Microsoft®
BASIC Compiler

for the XENIX® Operating System

Microsoft Corporation

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document Number 290700003-570-R00-0686
Part Number 007-092-009

Contents

Part 1 — User's Guide

1 Introduction 1

1.1 Welcome 3
1.2 Package Contents 3
1.3 System Requirements 4
1.4 Using This Manual 4
1.5 Notational Conventions 6

2 Getting Started 9

2.1 Installing BASIC 11
2.2 Practice Session 13

3 Compiling a BASIC Program 15

3.1 Invoking the BASIC Compiler 17
3.2 Compiling and Linking a BASIC Source File 19
3.3 Compiling Several Source Files 20
3.4 Producing Object Files 20
3.5 Naming the Executable File 21
3.6 Producing a Source Listing File 21
3.7 Producing a Disassembly Listing 22
3.8 Producing a Map File 22
3.9 Generating Debugging and Error-Handling Messages 23
3.10 Using Error-Handling Statements 24
3.11 Writing Quoted Strings to Disk Instead of Memory 24
3.12 Storing Arrays in Row Order 25
3.13 Using Assembly Language Source Files 25
3.14 Stripping the Symbol Table 26
3.15 BASIC Compiler Messages 26

4 Linking Object Modules 27

4.1 ld Command Syntax 29
4.2 Creating Executable Programs From Object Files 30
4.3 Naming the Output File 30
4.4 Linking to Special Libraries 31
4.5 Producing a Map File 32

5 Working With Files and Devices 33

5.1 Device-Independent Input/Output 35
5.2 Filenames and Paths 36
5.3 Handling Files 37
5.4 Data Files: Sequential, Random Access, and ISAM 37
5.5 BASIC and Child Processes 69

6 Using Subprograms 71

6.1 Creating Subprograms 73
6.2 Calling Subprograms 75
6.3 Passing Variables with CALL 76
6.4 Passing Arrays with CALL 77
6.5 Passing Expressions with CALL 79
6.6 Accessing Parameters with SHARED 80
6.7 Passing Variables and Arrays Between Modules 83
6.8 Error Handling 85
6.9 Using GOSUB..RETURN 86
6.10 Variable Scoping Using SHARED and COMMON: An Extended Example 86
6.11 Common Errors 89

7 Interfacing With Other Languages 91

7.1 Loading Assembly Language Files 93
7.2 Calling Assembly Language Subroutines 93
7.3 The Run-Time Memory Map 100

Part 2 — Reference

8 Language Elements 103

8.1 Character Set 105
8.2 The BASIC Line 106
8.3 Data Types 109
8.4 Constants 110
8.5 Variables 112
8.6 Expressions and Operators 116
8.7 Type Conversion 125

9 Compiler-Interpreter Language Differences 127

9.1 Dynamic and Static Arrays 129
9.2 Using Metacommands 132
9.3 New BASIC Statements and Functions 136
9.4 Compiler-Interpreter Differences 137
9.5 Enhanced Statements and Functions 144

10 Statement and Function Reference 147

Part 3 — Appendixes

A ASCII Character Codes 347

B Microsoft BASIC Reserved Words 349

C Summary of Commands 351

C.1 Compiler Options 353
C.2 Linker (ld) Options 354
C.3 XENIX BASIC Metacommands 355

D ISAM Reference 357

D.1 Introduction 359
D.2 Writing an MS-ISAM Application 360
D.3 Parameters 363

D.4 MS-ISAM Subroutines 368
D.5 MS-ISAM Codes 381

E Rebuild 2.0 385

E.1 Introduction 387
E.2 Using Rebuild With Your Data Files 388
E.3 Invoking Rebuild 389
E.4 Definitions of Command Line Arguments 389
E.5 Using Rebuild as a Tool 391
E.6 Data Loss After a System Crash 394
E.7 Adding and Deleting Indexes 395
E.8 Creating and Using
a dd (ASCII) Text File 401

F Error Messages 407

F.1 Invocation Errors 409
F.2 Compile-Time Errors 410
F.3 Run-Time Errors 416

Glossary 423

Index 427

Figures

Figure 7.1 Stack Layout when CALL statement is activated 95
Figure 7.2 Stack Layout After ADD Statement Execution 97
Figure 7.3 XENIX BASIC Compiler Run-Time Memory Map 101
Figure E.1 Rebuild Interactive Mode Main Menu 398

Tables

Table 1.1 Locating Information 5

Table 3.1 XENIX BASIC Compiler Options 18

Table 5.1 Key Designation Assignments 54

Table 8.1 Variable-Type Memory Requirements 115

Table 8.2 Relational Operators and Their Functions 120

Table 8.3 Values Returned by Logical Operations 122

Table 9.1 Functions and Statements Described in This Chapter 129

Table 9.2 XENIX BASIC Metacommands 132

Table 9.3 Listing Format Commands 135

Table 9.4 New BASIC Statements and Functions 137

Table 9.5 Operational Differences 139

Table 9.6 Statements and Functions Not Accepted by the Compiler 143

Table 9.7 Statements Requiring Modification 144

Table 9.8 Enhanced Statements and Functions 145

Table C.1 XENIX BASIC Compiler Options 353

Table C.2 XENIX Linker Options 354

Table C.3 XENIX BASIC Metacommands 355

Table D.1 Values Returned by Function IXSTAT 381

Table D.2 Open Mode Definitions 382

Table D.3 Seek Modes 382

Table D.4 ICONTROL Requests 382

Table D.5 Data Types 383

Table D.6 ILOCK Requests 383

Part 1
User's Guide

1 Introduction 1

2 Getting Started 9

3 Compiling a BASIC Program 15

4 Linking Object Modules 27

5 Working With Files and Devices 33

6 Using Subprograms 71

7 Interfacing With Other Languages 91

Chapter 1
Introduction

1.1 Welcome 3
1.2 Package Contents 3
1.3 System Requirements 4
1.4 Using This Manual 4
1.5 Notational Conventions 6

1.1 Welcome

The Microsoft® XENIX® BASIC Compiler is intended for users who are familiar with BASIC, but want the size and speed advantages of compiled programs. The XENIX BASIC compiler offers the following improvements over interpreted BASIC:

- **Large program capability.** The compiler separates instruction and data space, giving you 64K (kilobytes) of data space and 64K of code space per module.
- **Flexible array dimensioning.** Dynamic arrays allow you to use variables to dimension arrays, providing more efficient use of memory.
- **Support for alphanumeric labels.** The compiler does not require line numbers, and allows the use of alphanumeric line labels. With alphanumeric labels, you can give your statements and subroutines descriptive names. This improves program readability, and can provide simpler debugging.
- **Separately compiled subprograms.** With the compiler, you can create BASIC subprograms that perform common tasks, compile them separately, and use them over and over in many different programs.

1.2 Package Contents

Your XENIX BASIC Compiler package contains the following software, stored on two floppy disks:

- The compiler software
- The libraries required by the compiler
- The XENIX linker, **ld**.
- The ISAM (Indexed Sequential Access Method) utilities and libraries.
- A shell script, "msinstall" that installs the compiler and linker on the system.

1.3 System Requirements

The XENIX BASIC Compiler requires:

- A computer with an INTEL® 80286 microprocessor.
- Microsoft XENIX System 3 or IBM® XENIX Version 1.0 or later.
- 375K free disk space.

1.4 Using This Manual

Part 1 of this manual, the "User's Guide" explains how to set up the XENIX BASIC Compiler, how to compile and run BASIC programs on your system, and discusses how to use some of XENIX BASIC's new programming and language features. Refer to the when you have questions about compiling and linking, or about creating programs in general.

Part 2 of this manual, the "Reference" defines the BASIC language and provides a complete alphabetical reference to the statements and functions supported by the XENIX BASIC Compiler. It also describes the differences between compiled and interpreted XENIX BASIC, and describes what you need to do to compile programs originally written for the BASIC interpreter. Use the "Reference" when you have questions about the syntax and use of the BASIC language.

Part 3 of this manual, the "Appendixes" contains additional information you may find helpful, such as a table of ASCII character codes, a list of BASIC reserved words, and explanations of the compiler, linker and run-time error messages.

Table 1.1 will help you find what you need to know to start using the

XENIX BASIC Compiler.

Table 1.1
Locating Information

For This Information	See
How to install the compiler and linker software	Chapter 2, "Getting Started"
How to run the compiler	Chapter 3, "Compiling a BASIC Program"
How to link a BASIC program	Chapter 4, "Linking Object Modules"
A list of devices supported by BASIC	Chapter 5, "Working With Files and Devices"
How to create random access files	Chapter 5, "Working With Files and Devices"
How to use the ISAM utility	Chapter 5, "Working With Files and Devices"
How to create modular programs	Chapter 6, "Using Subprograms"
How to use assembly-language routines in your BASIC programs	Chapter 7, "Interfacing With Other Languages"
How to use alphanumeric labels	Chapter 8, "Language Elements"
What changes you need to make to your interpreted programs before you can compile them	Chapter 9, "Compiler-Interpreter Language Differences"
What metacommands are and how to use them	Chapter 9, "Compiler-Interpreter Language Differences"
A list of ASCII character codes	Appendix A, "ASCII Character Codes"
A list of BASIC reserved words	Appendix B, "BASIC Reserved Words"
A summary of all the BASIC Compiler options	Appendix C, "Summary of Commands"
A summary of all the BASIC Compiler metacommands	Appendix C, "Summary of Commands"
A complete ISAM reference	Appendix D, "ISAM Reference"
How to use the Rebuild Utility with ISAM files	Appendix E, "Rebuild 2.0"
Explanations of error messages given by the compiler, the linker, and the run-time system	Appendix F, "Error Messages"
An explanation of the terms used in this manual	The glossary

1.5 Notational Conventions

The following notational conventions are used in this manual:

Notation	Description
`Examples`	The typeface shown in the left column is used to simulate the appearance of information that would be printed on your screen. For example, the following program statement is printed in this special typeface: `INPUT "What is your name";A$` When discussing this statement in text, words appearing in the statement, such as `INPUT` and `A$`, also appear in the special typeface.
KEYWORDS	Bold capital letters indicate BASIC keywords. These keywords are a required part of the statement syntax, unless they are enclosed in double brackets as explained below. In programs you write, you must enter keywords exactly as shown. However, you can use uppercase (capital) letters and/or lowercase letters. In the following statement, **IF** and **THEN** are BASIC keywords: **IF** *expression* **THEN** *statement*
command names	Bold type within the text indicates command names, as in the following sentence: It is not necessary to specify the library name on the **bascom** command line.
Apostrophes `''`	An apostrophe is entered as a single right quotation mark (’), not a single left quotation mark (‘). Note that in the typeface used in examples, such as `'index variable`, apostrophes look like this: `'`.
placeholders	Items in italics are placeholders for types of information you must supply, such as a *filename*. In the following statement, *linelabel* is italicized to show that this is a general form for the **GOTO** statement:

GOTO *linelabel*

In an actual program statement, the placeholder *label* must be replaced by a specific line label, as in the following example:

```
GOTO fixup
```

[[optional items]]

Items inside double square brackets are optional. The following statement, for example, shows that entering an *argumentlist* is optional in the **CALL** statement.

CALL *name* **[[**(*argumentlist*)**]]**

Either of the following **CALL** statements is correct:

```
CALL PROG2
CALL NEXTPROG(VALUES, INDEX)
```

{ *choice1***|**_choice2_**}**

This syntax indicates that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items, unless all of the items are also enclosed in double square brackets. For example, the following is the syntax of the **RETURN** statement:

RETURN [[{ *linenumber***|**_linelabel_**}]]**

This indicates the arguments are optional, but if you use an argument it must be either a line number or a line label.

Repeating elements...

Three dots following an item indicate that more items having the same form may appear. For example, this is the syntax of the **DATA** statement:

DATA *constant1***[[**,_constant2_**]]...**

	The dots following [[*constant2*]] indicate you can enter more arguments, provided the arguments are separated by commas.

`Program`
`.`
`.`
`.`
`Fragment`
	A column of dots is used in program examples to show a portion of the program that has been omitted. For example, in the following program fragment, only two lines are shown:
	
	`SUB PROG2 STATIC`
	`.`
	`.`
	`.`
	`END SUB`

KEY NAMES	Small capital letters are used for the names of keys and key sequences, such as ENTER and DELETE.
`Input text`	This type is used to indicate input you enter in response to a prompt. In the following example, "John" is entered in response to the "Enter your name" prompt: `Enter your name: John`

Note

Bold type and italics are also used occasionally in the text for emphasis.

Chapter 2
Getting Started

2.1 Installing BASIC 11
2.2 Practice Session 13

The Microsoft BASIC Compiler must be installed on your hard disk before you can use it to create executable programs. This chapter explains how to install the compiler.

There is a brief Practice Session at the end of this chapter that demonstrates how to create, compile, link and run a sample program.

2.1 Installing BASIC

Before you can use the compiler, you must install it and the related software on your system. A shell program that performs the installation automatically is included on the first compiler disk. This section explains how to install the system on your hard disk.

You must have super-user privileges to install the BASIC compiler. To install the software, follow the instructions below. If you have any problems while you are installing the compiler, press DELETE to stop the installation, and start again from Step 2.

1. Log in as *root*.

   Do not install the BASIC compiler in system maintenance mode. If you do, the compiler will not be installed in the correct file system.

2. Us the **cd** command to change your current directory to "/tmp":

   ```
   cd /tmp
   ```

   Important

   BASIC is distributed on 360K byte, double-sided 9-sector floppy disks. You must determine the device name for your floppy disk drive before you can proceed with the installation. On the IBM AT the drive name is "/dev/fd048ds9". The name may be different on your machine.

   ---

3. Place the first disk (the disk labeled "1 of 2") in the top floppy disk drive and shut the drive door. Use the **tar** command to extract the installation program, *msinstall*, from the disk:

```
tar xvf drive msinstall
```

For *drive*, use the name of your floppy disk drive.

4. If you have an IBM AT computer, when the system prompt returns type

   ```
   msinstall basic
   ```

   If you are not installing BASIC on an IBM AT, type the following:

   ```
   msinstall -d drive basic
   ```

   Substitute the name of your floppy disk drive for *drive*. Press RETURN to execute the installation program.

   The *msinstall* program extracts the files from the disk and places them in the appropriate directories.

5. When you see the message:

   ```
   Are you ready to begin the installation [y,n]?
   ```

   Type `y` and press return. The next prompt asks

   ```
   First disk?
   ```

   If you removed the first disk from the drive, put it back in. Type `y` and press RETURN.

6. You will be prompted to change disks as each disk is copied. When the disk drive light goes out and you see the message:

   ```
   Next disk [y,n]?
   ```

   insert the next disk, type `y` and press RETURN. Do this for each disk in the package. When all the disks are installed, type `n` at the "Next disk" prompt. When you see the message:

   ```
   Installation complete
   ```

   go on to the next step.

7. Type

   ```
   rm /tmp/msinstall
   ```

   to delete the installation script.

The Microsoft XENIX BASIC Compiler is ready to use. Remove the last disk from the drive and store the disks in a safe place. If your hard disk is ever damaged or erased, you will need these disks to reinstall the compiler.

The installation may leave a file called "README.DOC" in "/tmp". This file, if present, contains important information about the compiler. It is recommended that you move this file to a permanent place and that you read it before you use the compiler.

2.2 Practice Session

This section shows you the steps involved in compiling and linking a program using the Microsoft XENIX BASIC compiler. By following these steps you can produce and run an executable program file.

Note

The **msinstall** program places the compiler driver in /usr/bin. To run the compiler, the PATH variable in your .profile file must include this directory.

For the Practice Session, it is recommended that you *not* be logged in as *root*. This way you can see if the compiler is working correctly for general users.

Before you can begin compiling, you must create a BASIC source file. Create your source file with any available text editor, or with the BASIC Interpreter. (If you create your program with the interpreter, be sure you save it with the ",A" option.) It is recommended that you write a very simple program, such as the one in the following example, so you can concentrate on compiling and linking, instead of debugging the program.

1. Enter the following program and save it in a file called "sample.bas:"

```
for a = 1 to 10
    print "This is a sample program"
next a
```

2. Once you have entered the program you are ready to compile and link it. The **bascom** command compiles and links your program with a single command. To create an executable program from the source file you just created, at the system prompt type

```
bascom sample.bas
```

The **bascom** command controls the compiler and linker. Arguments to **bascom** allow you to name the executable file, create additional listing files, link with the ISAM library, and suppress linking. These and other options are discussed in Chapter 3, "Compiling a BASIC Program" and Chapter 4, "Linking Object Modules."

3. After compilation, your program is automatically linked with the BASIC library, BCOM.A. The executable file is placed in a file in your current directory named "a.out."

4. You can run your program as soon as the compiling and linking process is complete, and the XENIX system prompt reappears. To run your program, type

   ```
   a.out
   ```

   and press RETURN. If the program you are using is the sample program listed above, the following message will appear on your screen when you run the program:

   ```
   This is a sample program.
   This is a sample program.
   This is a sample program.
   This is a sample program.
   This is a sample program.
   This is a sample program.
   This is a sample program.
   This is a sample program.
   This is a sample program.
   This is a sample program.
   ```

Chapter 3
Compiling a BASIC Program

3.1 Invoking the BASIC Compiler 17
3.2 Compiling and Linking a BASIC Source File 19
3.3 Compiling Several Source Files 20
3.4 Producing Object Files 20
3.5 Naming the Executable File 21
3.6 Producing a Source Listing File 21
3.7 Producing a Disassembly Listing 22
3.8 Producing a Map File 22
3.9 Generating Debugging and Error-Handling Messages 23
3.10 Using Error-Handling Statements 24
3.11 Writing Quoted Strings to Disk Instead of Memory 24
3.12 Storing Arrays in Row Order 25
3.13 Using Assembly Language Source Files 25
3.14 Stripping the Symbol Table 26
3.15 BASIC Compiler Messages 26

This chapter explains how to use the **bascom** command. In particular, it explains how to:

- Compile and link BASIC source files
- Interpret the error and warning messages produced by the compiler
- Use the compiler options to perform the functions described in Table 3.1

3.1 Invoking the BASIC Compiler

The **bascom** command is all you need to compile and link your BASIC source files using the Microsoft BASIC Compiler. The **bascom** command has the form

bascom [[*options*]] *files*

where each *option* is a command option, and each *file* specifies the file or files to be processed. *Files* can be the names of BASIC source files, assembly language source files, object files, or libraries. You can give more than one option or filename, but you must separate each item by one or more spaces. Table 3.1 summarizes the **bascom** command options described in this chapter.

Note

Compiler options are case-sensitive. Be careful to use the options *exactly* as listed below.

Table 3.1
XENIX BASIC Compiler Options

Option	Description
-A	Includes a listing of the disassembled object code in the source listing.
-c	Suppresses linking.
-D	Generates debugging code for run-time error checking and enables the DELETE key.
-E	Indicates the presence of **ON ERROR GOTO** with **RESUME** *linenumber* statement.
-i	Links with the ISAM library as well as the BASIC library.
-L	Generates a source listing file.
-m	Generates a linker map listing file.
-o	Allows you to specify the name of the executable file.
-R	Stores arrays in row order.
-s	Causes the linker to strip local symbols out of the symbol table when linking user object files with object files generated by the compiler.
-S	Writes quoted strings to .OBJ file instead of symbol table.
-X	Indicates presence of **ON ERROR GOTO** with **RESUME**, **RESUME NEXT**, or **RESUME 0**.

