stmt$ & "FROM Q.ORG ORDER BY DEPTNAME"
210 DO
220   CALL SQL (stmt$, sqlrc(), sqlmsg$, dno, dnm$, mgr)
230   EXIT IF SQLRC(1) GT 4
240   PRINT dnm$
250   stmt2$ = "SELECT ID, NAME FROM Q.STAFF "
260   stmt2$ = stmt2$ & "WHERE DEPT=" & STR$(dno) & " ORDER BY NAME"
270   DO
280     CALL SQL (stmt2$, sqlrc(), sqlmsg$, id, name$)
290     IF  sqlrc(2) EQ 100 THEN PRINT
300     EXIT IF sqlrc(1) GT 4
310     IF id = mgr THEN flag$ = "* " ELSE flag$ = "  "
320     PRINT TAB(16); flag$; name$
330   LOOP
340   EXIT IF sqlrc(1) GT 4 AND sqlrc(2) NE 100
350 LOOP
360 IF sqlrc(2) = 100 THEN
370   PRINT "*** End of Directory ***"
380 ELSE
390   PRINT "*** Data Base or SQL Statement Error:"
400   PRINT "       SQL Statement Text:  ";STMT$
410   PRINT "    BASIC/SQL Return Code:  ";SQLRC(2)
420   PRINT "    Data Base Return Code:  ";SQLRC(8)
430   PRINT "    Error Message Text: ";SQLMSG$
440 END IF
450 END
```

Notes:

- Line 260: Notice the use of BASIC's STR$ function to convert the numeric value in the department number variable, DNO, to character. (Remember, SQL statements must be character strings.)

- See "Example: Retrieval with User-Specified WHERE Clause" on page 243 for an example of the use of index values with SELECT queries. Generally speaking, using the index value technique is a preferable alternative to "hard coding" the value of DNO directly into the SQL statement.

Example: Retrieval with User-Specified WHERE Clause

In the previous retrieval examples, the SELECT statement is used to retrieve data from tables using a fixed SQL statement. But what about the case where the selection criterion is variable, based on user input? Also, how do you signal BASIC to terminate a selection without affecting other selections or overall commit status?

The following example illustrates the above points. The program uses BASIC's PRINT FIELDS and INPUT FIELDS statements to operate in "full-screen mode." The user is prompted for an employee name and, once it is entered, the corresponding data from the Q.STAFF table is retrieved. This requires that the SQL SELECT statement specify a WHERE clause containing the employee name. After the employee information is displayed, the user is prompted again, and the process is repeated. Because we never retrieve beyond the first row (we assume, here for the sake of simplicity, that no two employees have the same name), we never obtain an end of table condition for a selection. Hence, each selection must be "closed" or deactivated to avoid accumulating unnecessary active SQL requests to the data base.

```
100 REM: SELECT with user specified WHERE and
105 REM: selection termination
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256
130 DIM hdgs_def$(10), hdgs_text$(10)*40, col_def$(7)
140 !-- initialize heading field definitions -------------------
150 DATA " 1,1,v60,n", " 4,33,v3,n",  " 6,31, v5,n"
160 DATA " 8,31,v5,n", "10,32,v4,n",  "12,30, v6,n"
170 DATA "14,29,v7,n", "16,31,v5,n",  "22,12,v25,h", "24,1,v60,h"
180 MAT READ hdgs_def$
190 !-- initialize heading fields ------------------------------
200 DATA "Employee Data from Q.STAFF Table"
210 DATA "ID:", "Name:","Dept:", "Job:", "Years:"
215 DATA "Salary:", "Comm:"
220 DATA "Enter Employee Name ===> ", "PF3 to EXIT"
230 MAT READ hdgs_text$
240 !-- initialize field definitions for column data -----------
250 DATA " 4,37,n3,h", " 6,37,v9,  h", " 8,37,n2,  h", "10,37,v5,h"
260 DATA "12,37,n2,h", "14,37,n9.2,h", "16,37,n9.2,h"
270 MAT READ col_def$
280 !-- field definitions for user input and message -----------
290 in_fld$  = "22,37,v9" : msg_fld$ = "23,1,v50,h"
300 ON SKEY GOTO test_pfk                ! trap for function key 3
310 stmt$ = "SETOPT NULL 0"              ! translate nulls to zero
320 CALL SQL (stmt$, sqlrc(), sqlmsg$)
330 IF sqlrc(1) NE 0 THEN GOTO sql_error
340 PRINT NEWPAGE: PRINT FIELDS MAT hdgs_def$: MAT hdgs_text$
350 DO
360   INPUT FIELDS in_fld$: n$
370   PRINT FIELDS msg_fld$: "Stand by: Retrieving from data base"
380   stmt$ = "SELECT ID, DEPT, JOB, YEARS, SALARY, COMM "
390   stmt$ = stmt$ & "FROM Q.STAFF WHERE NAME = :7"
400   CALL SQL (stmt$, sqlrc(), sqlmsg$, i1, d, j$, y, s, c, n$)
410   EXIT IF sqlrc(1) GT 4 AND sqlrc(2) NE 100
420   PRINT FIELDS in_fld$: " "        ! Blank out input area
430   IF sqlrc(2) EQ 100 THEN
440     PRINT FIELDS msg_fld$: "*** No employee named " & n$
450   ELSE
460     PRINT FIELDS msg_fld$ : " "  ! Blank out message area
470     PRINT FIELDS MAT col_def$: i1, n$, d, j$, y, s, c
480     sqlrc(3) = -1   ! neg. value will deactivate SQL request
490     CALL SQL (stmt$, sqlrc(), sqlmsg$)
500   END IF
510 LOOP UNTIL (sqlrc(1) GT 4 AND sqlrc(2) NE 100)
520 sql_error:                                ! Fatal SQL errors
530 IF sqlrc(1) GT 4 AND sqlrc(2) NE 100 THEN
540    PRINT "*** Data Base or SQL Statement Error:"
550    PRINT "       SQL Statement Text:  ";stmt$
560    PRINT "    BASIC/SQL Return Code:  ";sqlrc(2)
570    PRINT "    Data Base Return Code:  ";sqlrc(8)
580    PRINT "    Error Message Text: ";sqlmsg$
590 END IF
600 STOP
610 test_pfk: IF KEYNUM = 3 THEN PRINT NEWPAGE ELSE CONTINUE
620 END
```

Notes:

- Lines 140-290 appear complicated, but are actually various initializations for the full-screen I/O parameters. The main program logic doesn't begin until statement 300.

- Line 300: The user exits from the program by pressing program function key 3, when prompted for input. This statement "traps" PF-Key interrupts and branches to TEST__PFK (line 610) to determine if PF3 was pressed.

- Lines 310-330: Since we may encounter nulls in the COMM column, we choose to translate them to zeros.

- Line 370: Because data base operations sometimes involve response-time delays, an informational message is displayed so the user doesn't get too anxious.

- Line 400: Notice that the CALL SQL parameter list contains N$ as the 7th host variable. This matches the :7 index value in the SQL statement. Thus, the first 6 elements of the value list (I, D, ... C) will contain the results of the selection for the ID, DEPT, ... COMM columns. Whenever index values are used with SELECT statements, it is recommended that their corresponding value list variables be located *after* the value list variables used to receive the query results.

  Using index variables in this manner is often more efficient than catenating the WHERE clause arguments, as was done in the example "Retrieval With Multiple Concurrent Selections" on page 241. In addition to being more efficient, the index value technique usually results in "cleaner" and more explicit BASIC programs. For example, if we were to modify this example to catenate N$ into STMT$, we would have to surround the N$ variable with single quotes, in order to properly represent its value as a character constant in the SQL statement.