The **bascom** command invokes the compiler driver, */bin/bascom*, which controls execution of the compiler files. When you give the **bascom** command without options, it automatically performs both compilation and linking to produce an executable program file. Using the **bascom** options, you can control and modify the tasks performed by the command. For example, you can direct **bascom** to create a disassembled object code listing, or to write quoted strings to disk instead of memory.

An option consists of a dash (-) followed by a combination of one or more letters that have special meaning to **bascom**. If you are using more than one option, each individual option must be preceded by a dash.

When **bascom** processes the file, it looks at the filename extension to determine whether it should start processing at the compiling, assembling, or linking stage. Files needn't be at the same stage to be combined on a **bascom** command line.

Many options are described in the following sections, and in Chapter 4, "Linking Object Modules." For a complete summary of all the **bascom** options, see Appendix C, "Command Summary."

3.2 Compiling and Linking a BASIC Source File

You can use the **bascom** command to create executable programs from BASIC source files. Source files must be in ASCII format. A file's contents are identified by the filename extension, and BASIC source files must have the extension ".bas". Compile a BASIC source file by giving the name of the file when you invoke the **bascom** command. The command compiles the program in the file, links with the appropriate libraries, then copies the executable program to the default output file `a.out`.

To compile a source program, type

```
bascom file
```

where *file* is the name of the file containing the program.

For example, to compile a program stored in a file named "math.bas," type

```
bascom math.bas
```

The compiler compiles the program, then links the program with the standard BASIC library. Finally, it copies the executable program to the file `a.out`.

You can execute the new program by typing

```
a.out
```

3.3 Compiling Several Source Files

Large programs are often split into several files to make them easier to understand, update, and edit. You can compile such a program by giving the names of all the files belonging to the program when you invoke the **bascom** command. The command reads and compiles each file in turn, then links all the object files together, and copies the new executable program to the file *a.out*.

To compile several source files, type

```
bascom file1 file2 ...
```

where each *file* is separated from the next by at least one space.

3.4 Producing Object Files

When source files are compiled, the **bascom** command creates object files to hold the binary code generated for each source file. These object files are then linked to create an executable program. Object files have the same basename as their source files, but are given the ".o" file extension.

For example, when you compile the two source files *math.bas* and *add.bas*, the compiler produces the object files *math.o* and *add.o*. The object files are permanent files, i.e., the **bascom** command does not delete them after completing its operation. Note that the **bascom** command creates an object file even if only one source file is compiled.

You may want to save useful functions as object files, and use these object files to create programs at a later time. Object files contain the compiled or assembled instructions of your source file, so they save you the time and trouble of recompiling the functions each time you need them. You can create just an object file from a given source file by using the **–c** (compile) option. This option directs **bascom** to compile the source file without creating an executable program. Object files are produced for all files named on the command line.

To make object files for the source files *math.bas* and *add.bas* type

```
bascom -c math.bas add.bas
```

This command compiles each file and writes the compiled source files to the

object files *math.o* and *add.o*. It does not link these files; no executable program is created.

3.5 Naming the Executable File

You can give the executable program file any valid filename by using the **-o** option. The option has the following form:

-o *file*

file is the name of the executable program file.

For example, the command

```
bascom -o math math.o add.o
```

causes the compiler to create the executable program file *math* from the object files *math.o* and *add.o*. You can execute this program by typing

```
math
```

The **-o** option does not create an *a.out* file, nor does it affect any existing *a.out* file. This means that the **bascom** command does not change the current contents of *a.out* if the **-o** option has been given.

3.6 Producing a Source Listing File

The **-L** option produces an a source listing file. The listing contains the memory address of each line in your source file, the text of the source file, its size, and any error messages produced during compilation. The option has the following form:

-L

A listing file is produced for all subsequent source files. Each listing file is given the basename of its source file and the filename extension ".I", and placed in the current directory.

The following command produces a source listing file from the program *sample.bas*:

```
bascom -L sample.bas
```

3.7 Producing a Disassembly Listing

You can direct the compiler to include a disassembly listing in your source listing by using the **-A** option with the **-L** option. The **-A** option adds a listing of the disassembled code, instructions, and addresses relative to the start of the program or module to the source listing, and greatly increases the length of the listing file. In the listing file, the first column contains the line number of the source file (L) and internal labels (I) used by the code generator. The asterisks are followed by the relative address of the instruction, the opcode, and the operand.

Disassembly listing files are typically used

- To look for inefficient instruction sequences that can be improved by changing the source file.
- To rewrite a routine in assembly language. You can write the routine in BASIC first, create an assembly language source file, then use the instructions as a basis for a more efficient assembler routine.
- To learn more about how the compiler works.

The **-A** option applies to source files only; the compiler cannot create an assembly language listing file from an existing object file.

3.8 Producing a Map File

The **-m** option produces a linker "map" file. The map file shows the address of every code and data segment in your program, relative to the start of the program, and lists all the global symbols defined in an object file. The symbols are listed both alphabetically by symbol name, and sorted by symbol address. The addresses of the global symbols are in frame: offset format, showing the location relative to the starting address of the program or module.

You can create a map file from either source or object files. The -m option has the following form:

-m *file*

The link map is placed in *file* in the current directory.

3.9 Generating Debugging and Error-Handling Messages

The -D option forces the compiler to include code that generates debugging and error-handling messages at run time. -D allows use of **TRON** and **TROFF** in the compiled file. If -D is not set, **TRON** and **TROFF** are ignored and a warning is issued.

The -D option causes the BASIC Compiler to generate larger and slower code that checks the following:

- Arithmetic overflow

  All arithmetic operations, both integer and floating-point, are checked for overflow and underflow.

- Array bounds

  All array references are checked to see if subscripts are within the bounds specified in **DIM** statements.

- Line numbers

  The generated binary code includes line numbers so that the run-time error listing can indicate on which line an error occurs.

- **RETURN** statements

  Each **RETURN** statement is checked for a prior **GOSUB** statement.

- CONTROL-BREAK

  After each line the compiler checks to see if the user has pressed CONTROL-BREAK. If CONTROL-BREAK is pressed, the following message appears and program execution stops:

  ```
  *Break*
  ```

```
STOP in line n of module m at address segment:offset
```

If you don't use the **-D** option, array bound errors, **RETURN** without **GOSUB** errors, and arithmetic overflow errors do not generate error messages at compile time or run time. The results may be unpredictable.

If a program is not compiled with the **-D** option, a user cannot exit the program by pressing Control-Break, except when entering data in response to an **INPUT** statement prompt. In this case, to exit the program you must restart your computer.

3.10 Using Error-Handling Statements

The **-E** option tells the compiler that the program contains an **ON ERROR GOTO...RESUME** *line number* construction. To trap errors properly, the compiler must generate extra code for the **GOSUB** and **RETURN** statements. The compiler also generates a line-number address table (one entry per line number) in the binary file, so each run-time error message can include the number of the line in which the error occurs. Do not use this option unless the program contains an **ON ERROR GOTO** statement.

The **-X** option tells the compiler that the program contains one or more **RESUME**, **RESUME NEXT**, or **RESUME 0** statements. If a **RESUME** statement other than **RESUME** *line number* is used with the **ON ERROR GOTO** statement, use the **-X** instead of the **-E** option.

3.11 Writing Quoted Strings to Disk Instead of Memory

The **-S** option forces the compiler to write quoted strings that exceed four characters in length to an object file on disk as they are encountered, instead of retaining them in memory during the compilation of the program. If this option is not set, and the program contains a large number of long quoted strings, the program may run out of memory at compile time.

Although the **-S** option reduces the amount of memory used at compile time, it may *increase* the amount of memory needed in the run-time environment, since multiple instances of identical strings will exist in the program. Without **-S**, references to multiple identical strings are combined

so that only one instance of the string is necessary in the executable program.

3.12 Storing Arrays in Row Order

The compiler normally stores arrays in column order. For example, the element ARRAY(2,1) is followed by ARRAY(3,1). The **-R** option instructs the compiler to store arrays in row order, where the element ARRAY(2,1) would be followed by ARRAY(2,2). This option is useful if you are using assembly-language routines that store arrays in row order.

The BASIC Interpreter stores and accesses arrays in column order.

3.13 Using Assembly Language Source Files

You can use the **bascom** command to create executable programs from a combination of BASIC source files and 80286 assembly language source files. Assembly language source files must contain 80286 instructions, and must have the extension ".s".

When assembly language source files are given, the **bascom** command invokes the XENIX assembler to assemble the instructions and create an object file. The object file can then be linked with object files created by the compiler. For example, the command

```
bascom math.bas add.s
```

compiles the BASIC source file *math.bas* and assembles the assembly language source file *add.s*. The resultant object files, *math.o* and *add.o*, are then linked to form a single executable program, *a.out*.

When using assembly language routines with BASIC programs, be sure to provide the correct interface for calls to and from BASIC language functions. BASIC functions require a specific calling and return sequence. Assembly language functions that fail to call and return properly will cause errors. For information about writing assembly language programs for use with the BASIC compiler, see Chapter 7, "Interfacing With Other Languages."

3.14 Stripping the Symbol Table

The **-s** option forces the linker to strip the symbols from the symbol table. This option is useful when when the program has been completely debugged and local symbols are no longer required. Using the **-s** option produces a smaller executable file.

3.15 BASIC Compiler Messages

The Microsoft BASIC Compiler generates error and warning messages to help you locate potential problems in programs. Error messages are produced for many syntactical and semantic errors. Warnings are produced for some special cases, such as using an inappropriate option with the **bascom** command. If an error is severe, the compiler displays a message and terminates the compilation. Otherwise, the compiler continues looking for other errors, but creates no object file. If only warning messages are displayed, the compiler completes compilation and creates an object file.

In addition to the compiler messages, the **bascom** command displays error messages generated by the XENIX linker. There is a complete list of compiler and linker error messages in Appendix F, "Error Messages."

Chapter 4
Linking Object Modules

4.1 ld Command Syntax 29
4.2 Creating Executable Programs From Object Files 30
4.3 Naming the Output File 30
4.4 Linking to Special Libraries 31
4.5 Producing a Map File 32

The **bascom** command automatically links object modules with modules from the XENIX BASIC library and creates an executable program file. You can also invoke the XENIX linker separately with the **ld** command. The **ld** command is useful if you want to link object files created by separate compilations. This chapter explains how to start the linker and options you can use with **ld** to:

- Create executable programs from object files.
- Give the executable program file a name other than *a.out*.
- Link to functions in libraries.
- Produce a map file.

4.1 ld Command Syntax

The **ld** command has the following form:

ld -P [**-Ml**[**e**]] [**-F** *num*] [**-C**] [**-S** *num*] [**-s**] [**-o** *outputfile*] [**-m** *mapfile*] *objectfiles* [**-l** *library*] [**-l** *library* ...]

where

Argument	Description
-P	Keeps segments defined in an assembly language or compiled BASIC program separate. This argument is required or the program will not run.
-Ml[**e**]	The **-Ml** portion of the argument creates a large model program. The **e** portion of the argument is required when using ISAM libraries; it permits mixed model linking. You must use **-Mle** with ISAM libraries.
-F *num*	Sets the size of the stack to *num* bytes.
-C	Causes the linker to ignore the case of symbols.
-S *num*	Sets the maximum number of data segments to *num*.
-s	Strips local symbols from the symbol table.

-o *outputfile*	assigns the name *outputfile* to the executable file.
-m *mapfile*	produces a linker map file named *mapfile.*
objectfiles	are the names of the files being linked.
-l *library*	specifies other libraries to be linked.

The rest of this chapter explains some of the linker commands in detail.

4.2 Creating Executable Programs From Object Files

To create an executable program, give the names of the object files you wish to link. For example, if the file math.o contains calls to the subprograms *add* and *mult* (saved in the object files *add.o* and *mult.o*), you can create an executable program by typing

```
ld math.o add.o mult.o
```

In this case, *math.o* is linked with *add.o* and *mult.o* to create the executable file *a.out.*

4.3 Naming the Output File

You can give the executable program file any valid filename by using the –o option. The option has the following form:

-o *file*

file is the name of the executable program file.

For example, the command

```
ld -o math math.o add.o
```

causes the compiler to create the executable program file *math* from the object files *math.o* and *add.o.* You can execute this program by typing

```
math
```

Note that the **-o** option does not create an *a.out* file, nor does it affect any existing *a.out* file. This means that the **ld** command does not change the current contents of *a.out* if the **-o** option is given.

4.4 Linking to Special Libraries

If you wish to link to special libraries that are unique to your installation, or to libraries that are not automatically linked by **ld**, you can use the **-l** option. This option directs **ld** to search the given library for the functions called in the source file. If the functions are found, the linker links them to the program file.

The **-l** option can only be used with libraries in the */lib* and */usr/lib* directories. Functions in libraries that are not in the directories */lib* or */usr/lib* are linked when declared as arguments on the **ld** command line.

The option has the form

-l *name*

The **–l** option takes as its argument a truncated filename. Actual library filenames begin with the four letters "Slib," "Mlib," or "Llib," but are referred to in the **–l** option by the letters that appear after this prefix and before the filename extension. ("S," "M," and "L" stand for small, medium and large model. XENIX BASIC only uses large model libraries.)

For example, the command

```
ld math.o -l termlib
```

links the library */usr/lib/Llibtermlib.a* (terminal handling routines) with the source file *math.o*.

The **ld** command searches the libraries in the order they are given until all references are resolved to functions not explicitly defined in your source file. The order in which libraries appear on the command line is significant only if they contain symbols with duplicate names.

4.5 Producing a Map File

The **-m** option produces a linker "map" file. The map file shows the address of every code and data segment in your program, relative to the start of the program, and lists all the global symbols defined in an object file. The symbols are listed both alphabetically by symbol name, and sorted by symbol address. The addresses of the global symbols are in frame: offset format, showing the location relative to the starting address of the program or module.

To create a map file, use the **-m** option on the **ld** command line. You can create a map file from both source and object files. This option has the following form:

-m *filename*

The link map is placed in *filename* in the current directory.

Chapter 5
Working With Files and Devices

5.1 Device-Independent Input/Output 35
5.2 Filenames and Paths 36
5.2.1 Filename Specifications 36
5.2.2 Pathnames 37
5.2.3 Default Directory 37
5.3 Handling Files 37
5.4 Data Files: Sequential, Random Access, and ISAM 37
5.4.1 Sequential Files 37
5.4.2 Random Access Files 41
5.4.3 ISAM Files 47
5.4.3.1 Updating an ISAM File 56
5.4.3.2 Searching in an ISAM File 62
5.4.4 Protecting Files in Multi-User Programs 65
5.4.4.1 Protecting Sequential And Random Files 65
5.4.4.2 Protecting ISAM Files 66
5.4.4.3 Sharing ISAM Files in XENIX 67
5.5 BASIC and Child Processes 69

This chapter discusses the way files and devices are used and addressed in BASIC, and the way information is input and output through the system.

5.1 Device-Independent Input/Output

BASIC provides device-independent input/output that permits a flexible approach to data processing. This generalized device I/O permits the user to access "devices" other than disk files by using the same syntax that Microsoft BASIC uses to access disk files.

This version of Microsoft BASIC supports the following devices:

SCRN:	You can open files for output to this device. The data written to a file opened to SCRN: is directed to the standard output device (the screen). You cannot open a SCRN: file for input.
KYBD:	You can open files to this device for input. The data read from a file opened under this device comes from the standard input device (the keyboard). You cannot open a KYBD: file for output.
LPT1:	You can open files for output to this device. The data written to a file opened to LPT1: is directed to the line printer.
PIPE:*command*	You can use this device to open sequential files for either input or output, or to open random access files for both input and output. When this is done, a pipe is opened, a process is forked, and a specified child process is executed. The *command* parameter is the command or process that input is piped from or output is piped to.

For example

```
OPEN "PIPE:ls" FOR INPUT AS #1
```

permits the directory listing to be accessed as file #1.

For files opened to PIPE:, **LOC(1)** and **LOF(1)** both return 1 if characters are ready to be read from the pipe. If no characters are ready, they both return 0. **EOF** returns −1 (true) if no processes have the pipe open for output and no data is available to be read from the pipe. If the child process is still active, **EOF** returns 0 (false).

The following program opens a sequential input pipe to the XENIX facility "who." The program checks for input before attempting to read the data. This can prevent unexpected "Read past end" errors. Next, the output from "who" is stored in the array WHO$, which is then printed on the screen in lines 100 to 130.

```
10 DIM WHO$(20)
20 OPEN "PIPE:who" FOR INPUT AS #1
30 I = 0
40 WHILE NOT EOF(1)
50       IF LOF(1) = 0 THEN GOTO 90
60       LINE INPUT#1, WHO$(I)
70       I = I + 1
80 WEND
90 I = 0
100 WHILE LEN(WHO$(I)) <> 0
110      PRINT WHO$(I)
120      I = I + 1
130 WEND
140 END
```

5.2 Filenames and Paths

BASIC uses the XENIX hierarchical directory structure, which allows files to be accessed through their pathname.

5.2.1 Filename Specifications

Filename specifications follow XENIX naming conventions. All file specifications can begin with a directory or device specification such as /usr/Basic or LPT1:. If no device is specified, the current directory is assumed. For example,

```
RUN "newfile.bas"
RUN "KYBD:newfile"
```

5.2.2 Pathnames

A pathname is a sequence of directory names followed by a simple filename. A slash (/) separates each component of the pathname from the next.

5.2.3 Default Directory

When a file specification is given (in commands or statements such as **FILES**, **OPEN**, and **LOCK**), the default (current) directory is the directory from which the BASIC compiler was invoked.

5.3 Handling Files

File I/O procedures for the beginning BASIC user are examined in this section. If you are new to BASIC, or if you are encountering file-related errors, read through these procedures and program examples to make sure you are using all the file statements correctly.

5.4 Data Files: Sequential, Random Access, and ISAM

BASIC programs can create and use three types of data files: sequential, random access, and ISAM (Indexed Sequential Access Method) files.

5.4.1 Sequential Files

Sequential files are the easiest to create, but are limited in flexibility and speed when it comes to locating data. The data written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order sent. The data is read back sequentially, in the same order as written.

The following statements and functions are used with sequential data files:

Statements	Functions
CLOSE	**EOF**
INPUT#	**LOC**

INPUT$ **LOF**
LINE INPUT#
LOCK
OPEN
PRINT#
PRINT USING#
UNLOCK
WIDTH
WRITE#

Creating a Sequential File

Program 1 creates a sequential file, "DATA," from information you input at the keyboard.

Program 1—Create a Sequential Data File

```
10 OPEN "DATA" FOR OUTPUT AS #1
20 LINE INPUT "NAME";N$
30   IF N$ = "DONE" THEN CLOSE#1:END
40 LINE INPUT "DEPARTMENT";DEPT$
50 LINE INPUT "DATE HIRED";HIREDATE$
60 PRINT #1,N$;",";DEPT$;",";HIREDATE$
70 PRINT
80 GOTO 20
```

When the program is executed, a sample session might look like this:

```
NAME? SAMUEL GOLDWYN
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? MARVIN HARRIS
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? DEXTER HORTON
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/81

NAME? STEVEN SISYPHUS
DEPARTMENT? MAINTENANCE
```

```
DATE HIRED? 08/16/81

NAME? DONE
```

As illustrated in Program 1, the following program steps are required to create a sequential file:

1. **OPEN** the file in **OUTPUT** mode.
2. Write data to the file using the **PRINT#** statement. (**WRITE#** can be used instead.)
3. **CLOSE** the file so that data can be written to it. You cannot read data from a sequential file that is opened for output.

Reading Data From a Sequential File

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1981.

Program 2—Accessing a Sequential File

```
10 OPEN "DATA" FOR INPUT AS #1
15 WHILE NOT EOF(1)
20      INPUT#1,N$,DEPT$,HIREDATE$
30      IF RIGHT$(HIREDATE$,2) = "81" THEN PRINT N$
40 WEND
50 END
```

When the program is executed, the output might look like this:

```
DEXTER HORTON
STEVEN SISYPHUS
```

Program 2 reads, sequentially, every item in the file, and prints the names of employees hired in 1981. When all the data has been read, the program exits the **WHILE** loop and displays the system prompt.

A program that creates a sequential file can also write formatted data to the file with the **PRINT# USING** statement. For example, the statement

```
PRINT#1,USING"####.##,":A,B,C,D
```

could be used to write numeric data to the file without the explicit comma delimiters shown in the example. The commas at the end of the format string separate the items in the file.

If you want commas to appear in the file as delimiters between variables, use the **WRITE#** statement. For example, the statement

```
WRITE#1, A, B$
```

can be used to write these two variables to the file with commas automatically separating them.

The **LOC** function, when used with a sequential file, returns the number of records that have been written to or read from the file since it was opened. A record is a 128-byte block of data.

Adding Data to a Sequential File

If you have a sequential file residing on disk and want to add more data to the end of it, you cannot simply open the file in **OUTPUT** mode and start writing data. As soon as you open an existing sequential file in the output mode, you destroy its current contents.

Instead, use **APPEND** mode. If the file doesn't already exist, the open statement will work exactly as it would if output mode had been specified.

The following procedure can be used to add data to an existing file called "FOLKS":

Program 3—Adding Data to a Sequential File

```
110 OPEN "FOLKS" FOR APPEND AS #1
120 REM ADD NEW ENTRIES TO FILE
130  LINE INPUT "NAME";N$
140  IF N$="" THEN 210 'CARRIAGE RETURN EXITS INPUT LOOP
150  LINE INPUT "ADDRESS? ";ADDR$
160  LINE INPUT "BIRTHDAY? ";BIRTHDATE$
170  PRINT#1,N$
180  PRINT#1,ADDR$
190  PRINT#1,BIRTHDATE$
200 GOTO 120
210 CLOSE #1
220 END
```

5.4.2 Random Access Files

Creating and accessing random access files requires more program steps than creating and accessing sequential files. There are, however, advantages to using random access files. One advantage is that random access files require less room on the disk, since BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage of using random access files is that data can be accessed randomly, i.e., anywhere in the file. It is not necessary to read through all the information from the beginning of the file, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, each of which is numbered.

The following statements and functions are used with random access files:

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

Creating a Random Access File

To create a random access file, you must take certain steps that are not required to create a sequential file, as shown in Program 4:

Program 4—Create a Random File

```
10 OPEN "FILE" AS #1
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
25 LET CONTINUE$ = "Y"
30 WHILE LEFT$(CONTINUE$,1) = "Y"
35    LINE INPUT "2-DIGIT CODE";CODE%
40    LINE INPUT "NAME";PERSON$
50    LINE INPUT "AMOUNT";AMOUNT
```

```
60     LINE INPUT "PHONE";TELEPHONE$
65     PRINT
70     LSET N$=PERSON$
80     LSET A$=MKS$(AMOUNT)
90     LSET P$=TELEPHONE$
100    PUT #1,CODE%
110    LINE INPUT "Enter Another? ", CONTINUE$
120 WEND
130 END
```

As illustrated in Program 4, the following program steps are required to create a random access file:

1. **OPEN** the file for random access. The following example specifies a record length of 32 bytes:

   ```
   10 OPEN "FILE" AS #1 LEN=32
   ```

2. Use the **FIELD** statement to allocate space in a random buffer for the variables that will be written to the random access file:

   ```
   20 FIELD #1,20 AS N$,4 AS A$,8 AS P$
   ```

3. Use **LSET** to move the data into the random access buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: **MKI$** to make an integer value into a string, **MKS$** to make a single precision value into a string, and **MKD$** to make a double precision value into a string:

   ```
   70     LSET N$=PERSON$
   80     LSET A$=MKS$(AMOUNT)
   90     LSET P$=TELEPHONE$
   ```

4. Write the data from the buffer to the file using the **PUT** statement. You can use a variable to specify which number record you wish to insert, as in this example:

   ```
   100    PUT #1,CODE%
   ```

Program 4 takes information typed at the keyboard and writes it to a random access file. Each time BASIC executes the **PUT** statement, it writes a record to the file. The two-digit code that is input in line 35 becomes the

record number.

Warning

Do not use a fielded string variable in an **INPUT** or **LET** statement. Doing so causes that variable to be redeclared. BASIC will no longer associate that variable with the file buffer, but with the new program variable.

Accessing a Random Access File

Program 5 accesses the random access file **FILE** that was created in Program 4. By entering a three-digit code at the keyboard, you can display and read the information associated with that code.

Program 5—Access a Random File

```
10 OPEN "FILE" AS #1 LEN=32
15 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
20 LET ANSWER$ = "YES"
25 WHILE LEFT$(ANSWER$,1) = "Y"
30   LINE INPUT "2-DIGIT CODE";CODE%
40   GET #1, CODE%
50   PRINT N$
60   PRINT USING "$$###.##";CVS(A$)
70   PRINT P$:PRINT
80   LINE INPUT "More? ", ANSWER$
90 WEND
```

As shown above, the following program steps are required to access a random access file:

1. **OPEN** the file in "R" mode. For example,

   ```
   10 OPEN "FILE" AS #1 LEN=32
   ```

2. Use the **FIELD** statement to allocate space in the random access buffer for the variables that will be read from the file. For example,

   ```
   15 FIELD #1,20 AS N$,4 AS A$,8 AS P$
   ```

Note

In a program that performs both input and output on the same random access file, you can often use just one **OPEN** statement and one **FIELD** statement.

3. Use the **GET** statement to move the desired record into the random access buffer. For example,

```
40 GET #1,CODE%
```

4. The data in the buffer can now be accessed by the program. Numeric values that were converted to strings by the **MKS$**, **MKD$**, or **MKI$** statements must be converted back to numbers using the "convert" functions: **CVI** for integers, **CVS** for single precision values, and **CVD** for double precision values. The **MKI$** and **CVI** processes mirror each other, the former converting a number into a format for storage in random files, the latter converting the random file storage into a format usable by the program. For example,

```
50   PRINT N$
60   PRINT USING "$$####.##";CVS(A$)
```

When used with random access files, the **LOC** function returns the "current record number." The current record number is the last record number that was used in a **GET** or **PUT** statement. For example, the statement

```
IF LOC(1) > 50 THEN END
```

ends program execution if the current record number in file number 1 is greater than 50.

Random File Operations

Program 6 is an inventory program that illustrates random file access.

Program 6—Inventory

```
120 OPEN "INVEN.DAT" AS #1 LEN=39 :REM* Open the file
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
    :REM* Set up Buffer
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT "1,INITIALIZE FILE"
140 PRINT "2,CREATE A NEW ENTRY"
150 PRINT "3,DISPLAY INVENTORY FOR ONE PART"
160 PRINT "4,ADD TO STOCK"
170 PRINT "5,SUBTRACT FROM STOCK"
180 PRINT "6,DISPLAY ALL ITEMS BELOW REORDER LEVEL"
190 PRINT "7,EXIT PROGRAM"
220 PRINT:PRINT:LINE INPUT"Choice? ",FUNCTION
225 IF (FUNCTION < 1) OR (FUNCTION > 7) THEN PRINT
    "BAD FUNCTION NUMBER":GOTO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680,1000
240 GOTO 220
250 REM ** BUILD NEW ENTRY **
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT "OVERWRITE"; ADDR$: _
    IF ADDR$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESCRIPTION$
300 LSET D$=DESCRIPTION$
310 LINE INPUT "QUANTITY IN STOCK";QUANTITY%
320 LSET Q$=MKI$(QUANTITY%)
330 LINE INPUT "REORDER LEVEL";REORDER%
340 LSET R$=MKI$(REORDER%)
350 LINE INPUT "UNIT PRICE";PRICE
360 LSET P$=MKS$(PRICE)
370 PUT#1,PART%
380 RETURN
390 REM ** DISPLAY ENTRY **
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:LINE INPUT "QUANTITY TO ADD ";ADDITIONAL%
520 Q%=CVI(Q$)+ADDITIONAL%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
```

```
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 LINE INPUT "QUANTITY TO SUBTRACT";LESS%
610 Q%=CVI(Q$)
620 IF (Q%-LESS%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-LESS%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%; _
    " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710   GET#1,I
720   IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY"; _
      CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 LINE INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER": _
    GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 LINE INPUT "ARE YOU SURE";CONFIRM$:IF CONFIRM$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940   PUT#1,I
950 NEXT I
960 RETURN
000 REM *Exit Routine*
020 CLOSE
040 END
```

In this program, the record number is used as the part number. It is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the various inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

5.4.3 ISAM Files

ISAM files use the most sophisticated and powerful file organization provided by Microsoft BASIC. These additional capabilities add additional complexities. For this reason, we advise only experienced programmers to attempt using ISAM files in their BASIC programs.

ISAM enables you to use records sequentially or randomly, and provides you with the ability to quickly find all records that meet a desired criterion. ISAM files enable you to produce complex database-type application programs in Microsoft BASIC.

About the ISAM Utility

The ISAM software provided with this product is external to BASIC. It is written in assembly language so that it can work with other Microsoft language products, and each of the ISAM functions and statements are accessed through assembly language calls. ISAM, like other assembly language routines, cannot take advantage of any BASIC error trapping you have designed for your programs. ISAM has its own routines to identify possible errors and the status of procedure calls.

ISAM File Construction

Each ISAM file you use is actually *two* different files. The first is the data file, the actual information you want to use. The second file is the "key file."

Like BASIC random access and sequential files, an ISAM data file consists of data records. Unlike a BASIC data file, however, an ISAM file has a "data dictionary" at the beginning of the file. The data dictionary holds coded information about the contents of the data file. The coded information tells BASIC the size and type of the fields in the data file.

The key file holds the index information about the data file: it is similar to the index of a book. Each file record has a pointer in the key file, just as each subject in a book has its own "listing" in the index. BASIC uses this information to find records in the data file. You can use ISAM for speedy database-type applications because you can "key" each record with an identification or trait.

Compare this with a sequential or random file. With a random file, if you want to search for the first record with a certain attribute (Name = "Amalgamated Transport" or Debt > 1000, for example) you perform series of steps. You might build a test loop and check each record to see if it matches the condition you are looking for. You could follow with a set of statements to load the data into program variables.

On the other hand, with an ISAM file you create a key for any attribute (possibly a field for company name or credit limit) you are going to search for. Then you tell ISAM to find a specific record or the first record that matches your search criteria. The program needn't test each record to see if it matches; you can use ISAM to select records for you. Because of the information ISAM stores in the key file, BASIC can find specific records more quickly than it can look at and test individual records. This process is like using the table of contents to find a chapter in a book rather than leafing through each page to look for the title.

Data File Record Types

There are two kinds of data records: segmented records and non-segmented records. A data file can have one type or the other, but not both.

Non-segmented records contain key fields that are of uniform length. A non-segmented record can have *one* variable length key field if, and only if, the field is the last field in the record. Do not confuse a variable length key with a variable length record.

Note

Use non-segmented records whenever possible. This makes file creation and interfacing ISAM files with your BASIC programs much simpler.

A segmented record can have variable length fields. They are most frequently used for key fields that are strings, such as city names (you don't always know the length of the data), or in records requiring the most efficient use of file space.

Keys

You use the ISAM key to uniquely identify a particular record in the file. Once identified, you can quickly find a record in the ISAM file by specifying the key.

There are two key forms: regular keys and split keys. A regular key is best used when one field sufficiently identifies a record. An example of this is a social security number. Only one person can have a given social security number.

Split keys consist of non-contiguous fields that can be of different types. They are best used when duplication might occur in a key field and uniqueness is important. For example, if you have a data file of major league ball players, you could have a Reggie Smith, a Lonnie Smith, a Spectacular Smith and an Ozzie Smith. You would want to have first names and last names as separate data fields and have the pair of fields combine to be a split key.

By having separate fields you can do alphabetical listings that require last name and then first name, and mailing labels that require first name followed by last name. At the same time, by using a split key you can find a specific ball player because, while last and first names are separate fields, they constitute one key.

It is useful to be able to quickly search a large data file and get the record for the person you want. But in this file you can't merely search for a record with last name field of "Smith"; it could easily be the wrong Smith. Searching by just first name has the same consequence.

As in the previous case, split keys contain more than one field. In the preceding example, the split key includes two fields: the field you use for first names and the one you use for last names. This combination of the two fields, a split key, limits the number of erroneous Smiths that you read from your data file.

Split keys can use fields that are either adjacent or not adjacent. The fields can be of the same or different data types, and the fields can be either regular keys or non-regular keys.

Key Handles

Because more than one key field can be in a record, a program passing information about a key to ISAM must identify the key. The key handle is a number that identifies a particular key within a record. You use the key handle in statements, for example, to find specific records.

You assign as many key handles as there are keys. Give split keys, regardless of the number of fields that constitute the key, one key handle.

The key handle numbers needn't correspond to the order of key fields within the data file: key handle 1 could be the third field in the record, while key handle 2 could be the first field in the record.

Using ISAM Files in Your Programs

Writing applications that use ISAM files is very different from writing those that use BASIC's sequential or random files. This section shows small examples of Microsoft BASIC programs that use ISAM files. Each example shows a cross section of the commands used to create and use ISAM files. For a detailed and complete description of ISAM file topics, see Appendix D, "ISAM Reference." You will probably want to consult the appendix as you follow the program examples.

Building a Program that Uses ISAM

Before you start writing code for a program that uses ISAM files, you should completely define the contents, field sizes and keys you want to create for the ISAM data file. What keys will you use? Will there be split keys? Will the records be non-segmented or segmented?

Once your file design plan is polished, write a program that creates an ISAM file.

Creating an ISAM File

The following BASIC program sets up the ISAM routines and then opens a file by calling the **IOPEN** routine. When creating an ISAM data file, the program must fully describe the attributes of both the record and the key. After the program gives the descriptions to ISAM, it opens the file with the **IOPEN** call. The **IOPEN** call specifies the address of the stored file name, the file mode, the memory location at which the record description begins, and the memory location at which the key description begins. It also contains a variable to which ISAM returns a number; that number is used by

ISAM to keep track of the file. The program then prompts for customer information, which it stores in the ISAM file.

Program 1 — Creating an ISAM File

```
100 REM ** Initialize **
110  DEFINT A-Z
115  LET FILENO = 0
120  DIM RECDESCRIPTION(3):   DIM KEYDESIGNATION(9)
160  OPEN "/dev/null" AS #2 LEN = 68
170   FIELD #2, 4 AS CUSTOMERNO$, 20 AS CUSTNAME$, 20 AS ADDRESS$, _
        15 AS CITY$, 5 AS ZIP$, 4 AS OWE$
180 REM ** Record Descriptor Set-up **
200  LET RECDESCRIPTION(1) = 1: REM** This variable sets up # of Keys.
220  LET RECDESCRIPTION(2) = 0: REM** This declares record "non-segmented".
230  LET RECDESCRIPTION(3) = 0: REM** Minimum allocation.
240  LET FIELDNAME$ = "CUSTOMER NUMBER"
260 REM ** Key Descriptor Set-ups **
280 REM ** The key is customer number **
300  LET KEYDESIGNATION(1) = VARPTR(FIELDNAME$)
310   REM** Points to field name.
320  LET KEYDESIGNATION(2) = 0
330   REM** Future feature. This must be set to 0.
340  LET KEYDESIGNATION(3) = (0*256) + 5
350   REM** First byte 0, second is 5(data type=string)
360  LET KEYDESIGNATION(4) = 1
370   REM** If record is non-segmented, = 1.
380  LET KEYDESIGNATION(5) = 1
390   REM** Position of field within record = 1.
400  LET KEYDESIGNATION(6) = 4
410   REM** Size of key in bytes...
420  LET KEYDESIGNATION(7) = 1
430   REM** Key Handle #. This is first, so #1
440  LET KEYDESIGNATION(8) = (0*256) + 1
450   REM** First byte is 0, second is 1.
460  LET KEYDESIGNATION(9) = 0
470   REM** Future feature. This must be set to 0.
480 REM ** Now give subject data file a name **
500  LET FILENAME$ = "CUSTOMER.DAT"
600  LET RECORDSIZE = 68
650  REM   **  Open the file in Create Mode  **
890 CALL IOPEN(VARPTR(FILENAME$),3,VARPTR(RECDESCRIPTION(1)), _
                    VARPTR(KEYDESIGNATION(1)), FILENO)
900 REM ** Check File Status **
920     IF IXSTAT < > 0 THEN PRINT "Failed to open file": END
940 REM ** Now prompt for data entry **
980        LINE INPUT "4-Digit Customer Number('0' to stop) ",CUSNO$
990        IF CUSNO$="0000" OR CUSNO$="0" THEN GOTO 1640
1000        LSET CUSTOMERNO$ = CUSNO$
```

```
1020       LINE INPUT "Customer Name ", CUSNM$
1040        LSET CUSTNAME$ = CUSNM$
1060       LINE INPUT "Street Address ", ADDR$
1080        LSET ADDRESS$ = ADDR$
1100       LINE INPUT "City & State ", POLIS$
1120        LSET CITY$ = POLIS$
1140       LINE INPUT "Zip code ", ZCODE$
1160        LSET ZIP$ = ZCODE$
1180       LINE INPUT "Balance Owed  $", OWES!
1200        LSET OWE$ = MKS$(OWES!)
1400      REM ** Now try to write record **
1460          CALL IWRITE(FILENO, SADD(CUSTOMERNO$), RECORDSIZE)
1470          IF IXSTAT = 0 THEN GOTO 940
1480          IF IXSTAT <> 13 THEN GOTO 1600
1500          PRINT "Customer # is duplicate: reenter";
1520          LINE INPUT CUSNO$: LSET CUSTOMERNO$ = CUSNO$
1540          GOTO 1400
1580      REM ** End of Entry Loop **
1600 REM
1601 REM ** Handle unexpected error **
1605    PRINT "UNEXPECTED ERROR ON THE WRITE--IXSTAT= ";IXSTAT
1620 REM ** Now close the new file **
1640 CALL ICLOSE(FILENO)
1660 END
```

The following program shows a simple file creation using ISAM. It creates an ISAM file with one key: customer number. Lines 120 to 170 dimension the array variables that ISAM uses, and set aside an area of memory for the field variables.

```
120  DIM RECDESCRIPTION(3):   DIM KEYDESIGNATION(9)
160  OPEN "/dev/null" AS #2 LEN = 68
170   FIELD #2, 4 AS CUSTOMERNO$, 20 AS CUSTNAME$, 20 AS ADDRESS$, _
          15 AS CITY$, 5 AS ZIP$, 4 AS OWE$
```

ISAM requires the definition of heterogeneous fields—fields with different sizes and types. BASIC's strongest facility for doing this is the **FIELD** statement. This sets aside an area that you can use to define your data fields. To use the **FIELD** statement, you must first **OPEN** a file with that number. The file in this case is /dev/null, because this file will never be used as a file. It is being used so a **FIELD** can be defined for ISAM.