- Lines 480-490: By assigning a -1 to SQLRC(3) and then calling SQL with the same STMT$ value, we instruct BASIC to terminate the selection (so that this SQL statement is no longer active). Usually, the SQL return codes are only examined or tested by the program; however, SQLRC(3) is a special case because it can also be assigned before the CALL SQL, and used as a "close cursor" signal. When used in this manner, it's important to remember that only negative values will work. (Positive values are used to specify the number of rows to be returned when performing multiple row retrievals, provided the positive value is less than the dimension of the receiving arrays.) Because you're not actually retrieving any data, you don't have to pass any variables in the CALL SQL value list. If you do, they will be ignored.

  Note: The first use of a SELECT statement (that is, the first CALL SQL specifying the SELECT) causes a cursor to be opened in the data base system. This cursor stays open until it is closed implicitly by a COMMIT WORK, ROLLBACK WORK, or end-of-table condition (SQL Return Code 100), or else explicitly by the user via the negative value in SQLRC(3) shown above. Because each different SELECT that is active will require its own cursor, and because there is a limit of 20 open cursors allowed (due to system resource constraints, etc.), it is important to close any unneeded cursors that are open. A COMMIT WORK or ROLLBACK WORK SQL statement can also be used to close cursors, and, in the above example, is an alternative to the -1 technique. However, in general, a COMMIT WORK or ROLLBACK is not always appropriate, because it will close all open cursors and also affect any table modification activity.

Example: Retrieval and Update with User-Specified WHERE Clause

This example is a modification of "Example: Retrieval with User-Specified WHERE Clause" on page 243 and allows you to update the employee data that is displayed. To aid you in identifying the changes, program lines that are identical with the original example have matching line numbers (which are always multiples of 10 from 220 on), and new or changed lines have unique line numbers which end in digits 1-9. For example, to modify the original example, delete lines 460 and 480, and enter lines 471-479, and 491-492.

Note: Lines 140-290 are not shown in order to conserve space. However, they are identical with the original.

Before you run this program, you should create your own special copy of the Q.STAFF table. This is necessary because updates are not allowed against Q.STAFF. Hence, you should use the "generalized" program shown on "Example: Filling a Table by Copying Rows From Another Table" on page 179, run the program three times, and respond to the prompt message:

```
Enter SQL statement lines - null line when finished
  SQL ===>
```

as follows:

Step 1 - Table Creation

```
SQL ===> CREATE TABLE MYSTAFF
SQL ===> (ID SMALLINT NOT NULL,
SQL ===> NAME VARCHAR(9),
SQL ===> DEPT SMALLINT,
SQL ===> JOB CHAR(5),
SQL ===> YEARS SMALLINT,
SQL ===> SALARY DECIMAL (7,2),
SQL ===> COMM DECIMAL (7,2))
SQL ===>
```

This creates a table named MYSTAFF that has the same data characteristics as the Q.STAFF table.

Note: You may want to add an IN clause to this SQL statement. See the discussion of line 160 in "Example: Table Creation" on page 177.

Step 2 - Insert Rows

```
SQL ===> INSERT INTO MYSTAFF
SQL ===> SELECT * FROM Q.STAFF
SQL ===>
```

This simply copies all the data rows from Q.STAFF to MYSTAFF. MYSTAFF is now identical to Q.STAFF.

Step 3 - Create Index

```
SQL ===> CREATE UNIQUE INDEX INDX_1 ON MYSTAFF (ID)
SQL ===>
```

This step is included to guarantee that the ID column always contains unique values. This is necessary because otherwise it would be possible to introduce duplicate ID values when updating the table. Then, subsequent updates could affect more than one row, which contradicts our intent of updating only the displayed data.

```
100 REM: SELECT and UPDATE with user specified WHERE
110 OPTION BASE 1: INTEGER sqlrc
120 DIM sqlrc(20), stmt$*1000, sqlmsg$*256
130 DIM hdgs_def$(10), hdgs_text$(10)*40, col_def$(7)
 .
 .
 .
300 ON SKEY GOTO test_pfk               ! trap for function key 3
310 stmt$ = "SETOPT NULL 0"             ! translate nulls to zero
320 CALL SQL (stmt$, sqlrc(), sqlmsg$)
330 IF sqlrc(1) NE 0 THEN GOTO sql_error
340 PRINT NEWPAGE: PRINT FIELDS MAT hdgs_def$: MAT hdgs_text$
350 DO
360   INPUT FIELDS in_fld$: n$
370   PRINT FIELDS msg_fld$: "Stand by: Retrieving from data base"
380   stmt$ = "SELECT ID, DEPT, JOB, YEARS, SALARY, COMM "
390   stmt$ = stmt$ & "FROM MYSTAFF WHERE NAME = :7"
400   CALL SQL (stmt$, sqlrc(), sqlmsg$, i1, d, j$, y, s, c, n$)
410   EXIT IF sqlrc(1) GT 4 AND sqlrc(2) NE 100
420   PRINT FIELDS in_fld$: " "         ! Blank out input area
430   IF sqlrc(2) EQ 100 THEN
440     PRINT FIELDS msg_fld$: "*** No employee named " & n$
450   ELSE
470     PRINT FIELDS MAT col_def$: i1, n$, d, j$, y, s, c
471     PRINT FIELDS msg_fld$: "Update employee data; press ENTER"
472     INPUT FIELDS MAT col_def$: i2, n$, d, j$, y, s, c
473     PRINT FIELDS msg_fld$ : "Stand by: Updating data base"
474     stmt$ = "UPDATE MYSTAFF SET ID=:1, NAME=:2, DEPT=:3, "
475     stmt$ = stmt$ & "JOB=:4, YEARS=:5, SALARY=:6, COMM=:7 "
476     stmt$ = stmt$ & "WHERE ID= :8"
477     CALL SQL(stmt$,sqlrc(),sqlmsg$,i2,n$,d,j$,y,s,c,i1)
478     IF sqlrc(1) GT 4 THEN GOTO sql_error
479     stmt$ = "COMMIT WORK"
490     CALL SQL (stmt$, sqlrc(), sqlmsg$)
491     IF sqlrc(1) GT 4 THEN GOTO sql_error
492     PRINT FIELDS msg_fld$ : "Update complete: Enter next name"
500   END IF
510 LOOP UNTIL (sqlrc(1) GT 4 AND sqlrc(2) NE 100)
520 sql_error:                              ! Fatal SQL errors
530 IF sqlrc(1) GT 4 AND sqlrc(2) NE 100 THEN
540   PRINT "*** Data Base or SQL Statement Error:"
550   PRINT "       SQL Statement Text:  ";stmt$
560   PRINT "    BASIC/SQL Return Code:  ";sqlrc(2)
570   PRINT "    Data Base Return Code:  ";sqlrc(8)
580   PRINT "    Error Message Text: ";sqlmsg$
590 END IF
600 STOP
610 test_pfk: IF KEYNUM = 3 THEN PRINT NEWPAGE ELSE CONTINUE
620 END
```

Notes:

- Line 471: A message has been added inviting you to modify the displayed values.

- Line 472: INPUT FIELDS is used to assign the screen data back to program variables. Notice that the I2 is used for the new ID variable name, since we need to save the original value (I1) for use in the SQL UPDATE statement.

- Lines 474-478: a SQL UPDATE is issued which updates all columns to their current values. The WHERE clause specifies that the original row (identified as having ID value = I1) is updated. Since there is a unique index for the ID column, we are guaranteed that only one row will be updated.

- Lines 479-482: The SQL COMMIT WORK statement causes the updates to be made permanent, and also terminates the selection (that is, it is unnecessary to explicitly terminate the selection using the negative SQLRC(3) technique, since COMMIT WORK automatically terminates all selections.)

Appendix D. The SQL Sample Tables

The following sample tables are used by the examples in the section "Using SQL to Access Relational Data Bases" on page 156. The tables are distributed with the Query Management Facility product (QMF); they relate to a mythical company called the J & H Supply Company, which has a head sales office and several geographical divisions.

The Q.STAFF Sample Table

The table holds data on the employees of the J & H Supply Company. Each row represents a single, unique employee.

The columns are as follows:

- ID contains the employee serial number. The values of the ID fields must be distinct.
- NAME is the name of the employee.
- DEPT contains the employee's department number.
- JOB contains the job classification for the employee's job and is a descriptive field such as "SALES."
- YEARS contains the number of years the employee has worked for the company.
- SALARY contains the employee's salary in dollars and cents.
- COMM is the commission the employee has earned and is shown in dollars and cents.

The following table provides information about the data characteristics of the Q.STAFF table.

Column Heading	Data Type	Length	Definition
ID	SMALLINT	2	NOT NULL
NAME	VARCHAR	9	
DEPT	SMALLINT	2	
JOB	CHAR	5	
YEARS	SMALLINT	2	
SALARY	DECIMAL	7,2	
COMM	DECIMAL	6,2	

Figure 38. Q.STAFF Data Characteristics

The contents of the Q.STAFF sample table are shown below:

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	SANDERS	20	MGR	7	18357.50	-
20	PERNAL	20	SALES	8	18171.25	612.45
30	MARENGHI	38	MGR	5	17506.75	-
40	O'BRIEN	38	SALES	6	18006.00	846.55
50	HANES	15	MGR	10	20659.80	-
60	QUIGLEY	38	SALES	-	16808.30	650.25
70	ROTHMAN	15	SALES	7	16502.83	1152.00
80	JAMES	20	CLERK	-	13504.60	128.20
90	KOONITZ	42	SALES	6	18001.75	1386.70
100	PLOTZ	42	MGR	7	18352.80	-
110	NGAN	15	CLERK	5	12508.20	206.60
120	NAUGHTON	38	CLERK	-	12954.75	180.00
130	YAMAGUCHI	42	CLERK	6	10505.90	75.60
140	FRAYE	51	MGR	6	21150.00	-
150	WILLIAMS	51	SALES	6	19456.50	637.65
160	MOLINARE	10	MGR	7	22959.20	-
170	KERMISCH	15	CLERK	4	12258.50	110.10
180	ABRAHAMS	38	CLERK	3	12009.75	236.50
190	SNEIDER	20	CLERK	8	14252.75	126.50
200	SCOUTTEN	42	CLERK	-	11508.60	84.20
210	LU	10	MGR	10	20010.00	-
220	SMITH	51	SALES	7	17654.50	992.80
230	LUNDQUIST	51	CLERK	3	13369.80	189.65
240	DANIELS	10	MGR	5	19260.25	-
250	WHEELER	51	CLERK	6	14460.00	513.30
260	JONES	10	MGR	12	21234.00	-
270	LEA	66	MGR	9	18555.50	-
280	WILSON	66	SALES	9	18674.50	811.50
290	QUILL	84	MGR	10	19818.00	-
300	DAVIS	84	SALES	5	15454.50	806.10
310	GRAHAM	66	SALES	13	21000.00	200.30
320	GONZALES	66	SALES	4	16858.20	844.00
330	BURKE	66	CLERK	1	10988.00	55.50
340	EDWARDS	84	SALES	7	17844.00	1285.00
350	GAFNEY	84	CLERK	5	13030.50	188.00

Figure 39. Q.STAFF Sample Table

Q.ORG Table

This table provides information on the organization of the company — department within division. Each row represents a single unique department.

The columns are as follows:

- DEPTNUMB contains the department number. The values of the DEPTNUMB field must be unique.
- DEPTNAME contains the department's descriptive name such as "MID ATLANTIC."
- MANAGER contains the employee number of the manager of the department.
- DIVISION contains the division to which the department belongs.
- LOCATION contains the city location of the department such as "NEW YORK."

The following table provides information about the data characteristics of the Q.ORG table.

Column Heading	Data Type	Length	Definition
DEPTNUMB	SMALLINT	2	NOT NULL
DEPTNAME	VARCHAR	14	
MANAGER	SMALLINT	2	
DIVISION	VARCHAR	10	
LOCATION	VARCHAR	13	

Figure 40. Q.ORG data characteristics

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
10	HEAD OFFICE	160	CORPORATE	NEW YORK
15	NEW ENGLAND	50	EASTERN	BOSTON
20	MID ATLANTIC	10	EASTERN	WASHINGTON
38	SOUTH ATLANTIC	30	EASTERN	ATLANTA
42	GREAT LAKES	100	MIDWEST	CHICAGO
51	PLAINS	140	MIDWEST	DALLAS
66	PACIFIC	270	WESTERN	SAN FRANCISCO
84	MOUNTAIN	290	WESTERN	DENVER

Figure 41. Q.ORG Sample Table

Appendix E. The STATS Subprogram

The following is a general purpose subprogram that is used by "Example: Multiple Row Retrieval for an Entire Table" on page 172. STATS uses a single numeric array as an argument, and calculates and prints various statistical measures.

```
400 SUB stats(x())
410 REM: Mean, median, st. dev., variance, range, max and min
420 OPTION BASE 1
430 INTEGER i,n                              ! n is number of values
440 n=SIZE(x)
450 min,max=x(1)
460 !-- loop through the data---------------------------------
470 FOR i=1 TO n
480    num=x(i)
490    tot=tot+num
500    totsq=totsq+(num**2)                  ! compute sum of squares
510    IF (num>max) THEN
520         max=num
530       ELSE
540         IF (num<min) THEN min=num
550    END IF
560 NEXT i
570 !-- compute ---------------------------------------------
580 mean=tot/n
590 variance=(totsq-((tot**2)/n))/(n-1)
600 sdev=SQR(variance)
610 range=max-min
620 !------find the median------------------------------------
630 MAT x=asort(x)                           ! sort to find median
640 center=n/2
650 IF MOD(n,2)=0 then                       ! even number of elements
660    median=(x(center)+x(center+1))/2
670 ELSE
680    median=x(center)
690 END IF
700 !-- print the results-------------------------------------
710 PRINT "n",n
720 PRINT "mean",mean
730 PRINT "median",median
740 PRINT "standard deviation",sdev
750 PRINT "variance",variance
760 PRINT "range",range
770 PRINT "maximum",max
780 PRINT "minimum",min
790 END SUB
```

Glossary

The terms in this glossary are defined in accordance with their meanings in BASIC. These terms may or may not have the same meanings in other languages.

Definitions from the *Draft Proposed American National Standard for BASIC* dated February 15, 1982, are preceded by **(B)**.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from *American National Standard Vocabulary for Information Processing, ANSI X3.12-1970* (copyright 1970 by American National Standards Institute, Inc.), which was prepared by the subcommittee on Terminology and Glossary, X3.5.

American National Standard definitions are preceded by an asterisk (*).

alphabetic character. Any of the 26 letters (A through Z) of the English alphabet or any of the alphabet extenders.

argument. An expression appearing in parentheses following a function or subprogram name when either is invoked. The expression represents a value that the function or subprogram is to act upon.

arithmetic constant. Also called numeric constant. A constant with a numeric value. The forms of arithmetic constants permitted in IBM BASIC are integer, fixed point, and floating point.