In lines 200 to 230, the program assigns values to three ISAM variables:

RECDESCRIPTION(1)	tells ISAM how many keys the data file will have.
RECDESCRIPTION(2)	tells ISAM whether the file is segmented or non-segmented.
RECDESCRIPTION(3)	tells ISAM how many bytes to allocate for the record. If this is set to 0, ISAM allocates the default, 8 bytes per record.

Lines 300 to 470 assign values to the KEYDESIGNATION array. This array describes information about the key to this file. Table 4.1 describes the significance of each array element. Seven of the nine array variables contain one or two specific pieces of information about the key. Two of them give no information, and you must set them to zero. The actual name you use for the array is not important. ISAM locates the information because the `VARPTR(KEYDESIGNATION(1))` argument to the **IOPEN** call in line 890 holds the pointer to the beginning of the array. ISAM copies the entire array during the **IOPEN** call.

Table 5.1
Key Designation Assignments

Variable	Description
KEYDESIGNATION(1)	The memory location of the field name.
KEYDESIGNATION(2)	This spot is saved for future use. For now, set it to zero.
KEYDESIGNATION(3)	Two different things. The first byte in this two-byte integer variable is used for future features. For now, you must set it to zero. The second byte describes the data type of the field.
KEYDESIGNATION(4)	The number of a segmented field which holds the field. For the usual form of ISAM file, non-segmented, you always set this to one.
KEYDESIGNATION(5)	The offset from the beginning of the segment to the beginning of the key field in a segmented record. For the usual form of ISAM file, non-segmented, you always set this to one.
KEYDESIGNATION(6)	The number of bytes in the key field. If you are using segmented records with variable length key fields, you should set this number to zero.
KEYDESIGNATION(7)	The number of the key's handle.
KEYDESIGNATION(8)	Two different things. The first byte in this two-byte integer variable sets option flags. The second byte, the "low byte," tells if the field is part of a key and whether or not the key is split or not.
KEYDESIGNATION(9)	This spot is saved for future use. For now, set it to zero.

See Appendix D, "ISAM Reference," for details on each of the KEYDESIGNATION array variables.

Line 500 gets the pointer to a memory location that the **IOPEN** call uses later. Line 600 assigns the size of the record:

```
500  LET FILENAME$ = "CUSTOMER.DAT"
600  LET RECORDSIZE = 68
```

Line 890 calls the ISAM routine, **IOPEN**:

```
890 CALL IOPEN(VARPTR(FILENAME$), 3, VARPTR(RECDESCRIPTION(1)), _
                    VARPTR(KEYDESIGNATION(1)), FILENO)
```

The arguments to **IOPEN** are

- The pointer to a variable that holds the pathname of the ISAM data file.
- A number that represents the mode of the file. Modes include read-only, create-and-write-only, read-and-write, and create-and-read-and-write. In this case, the argument is 3, which is write.
- A pointer to the start of the record description.
- A pointer to the start of the key description.
- A variable that will hold a number. ISAM assigns a file number to the file, and returns it here. The program needs it later to identify the file. Notice that line 115 initializes the variable. You must initialize a variable you will be using to take a function value.

Line 920 tests the ISAM status variable to check if the file is actually open:

```
920      IF IXSTAT < > 0 THEN PRINT "Failed to open file": END
```

Lines 960 through 1200 are the data entry and size verification loop. The operator exits this loop by entering 0000 as the customer number. Note that the BASIC operations in the loop do not use the field variable names. They use program variables that, when verified for proper size, are loaded

(**LSET**) into the field variables.

Lines 1400 to 1540 make the part of the loop that checks to see if a record with the key exists. If it does already exist, the program prompts the user to enter a new number:

```
1400   REM ** Now try to write record **
1460       CALL IWRITE(FILENO. SADD(CUSTOMERNO$), RECORDSIZE)
1470       IF IXSTAT = 0 THEN GOTO 940
1480       IF IXSTAT <> 13 THEN GOTO 1600
1500       PRINT "Customer # is duplicate; reenter";
1520       LINE INPUT CUSNO$: LSET CUSTOMERNO$ = CUSNO$
1540       GOTO 1400
```

Line 1640 closes the data file by calling ISAM's **ICLOSE** routine:

```
1640 CALL ICLOSE(FILENO)
```

The program specifies a single argument when calling **ICLOSE**. The argument is the file number that ISAM passed to the program at the **IOPEN** call.

5.4.3.1 Updating an ISAM File

Once you have a data file, the next thing you are likely to do is update it. You might want to add or delete records, or change information in an existing record. Although the ISAM calls for these procedures differ from those used to create ISAM files, the processes are similar.

You first open the old file using the **IOPEN** statement. Because you are opening an existing file, some of the information passed in the **IOPEN** call in the preceding program is unnecessary. The pointers to the beginnings of the record description and key description are not required, because both descriptions are already in the data dictionary in the ISAM data file.

Then you look for the record to be changed.

To search for records	use the **ISEEK** call
To add records	use the **IWRITE** call (see Program)
To delete records	use the **IDELETE** call
To change records	use the **IREWRITE** call

Internal ISAM procedures for writing new records are different than those for changing existing records—just as changing a book index entry is

different than adding a new entry.

The following BASIC program allows a user to update ISAM file records:

Program 2—Main Menu

```
100 REM ** Initialize **
105  DEFINT A-Z
110  LET ZERO=0: LET FILENO=0
120  LET FILENAME$ = "CUSTOMER.DAT"
150  RECORDSIZE=68
160  OPEN "/dev/null" AS #2 LEN = 68
180   FIELD #2, 4 AS CUSTOMERNO$, 20 AS CUSTNAME$, 20 AS ADDRESS$, _
            15 AS CITY$, 5 AS ZIP$, 4 AS OWE$
210  REM ** Open the file **
220  CALL IOPEN(VARPTR(FILENAME$),2,VARPTR(ZERO),VARPTR(ZERO), FILENO)
230  IF IXSTAT <> 0 THEN PRINT "ERROR ON OPEN. IXSTAT=";IXSTAT: GOTO 4020
280 REM ** Provide a menu of update choices **
300  LET CHOICE% = 0
320  WHILE (CHOICE% < 1 OR CHOICE% > 4)
340    CLS
360    LOCATE 3, 22: PRINT "*****Update Customer Information*****"
380    LOCATE 5, 22: PRINT "         1. Add A Record"
400    LOCATE 6, 22: PRINT "         2. Delete A Record"
420    LOCATE 7, 22: PRINT "         3. Change A Record"
440    LOCATE 8, 22: PRINT "         4. QUIT"
460    LOCATE 9, 22: LINE INPUT "Enter the number(1,2,3 or 4)of your choice ",CHOICE%
480  WEND
500 REM ** Route to appropriate section of code**
520  ON CHOICE% GOTO 1000, 2000, 3000, 4000
560 REM ** End of main
580 REM **  menu routine.
```

Program 2 has subroutines that show the three main updates to an existing file: additions, deletions, and changes.

The main module first sets up a field buffer for the information. It does this by opening a dummy file and using a **FIELD** statement:

```
160  OPEN "/dev/null" AS #2 LEN = 68
180  FIELD #2, 4 AS CUSTOMERNO$, 20 AS CUSTNAME$, 20 AS ADDRESS$, _
            15 AS CITY$, 5 AS ZIP$, 4 AS OWE$
```

Again, you use /dev/null as a dummy file, so you can use the convenient **FIELD** statement.

Then the program opens the ISAM file by calling the **IOPEN** routine:

```
220 CALL IOPEN(VARPTR(FILENAME$),2,VARPTR(ZERO),VARPTR(ZERO),FILENO)
```

The arguments to **IOPEN** are

- The pointer to a variable that holds the pathname of the ISAM data file.
- A number that represents the mode of the file. Modes include read-only, create-and-write-only, read-and-write, and create-and-read-and-write. In this case, the argument is 2, which is read-and-write.
- A pointer to the start of the record description. This is set to 0 with the variable VARPTR(ZERO) (the pointer to a variable that holds a 0), because you need this argument only when you create a data file. Here, you are not creating a file, but using an existing one.
- A pointer to the start of the key description. This is set to VARPTR(ZERO), for the same reason as the record description.
- A variable which will hold the file number ISAM assigns. The program needs it later to identify the file.

The program then gives a user a chance to add, delete or change records.

Program 2—Subroutine to Add a Record

```
1000 REM ** This routine adds a record to the file **
1020  LET CONTINUE$ = "Y"
1040  WHILE CONTINUE$ = "Y" OR CONTINUE$ = "y"
1060   CLS
1080   LINE INPUT "4-Digit Customer Number('0' to stop) ",CUSNO$
1100    LSET CUSTOMERNO$ = CUSNO$
1120   LINE INPUT "Customer Name ", CUSNM$
1140    LSET CUSTNAME$ = CUSNM$
1160   LINE INPUT "Street Address ", ADDR$
1180    LSET ADDRESS$ = ADDR$
1200   LINE INPUT "City & State ", POLIS$
1220    LSET CITY$ = POLIS$
1240   LINE INPUT "Zip code ", ZCODE$
1260    LSET ZIP$ = ZCODE$
1280   LINE INPUT "Balance Owed  $", OWES!
1300    LSET OWE$ = MKS$(OWES!)
1320   LET CONDITION% = (-1)
1340   WHILE CONDITION%
1360     CALL IWRITE(FILENO,SADD(CUSTOMERNO$), RECORDSIZE)
1370  IF IXSTAT = 0 THEN DUPLICATE%=0: GOTO 1440
```

```
1375  IF IXSTAT<>13 THEN PRINT "ERROR WRITING A NEW RECORD,IXSTAT=";IXSTAT:GOTO 4020
1380     IF IXSTAT = 13 THEN DUPLICATE% = (-1) ELSE DUPLICATE% = 0
1400      IF DUPLICATE% THEN PRINT "Customer # is duplicate; reenter";
1420      IF DUPLICATE% THEN INPUT CUSNO$: LSET CUSTOMERNO$ = CUSNO$
1440      IF DUPLICATE% = 0 THEN LET CONDITION% = 0
1460   WEND
1480   REM ** Check for more **
1500    LINE INPUT "Add Another (Y or N)? ", CONTINUE$
1520  WEND
1560  GOTO 300
```

If the choice is to add records, program control transfers to the subroutine that starts at line 1000. The routine collects the information and then writes it to the file:

```
1360      CALL IWRITE(FILENO,SADD(CUSTOMERNO$), RECORDSIZE)
```

The arguments to the **IWRITE** routine are

- The number of the file. The **IOPEN** routine returns this number.
- The memory location of the beginning of the key.
- The length of the data record.

The program then tests an ISAM variable, **IXSTAT**, to check if the attempted write operation was stopped because the key duplicates an existing record:

```
1380      IF IXSTAT = 13 THEN DUPLICATE% = (-1) ELSE DUPLICATE% = 0
```

Program 2—Subroutine to Delete a Record

```
2000 REM ** This routine deletes a record from the file **
2020  LET CONTINUE$ = "Y"
2040  WHILE CONTINUE$ = "Y" OR CONTINUE$ = "y"
2060     CLS
2080     LINE INPUT "Delete customer number #",  CUSNO$
2100     LSET CUSTOMERNO$ = CUSNO$
2120     CALL ISEEK(FILENO,1,SADD(CUSTOMERNO$),4,2)
2125 IF IXSTAT = 0 THEN GOTO 2140
2130 IF IXSTAT = 10 THEN PRINT "RECORD NOT FOUND; PLEASE RE-ENTER": GOTO 2080
2135 IF IXSTAT <> 10 THEN PRINT _
     "ERROR SEEKING RECORD TO DELETE, IXSTAT=";IXSTAT: GOTO 4020
2140     CALL IDELETE(FILENO)
2150 IF IXSTAT <> 0 THEN PRINT "ERROR DELETING RECORD--IXSTAT=";IXSTAT: GOTO 4020
2160     PRINT "Deleted customer #";CUSNO$;" record"
2180     LINE INPUT "Delete another (Y or N)? ", CONTINUE$
```

```
2200  WEND
2240  GOTO 300
```

If the choice is to delete records, the program shifts control to the subroutine that starts at 2000. The subroutine prompts for the customer number and then seeks the record specified:

```
2120      CALL ISEEK(FILENO,1,SADD(CUSTOMERNO$),4,2)
```

The five parameters of the **ISEEK** routine are

- The file number. This is returned by the **IOPEN** routine.
- The key handle for your search. You assign the key handle (number) when you create the ISAM file.
- The pointer to the buffer that holds the value of the key to search for. In this program, **ISEEK** is aiming for the record specified in **CUSTOMERNO$**.
- The size of the key in bytes.
- The seek mode. Seek modes include first-record-with-key, last-record-with-key, record-equal-to-key, record-greater-than-key and key-greater-than-or-equal-to-key.

When it finds the record, it then calls the ISAM routine for deletion:

```
2140      CALL IDELETE(FILENO)
```

Program 2—Subroutine to Change a Record

```
3000 REM ** This routine changes a record in the file **
3020 LET CONTINUE$ = "Y"
3040 WHILE CONTINUE$ = "Y" OR CONTINUE$ = "y"
3080     CLS
3120        LOCATE 2, 12
3140        LINE INPUT  "Change info on customer number #", FIND$
3150     LSET CUSTOMERNO$ = FIND$
3160        CALL ISEEK(FILENO,1,SADD(CUSTOMERNO$),4,2)
3165     IF IXSTAT = 0 THEN  GOTO 3195
3180        IF IXSTAT=10 THEN BEEP: LOCATE 5,12
3185        IF IXSTAT=10 THEN PRINT FIND$;"Not Found. Try Again.": GOTO 3120
3190     PRINT "ERROR SEEKING RECORD TO CHANGE--IXSTAT=";IXSTAT: GOTO 4020
3195     CALL IREAD(FILENO,SADD(CUSTOMERNO$),RECORDSIZE)
3200     IF IXSTAT <> 0 THEN PRINT _
            "ERROR READING RECORD -IXSTAT=";IXSTAT: GOTO 4020
3260      CLS: LOCATE 1,12: PRINT "***Customer #";CUSTOMERNO$;"***"
3280      PRINT "Press the return key to leave old info."
```

```
3380     LINE INPUT "Change Customer Name (";CUSTNAME$;") to ", CUSNM$
3400     IF CUSNM$ <> "" THEN LSET CUSTNAME$ = CUSNM$
3500    LINE INPUT "Change Street Address to ", ADDR$
3520     IF ADDR$ <> "" THEN LSET ADDRESS$ = ADDR$
3580     LINE INPUT "Change City & State to ", POLIS$
3700     IF POLIS$ <> "" THEN LSET CITY$ = POLIS$
3800     LINE INPUT "Change Zip code to", ZCODE$
3810     IF ZCODE$ <> "" THEN  LSET ZIP$ = ZCODE$
3860     LINE INPUT "Balance Owed  $", OWES!
3870     IF OWES! <> 0 THEN LSET OWE$ = MKS$(OWES!)
3920   CALL IREWRITE(FILENO,SADD(CUSTOMERNO$),RECORDSIZE)
3925   IF IXSTAT <> 0 THEN PRINT _
        "UNEXPECTED ERROR DURING REWRITE--IXSTAT="; IXSTAT: GOTO 4020
3930     REM ** Check for more **
3940     LINE INPUT "Change Another (Y or N)? ", CONTINUE$
3950   WEND
3970   GOTO 300
```

If the choice is to change information in existing records, the program routes control to the subroutine that begins at line 3000. The program prompts for the customer number and seeks the record:

```
3160          CALL ISEEK(FILENO,1,SADD(CUSTOMERNO$),4,2)
```

Then it prompts for information on each field in the record. With the updated information in the buffer, the program uses the **IREWRITE** function to put in the new data:

```
3920   CALL IREWRITE(FILENO,SADD(CUSTOMERNO$),RECORDSIZE)
```

Whenever you change an existing record, use the **IREWRITE** function and not the **IWRITE** function.

Program 2—Program Exit Statements

```
4000 CLS
4020 CALL ICLOSE(FILENO)
4040 END
```

The program closes the ISAM file before ending, using the **ICLOSE** routine:

```
4020 CALL ICLOSE(FILENO)
```

5.4.3.2 Searching in an ISAM File

When you have an ISAM file, it is probable that you want to access records without updating them. This querying process is different from updating.

Use the **IOPEN** call (as in Program 2, you needn't describe the entire record). Then use the **ISEEK** call to find a record you want to look at. If you are browsing through a file, you might want to look at the following or preceding record. For these processes, use the **IPREV** and **INEXT** calls.

Program 3 is an example of searching a data file one record at a time, going to the next record or going to the previous record.

Program 3—Searching an ISAM File

```
100 REM **Initialize**
110  DEFINT A-Z
120  LET ZERO = 0
125  LET FILENO = 0
130  LET FILENAME$ = "CUSTOMER.DAT"
140  LET RECORDSIZE=68
160  OPEN "/dev/null" AS #2 LEN = 68
180   FIELD #2, 4 AS CUSTOMERNO$, 20 AS CUSTNAME$, 20 AS ADDRESS$, _
            15 AS CITY$, 5 ZIP$, 4 OWE$
200  REM ** Open the file **
230  CALL IOPEN(VARPTR(FILENAME$),2,VARPTR(ZERO), _
          VARPTR(ZERO),FILENO)
235  IF IXSTAT <> 0 THEN PRINT "UNABLE TO OPEN": END
236  REM ** Open the file & position at first record.
237  CALL ISEEK(FILENO, 1, ZERO, ZERO, 0)
238  CALL IPREV(FILENO, 1)
240  LET CHOICE% = 0
260  WHILE (CHOICE% < 1 OR CHOICE% > 4)
280    CLS
300    LOCATE 3, 22: PRINT "*****Browse Customer Information*****"
320    LOCATE 5, 22: PRINT "         1. Seek A Specific Record"
340    LOCATE 6, 22: PRINT "         2. Look at Next Record"
360    LOCATE 7, 22: PRINT "         3. Look at Previous Record"
380    LOCATE 8, 22: PRINT "         4. QUIT"
400    LOCATE 9, 22: INPUT "Enter the number of your choice ",CHOICE%
420  WEND
440 REM ** Route to appropriate code **
460  ON CHOICE% GOTO 1000, 2000, 3000, 5000
500 REM **End of main menu**
1000 REM ** Subroutine for finding a specified record**
1020   CLS
1100    LOCATE 2, 12
1120    LINE INPUT  "***Find Customer #:", FIND$
```

```
1125    LSET CUSTOMERNO$=FIND$
1130    PRINT "FIND$=";FIND$
1140    CALL ISEEK(FILENO,1,SADD(CUSTOMERNO$),4,2)
1160    IF IXSTAT = 0 THEN GOTO 1250
1170    IF IXSTAT = 10 GOTO 1180 ELSE GOTO 4500
1180    PRINT FIND$;" Not Found."
1190    LINE INPUT "Press return try again.",A$
1200    GOTO 1120
1250 '    Here we read the record we found with the ISEEK
1260    CALL IREAD(FILENO,SADD(CUSTOMERNO$),RECORDSIZE)
1265 '    If the read is successful, print it to the screen.
1266 '    Otherwise, go to an error handling routine.
1270    IF IXSTAT = 0 THEN GOTO 4000 ELSE GOTO 4500
2000 REM **This subroutine brings in the next record**
2020  CALL INEXT(FILENO,1)
2040  IF IXSTAT = 0 THEN GOTO 2150
2050  IF IXSTAT <> 11 THEN GOTO 4500  'Go to error handler if error is not EOF
2060  INPUT "End of file reached. Press RETURN to continue",A$:GOTO 240
2150  CALL IREAD(FILENO,SADD(CUSTOMERNO$),RECORDSIZE)
2160  IF IXSTAT = 0 GOTO 4000 ELSE GOTO 4500
2170 ' **If the read was successful, print it to the screen.
2180 '   Otherwise, go to an error handling routine.
3000 REM **This subroutine brings in the previous record**
3020  CALL IPREV(FILENO,1)
3040  IF IXSTAT = 0  THEN GOTO 3150
3050  IF IXSTAT<>11 THEN GOTO 4500 'Go to the error handler if error isn't EOF
3060  INPUT "Beginning of file reached. Press RETURN to continue",A$: GOTO 240
3150  CALL IREAD(FILENO,SADD(CUSTOMERNO$),RECORDSIZE)
3160  IF IXSTAT=0 THEN GOTO 4000 ELSE GOTO 4500
3170 '  **If the read was successful, print it to the screen.
3180 '    Otherwise, go to the error handling routine.
4000 REM ** This subroutine displays a chosen record **
4010    CLS: LOCATE 2, 27: PRINT"***Customer Information***"
4020    LET NUMBER$ = CUSTOMERNO$
4040    LOCATE 4,24: PRINT "Number:"; NUMBER$
4060    LET TITLE$ = CUSTNAME$
4080    LOCATE 6,24: PRINT "Customer: "; TITLE$
4100    LET PLACE$ = ADDRESS$
4120    LOCATE 8,34: PRINT PLACE$
4140    LET CODE$ = CITY$+" "+ZIP$
4160    LOCATE 10,34:PRINT CODE$
4180    LET DEBT! = CVS(OWE$)
4200    LOCATE 12,34:PRINT "Owes us $"; DEBT!
4220    INPUT "Press return to continue", CONTINUE$
4240   GOTO 240  '  **Loop back to select another customer number
4500 REM** This is the error handling routine
4510 PRINT "Unexpected error has occurred---IXSTAT=",IXSTAT
5000 REM** Clean-up**
5020 CALL ICLOSE(FILENO)
5040 END
```

Program 3 first opens the ISAM data file:

```
230 CALL IOPEN(VARPTR(FILENAME$),2,VARPTR(ZERO), _
        VARPTR(ZERO),FILENO)
```

It then sets the pointer to the first record in the file:

```
236  REM ** Open the file & position at first record.
237  CALL ISEEK(FILENO, 1, ZERO, ZERO, 0)
238  CALL IPREV(FILENO, 1)
```

Notice that it moves the pointer so that **INEXT** will move to the first record.

If the user selects a specific record, the program routes control to the subroutine at line 1000. It seeks out the specified record and checks whether it found the record:

```
1140     CALL ISEEK(FILENO,1,SADD(CUSTOMERNO$),4,2)
1160     IF IXSTAT = 0 THEN GOTO 1250
```

If the user chooses to advance sequentially to the next record, the program routes control to the subroutine at line 2000. It seeks the next record, and checks to see if the end of the file was reached:

```
2020  CALL INEXT(FILENO,1)
2040  IF IXSTAT = 0 THEN GOTO 2150
2050  IF IXSTAT <> 11 THEN GOTO 4500
```

If the user chooses to move sequentially to the previous record in the data file, the program routes control to the subroutine at line 3000. It seeks the previous record, and checks to see if the beginning of the file was reached:

```
3020  CALL IPREV(FILENO,1)
3040  IF IXSTAT = 0  THEN GOTO 3150
3050  IF IXSTAT<>11 THEN GOTO 4500
```

This section has demonstrated creating, updating and searching through ISAM files—only a handful of the most common ISAM functions. For information on the entire range of ISAM capabilities, see Appendix F, "ISAM Reference."

5.4.4 Protecting Files in Multi-User Programs

This version of BASIC provides additional commands and facilities for programming multi-user applications. Two design issues are critical for multi-user programs: preventing simultaneous data changes by multiple users, and avoiding the "deadly embrace," in which each user must wait for the other to release control of the other's data.

BASIC uses variations of the **LOCK** statement to protect data from other users. Use **LOCK** on sequential and random files, and the ISAM call **ILOCK** on ISAM files. The following sections discuss the **LOCK** process.

5.4.4.1 Protecting Sequential And Random Files

The **LOCK** statement restricts access by other programs to any part of a data file. If you use it on a random access file, **LOCK** can protect individual records or ranges of records.

Locks can be either partial or total. Total locking is the default for the **LOCK** statement, and is applied to the file if you do not use the **READ** keyword. Total locking prevents any access by another program to the locked portion of the file.

Partial locking allows another program to read the file, but prevents that program from modifying the locked portion of the file.

If another program has locked a portion of the file you want, you have two options. The first is to return control to the program immediately with an accompanying error message. All of BASIC's usual error handling routines can trap and examine this error. If error trapping is not active, the error message is "Permission denied." This is the default option.

The second option is to wait until the program that issued the original **LOCK** unlocks the requested region of the file. The presence of the **WAIT** keyword forces this second option. It is possible to interrupt this wait by pressing DELETE; you can then continue the wait by entering **CONT**.

It is possible to get into a deadlock situation when waiting for a **LOCK** request. For example, a deadlock situation might occur when:

- A program has File1 open and locked.

- A different program has File2 open and locked.
- The first program executes a **LOCK** request with the **WAIT** option on File2.
- The second program executes a **LOCK** request with the **WAIT** option on File1.

If XENIX detects a deadlock, it returns a "Deadlock" error message.

Multiple **LOCK** statements have a cumulative effect. That is, a record that the program locked multiple times with different locking characteristics (**READ** or **WAIT**) retains the characteristics of the latest **LOCK** statement.

The program unlocks locked records with the **UNLOCK** statement.

5.4.4.2 Protecting ISAM Files

You can lock and unlock individual ISAM records in an open ISAM file by using the **ILOCK** routine. The **ILOCK** call requires two arguments: the filenumber returned by ISAM in the **IOPEN** call, and the specific kind of locking request.

ISAM has two forms of record locking: automatic and manual. Automatic is the default and is suitable for most applications. Both forms of locking are discussed in the following sections.

Automatic Locking

If you have ordered automatic locking on the ISAM file, the current record is locked as you use it. In other words, as you move from record to record, ISAM removes the old record lock and places one on the new current record. Invoke automatic locking with the **L_AUTO** request to the **ILOCK** call. Turn it off with another **ILOCK** call, with the **L_RELEASE** request.

Manual Locking

In manual mode, ISAM locks no record unless the program specifically requests it with the **L_LOCK** request to the **ILOCK** call. When such a request occurs, ISAM locks the current record. You may do this on several records, locking multiple records in the process. The **L_RELEASE** request to an **ILOCK** call unlocks *all* records in the ISAM file.

Trying to Open a Locked Record

The **ILOCK** call comes with two options for programs trying to open a locked record. The first is wait mode, in which your program waits for the locked record to be unlocked before proceeding. You invoke wait mode with the **L_WAIT** request to the **ILOCK** call.

The second option is the no wait mode, in which the program returns an error message explaining that the record is locked. You invoke no wait mode with the **L_NOWAIT** request to the **IOPEN** call.

For complete details on the **ILOCK** call and its options, see Appendix D, "ISAM Reference."

5.4.4.3 Sharing ISAM Files in XENIX

ISAM files are created with permissions limited to read and write by the owner. The owner is the user id of the process that created the file. You cannot change these permissions without risk of abnormal termination in cases where multiple users attempt simultaneous file updates.

In order to have multiple users work with the same ISAM files, all users must have the same user id. Use one of the following methods to establish a common user id:

Multiple users of a common set of files can log in under the same name. This name can be one user's identification, or a project identification.

Personal security issues can arise if you use one person's id to login: every person in the project will know the user's password and can gain access to all files—even those (such as mail) unrelated to the shared project. By using a project identification for login, you can avoid these security difficulties.

Assume, for example, that a number of people are updating inventory, and all use the same set of files and programs. You can create an account for the project, and have each person log in under a project account name, such as "stock." By using the common project user id, each user can have access to the program and the required files. Upon completing work on a project, the user can log off, and log back in under his own user id.

Using the ishare Utility.

Microsoft provides a utility called *ishare*. This program changes the user id of the program invoked by ishare to that of the owner of ishare.

To make use of ishare, have a common copy of ishare for each group working with a set of ISAM files. Use ishare to invoke msbasic as follows:

```
ishare msbasic
```

The system subsequently believes that the owner of ishare is the owner of msbasic and any ISAM file accessed by this invocation of the interpreter. In addition, random and sequential data files created or modified by this invocation of the interpreter are owned by the owner of ishare.

Each group sharing a set of programs and files should have a separate copy of the ishare utility. This separate version should be accessible only to that group.

To use the ishare utility, make a number of copies of ishare and distribute them to each logical project directory. Each copy should have the project as the owner of the ishare utility. The program is not large, so making multiple copies won't significantly affect available storage.

To create a copy of ishare that only one project, "stock" ,for example, can use:

1. Log in as a user in the stock project:
2. Make a new copy of ishare:

   ```
   cp /usr/lib/ishare  /usr/stock/bin/ishare1
   ```

3. Give ishare1 "setuid" mode, that is:

   ```
   chmod u+s /usr/stock/bin/ishare1
   ```

4. Give ishare1 group execute permission:

   ```
   chmod g+x /usr/stock/bin/ishare1
   ```

Users invoking ishare1 can access ISAM files owned by stock or other files with permissions set for the stock project. To allow people outside the project to access these ISAM files, give execute privileges for ishare1 to others:

```
chmod o+x /usr/stock/bin/ishare1
```

5.5 BASIC and Child Processes

The XENIX Microsoft BASIC Compiler is able to use one of the most powerful features of XENIX : the ability to create child processes. **SHELL** enables you to run part of a BASIC program, temporarily exit to XENIX to perform a specified function, and return to the BASIC program at the statement after the **SHELL** statement to proceed with the program.

This can be very useful when you want to use XENIX utilities instead of coding new BASIC routines. For example, to sort a file of addresses by zip code for mailing labels, you could use one BASIC routine to read the records into memory, another to sort them, and a third to write the updated file back to disk. It is easier, however, to use the XENIX sort command through **SHELL**.

This version of BASIC offers two options. If you use the **SHELL** function, *sh* executes the XENIX command you provide, and immediately resumes execution of the BASIC program. For example,

```
1030 LET SORTLABELS$ = "sort -n deliveries.f >label.list"
1050 PROCESS! = SHELL(SORTLABELS$)
```

The other option, the **SHELL** statement, is useful for starting shell processes whose output you need later in the program. If you use the statement, the BASIC program resumes execution only when the XENIX process is completed and control returns to BASIC. For example,

```
1030 LET SORTLABELS$ = "sort -n deliveries.f >label.list"
1050 SHELL(SORTLABELS$)
```

If you provide no argument to the **SHELL** statement (i.e., SHELL( )), a child shell is created in which you can execute normal XENIX commands. When you type Control-D you return to BASIC command level.

Chapter 6
Using Subprograms

6.1 Creating Subprograms 73
6.2 Calling Subprograms 75
6.3 Passing Variables with CALL 76
6.4 Passing Arrays with CALL 77
6.5 Passing Expressions with CALL 79
6.6 Accessing Parameters with SHARED 80
6.6.1 Sharing fom the Main Program: The SHARED Attribute 80
6.6.2 Overriding a SHARED Variable with a STATIC Statement 81
6.6.3 Sharing from the Subprogram Using the SHARED Statement 82
6.7 Passing Variables and Arrays Between Modules 83
6.8 Error Handling 85
6.9 Using GOSUB...RETURN 86
6.10 Variable Scoping Using SHARED and COMMON: An Extended Example 86
6.11 Common Errors 89

Programs, especially large programs, are easier to maintain and debug if they are broken down into smaller parts. Creating your own functions with the **DEF FN** statement is one way to break your programs into separate pieces. BASIC provides another way to segment programs, the *subprogram*. Subprograms are units of code, delimited by **SUB** and **END SUB** statements, that can be compiled separately and joined to the main body of the program at link time. Subprograms are different from functions because they do not return a value that is associated with the subprogram name (and thus cannot be used in expressions), and they can be called before they are defined. To distinguish between subroutines written in BASIC and subroutines written in assembly language, in this chapter the word *subprogram* defines an independently compilable segment of code used by another program (called the "main program.")

Using subprograms has three major benefits:

1. **Clarity.** Each subprogram has a specific task it performs on a small number of arguments or common variables. This can make the program much easier to understand.
2. **Independence.** Subprograms can be developed as independent units, permitting repeated use in many different.
3. **Flexibility.** Subprograms can be modified and rewritten without affecting other programs.

For many programs, a single source file is adequate. However, in some cases you can develop large programs more efficiently as a system of individual source files that are compiled separately, then linked to form a single executable program. Using this approach, you can create separate source files that perform specific tasks and use them over and over, in one program or in many different programs. In this chapter, each such source file is called a *module*.

6.1 Creating Subprograms

A subprogram is a unit of BASIC code that can be compiled independently, then linked to the rest of the program. Subprograms are useful for coding routines that are used over and over in programs; one subprogram can be used in many different main programs. Subprograms are delimited by **SUB** and **END SUB** statements. The **SUB** statement has the following form:

SUB *subprogram-name* [[(*parameter-list*)]] **STATIC**

where

subprogram-name	is a name up to 31 characters long. The name cannot appear in any other SUB statement used in the program.
parameter-list	contains variables and arrays that represent corresponding variables and arrays passed from the main program. Parameters are separated by commas.
STATIC	indicates that the subprogram is nonrecursive; that is, it does not call itself, nor does it call another subprogram that calls this subprogram. Only nonrecursive subprograms are supported, and a warning error is generated if **STATIC** is omitted.

Variable types must correspond to the variable type being passed from the main program. Arrays must be declared in the parameter list in the following form:

array-name (*dim-num*)

where

array-name	is the name of the array.
dim-num	is an integer representing the number of dimensions in the array. One dimension is the default.

If your program is a single module, subprograms should be placed after the **END** statement in the main program. The subprograms can appear in any order. If the main program and subprograms are in separate modules, they can be linked in any order.

All BASIC expressions are allowed within a subprogram, except the following:

- User-defined functions.
- A **SUB...SUB END** block. Subprograms cannot be nested.

Example

The following subprogram multiplies the values passed to it in an array VALUES. The index value of the array is passed to the subprogram in variable S.

```
SUB MULT (VALUES(1),S) STATIC
    FOR INDEX=1 TO S
        PRINT VALUES(INDEX) * 16
    NEXT INDEX
END SUB
```

6.2 Calling Subprograms

When the main program transfers execution to the subprogram, it "calls" the subprogram with the **CALL** statement. The **CALL** statement can invoke BASIC subprograms or assembly-language subroutines. This section and the rest of this chapter apply only to **CALL** when it is used with subprograms. Calling assembly-language routines is described in Chapter 7, "Interfacing With Other Languages."

The **CALL** statement has the following syntax:

CALL *name* [[(*argument-list*)]]

where

name	is the name of the called subprogram. The name is limited to 31 characters.
argument-list	is a list of variables and array elements passed to the called subprogram.

The number and type of the arguments in the **CALL** statement must match the number and type of parameters in the **SUB** statement or stack space will not be allocated correctly. Arguments and parameters must be given in the same order. The compiler does not check to see if **CALL** statement arguments and **SUB** statement parameters match, and no error

message is generated if they do not.

A complete subprogram call looks like this:

```
(main program)
.
.
.
CALL subprogram-name (argument-list)
.
.
.
(end of main program)

SUB subprogram-name (parameter-list) STATIC

    (body of subprogram)

END SUB
```

6.3 Passing Variables with CALL

Variables can be passed from the main program to subprograms through the **CALL** statement argument list. The following example shows how a **CALL** statement invokes a subprogram and passes variables A and B to it:

```
A=5 : B=0
CALL SQUARE(A,B)
PRINT "Main program variable values are " A "and " B
END

SUB SQUARE(X,Y) STATIC
    Y=X*X
    PRINT "In subprogram, Y =" Y
END SUB
```

The above example prints the results:

```
In subprogram, Y = 25
Main program variable values are 5 and 25.
```

The subprogram changes the value of variable B by assigning a new value to the **SUB** statement's corresponding parameter, Y. When the subprogram SQUARE begins executing, the initial value of Y is 0. The subprogram

computes a new value for Y (in this case, 25) and stores it in the passed parameter.

Variables passed using the above method are passed *by reference*, i.e., their location in memory is passed to the subprogram. The subprogram acts on the value in that location, and when control returns to the calling program, the value in that location may be changed. When you don't want the subprogram to change the value of a variable you must pass the actual value of the variable. When an actual value is passed, the subprogram copies the value into a temporary location for its own use and removes the temporary location before control is returned to the calling program. The original value is unchanged.

To pass a variable by value, enclose it in parentheses. For example, if you change the CALL SQUARE statement in the above example to

```
CALL SQUARE(A,(B))
```

B is passed by value, which is 0. In the example, subprogram SQUARE cannot change the value of B. The subprogram can change the value of Y, but this value is local to the subprogram and cannot be passed back to the main program through variable B. The revised example prints:

```
In subprogram Y = 25
Main program variable values are 5 and 0
```

Note that you can also pass expressions as parameters, as in the following:

```
CALL PROG2(X,(Y+1)) STATIC
```

Expressions are passed by value.

6.4 Passing Arrays with CALL

Arrays can be passed to subprograms with the **CALL** statement. The **CALL** statement with an array argument has the following form:

CALL *subprogram* (*array-name*())

where

Argument	Description
subprogram	is the name of the subprogram being called.
array-name	is the name of the array being passed.

Arrays use the same type matching rules as variables.

Example

The following example creates an array, then passes the array to a subprogram that multiplies each array element by 16:

```
DEFINT I-L
OPTION BASE 1
LIMIT=4
DIM ARRAY(LIMIT)
FOR INDEX=1 TO LIMIT
   ARRAY(INDEX)=INDEX
NEXT INDEX
CALL MULT(ARRAY(),LIMIT)
END

SUB MULT(VALUES(1),SUBSCRIPT) STATIC
   FOR ELEMENT=1 TO SUBSCRIPT
       PRINT VALUES(ELEMENT) "times 16 is " VALUES(ELEMENT)*16
   NEXT ELEMENT
END SUB
```

Passing Individual Array Elements

If a subprogram does not require an entire array, you can pass the values of individual array elements instead. To pass separate elements of an array, enclose their subscripts in parentheses. The statement in the following example passes the value of the element in row 3, column 9 of ARRAY1 to PROGNAME:

```
CALL PROGNAME(ARRAY1(3,9))
```

Using Array Bound Functions

The **LBOUND** and **UBOUND** functions, which are described in the "Reference," are useful for determining the size of an array passed to a subprogram. The **LBOUND** function finds the smallest index value of an array subscript; **UBOUND** finds the largest index value. They are especially useful for passing dynamic arrays, because the calling program does not have to pass the size of each dimension to the subprogram.