arithmetic data item. Also called numeric data item. Data having a numeric value

arithmetic expression. Also called numeric expression. An arithmetic constant, a simple arithmetic variable, a scalar reference to an arithmetic array, an arithmetic-valued function reference, or a sequence of the above appropriately separated by arithmetic operators and parentheses.

arithmetic operator. Also called numeric operator. A symbol representing an operation to be performed upon arithmetic data. The arithmetic operators are:

+ addition and unary plus sign
- subtraction and unary minus sign
* multiplication
/ division
** or ¬ or ^ exponentiation

arithmetic variable. Also called numeric variable. The name of an arithmetic data item whose value is assigned and/or changed during program execution. The name consists of one to 40 alphabetic characters. The first character must be a letter, and the remaining may be a mixture of letters, numbers and underlines. The last character can also be used to designate the type of variable (# for decimal, % for integer).

array. (1) * An arrangement of elements in one or more dimensions. (2) In BASIC, a named list or table of data items, all of which are the same type, arithmetic or character, and all of which have the same maximum length. IBM BASIC allows up to seven dimensions for arrays.

array declaration. The process of naming an array and assigning dimensions to it either explicitly (by the DIM or COM statement) or implicitly through usage.

array element. See array member.

array expression. An arithmetic expression or a character expression representing an array of values rather than a single value. It may be used only in an array assignment statement.

array member. A single data item in an array; its position is indicated by a subscripted array reference.

array variable. An entire named array. The name consists of alphabetic characters (for arithmetic arrays) or alphabetic characters followed by the dollar sign character, $, (for character arrays).

*** ASCII.** American National Code for Information Interchange. The standard code using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

assignment. The process of giving values to variables; for example, via LET statements, READ statements, INPUT statements, etc.

assignment symbol. The symbol =, which is used in an assignment statement to give a value to one or more variables.

BASIC. A term applied as a name to members of a special class of languages which possess similar syntaxes and semantic meanings; acronym for Beginner's All-purpose Symbolic Instruction Code.

(B) batch mode. The processing of programs in an environment where no provision is made for user interaction.

binary operator. A numeric operator with two terms. The binary operators that can be used in arithmetic expressions are: addition (+), subtraction (-), multiplications(*), and division (/).

*** branch.** In the execution of a computer program, to select one from several alternative sets of instructions.

built-in function. See intrinsic function.

character constant. A constant with a character value. It is usually enclosed by quotation marks, but character constants in DATA statements and INPUT replies need not have enclosing quotation marks.

character data. Data having a character value, as opposed to a numeric value.

character expression. A character constant, a simple character variable, a reference to a character array element, a character-valued function reference, a substring of a character variable or array element, or a sequence of the above appropriately separated by concatenation operators and parentheses.

character string. (1) * A string consisting only of characters. (2) A connected sequence of characters.

CMS. Conversational Monitor System.

collate native. To sequence data in the EBCDIC mode. (See *IBM BASIC Language Reference.*)

collate standard. To sequence data in the ASCII mode. (See *IBM BASIC Language Reference*).

comment. A remark or note included in the body of a program by the programmer. It has no effect on the execution of the program; it merely documents the program. Comments are written as a string of characters, preceded by an exclamation mark (!), or as individual lines that start with the keyword REM. May be included on all BASIC statements except the DATA, FORM, and IMAGE statements.

concatenate. To link or join together.

concatenation operator. An operator used in a character expression to join two character expressions. The concatenation operator is an ampersand (&).

constant. A value that never changes. IBM BASIC has two types of constants: numeric and character.

control specification. One of the specifications X, POS, SKIP, or PAGE used in the FORM statement to specify formatting of records in record-oriented files, or to control print line formatting at a terminal.

current line. The line at which operations are being performed, or at which operations on a group of lines begin.

data file. See file.

data form specification. (1) One of the specifications B, C, NC, ZD, PD, S, L, or PIC, used in the FORM statement to specify formatting of character and numeric values in record-oriented files. (2) One of the specifications C or PIC, used in the FORM statement to format character and numeric values on a printed line. (3) One of the specifications used in IMAGE to specify formatting of character and numeric values on a printed line.

data item. A single unit of data; that is, a constant, a variable, an array element, or a function reference.

data table. The values contained in the DATA statements of your program. DATA statements are processed in statement number sequence (lowest to highest). The values in each DATA statement are collected and placed conceptually in a single table in order of their appearance (left to right).

data table pointer. An indicator that moves sequentially through the data table, pointing to each value as it is assigned to a corresponding variable in a READ statement. Initially, the indicator refers to the first item in the table. It can be repositioned to the beginning of the table at any time by the RESTORE statement.

declaration. See explicit declaration and implicit declaration.

*** delimiter.** A character that groups or separates words or values in a line of input.

digits. Any of the numerals 0 through 9.

dimension specification. The specification of the size of an array and the arrangement of its members into one to seven dimensions.

direct access. The facility to obtain data from a storage device, or to enter data into a storage device in such a way

that the process depends only on the location of that data and not on a reference to data previously accessed.

display format file. A sequentially organized file designed to be output in readable form.

EBCDIC. External binary coded decimal interchange code.

EBCDIC collating sequence. The ordering of character data items according to the Extended Binary Coded Decimal Interchange Code.

* **error message.** An indication that an error has been detected.

(B) exception. A circumstance arising in the course of executing a program when an implementation recognizes that the semantic rules of the BASIC standard cannot be followed or that some resource constraint is about to be exceeded. Certain exceptions (nonfatal exceptions) may be handled by automatic recovery procedures specified in the BASIC standard. These and other exceptions may also be handled by recovery procedures specified in the program. If no recovery procedure is given (fatal exceptions) or if restrictions imposed by the hardware or operating environment make it impossible to follow the given procedure, then the exception shall be handled by terminating the program.

executable statement. A program statement that causes an action to be performed by the computer.

execution error. Execution errors are called exceptions. An error discovered during execution of an IBM BASIC program (for example, dividing by zero, branching to a nonexistent statement number, etc.)

explicit declaration. The use of a DIM or COMMON statement to specify the number of members in an array, the number of dimensions in an array, or the length of a character variable. The use of DECIMAL, INTEGER, and REAL statements to specify the numeric type of variables, arrays, and functions.

exponent (of floating-point format number). An integer constant specifying the power of 10 (or 16) by which the base (mantissa) of the decimal (or binary) floating-point number is to be multiplied.

exponentiation. Raising a value to a power.

expression. A notation, within a program, that represents a numeric or character value. Expressions can be constants, variables, or functions appearing alone, or in combination with operators. Five forms of expressions are defined in IBM BASIC: numeric, character, relational, logical, and array.

* **file.** A set of related records treated as a unit, for example, in stock control, a file could consist of a set of invoices.

filename. The name of a file.

fixed-point. A mathematical notation (as in a decimal system) in which the point separating whole numbers and fractions is fixed. See floating-point.

fixed-point constant. An arithmetic constant consisting of one or more digits and a decimal point, and optionally preceded by a sign.

fixed-point format. The form used to express a fixed-point constant.

floating-point. Involving or being a mathematical notation in which a quantity is denoted by one number multiplied by a power of the number base. See fixed-point.

floating point constant. An arithmetic constant consisting of an integer or fixed-point constant (the mantissa) followed by the letter E, followed by an optionally signed one or two digit integer constant (the exponent).

floating-point format. The form used to express a floating-point constant.

formatted data. Data that has been input or output in accordance with the IMAGE or FORM statement.

function. A named expression that computes a single value. See also intrinsic function and user-written function.

function reference. The appearance of an intrinsic function name or a user-written function name in an expression.

hard copy. A printed copy of machine output in a visually readable form; for example, printed reports, listings, documents, and summaries.

identifier. A character string used to name a variable, an array, a function, a subprogram, or a program.

implicit declaration. (1) The specification of the number of members in an array or the number of dimensions in an array, either by a reference to a member of an array or by context (without the array being explicitly specified in a DIM or COM statement). (2) The specification of the length of a character variable by context (without the variable being explicitly defined in a DIM statement).

input. The transfer of data from an external medium to internal storage.

input list. A list of variables to which values are assigned from input data; the list can be composed of scalar variables, array member references, array references, and array references with redimensioning.

input/output. The transfer of data between an external medium (that is, the terminal typewriter or a file) and a workspace.

(B) interactive mode. The processing of programs in an environment which permits you to respond directly to the actions of individual programs and to control the initiation and termination of these programs.

integer constant. An arithmetic constant containing one or more digits, optionally preceded by a sign.

integer format. The form used to express an integer constant.

internal format file. A file created by IBM BASIC WRITE, or PUT statements containing self-identifying, sequentially organized data.

internal storage. A computer's main storage.

internal text file. A sequential file composed of both binary and EBCDIC data records in a format unique to IBM BASIC.

* **interrupt.** To stop a process in such a way that it can be resumed.

intrinsic function. A function supplied by IBM BASIC (for example, SIN, COS, SQR, etc.) See Function.

* **key.** One or more characters, within a set of data, that contains information about the set, including its identification.

key-sequenced file. A record-oriented file whose records are stored and accessed according to key values embedded in the records.

line. An ordered sequence of characters which terminates with an end-of-line.

listing files. Files with records that contain only EBCDIC characters; such files can be listed on an external device.

literal. A symbol or quantity in a source program that is itself data, rather than a reference to data. Same as data.

Logging on. Entering the procedure in force in your organization to allow you access to the system.

logical operator. An operator that is used in a logical expression. The logical operators are: AND, OR, and NOT.

loop. A sequence of instructions that is executed repeatedly until a terminating condition is reached. The FOR statement identifies the beginning of one type of loop; the NEXT statement identifies the end of it. The DO statement identifies the beginning of another type; the LOOP statement identifies the end of it.

mantissa. In floating-point notation (floating-point format), the number that precedes the E. The value represented is the product of the mantissa and that power of the base specified by the exponent.

* **matrix (mathematical).** A rectangular array of elements, arranged in rows and columns, that may be manipulated according to the rules of matrix algebra.

multiline function. A user-defined function that is defined with more than one statement.

native format file. A file created by an IBM BASIC output statement which uses the FORM statement. The file contains user-defined data, organized sequentially, by record number or by key.

nesting. (1) The occurrence of one or more loops within another loop. (2) The occurrence of a GOSUB statement when one or more GOSUB statements are already active. (3) The use of more than one set of parentheses to indicate the order of evaluation in a complex arithmetic expression.

nonexecutable statement. A statement that that is used to specify information to a compiler, but that does not explicitly result in executable code; for example, a declaration.

null character string. Two adjacent single or double quotation marks.

null delimiter. One or more blanks or no characters at all (that is, one data item directly following another data item with no intervening space or delimiter).

numeric character. Any of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

numeric constant. See arithmetic constant.

numeric data item. See arithmetic data item.

numeric expression. See arithmetic expression.

numeric operator. See arithmetic operator.

numeric variable. See arithmetic variable.

object file (module). A file of internally coded data records appropriate for input to a linkage editor or loader.

operand. A constant, a variable, an array member reference, a function reference, or a subexpression on which an operation is to be performed.

operator. A symbol specifying an operation to be performed. See also arithmetic operator, binary operator, concatenation operator, logical operator, relational operator, and unary operator.

output. The transfer of data from internal storage to an external medium.

output list. A list of variables from which values are written to an output file; the list can be composed of scalar expressions and array references.

padding. The addition of one or more blanks to the right or left of a character string to extend the string to a required length.

parameter. A simple variable or empty array declaration enclosed in parenthesis in a DEF statement or a SUB statement, and then used within that function or subprogram. The function or subprogram performs its calculations on the values substituted for the parameters when the function or subprogram is called.

precision. The number of digits for which significance can be expressed, or the accuracy with which a number can be presented.

priority. A rank assigned to a numeric or logical operator; it is used when evaluating a numeric expression. The order of priorities, from high to low, is: exponentiation, unary operations, multiplication and division, addition and subtraction, relational operators, NOT, AND, OR. Operations at the same priority level are evaluated as they are encountered (from left to right in the expression).

program. A logically self-contained sequence of BASIC statements that can be executed by the computer to attain a specific result.

record. A collection of related data items treated as a unit.

record-oriented file. A file in which items are stored in records.

redimension specification. The assignment of a new dimension specification to an already existing array, via an array assignment statement, a READ statement, an INPUT statement, a GET statement, a READ file statement, or a REREAD file statement.

redimensioning. The changing of the number of dimensions or the number of members in each dimension of a previously declared array.

relational operator. An operator used in a logical subexpression. The relational operators are:

EQ or = equal to
NE or <> not equal to
GT or > greater than
LT or < less than
GE or >= greater than or equal to
LE or <= less than or equal to

relative-record file. A file whose records are loaded into fixed-length, fixed-location slots, according to their record number.

relative-record number. A number that identifies not only the slot in a relative-record data set but also the record occupying the slot.

remark. See comment.

replication factor. The number of times a single data FORM specification is to be used.

scalar. A single data item (as opposed to an array of items).

scalar expression. An arithmetic expression or a character expression representing a single value rather than an array of values.

sequential access. The retrieval of data according to the order in which the data is stored in a file.

significant digits. All the digits of a number starting with the leftmost nonzero digit.

simple variable. A scalar variable (but not an array member).

single-line function. A user-defined function that is defined in one statement (that is, the DEF statement).

slot. The space for a data record in a relative-record file.

source file. A sequentially accessible file containing EBCDIC coded records of language statements or commands.

special characters. Any characters allowed in IBM BASIC that are not alphabetic characters or digits.

stream-oriented file. A file in which items are stored as a stream of data items and retrieved in sequential order.

subprogram. A program unit beginning with a SUB statement and ending with an END SUB statement. Control is transferred to a subprogram by a CALL statement. Control is returned by the SUBEXIT statement.

subroutine. A program segment (sequence of statements) branched to by a GOSUB statement. The last statement of a subroutine must be a RETURN statement which directs the computer to return and execute the statement following the GOSUB statement.

subscript. Any valid arithmetic expression (whose rounded integer value is greater than or equal to zero) used to refer to a particular member of an array.

substring. A part of a character string.

terminal. A device, usually equipped with a keyboard and some type of display, capable of sending and receiving information over a communication channel.

text file. A file of internally coded data records appropriate for input to a linkage editor or loader.

* **truncation.** The deletion or omission of either a leading or trailing portion of a string in accordance with specified criteria.

unary. Having or consisting of a single component, element, or item.

* **unary operator.** A numeric operator having only one term. The unary operators that can be used in numeric expressions are: (positive) + and (negative) -.

user-written function. A function defined by the user in a single-line or multiline function definition.

variable. A data item whose value may change during execution of a program.

vector. (1) * A quantity usually represented by an ordered set of numbers. (2) A collection of array data having a single dimension.

VS. Virtual Storage.

VSAM. Virtual Storage Access Method.