For instance, the subprogram in the first example uses **LBOUND** and **UBOUND** instead of passing the upper bounds of the array explicitly. The **LBOUND** statement initializes the variable `row` to the lowest subscript value of `array`, and the **UBOUND** statement limits the number of times the FOR loop executes to the number of elements in the array.

```
SUB PROG2(ARRAY(2)) STATIC
    FOR ROW=LBOUND(ARRAY, 2) TO UBOUND(ARRAY, 2)
.
.
.
```

The following example uses **LBOUND** and **UBOUND** to compute the limits of the array, instead of a variable value as in the previous example.

```
OPTION BASE 1
DIM ARRAY(4)
FOR INDEX=1 TO UBOUND(ARRAY)
   ARRAY(INDEX)=INDEX
NEXT INDEX
CALL MULT(ARRAY())
END

SUB MULT(VALUES(1)) STATIC
   FOR INDEX=LBOUND(VALUES) TO UBOUND(VALUES)
       PRINT VALUES(INDEX) "times 16 is " VALUES(INDEX)*16
   NEXT INDEX
END SUB
```

6.5 Passing Expressions with CALL

You can also pass expressions as arguments to subprograms. An argument can be any valid BASIC expression except simple variables and array-element references, for example:

```
CALL PROG2(VAR*16,X) STATIC
```

6.6 Accessing Parameters with SHARED

BASIC supports shared variables and provides two ways for their values to be preserved across subprogram calls within a single module:

1. The **SHARED** attribute to main program **COMMON**, **DIM**, and **REDIM** statements permits the use of main program variables by all subprograms within the module. This method is most useful if all variables will be shared by all subprograms.
2. The **SHARED** statement permits a subprogram to use variables declared in a main program. This method is most useful when different subprograms use different combinations of main program variables.

Sections 6.6.1 and 6.6.2 discuss these two ways to pass parameters.

6.6.1 Sharing fom the Main Program: The SHARED Attribute

You can access variables from the main program without passing them as parameters to the **CALL** statement by using the **SHARED** attribute with **COMMON**, **DIM**, or **REDIM** statements in the main program. Within a module, subprograms can use all the main program variables declared with the **SHARED** attribute. If all the variables in the main program are also used by all the subprograms, this is easiest and most efficient way to share variables. To use the **SHARED** attribute, place the word "SHARED" directly after the keyword in a **COMMON**, **DIM**, or **REDIM** statement.

Note

The **SHARED** attribute and the **SHARED** statement, described in Sections 6.6.1, 6.6.2, and 6.6.3 only share variables within a single compiled module. They do not share variables with subprograms compiled separately.

Example

The following statements declare variables shared across all subprograms within a module:

```
COMMON SHARED a,b,c
COMMON SHARED /length/ short, medium, long
SHARED x,y
DIM SHARED array (10,10)
REDIM SHARED alpha(n%)
```

In the following module, the main program passes the value of A to subprogram PROG2. When the executable program is run, it prints the number "10."

```
COMMON SHARED A 'variable A can be used by any
A=6+4           'subprogram in this module
CALL prog2

SUB prog2 STATIC
    PRINT A     'prints 10
END SUB
```

Without the **COMMON SHARED** statement, the above program would look like this:

```
A=6+4
CALL prog2(A)

SUB prog2(X) STATIC
    PRINT X
END SUB
```

The significance of the **SHARED** attribute becomes apparent when large numbers of variables are being passed from the main program.

6.6.2 Overriding a SHARED Variable with a STATIC Statement

Variables declared in the main program with the **SHARED** attribute are global. Variables declared in subprograms are local to the subprogram. In a subprogram you may occasionally need to use a local variable name that has been previously declared a global variable. You can use the **STATIC** statement to override global variables at the subprogram level.

Note that when you use the **STATIC** statement with an array variable, you must immediately dimension the array, as in the following example:

```
SUB PROG2(x) STATIC
    STATIC array%(1)
    DIM array%(25)
.
.
.
END SUB
```

Example

The output of the following program illustrates the effect of the **STATIC** statement on a global variable.

```
FOR I = 1 to 10
    X=X+2        'X is a global variable
    CALL PROG2
    PRINT "main program X =" X
NEXT I

SUB PROG2 STATIC
    STATIC X     'declares X local to the subprogram
    PRINT "subprogram X =" X
END SUB
```

The output of the program is:

```
subprogram X = 0
main program X = 2
subprogram X = 0
main program X = 4
subprogram X = 0
.
.
.
```

6.6.3 Sharing from the Subprogram Using the SHARED Statement

Within a single module, use of the **SHARED** statement in a subprogram allows the subprogram to use main program variables. The **SHARED** statement has the form

SHARED *variable* [[,*variables*]]

The argument *variable* is the name of any variable declared in the main program. A **SHARED** statement with array arguments has the form:

SHARED *arrayname*() [[,**arrayname**()]]

Variables and arrays can bogh be used as arguments to a single **SHARED** statement.

Example

To use variables A and D declared in the main program, the subprogram must declare them in a **SHARED** statement before they are used, as in the following example:

```
A=4:D=6
CALL prog2

SUB prog2 STATIC
    SHARED A,D
    X=A+D
    PRINT X      'prints 10
END SUB
```

6.7 Passing Variables and Arrays Between Modules

If the main program and subprograms are in separate modules, use the **COMMON** statement to share variables and arrays. The **COMMON** statement has the form:

COMMON [[**SHARED**]] */blockname/ item-list*

where

SHARED	is an optional attribute. Use the **SHARED** attribute when you want to share all items in the *item-list* among all subprograms in the module.
/blockname/	is any name up to 31 characters long. This name is used internally to identify a specific **COMMON** variable group. Use *blockname* when every subprogram does not share every **COMMON** variable or

array.

item-list is a list of variables and arrays that will be used by the subprograms. Arrays have the form:

array-name (*dim-num*)

Where *array-name* is the name of the array and *dim-num* is the number of dimensions in the array.

The **COMMON** statement useful when there are many variables to be shared, or when not all variables are shared by all subprograms.

Order is important in **COMMON** statements. If the main program's **COMMON** statement looks like this:

```
COMMON a,d,e
a=5:d=8:e=10
```

And the subprogram **COMMON** statement looks like this:

```
COMMON a,e,d
```

the order of the values is maintained; thus in the subprogram e=8 and d=10.

Named COMMON

Named **COMMON** separates shared variables into groups that are identified by their *blockname*. It is useful when subprograms need to share specific groups of variables from the main program. The following example calculates the volume and density of a 3-dimensional rectangle using data supplied from the main program. Subprogram VOLUME only needs to share the variables representing the lengths of the sides, in **COMMON** block *sides*. Subprogram DENSITY needs variables representing the sides of the rectangle and the weight.

```
'main program
DIM a(3)
COMMON /sides/ a()
COMMON /weight/ c

c=52
a(1)=3:a(2)=3:a(3)=6
CALL volume
CALL density
```

```
'subprogram VOLUME in separate module
DIM a(3)
COMMON SHARED /sides/ a()

SUB volume STATIC
    vol=1
    FOR x = 1 TO 3
        vol=a(x)*vol
    NEXT x
    PRINT "The volume is " vol
END SUB


'subprogram DENSITY in separate module
DIM a(3)
COMMON SHARED /sides/ a()
COMMON SHARED /weight/ w

SUB density STATIC
    vol=1
    FOR x = 1 TO 3
        vol=a(x)*vol
    NEXT x
    dens=w/vol
    PRINT "The density is " dens
END SUB
```

6.8 Error Handling

Error handling subroutines must be located *outside* subprograms. Error handling subroutines located inside **SUB...END SUB** blocks will cause SB error messages, and may not return correctly. The correct structure for error handling within a subprogram is:

```
        SUB TEST STATIC
        .
        .
        .
        ON ERROR GOTO 5000
        .
        .
        .
        END SUB

5000: 'Error handling routine begins here
```

```
(body of routine)

RESUME
```

Error-handling routines in separately compiled modules apply only to that module. In other words, if an error-handling routine is to apply to the entire program, the entire program must be in a single source file.

6.9 Using GOSUB...RETURN

Subprograms are intended to be entirely independent of the main program. To preserve this independence, it is recommended that subprograms be entered only with the **CALL** statement, and exited only with the **END SUB** statement. This ensures that the stack is maintained properly, and prevents unexpected errors at run time. Entering and exiting subprograms with **GOTO**, **GOSUB**, or **RETURN** *linenumber* statements is not recommended. Within a subprogram, the objects of **GOTO** and **GOSUB** statements should be located inside the subprogram. Improperly placed **GOTO**, **GOSUB**, or **RETURN** *linenumber* statements will cause a "Subroutine error" message at run time.

6.10 Variable Scoping Using SHARED and COMMON: An Extended Example

Unless specified otherwise, variables and arrays are considered local to the subprogram or main program in which they are declared.

There are three different ways to share variables and arrays:

1. Within a module,
2. Across all modules
3. Across selected modules

This section summarizes the different ways of passing parameters with **SHARED** and **COMMON**.

In the following example, the array A is not shared by the main program and the subprograms:

```
'main program
DIM A(10)
A(1)= 256
PRINT A(1)       'prints 256
CALL prog1
CALL prog2

'subprogram 1 in same module
SUB prog1 STATIC
   DIM A(10)
   PRINT A(1)    'prints 0
END SUB

'subprogram 2 in same module
SUB prog2 STATIC
   DIM A(10)
   PRINT A(1)    'prints 0
END SUB
```

To share array A with both subprograms in the module, (and thus give A(1) the value 256 throughout) add the **SHARED** attribute to the main program's **DIM** statement. (You must also remove the **DIM** statements from the subprograms or you will get a DD "Array Already Dimensioned" error.) The **SHARED** attribute shares variables and arrays with all subprograms called by that program.

```
'main program
DIM SHARED A(10)
A(1)= 256
PRINT A(1)       'prints 256
CALL prog1
CALL prog2

'subprogram 1 in same module
SUB prog1 STATIC
   PRINT A(1)    'prints 256
END SUB

'subprogram 2 in same module
SUB prog2 STATIC
   PRINT A(1)    'prints 256
END SUB
```

If only subprogram 2 needs to share array A with the main program, add the **SHARED** statement to subprogram 2 instead. The **SHARED** statement allows the subprogram to share variables and arrays from the calling

program. (If you compile the following program you will get a warning error.)

```
'main program
DIM A(10)
A(1)= 256
PRINT A(1)       'prints 256
CALL prog1
CALL prog2

'subprogram 1 in same module
SUB prog1 STATIC
   PRINT A(1)    'prints 0
END SUB

'subprogram 2 in same module
SUB prog2 STATIC
   SHARED A()
   PRINT A(1)    'prints 256
END SUB
```

If the main program is in one module, and the subprograms are in one or more separate modules, use **COMMON** to share variables and arrays among modules. There are two steps to using **COMMON** with subprograms in separate modules:

1. The shared variables and arrays must be passed to each module with a **COMMON** statement.
2. The shared variables and arrays must be passed to each individual subprogram by adding the **SHARED** attribute to the **COMMON** statement at the beginning of each module, or by adding the **SHARED** statement to each subprogram.

For example, in the following program A(1) is passed to *module* mod1.bas, but not to *subprogram* prog1. (If you compile mod1.bas you will get a warning error.)

```
'main program in file main.bas
DIM A(10)
COMMON A()

A(1)=256
PRINT A(1)       'prints 256
CALL prog1
END

'prog1 in file mod1.bas
```

```
DIM A(10)
COMMON A()

SUB prog1 STATIC
   PRINT A(1)    'prints 0
END SUB
```

There are two ways to pass array A to subprogram prog1:

1. Add the **SHARED** attribute to the **COMMON** statement in mod1.bas:

```
DIM A(10)
COMMON SHARED A(10)

SUB prog1 STATIC
   PRINT A(1)  'prints 256
END SUB
```

2. Add a **SHARED** statement to subprogram prog1:

```
DIM A(10)
COMMON A()

SUB prog1 STATIC
    SHARED A()
    PRINT A(1) 'prints 256
END SUB
```

6.11 Common Errors

Two errors are commonly made when calling subprograms:

- Mismatched argument and parameter lists
- Variable aliasing

Mismatched argument and parameter list errors are caused when the order, type, or number of arguments passed to a BASIC subprogram do not correspond to the parameters in the subprogram. The BASIC Compiler does not check for this discrepancy, and no error message is generated. However, noticeable side effects will probably occur.

Example

In the following program, the main program passes a string to a subprogram that is expecting an integer. Although the program compiles and links without errors, when the program is executed, it will eventually cause a "String Space Corrupt" error.

```
A$="This is a string"
CALL prog2(A$)

SUB prog2(X%) STATIC
    X%=X%*X%
END SUB
```

A common mistake in long programs containing many variables is variable aliasing. Variable aliasing occurs when more than one name refers to the same location in memory. In modular programs, variable aliasing occurs when an argument passed to a subprogram can be referenced in the subprogram in more than one way. This often happens when the same variable is used twice as a parameter to **CALL**, or when a variable passed as a parameter is also accessed by means of the **SHARED** statement or the **SHARED** attribute. To avoid aliasing problems, pass arguments by value and make minimal use of **SHARED**.

Example

The following is a simple example that illustrates how unexpected variable aliasing can occur.

```
COMMON SHARED A
A=4
CALL prog2 (A)

SUB prog2 (X) STATIC
    PRINT "Half of"; X; "plus half of"; A; "is";
    X=X/2
    A=A/2
    PRINT (A+X)
END SUB
```

When run, this program displays the following message:

```
Half of 4 plus half of 4 is 2
```

In the above example, A passed in the **COMMON SHARED** statement is the variable the subprogram references from the main program, and it is considered global data. When the subprogram modifies X, it is in effect modifying A, so when further operations are performed on A with the assumption A=4, the results are false.

Chapter 7
Interfacing With Other Languages

7.1 Loading Assembly Language Files 93
7.2 Calling Assembly Language Subroutines 93
7.2.1 Assembly Language Coding Rules 97
7.2.2 Calling FORTRAN Subroutines 99
7.2.3 Calling C Language Modules 99
7.3 The Run-Time Memory Map 100

You may wish to incorporate assembly language, FORTRAN or C subroutines into your BASIC programs. This section describes the necessary procedures and some special features of the Microsoft BASIC Compiler.

Under the XENIX operating system, memory space for BASIC is separated into instruction and data space. Any subroutine will reside in the instruction space and will use the data space.

7.1 Loading Assembly Language Files

You can assemble and link assembly language source files with BASIC programs on the **bascom** command line. The compiler automatically invokes the assembler on files that have the ".s" extension. The assembly language file can also be a previously assembled object file.

For example, the following command assembles the assembly language files asm1.s and asm2.s, and links them with the BASIC program math.bas:

```
bascom math.bas asm1.s asm2.s
```

7.2 Calling Assembly Language Subroutines

You can incorporate assembly language subroutines in your BASIC program using the **CALL** or **CALLS** statements. The **CALL** statement calls an assembly-language subroutine and passes the subroutine the unsegmented addresses of the statement arguments. The **CALLS** statement has the same syntax and purpose as **CALL**, but passes the segmented addresses of its argument. The syntax of the **CALL** statement is

CALL *name* [[(*argument-list*)]]

where

name	The name of the assembly-language subroutine being called. It is limited to 31 characters.
argument-list	parameter contains the arguments you pass to the subroutine.

There is a detailed description of the **CALL** statement in the "Reference."

Invoking the **CALL** statement causes the following to occur:

- Arrays are passed by pushing an array descriptor on the stack.
- Array elements are passed by pushing a 2-byte pointer to the element onto the stack. Note that elements adjacent to the pointer are not guaranteed to contain the values of adjacent elements of the array.
- Strings are passed by pushing a pointer to the string descriptor.
- For each non-array argument in the argument list, BASIC pushes the 2-byte offset of the argument's location within the data segment (DS) onto the stack.
- BASIC pushes the return address code segment (CS), and offset (IP) onto the stack.
- BASIC transfers control to the subroutine.

Figure 7.1 illustrates the state of the stack at the time of the **CALL** statement. Figure 7.2 shows the condition of the stack during execution of the called subroutine.

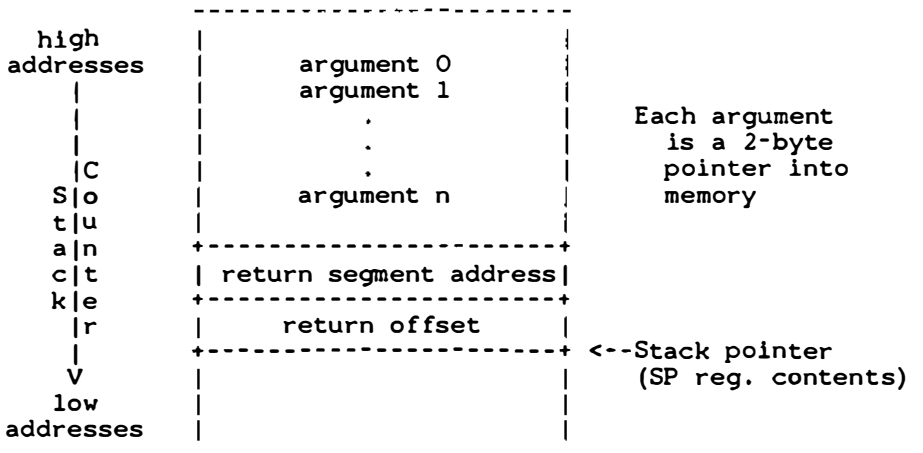

Figure 7.1 Stack Layout when CALL statement is activated

The subroutine now has control. The program can reference arguments by moving the Stack Pointer (SP) to the Base Pointer (BP) and adding a positive offset to (BP). For example,

```
push    bp
mov     bp, sp
        .
        .
```

You can calculate the offset of argument n, where n is a 2-byte variable, as BP + 4. Argument $n-1$ is at BP + 6. Locate any argument, k for example, by using the following formula, where k is the argument and n is the total number of arguments made by the program:

Location k = BP + 4 + (2 * (n − k))

Calculate the location of argument 0 by using the following formula:

Location of Argument 0 = BP + 4 + 2n

If you have local variables, you should begin the assembly language program with language similar to the following:

```
push    bp
mov     bp, sp
add     sp, space
        .
        .
mov     sp, bp
```

The *space* argument is equal to two times the number of local variables in the program.

You can calculate the location of local variable 0 with the following formula:

Location of local variable 0 = BP - 2

You can also calculate the locations of subsequent local variables. For each variable, subtract an additional two bytes from BP. This gives you a formula to calculate the location of local variable k if you know n, the total number of local variables.