workspace. An area assigned for each user for entering, changing, or executing a program.

zero suppression. The elimination of leading nonsignificant zeros in a number.

Index

Special Characters

& (ampersand) 49
  continuing statements 11, 86
! (exclamation point) 85
$ (dollar sign) 48
* (asterisk) 8
/ (slash) 94
% (percent sign) 43
? (question mark) 93, 95
(number sign) 43
' (apostrophe or single quote) 48
" (quotation mark) 48

A

accessing relational data bases 156
adding records to a file 117
addition 45
ampersand 11
AND operator 65
arguments
  intrinsic functions 60
  user-defined functions 63
array
  assignments 55
  character 53
  defining 51
  element, definition 51
  explicit dimensioning 53
  expression 56
  implicit dimensioning 54
  matrix multiplication 57
  numeric 53
  reading 122
  redimensioning 58
  subscript, definition 51
  subscripts 54
  writing 122
array dimensioning 54
ASSEMBLER
  argument list 143
  calling sequence 146
  linking subprograms 147
assigning values to variables 46
assignment statement (LET) 46
assignment, array 55
asterisk 8
attention interrupt 191
ATTN 130
AUTO command 10
automatic line number 10
automatic line number ending 11

B

BASE in OPTION statement 52
BASIC environment 2
  inside 2
  outside 2
BASIC, general description 1
BASICRUN 197
BOTTOM option, MARGIN statement 106
branching
  conditional 77, 78
  unconditional 76
BREAK command 208
BREAK Statement 210

C

CALL BASIC statement 147
CALL GDDM statement 148
CALL SQL statement
  examples 168, 241
  how to use 159
  message parameter 164
  return code parameter 162
  value-list parameter 164
CALL statement 141, 142
CALL SYSTEM statement 147
calling
  ASSEMBLER 142
  COBOL 141
  FORTRAN 141
  GDDM 148
  PL/I 141
  programs 81
  SQL 156
  SYSTEM 147
CASE ELSE statement 73
CASE statement 73
CAUSE statement 132
causing an Exception 132
CHAIN statement
  COMMON statement and 81
  description 81
  files 82
  USE option 82
  variables 82
chaining programs 81
CHANGE command 20
CHANGE command, line or scroll mode 21
changing attributes of variables 213
changing character strings 20

changing the program
CHANGE command 20
COPY command 21
delete 12, 14
DELETE command 22
EXTRACT command 22
insert 8
MERGE command 23
replace 12, 14
character
arrays 53
constant 47
function 63
substrings 49
variable 48
character string
current length 49
definition 256
input 93
length 48
line input 94
overflow 91, 105
PAUSE statement 80
character substrings 49
checking SQL return codes 162
circumflex 45
closing files
CLOSE statement 107
with CHAIN 82
with CLOSE 108
with END 79, 108
with INITIALIZE 16
with QUIT 25
with STOP 79, 108
COBOL 141
column positioning 97
COM statement 53, 82
comma
input 93, 94
ON GOSUB statement 78
print 95
command execution, interrupting 33
command, definition 7
commands
AUTO 10
BREAK 208
CHANGE 20
COMPILE 194
COPY 21
DELETE 22
DROP 213
EXTRACT 22
FIND 19
GO 209
GO STEP 209
HELP 34, 216
INITIALIZE 16
LIST 16
LOAD 26
MERGE 23
PROFILE 29
PURGE 128
QUERY 127
QUIT 35
RENAME 24
RENUMBER 18
RUN 189, 195
SAVE 25
SET LOG 31
SET MSG 31
SET OPTION 32
SET PROFILE 30
SET TERM 32, 147
SYSTEM 147
comments 84
! 85
continuation 86
REM statement 85
common programming errors 86
COMMON statement 53, 82
COMPILE command 194
compiled execution 196
compiler options
FIPS 199
FLAG 199
LIST 199
LPREC 204
MAP 201
NOFIPS 199
NOLIST 199
NOMAP 201
NOOBJECT 203
NOPROF 204
NOSOURCE 204
NOXREF 204
OBJECT 203
PROFILE 204
SOURCE 204
specifying 198
SPREC 204
XREF 204
compiling a program
batch mode 196
interactively 194
concatenation 49
conditional branching
IF statement 72, 77
ON GOSUB statement 78
ON GOTO statement 78
conditional statements
IF/ELSE block 72
SELECT block 73
connecting with IBM BASIC 5
constants
character 47
decimal 41
floating-point 41
integer 40
numeric 43
continuation lines

deleting 13
editing 12
inserting 14
statement 11
CONTINUE statement 133, 134, 193
control structures
branching 76
controlling program flow with 67
DO WHILE loops 69
DO/UNTIL loops 68
END statement 79
EXIT IF statement 71
FOR/NEXT loops 70
general looping considerations 71
GOSUB/RETURN statements 76
GOTO statement 76
halting execution 79
IF statement 72, 77
IF/ELSE blocks 72
nested loops 71
program loops 68
SELECT/CASE blocks 73
STOP statement 80
controlling exceptions
CONTINUE 133
ON condition statement 129
RETRY 133
CONV condition 123, 130
conversion exceptions 105
COPY command 21
correcting a line. 9
COS function 60
creating a program 8
current line and
CHANGE command 21
COPY command 22
DELETE command 22
EXTRACT command 23
FIND command 19
LIST command 17
LOAD command 26
RENUMBER command 19
SAVE command 26

D

data
DATA statement 89
reading internal 90
real 42
RESTORE statement 92
reusing internal 92
specifying internal 89
type 39
DATA statement 89
data type
decimal 41
integer 40
real 42
DATABASE 2 (DB2) 156
DB2 156
deactivating SQL requests 162
DEBUG statement 210
debugging
a program 207
TRACE statement and 214
DECIMAL array 53
decimal numbers 41
DECIMAL statement 44
declaring arrays 52
DEF 62
DELETE command 22
deleting
a program from workspace 35
deleting a program from workspace 22
files 128
program lines 12, 22
programs 128
delimiter 19
DIM statement 53
dimensioning arrays
explicit 53
implicit 54
DISPLAY file
format 110
organization 112
reading 110
writing 110
displaying items at terminal 95
displaying lines in workspace 17
division 45
division by zero 130
DO loop 68
DO UNTIL/LOOP block 68
DO WHILE/LOOP block 69
DROP command 213
DUPKEY condition 123
duplicate key 123
duplicate record 123
duplicating lines 21
DUPREC condition 123

E

editing
full-screen 14
with commands 16
editing a program 12
ELSE clause 72, 78
END IF statement 72
end of file 123
end program 79
END SELECT statement 73
END statement 79
END SUB statement 137
ending automatic line numbering 11
ENDPAGE 130
ENDPAGE condition 123

entering a program 8
EOF condition 123
error codes
  CODE 132
  ERR 132
  explanation 31
  LINE 132
error conditions
  input/output statements 125
error handling
  CAUSE statement 132
  considerations 134
  CONTINUE statement 133
  ON condition statement 129
  RETRY statement 133
error recovery in I/O
  See exceptions in I/O
errors 86
exception handling
  See also error handling
  description 129
  INPUT statement 105
  PRINT statement 105
exceptions
  listing 17
  multiple 124, 135
  within functions 136
  within loops 134
  within subprogram 140
exceptions in I/O
  CONV 131
  conversion 123
  duplicate key 123
  duplicate record 123
  end of file 123
  ENDPAGE 131
  EXIT 131
  exit to separate routine 123
  hardware error 123
  key doesn't exist 123
  page overflow 123
  record doesn't exist 123
  SOFLOW 131
  string overflow 123
executing a program
  batch mode 198
  interactive mode 195
EXIT condition 123
EXIT IF statement 71
EXIT option 105
exiting a subroutine 77
exponentiation 45
expressions
  array 55
  character 49
  hierarchy 45
  logical 65
  numeric 45
  numeric operators 45
  parentheses 46
  priority of evaluation 66
  rules 45
  using 63
EXTRACT command 22
extracting lines 22

F

FETCH command 26
file accessing
  CLOSE statement 107
  OPEN statement 107
file attributes 107
file format
  DISPLAY 110
  internal sequential 109
  native 109, 116
  stream 120
file organization
  KEYED 116
  relative 112
  SEQUENTIAL 108
file positioning
  KEYED 119
  relative 115
  sequential 112
  stream 122
file processing
  random by key 116
  random by record number 112
  sequential 108
file reference number 107
file specification 107
files
  closing 107
  defining 107
  displaying 126
  opening 107
  purging 126
FIND command 19
FIPS compiler option 199
FIXED option, OPEN statement 108
fixed-point 41
  fixed-point 41
FLAG compiler option 199
floating-point 41
FNEND statement 62
FOR/NEXT block 70
FORM statement 100
formatted output
  IMAGE statement 100
FORTRAN 141
full-screen editing 14
full-screen editor
  scroll mode vs. line mode 14
  with PF keys 16
function
  arguments 63

DEF statement 62
exceptions 136
FNEND statement 62
intrinsic 60
multi-statement 62
parameters 62, 63
single statement 62
user defined 62

G

GDDM 148
GET statement 123
GO command 209
GO STEP command 209
GOSUB statement 76
GOTO statement 76
Graphical Data Display Manager (GDDM) 148

H

halting a program
attention interrupt 191
BREAK command 208
BREAK statement 210
PAUSE statement 80, 210
STOP 212
halting program execution 79
HELP command 34, 216
horizontal spacing, controlling 95

I

I/O errors
See exceptions in I/O
I/O exceptions
See exceptions in I/O
IBM DATABASE 2 (DB2) 156
ICU 156
IF statement 72
IF/ELSE blocks 72
IMAGE statement 100
immediate statement
deleted or dropped 213
LET 212
MAT 212
MAT PRINT 211
PRINT 211
scope 213
STOP 212
immediate statements 211
immediate variables 213
arrays 214
creating 214
type (DECIMAL, INTEGER, REAL) 213
INITIALIZE command 16
INPUT FIELDS statement 98
INPUT statement 93
INPUT statement, exception handling 105
CONV option 105
SOFLOW option 105
input with
comma 94
prompting 93
slash 94
input/output
error conditions 125
exceptions 123
FORM statement formats 100
IMAGE statement formats 100
performing 89
terminal 92
inserting lines 8
numbering 8
INTEGER array 53
INTEGER statement 44
integers 40
Interactive Chart Utility (ICU) 156
interactive input/output 92
internal data
RESTORE statement 92
reusing 92
internal data, specifying 89
internal program representation 24
internal text 24, 26
interrupting command execution 33
interrupts
attention 192
AUTO command 33
BREAK command 208
CHANGE command 33
commands 33
FIND command 33
GO command 33
LIST command 33
LOAD command 33
MERGE command 33
PAUSE 198
program 192
RUN command 33
SAVE command 33
STORE command 33
intrinsic function 60
introduction 1
invoking
COBOL 141
DB2 156
FORTRAN 141
GDDM 148
PL/I 141
separate programs 81
SQL/DS 156
SYSTEM commands 147
IOERR condition 123

K

KEY 119
key length 126
key position 126
keyed file
  adding records 117
  changing a record 118
  format 116
  organization 116
  positioning 119
  reading by key 117
  reading sequentially 119
  writing 116
keyword, definition 43
KLN function 126
KPS function 126

L

LEFT option, MARGIN statement 106
LEN function 60
LET assignment statements 46
line continuation 11
LINE INPUT statement 94
line number
  automatic 10
  ending 11
  entering 8
  range 8
line numbering, automatic 10
LINPUT statement 94
LINPUT statement, exception handling 105
  CONV option 105
  SOFLOW option 105
LIST command 16, 17
  ERROR option 17
  OUT option 17
  XREF option 17
LIST compiler option 199
listing
  files 127
  program 16
literal
LOAD command 26
loading a program 24, 26
  LOAD 26
locating character strings
  CHANGE command 20
  FIND command 19
logging on 5
logical expressions 65
  AND operator 65
  NOT operator 65
  OR operator 65
logical programming errors 87
LOGON command 5
long precision 191
loops, nested 71
loops, program
  DO UNTIL statement 68
  DO WHILE statement 69
  EXIT IF statement 71
  FOR/NEXT statement 70
LPAD$ function 61
LPREC compiler option 204
LPREC option, direct execution 191
LPREC option, RUN command 191

M

MAP compiler option 201
margin setting 106
MARGIN statement
  BOTTOM option 106
  files 106
  LEFT option 106
  RIGHT option 106
  terminals 106
  TOP option 106
MAT statement 122
MAT, array assignment 56
matrix
  OPTION BASE statement 54
matrix operations
  addition 56
  assignment 56
  multiplication 57
  scalar assignment 56
  subtraction 56
mechanical programming errors 87
MERGE command 23
merging programs 23
messages from SQL 164
mixed expressions
  decimal 47
  integer 47
  real 47
mode
  line vs. scroll 2
multiple statements on a line 11
multiplication 45

N

nested
  loops 71
  subroutines 77
NEWPAGE 98
NEXT statement 70
NOFIPS compiler option 199
NOKEY condition 123
NOLIST compiler option 199

NOMAP compiler option 201
NONE clause 78
NOOBJECT compiler option 203
NOPROF compiler option 204
NOREC condition 123
NOSOURCE compiler option 204
NOT operator 65
NOXREF compiler option 204
null line 11
NULL values, SQL 166
numbering lines, automatic 10
numbers 40
numeric data 39
- constants 43
- decimal 41
- default type 41
- expressions 45
- floating-point 41
- integer 40
- real 42
- variables 43

O

OBJECT compiler option 203
object program 194, 195
OFLOW 130
ON ATTN statement 192
ON condition action
- GOTO 130
- IGNORE 130
- SYSTEM 130

ON condition statement 129
ON conditions
- ATTN 130
- CONV 130
- ENDPAGE 130
- OFLOW 130
- SKEY 130
- SOFLOW 130
- UFLOW 130
- ZDIV 130

ON GOSUB statement 78
ON GOTO statement 78
OPEN statement
- FIXED option 108
- VARIABLE option 108

operating system
- running a program under different 2

operators
- add 45
- divide 45
- exponentiation 45
- multiply 45
- subtract 45

OPTION statement 52, 83
- BASE option 54
- COLLATE option 64, 83
- FLAG option 84
- INVP option 84
- PRTZO option 95
- RD option 98
- SPREC/LPREC option 83

OR operator 65

P

page control 98
page dimensions and MARGIN statement 106
Parameter function, direct execution 191
parameters for
- CHAIN statement 82
- COBOL 141
- FORTRAN 141
- GDDM 148
- PL/I 141
- SYSTEM 147

PAUSE statement 80, 196, 210
performing input/output 89
PL/I 141
positioning
- by record 115
- relative 115

positioning files
- by KEY 119
- by SEARCH 119
- KEYED 119
- RESTORE/RESET 119
- sequential 112
- stream 122

PRINT FIELDS statement 98
PRINT statement 95
PRINT statement, exception handling 105
- CONV option 105
- SOFLOW option 105

PRINT USING statement
- IMAGE, examples 104

print zone 95
printing
- column control, TAB 97
- comma 95
- page control 98
- print zone 95
- semicolon 96

profile
- command 29
- compiler option 204
- displaying 30
- option 29
- using 28

program
- chaining 81
- changing lines 14
- comments 84
- copying lines 21
- creation 8
- data defined in 89
- debugging 207

deleting from workspace 22
deleting lines 9, 14, 22
designing 37
editing 12, 14
execution, END statement 79
execution, PAUSE 191
execution, STOP statement 79
extracting lines 22
inserting lines 8
interrupting 192
line number 8
loading 26
loops 68
merging 23
multiple statements 11
name 16, 24
renaming 24
renumbering lines 18
replacing lines 12, 14
saving 25
statements 7
stepping 209
testing 207
tracing 214
program execution 209
compiled 194
direct 189
RUN command 189
program-defined data 89
Programmable Function (PF) keys
in HELP panels 27
in programs 27
line and scroll mode 27
notes for using 27
SET PF command 27
with full screen editing 16
programming errors
avoiding 86
logical 87
mechanical 87
not saving a program 86
PROMPT clause, LINE INPUT statement 95
prompting for input 93, 95
PURGE command 128
PUT statement 123

Q

QUERY command 12, 127
QUIT command 25, 35

R

READ statement 90
reading data
DATA statement 90
display 110
internal 90
keyed record 117
READ statement 90
relative record 114
stream 120
reading from
terminal 93
real data 42
double precision 42
real double 42
real single 42
single precision 42
REAL statement 44
record file
definition 107
keyed 116
relative 113
sequential 108
RECORD keyword 113
redimensioning arrays 58
relational data bases, accessing 156
relational expression 64
relational operators 64
relational tables 156
relative file
accessing 113
format 113
organization 112
positioning 115
reading by record 114
reading sequentially 115
updating 115
using 112
writing 113
REM statement 84
remarks
See comments
RENAME command 24
RENUMBER command 18
renumbering lines 18
replacing ? on prompt 95
REREAD statement 123
RESET statement 119
RESTORE statement 92, 119
resuming program operation 209
RETRIEVE PF key 28
retrieving a program 24
retrieving data from SQL tables 161
RETRY statement 133, 134
return codes, SQL 162
RETURN statement 76, 77
conditional
IF statement 77

ON GOSUB statement 78
reusing internal data 92
REWRITE statement 123
RIGHT option, MARGIN statement 106
rounding 47
rounding decimal digits 98
RPAD$ function 61
RUN command 189, 195
interactive mode
after compilation 195

S

sample programs
CALC 227
CUST 232
EDITOR 219
ICU 240
LOANS 221
PIEBAR 153
SQL 241
STATS 253
TRIANGLE 151
SAVE command 25
saving a program 24
SAVE command 25
SAVE vs. STORE 24
scalar assignment, MAT 56
SCRATCH statement 123
screen format 98
SEARCH clause 119
SELECT block 73
CASE ELSE statement 73
CASE statement 73
END SELECT statement 73
SELECT statement 73
SELECT statement 73
SELECT/CASE block 73
semicolon, printing 96
SEQUENTIAL file
adding records 110, 112
format 108
organization 108
reading 108
sequential files
display 110
internal 109
native 109
positioning 112
writing 109
SET command
SET LOG 31
SET MSG 31
SET OPTION 32
SET PF 27
SET TERM 32
SET TERM, from a program 147
short precision 191
single-statement function 62
SKEY 130
SOFLOW condition 123, 130
SOURCE compiler option 204
source file 24
SPREC compiler option 204
SPREC option, direct execution 191
SPREC option, RUN command 191
SQL 156
advanced techniques 241
calling 159
examples 168, 241
language 156
messages 164
NULL values 166
requests, deactivating 162
tables 157
SQL/DS 156
SQR function 60
SREP$ function 61
statement
continuation 11
how to enter 8
immediate statements 7
multiple on a line 11
numbering, automatic 10
program statements 7
statements
BREAK statement 210
CALL BASIC 147
CONTINUE 133
END SUB 137
PAUSE 191
REAL 44
REM 85
RETRY 133
SUB 137
STEP 10, 209
STEP clause in FOR/NEXT block 70
stepping through a program 209
STOP statement 80
description 79
END statement 79
stopping a program 80
STORE command 26
stream file
accessing 120
definition 107
format 120
organization 120
positioning 122
reading 120
Structured Query Language (SQL) 156
Structured Query Language/Data System (SQL/DS) 156
SUB statement 137
subprogram
definition 137
END SUB statement 137
exceptions in 140
SUB statement 137
subroutine 77

subscript
    array access using 54
    definition 51, 260
substring 49
subtraction 45
syntax errors 9, 11
SYSTEM Commands 147
System defaults 32
    overriding with the SET OPTION command 32

T

TAB 97
terminal display 95
terminal input/output 92, 93
terminal, logging 31
terminology note 2
testing a program
    See debugging
THEN clause 72
TOP option, MARGIN statement 106
TRACE statement 214
transferring control 76
    unconditional
        GOSUB statement 76
        GOTO statement 76
        RETURN statement 76
typing variables 43

U

UFLOW 130
unconditional branching 76
USE option 82
user defined functions 62
USING clause
    PRINT statement 104

V

VARIABLE option, OPEN statement 108
variables
    character 48
    decimal 44
    integer 43
    real 44
    specifying numeric type 44
Virtual Storage Access Method 125
VSAM 125

W

workspace
    clearing 25
    loading 26
workspace files 24, 26
WRITE statement 123

X

XREF compiler option 204

Z

ZDIV 130
ZER function 59

Reader's Comment Form

IBM BASIC Programming Guide
SC26-4027-2

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ______________________

Previous TNL ____________________

Previous TNL ____________________

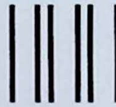
Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC26-4027-2

Reader's Comment Form

Fold and tape Please do not staple Fold and tape

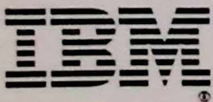


BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape Please do not staple Fold and tape



IBM BASIC Programming Guide (File No. S370-40) Printed in U.S.A. SC26-4027-2

Reader's Comment Form

IBM BASIC Programming Guide
SC26-4027-2

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ____________________

Previous TNL ____________________

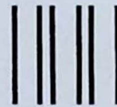
Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC26-4027-2

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

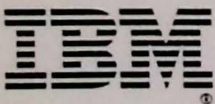


BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape



IBM BASIC Programming Guide (File No. S370-40) Printed in U.S.A. SC26-4027-2

Reader's Comment Form

IBM BASIC Programming Guide
SC26-4027-2

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ____________________

Previous TNL ____________________

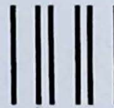
Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

Note: Staples can cause probl[illegible]ith automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC26-4027-2

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape



NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

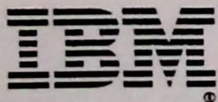
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape



IBM BASIC Programming Guide (File No. S370-40) Printed in U.S.A. SC26-4027-2

Reader's Comment Form

IBM BASIC Programming Guide
SC26-4027-2

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ____________________

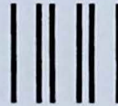
Previous TNL ____________________

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

SC26-4027-2

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

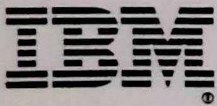


BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape



IBM BASIC Programming Guide (File No. S370-40) Printed in U.S.A. SC26-4027-2

IBM BASIC
Programming Guide

File Number S370-40

Printed in U.S.A.