Location of local variable k = BP - $2(n + 1)$

Important

It is critical that your program clean up the stack before exiting the called subroutine. Do this by popping BP, as in the following example:

```
push    bp
mov     bp, sp
add     sp, space
            .
            .
mov     sp, bp
pop     bp
ret
```

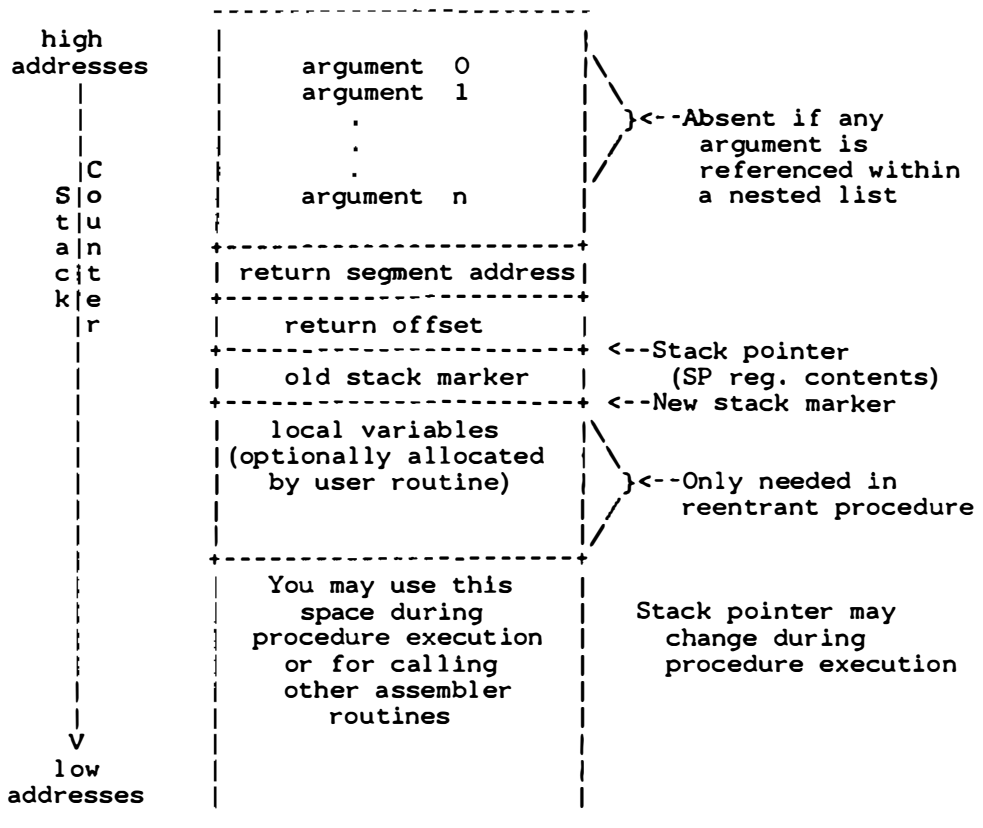

Figure 7.2 Stack Layout After ADD Statement Execution

BASIC does not preserve case. Therefore, all entry point names must be capitalized.

7.2.1 Assembly Language Coding Rules

You must observe the following rules when coding a subroutine:

- The subroutine the program calls is permitted to destroy the AX, BX, CX, and DX registers. It must preserve all other registers.

- The called subroutine must know the number and length of the arguments the BASIC program passes to it. References to arguments are positive offsets added to (BP) (assuming the called routine moved the current stack pointer into BP; i.e., MOV BP,SP).
- The subroutine the program calls must perform an intersegment return.
- BASIC receives returned values by including the variable name that receives the result in the argument list.
- If the argument is a string, its offset points to 4 bytes called the "string descriptor." The first 2 bytes of the string descriptor define the length of the string, and the second two point to the actual characters. See the following information about the floating point accumulator for details about the string descriptor.

Important

If the argument is a string literal in the program, the string descriptor points to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. For example,

```
20 A$ = "BASIC"+""
```

This forces BASIC to copy the string literal into string space. Now the string can be modified without affecting the program.

- User routines can alter strings, but they *must not* change the length of the strings. BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

The following sequence of assembly language demonstrates access of the arguments passed and storage of a return result in the variable C. BASIC accesses this routine through a statement like

```
CALL ENTRY(A%,B$,C%)
```

The assembly language routine can appear as follows:

```
push    bp
push    si
push    di
mov     bp,sp       ;Get current stack position BP.
mov     bx,8[bp]    ;Get address of B$ dope.
mov     cx,[bx]     ;Get length of B$ in CL.
mov     dx,2[bx]    ;get address of B$ text in DX.
        .
        .
        .
mov     si,10[bp]   ;Get address of 'A' in SI.
mov     di,6[bp]    ;Get pointer to 'C' in DI.
movs    word        ;Store variable 'A' in 'C'.
pop     bp
pop     di
pop     si
ret     6           ;Restore Stack, return.
```

Important

The subroutine that BASIC calls must know the variable type for numeric arguments passed. In the above example, the instruction

```
movs word
```

copies only 2 bytes. This works out if variables A and C are integers. You would have to copy 4 bytes if A and C were single precision and copy 8 bytes if they were double precision.

7.2.2 Calling FORTRAN Subroutines

Use the **CALLS** statement to access FORTRAN subroutines. **CALLS** works just like **CALL**, but each of the arguments on the stack is a 4-byte pointer into memory, rather than a 2-byte pointer.

7.2.3 Calling C Language Modules

If you want to call C language modules from your BASIC programs, you must take into consideration the different calling conventions of the two languages. BASIC uses a convention like the PL/M convention. Languages using the PL/M calling convention expect the *callee* to remove parameters pushed on the stack. Microsoft C uses its own convention, in which the

caller is expected to clean the stack. This means that the order that parameters are pushed onto the stack is different for each language. BASIC pushes parameters in the order of occurrence; C pushes parameters on the stack in reverse order. In BASIC, the called routine must clean up the stack before returning; C expects the calling routine to clean up the stack.

As an example, look at the following BASIC statement:

```
CALL AROUTINE(A%,B%,C%)
```

It is equivalent to the following assembly language code:

```
push    offset ds:a          ;Push addresses of arguments
push    offset ds:b          ;In the order encountered
push    offset ds:c
call    aroutine             ;Far Call
```

The Microsoft C language convention expects parameters to be pushed in the *reverse* of the order encountered. The C procedure definition to receive the above CALL statement would be

```
aroutine (c,b,a)             /* Note: parms in reverse order  */
short int *a, *b, *c;        /* BASIC passes pointers to data */
```

C Data Representation

Only short (16-bit) integers are certain to be in the same format in both languages. Strings, single and double precision floating point data are stored differently.

7.3 The Run-Time Memory Map

Figure 7.3 shows the run-time memory map for programs linked to the XENIX BASIC runtime libraries.

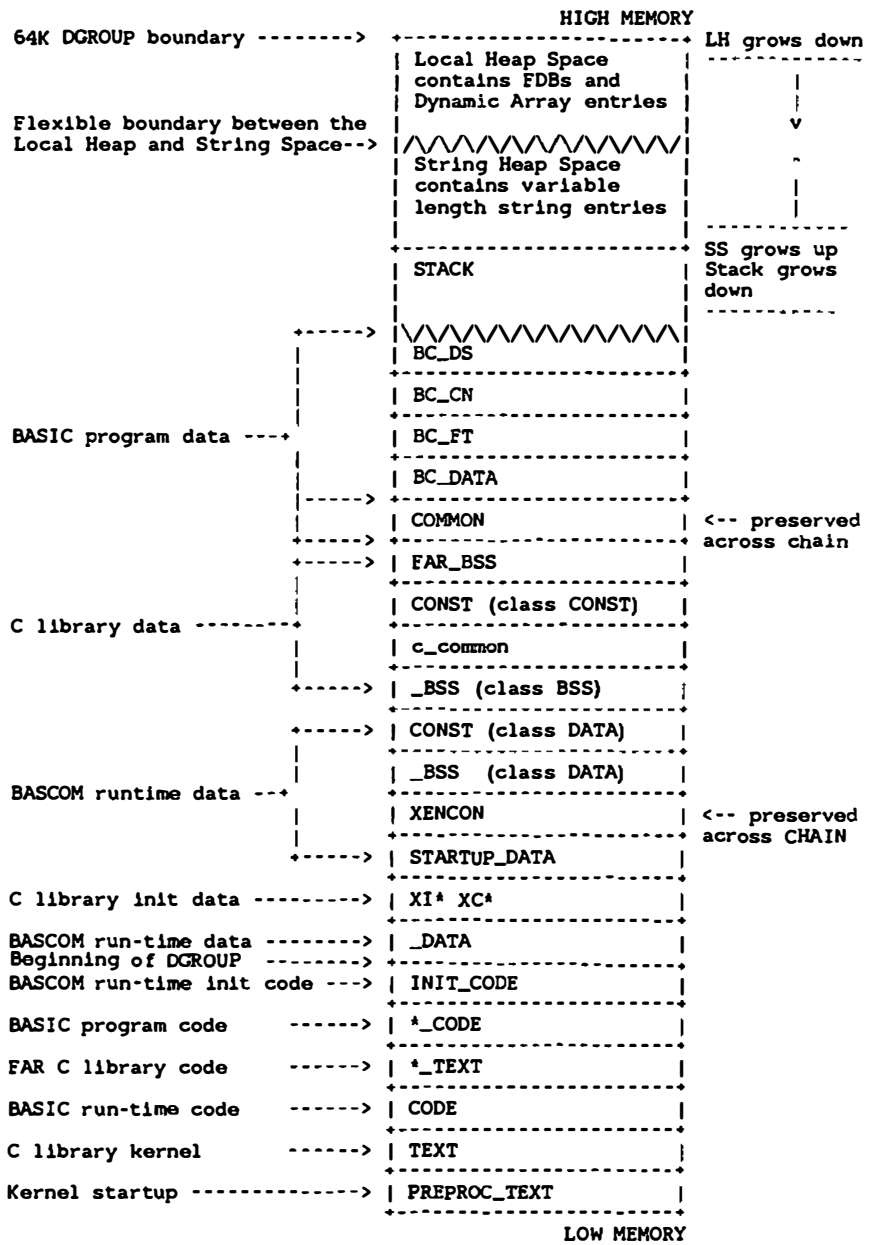

Figure 7.3 XENIX BASIC Compiler Run-Time Memory Map

Part 2
Reference

8 Language Elements 103
9 Compiler-Interpreter Language Differences 127
10 Statement and Function Reference 147

Chapter 8
Language Elements

8.1 Character Set 105
8.2 The BASIC Line 106
8.2.1 Using Line Labels 108
8.2.2 Creating Lines Longer than 255 Characters 108
8.3 Data Types 109
8.4 Constants 110
8.5 Variables 112
8.5.1 Variable Names 113
8.5.2 Declaring Variable Types 114
8.5.3 Array Variables 115
8.6 Expressions and Operators 116
8.6.1 Hierarchy of Operations 117
8.6.2 Arithmetic Operators 118
8.6.2.1 Integer Division 119
8.6.2.2 Modulo Arithmetic 119
8.6.2.3 Overflow and Division by Zero 119
8.6.3 Relational Operators 120
8.6.4 Logical Operators 121
8.6.5 Functional Operators 124
8.6.5.1 Intrinsic Functions 124
8.6.5.2 User-Defined Functions 124
8.6.6 String Operators 124
8.7 Type Conversion 125

This chapter presents the character set and the rules for the constants, variables, expressions, and operators used by the Microsoft XENIX BASIC language.

8.1 Character Set

The Microsoft XENIX BASIC character set consists of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in BASIC are the uppercase letters (A-Z) and lowercase letters (a-z) of the English alphabet.

The BASIC numeric characters are the digits 0-9. The letters a-f and A-F can be used as parts of hexadecimal numbers.

Special Characters

The following special characters are recognized by BASIC:

Character	Name or Function
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent (suffix for integer data type)
!	Exclamation point (suffix for single-precision data type)
#	Number (or pound) sign (suffix for double-precision data type)

$	Dollar sign (suffix for string data type)
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At symbol
_	Underscore
RETURN	Terminates input of a line

8.2 The BASIC Line

BASIC program lines have the following format:

[[*line-identifier*]] *statement* [[: *statement...*]] [[*comment*]]

Program lines can contain a maximum of 255 characters, and must end with a carriage return.

The XENIX BASIC Compiler supports two types of *line-identifiers*: line numbers and alphanumeric line labels.

1. A line number can be any combination of digits, from 0 to 65529. XENIX BASIC accepts lines with no space between the line number and the keyword. Spaces are allowed in line numbers, but the compiler uses only the digits preceding the space as the line number. The following are valid line numbers:

```
1
200
300PRINT "hello"        '300 is the line number
65000
25 10                   'The compiler considers this line 25
```

Using line number 0 is not recommended. Although line number 0 is allowed, error trapping statements (such as **ON ERROR GOTO** and **ON** *event* **GOTO**) interpret the presence of line number 0 to mean that error trapping is disabled. For example, the following statement will not branch to line 0 if an error occurs:

```
ON ERROR GOTO 0
```

Note that, if **RESUME 0** is present in a program that also contains line number 0, execution will *not* resume on line number 0, but on the line where the error occurred.

2. An alphanumeric line label can be any combination of from 1 to 40 letters and digits that ends with a colon. BASIC keywords are not permitted. The following are valid alphanumeric line labels:

```
alpha:
a16:
SCREENsub:
123:
```

Case is not significant in line labels. The following line labels are equivalent:

```
alpha:
Alpha:
ALPHA:
```

Line numbers and labels can begin in any column, as long as they are the first nonblank characters on the line. Spaces are allowed between an alphanumeric label and the colon following it. There cannot be more than one line label on a line.

A BASIC *statement* can be either executable or nonexecutable. An executable statement tells your program what to do next (such as read input, write output, open a file, or branch to another part of the program). All statements are executable, except the following:

COMMON
DATA
DEF*type*
DIM (static arrays only)
OPTION BASE
REM (includes metacommands)

A *comment* is a nonexecutable statement used to clarify a program's operation and purpose. A comment is introduced by a single quote character ('), or the **REM** statement.

More than one BASIC statement can be placed on a line, but each statement must be separated from the last by a colon (:).

8.2.1 Using Line Labels

The XENIX BASIC compiler does not require each line in a source program to have a line number or label. You can mix alphanumeric labels and line numbers in the same program, and you can use alphanumeric labels as objects of any BASIC statement where line numbers are permitted, except as the object of an **IF...THEN** statement. In **IF...THEN** statements, BASIC permits only a line number, unless you explicitly use a **GOTO** statement. For example, the following statement will compile and execute correctly:

```
IF A = 10 THEN 500
```

However, if the object of the **IF...THEN** statement is a line label, a **GOTO** statement is required, as follows:

```
IF A = 10 THEN GOTO INCOMEDATA
```

If you are trapping errors, the **ERL** function will return only the last line *number* encountered before the error. Line labels or numbers are not required with the **RESUME** and **RESUME NEXT** statements.

8.2.2 Creating Lines Longer than 255 Characters

You can use the underscore character (_) to create "logical" lines of more than 255 characters. Using this feature, you can improve program structure and readability.

Breaking lines is particularly effective with **FIELD** statements, as in the following:

```
FIELD #1,10 AS A$,_    'name
         10 AS B$,_    'address
         10 AS C$,_    'zip code
         10 AS D$_     'license number
```

8.3 Data Types

Four data types exist in BASIC. They are described in detail below.

1. String

   A string is a sequence of up to 32,767 characters. The codes for these characters range from 0 to 127 for ASCII (American Standard Code for Information Interchange) characters, and from 128 to 255 for non-ASCII characters (see Appendix A, "ASCII Character Codes.").

2. Integer numeric

   Integers are stored internally as signed (two's complement) 16-bit binary numbers ranging in value from –32,768 to 32,767.

3. Single-precision floating-point numeric

   Single-precision numbers are accurate to 7 decimal places, and have approximate ranges of –1.7E+38 to –2.9E–39 for negative values, true zero, and 2.9E–39 to 1.7E+38 for positive values.

   Single-precision numbers are stored internally as 32-bit (or 4-byte) binary numbers in sign-magnitude format. The number is negative if the most significant bit (or MSB) of the third byte is set (that is, if it is equal to 1). The number is positive if the MSB is clear (0).

   The unsigned magnitude of the number is contained in 2 parts: a 24-bit binary fraction, known as the mantissa, and an 8-bit exponent, or the power of 2 by which the mantissa must be multiplied to get the single-precision value.

   The mantissa is a normalized, unsigned fraction whose binary point is located to the left of its MSB (since this fraction is normalized, the MSB is implied). The remaining 23 bits of the mantissa comprise the seven right-most bits of the third byte, plus the 16 bits of the second and first bytes, respectively. The value of the mantissa ranges from 0.5 (decimal) up to, but not including, 1.

   The exponent is the value contained in the last (fourth) byte, minus

256 (80H). This is known as an "excess 256" representation. A zero value is defined as a binary representation whose fourth byte is 0, regardless of the contents of the other 3 bytes.

4. Double-precision floating-point numeric

   Double-precision numbers have the same allowable ranges as single-precision numbers, but are accurate to 15 decimal places. The binary representation of a double-precision number has 58 bits to accomodate this extra accuracy. The preceding discussion of single-precision numbers applies to double-precision numbers, with the following differences:

   The sign bit is the MSB of the seventh byte.

   The mantissa consists of an implied 1, the 7 right-most bits of byte 7, plus bytes 6 through 1, respectively.

   The exponent is the eighth byte. A value of 0 in this byte implies a value of 0 for the entire number.

8.4 Constants

Constants are predefined values that do not change during program execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 32,767 alphanumeric characters enclosed in double quotation marks. These alphanumeric characters can be any of the characters whose ASCII codes fall within the range 32 to 127, except the double quote character ("). These are all valid string constants:

```
"HELLO"
"$25,000.000"
"Number of Employees"
```

Numeric constants are positive or negative numbers. There are three types of numeric constants: integer, fixed-point, and floating-point.

1. Integer constants

   a. Decimal

      One or more decimal digits (0-9), with an optional sign prefix (+ or -). The range for decimal constants is -32,768 to +32,767.

b. Hexadecimal

One or more hexadecimal digits (0-9, a-f, or A-F) with the prefix &H or &h. The range for hexadecimal constants is &H0 to &HFFFF.

Examples:

```
&H76
&H32F
```

c. Octal

One or more octal digits (0-7) with the prefix &O, &o, or &. The range for octal constants is &O0 to &O17777.

Examples:

```
&0347
&1234
```

Integer constants do not contain decimal points.

2. Fixed-point constants

Positive or negative real numbers; that is, numbers that contain decimal points.

Example:

```
9.0846
```

3. Floating-point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A single-precision floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The exponent is the power of ten by which the mantissa is multiplied to get the value of the floating-point number. Double-precision floating-point constants are denoted by the letter D instead of E.

Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
4.35D-10 = .000000000435
```

Fixed-point and floating-point numeric constants can be either single-precision or double-precision numbers. Single-precision numeric constants are stored with six digits of precision (plus the exponent) and printed with up to six digits of precision. Double-precision numbers are stored with 14 digits of precision (plus the exponent) and printed with up to 14 digits of precision. A single-precision constant is any numeric constant that has one

of the following properties:

- Six or fewer digits with no "D" exponential character or trailing "#"
- Exponential form denoted by "E"
- A trailing exclamation point ("!")

A double-precision constant is any numeric constant that has one of the following properties:

- Seven or more digits with no "E" exponential character or trailing "!"
- Exponential form denoted by "D"
- A trailing number sign ("#")

The following are examples of numeric constants:

Single Precision	Double Precision
46.8	345692811
−1.09E-6	−1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Numeric constants in BASIC cannot contain commas.

8.5 Variables

Variables are named values that can change during the execution of a program, or from one execution of a program to the next. As with constants, there are two types of variables: numeric and string. A numeric variable can be assigned only a numeric value (either integer, single-precision, or double-precision); a string variable can be assigned only a character-string value. You can assign the following values to a variable:

1. A constant value, as in the following example:

```
A = 4.5
```

2. The value of another variable, as in the following example:

```
B$ = "ship of fools"
A$ = B$
```

3. The value obtained by combining other variables and/or constants with operators, as in the following example:

```
PI = 3.141593
CONVERSION = 180/PI
```

For more information on combining variables and constants, see Section 9.7, "Expressions and Operators."

In any case, the variable must always match the type of data (numeric or string) assigned to it.

Note

Before a variable is assigned a value, its value is assumed to be zero (for numeric variables) or null (for string variables).

8.5.1 Variable Names

A BASIC variable name can contain as many as 40 characters. The characters allowed in a variable name are letters, numbers, the decimal point, and the type-declaration characters %, !, #, and $ (see Section 9.6.2, "Declaring Variable Types"). The first character in a variable name must be a letter. If a variable begins with **FN**, it is assumed to be a call to a user-defined function that has been defined with the **DEF FN** statement. A variable name cannot be a reserved word, but embedded reserved words are allowed.

For example, the following statement is illegal because **LOG** is a reserved word:

```
10 LOG = 8
```

Reserved words include all BASIC commands, statements, function names, and operator names (see Appendix B, "BASIC Reserved Words," for a complete list of reserved words).

8.5.2 Declaring Variable Types

Variable names can be declared as either numeric values or string values. String variable names are written with a dollar sign ($) as the last character, as in the following example:

```
A$ = "SALES REPORT"
```

The dollar sign is the type-declaration character for string variables; that is, it "declares" that the variable will represent a string. Numeric variable names can declare integer values (denoted by a "%" suffix), single-precision values, (denoted by either a "!" suffix, or no suffix), or double-precision values (denoted by a "#" suffix). Computations with integer- and single-precision variables are less accurate than those with double-precision variables. However, you may want to declare a variable to have a lower precision type for one of the following reasons:

- Variables of higher precision use more memory.
- Arithmetic computation times are longer for higher precision numbers: a program with repeated calculations runs faster with integer variables.

The default type for a numeric variable is single precision.

The type-declaration characters for numeric variables and the memory requirements (in bytes) for storing each variable type are listed in Table 8.1.

Table 8.1
Variable-Type Memory Requirements

Declaration Character	Variable Type	Bytes Required
%	Integer	2
!	Single precision	4
#	Double precision	8
$	String	4 bytes for descriptor 2 bytes for string backpointer 1 byte for each character in string

The following are examples of BASIC variable names:

Variable Name	Variable Type
PI#	Double precision
MINIMUM!	Single precision
LIMIT%	Integer
FIRSTNAME$	String
ABC	Single precision (default type)

The BASIC statements **DEFINT**, **DEFSTR**, **DEFSNG**, and **DEFDBL** can be included in a program to declare the types for certain variable names. By using one of these **DEF***type* statements, you can specify that all variables starting with a given letter or range of letters will be of a certain variable type, without having to use the trailing declaration character. For more information on the **DEF***type* statements, see **DEF***type* in Chapter 10, "Statement and Function Reference."

8.5.3 Array Variables

An array is a group or table of values referred to by the same variable name. The individual values in an array are called elements. Array elements are also variables, and can be used in any BASIC statement or function which uses variables. Declaring the name and type of an array and setting the number of elements in the array is known as *dimensioning* the array.

Each element in an array is referred to by an array variable that is subscripted with an integer or an integer expression (single- and double-precision expressions can be used as array subscripts; however, they are truncated to integer values). An array-variable name has as many subscripts as there are dimensions in the array. For example, V(10) refers to a value in a one-dimensional array, while T$(1,4) refers to a value in a two-dimensional string array. The default maximum subscript value for any array dimension is 10; this maximum value can be changed by using the **DIM** statement. (See the reference pages for **DIM** in Chapter 10, "Statement and Function Reference" for more information). The maximum number of dimensions for an array is 63. The maximum number of elements per dimension is 32,768. The maximum amount of memory that can be taken for an array is 64K.

Note

The array-variable T and the simple variable T in the following example refer to two distinct variables:

```
DIM T(11)
T = 2 : T(0) = 1              'T is simple variable
FOR I% = 0 TO 10              'T(0) is element of array
    T(I% + 1) = T * T(I%)
NEXT
```

Array elements, like nonarray variables, require a certain amount of memory, depending on the variable type. See Table 8.1 for information on the memory requirements for storing arrays.

8.6 Expressions and Operators

An expression can be a string or numeric constant, a variable, or a single value obtained by combining constants, variables, and other expressions with operators. Operators perform mathematical or logical operations on values. The operators provided by BASIC can be divided into four categories, as follows:

1. Arithmetic
2. Relational
3. Logical
4. Functional

8.6.1 Hierarchy of Operations

The BASIC operators have an order of precedence; that is, when several operations take place within the same program statement, certain kinds of operations will be executed before others. Operations are executed in the following order:

A. Arithmetic Operators
   1. Exponentiation
   2. Negation
   3. Multiplication and Division
   4. Integer Division
   5. Modulo Arithmetic
   6. Addition and Subtraction

B. Relational Operators (=, >, <, <>, <=, and >=)

C. Logical Operators
   1. **NOT**
   2. **AND**
   3. **OR** and **XOR**
   4. **EQV**
   5. **IMP**

An exception to the order of operations listed above occurs when an expression has adjacent exponentiation and negation operators; in this case, the negation is executed first. For example, the following statement prints .0625 (4^{-2}), not -16 (-4^2):

```
print 4 ^ - 2
```

If the operations are of the same level, the left-most one is executed first, the right-most last, as in the following example:

```
A = 3 + 6 / 12 * 3 - 2  'A = 2.5
```

The order of operations in this example is as follows:

1. 6 / 12
2. 0.5 * 3
3. 3 + 1.5
4. 4.5 - 2

8.6.2 Arithmetic Operators

Use parentheses to change the order in which arithmetic operations are performed. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained. Here are some sample algebraic expressions and their BASIC counterparts:

Algebraic Expression	BASIC Expression
$\frac{X-Y}{Z}$	(X-Y)/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
X(−Y)	X*(-Y)

Generally, two consecutive operators must be separated by parentheses. Exceptions to this rule are * −, * +, ^ −, and ^ + ; therefore, the last expression in the right-hand column above could also be written X*−Y.

8.6.2.1 Integer Division

Integer division is denoted by the backslash (\) instead of the forward slash (/), which indicates floating-point division. Before integer division is performed, operands are rounded to integers, and hence must be greater than -32,768.5 and less than +32,767.5. The quotient of an integer division is truncated to an integer.

Example

```
PRINT 10\4, 10/4, -32768.499\10, -32768.499/10
```

Output:

```
 2              2.5            -3276           -3276.8499
```

8.6.2.2 Modulo Arithmetic

Modulo arithmetic is denoted by the modulus operator **MOD**. Modulo arithmetic provides the integer value that is the remainder of an integer division.

Example

```
X% = 10.4\4
REMAINDER% = INT(10.4) - 4*X%    '10\4 = 2, with remainder 2
PRINT REMAINDER%, 10.4 MOD 4
```

Output:

```
 2              2
```

8.6.2.3 Overflow and Division by Zero

If the evaluation of a numeric expression results in division by 0 (case 1), 0 being raised to a negative power (case 2), or overflow (case 3), then one of the following things will happen:

1. If the program was compiled with the Debug (/D) option, it will display a "Division by Zero" (case 1 or case 2) or "Overflow" (case 3) error message.

a. If there is no error-trapping routine, the program ends.

b. If there is an error-trapping routine to handle the error, control branches to the target label (see **ON ERROR GOTO** in Chapter 10, "Statement and Function Reference." for more information on error-trapping).

2. If the program was compiled without the Debug option, the error is not detected.

8.6.3 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (nonzero) or "false" (0). This result can then be used to make a decision regarding program flow.

Table 8.2

Relational Operators and Their Functions

Operator	Relation Tested	Expression
=	Equality[a]	X = Y
< >	Inequality	X < > Y
<	Less than	X < Y
>	Greater than	X > Y
<=	Less than or equal to	X <= Y
>=	Greater than or equal to	X >= Y

[a]The equal sign is also used to assign a value to a variable.

When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first. For example, the following expression is true if the value of `X` plus `Y` is less than the value of `T - 1` divided by `Z`:

```
X + Y < (T - 1)/Z
```

8.6.4 Logical Operators

Logical operators perform tests on multiple relations, bit manipulations, or Boolean operations, and return a true (nonzero) or false (zero) value to be used in making a decision.

Examples

```
IF D < 200 AND F < 4 THEN 80

WHILE I > 10 OR K < 0
.
.
.
WEND

IF NOT P THEN PRINT "Name not found"
```

There are six logical operators in BASIC; they are listed below in order of precedence:

Operator	Operation Performed
NOT	Logical complement
AND	Conjunction
OR	Disjunction (inclusive or)
XOR	Exclusive or (same precedence as **OR**)
EQV	Equivalence
IMP	Implication

Each operator returns results as indicated in Table 8.3. A "T" indicates a true value and an "F" indicates a false value. Operators are listed in order of operator precedence.

Table 8.3

Values Returned by Logical Operations

X	Y	NOT X	X AND Y	X OR Y	X XOR Y	X IMP Y	X EQV Y
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

In an expression, logical operations are performed after arithmetic and relational operations. Logical operators convert their operands to 16-bit, signed, two's complement integers in the range -32,768 to +32,767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1, respectively, as in the following example. (Note the similarity of the output from this program with Table 8.3: "T" becomes -1, and "F" becomes 0.)

Example

```
PRINT " X        Y       NOT      AND      OR       XOR      IMP      EQV"
PRINT
I = 10 : J = 15
X = (I = 10) : Y = (J = 15)      'X is true (-1); Y is true (-1)
GOSUB TRUTHTABLE
X = (I > 9) : Y = (J > 15)       'X is true (-1); Y is false (0)
GOSUB TRUTHTABLE
X = (I <> 10) : Y = (J < 16)     'X is false (0); Y is true (-1)
GOSUB TRUTHTABLE
X = (I < 10) : Y = (J < 15)      'X is false (0); Y is false (0)
GOSUB TRUTHTABLE
END

TRUTHTABLE:
   PRINT X "      " Y "      ";NOT X "      " X AND Y "      " X OR Y;
   PRINT "      " X XOR Y "      " X IMP Y "      " X EQV Y
   PRINT
RETURN
```

Output:

X	Y	NOT	AND	OR	XOR	IMP	EQV
-1	-1	0	-1	-1	0	-1	-1
-1	0	0	0	-1	-1	0	0
0	-1	-1	0	-1	-1	-1	0
0	0	-1	0	0	0	-1	-1

Logical operators compare each bit of the first operand with the corresponding bit in the second operand to compute the bit in the result; in these "bit-wise" comparisons, a 0 bit is equivalent to a "false" value (F) in Table 8.3, while a 1 bit is equivalent to a "true" value (T).

It is possible to use logical operators to test bytes for a particular bit pattern. For instance, the **AND** operator can be used to mask all but one of the bits of a status byte, while the **OR** operator can be used to merge two bytes to create a particular binary value.

Example

```
PRINT 63 AND 16          '    63 (binary 111111)
                         'AND 16 (binary  10000)
                         '------------------
                         '    16 (binary  10000)

PRINT -1 AND 8           '    -1 (binary  1111111111111111)
                         'AND  8 (binary              1000)
                         '---------------------------------
                         '     8 (binary              1000)

PRINT 10 OR 9            '    10 (binary 1010)
                         'OR   9 (binary 1001)
                         '-------------------
                         '    11 (binary 1011)

PRINT 10 XOR 10,         'Always 0

PRINT NOT 10, NOT 11, NOT 0      'NOT X = -(X + 1)
```

Output:

```
 16
 8
 11
```

```
 0          -11            -12            -1
```

8.6.5 Functional Operators

A function is used in an expression to call a predetermined operation to be performed on an operand. For example, **SQR** is a functional operator used twice in the following assignment statement:

```
A = SQR (20.25) + SQR (37)
```

BASIC incorporates two kinds of functions: intrinsic and user-defined.

8.6.5.1 Intrinsic Functions

Functions perform operations on their operands and return values. BASIC has many predefined functions built into the language. Examples are the **SQR** (square root) and **SIN** (sine) functions.

8.6.5.2 User-Defined Functions

Functions can be defined by the user with the **DEF FN** statement. These functions are defined only for the life of a given program, and are not part of the BASIC language. In addition to **DEF FN**, the Microsoft XENIX BASIC Compiler allows you to define subprograms. (For more information on defining your own functions and subprograms, see the entries for **DEF FN** and **SUB...END SUB** in Chapter 10, "Statement and Function Reference.")

8.6.6 String Operators

A string expression consists of string constants, string variables, and other string expressions combined by string operators. There are two classes of string operations: concatenation and string function.

The act of combining two strings is called concatenation. The plus symbol (+) is the concatenation operator for strings. For example, the following program fragment combines the string variables A$ and B$ to produce the value FILENAME.

```
A$ = "FILE": B$ = "NAME"
PRINT A$ + B$
PRINT "NEW " + A$ + B$
```

Output:

```
FILENAME
NEW FILENAME
```

Strings can be compared using the following relational operators:

< > = < > <= >=

Note that these are the same relational operators used with numbers. String functions are similar to numeric functions, except that the operands are strings rather than numeric values. String comparisons are made by taking corresponding characters from each string and comparing their ASCII codes. If the ASCII codes are the same for all the characters in both strings, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If the end of one string is reached during string comparison, the shorter string is said to be smaller if they are equal to that point. Leading and trailing blanks are significant. The following are examples of true statements:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"                'where B$ = "8/12/85"
```

Thus string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks. See Appendix A, "ASCII Character Codes," for more information on the ASCII collating sequence.

8.7 Type Conversion

When necessary, BASIC will convert a numeric constant from one type to another according to the following rules:

- If a numeric constant of one type is set equal to a numeric variable of a different type, the numeric constant will be stored as the type declared in the variable name, as in the following example:

  ```
  A% = 23.42
  PRINT A%
  ```

Output:

```
 23
```

However, if a string variable is set equal to a numeric value, or vice versa, a "Type Mismatch" error message is generated.

- During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision: that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision, as in this example:

```
X% = 2 : Y! = 1.5 : Z# = 100
PRINT X% / Y! * Z#
```

Output:

```
 133.3333373069763
```

Although the preceding result is displayed in double precision, (because of the double-precision variable Z#), it has only single-precision accuracy, since the first operation (X% / Y!) is calculated with single-precision accuracy. This explains the nonsignificant digits (73069763) after the fifth decimal place. Contrast this with the output from the following example:

```
X% = 2 : Y# = 1.5 : Z# = 100
PRINT X% / Y# * Z#
```

Output:

```
 133.3333333333333
```

- Logical operators convert their operands to integers and return an integer result. Operands must be in the range –32,768 to +32,767 or an "Overflow" error message is generated.

- When a floating-point value is converted to an integer, the fractional portion is rounded, as in this example:

```
TOTAL% = 55.88
PRINT TOTAL%
```

Output:

```
 56
```

Chapter 9
Compiler-Interpreter Language Differences

9.1 Dynamic and Static Arrays 129
9.2 Using Metacommands 132
9.2.1 Metacommand Syntax 133
9.2.2 Processing Additional Source Files: $INCLUDE 133
9.2.3 Dimensioned Array Allocation: $STATIC and $DYNAMIC 134
9.2.4 Source Listing Format: $LIST 134
9.2.5 Object Code Listing Format: $OCODE 135
9.2.6 Controlling the Listing Format 135
9.2.7 Changing Internal Module Names: $MODULE 136
9.3 New BASIC Statements and Functions 136
9.4 Compiler-Interpreter Differences 137
9.4.1 A Comparison of Compilation and Interpretation 138
9.4.2 Source File Format 139
9.4.3 Operational Differences 139
9.4.4 Implementation Differences 139
9.4.4.1 Expression Evaluation 140
9.4.4.2 Integer Variables 141
9.4.4.3 Double-Precision Arithmetic Functions 141

9.4.4.4 Double Precision Loop Control Variables 142
9.4.4.5 String Size 142
9.4.4.6 String Space Implementation 142
9.4.4.7 Newline Character 142
9.4.4.8 Program Line Extension Character 142
9.4.5 Statements and Functions Not Accepted by the Compiler 143
9.4.6 Statements Requiring Modification 143
9.5 Enhanced Statements and Functions 144

This chapter explains how to use some of the features added to XENIX BASIC to create more efficient and useful programs. It also describes the difference between interpretation and compilation, and lists the differences between interpreted BASIC and the XENIX BASIC compiler. Even if you are an experienced BASIC programmer, you should read this chapter to learn about the new features, and to find out if you need to change any source files created with the interpreter.

Table 9.1 lists the functions and statements covered in this chapter. See Chapter 10, "Statement and Function Reference" for a complete description of all the XENIX BASIC functions and statements.

Table 9.1

Functions and Statements Described in This Chapter

Enhanced in Compiler	New to Compiler†	May Require Modifying of Interpreter Programs
COMMON	COMMAND$	CALL
DEF FN	LBOUND	CALLS
END	LOCK	CHAIN
ERASE	REDIM	CLEAR
FOR...NEXT	SHARED	DEF*type*
FRE	STATIC	DIM
GOSUB	SUB...END SUB	FIELD
GOTO	UBOUND	RESUME
OPEN	UNLOCK	RUN
RETURN		USR
STOP		

9.1 Dynamic and Static Arrays

The XENIX BASIC compiler supports two types of arrays: static arrays and dynamic arrays. An array is static if space to hold its elements is allocated at compile time; it is dynamic if allocation occurs at run time. Static arrays occupy slightly less space in memory than dynamic arrays, and static array elements can be accessed faster. However, dynamic arrays provide more efficient use of memory because space for the array is not allocated until it is needed, and can be released and reused.

Arrays dimensioned with constant subscripts are considered static by default; arrays dimensioned with variable subscripts are considered dynamic. You can use the $STATIC and $DYNAMIC metacommands, which are explained in Section 9.3.3, to override these defaults. Both static and dynamic arrays are dimensioned with the **DIM** statement, which has the same syntax in the compiler as in the interpreter.

Dynamic arrays can be redimensioned with the **REDIM** statement. The **REDIM** statement changes the amount of space allocated to a dynamic array. When a **REDIM** statement is executed, the array it refers to is deallocated, then reallocated with the new dimensions. The old array-element values are lost. (Note that **REDIM** does not require a previous **DIM** statement, and thus can also allocate new arrays.) Static arrays cannot be reallocated with **REDIM**.

The **ERASE** statement operates differently on static and dynamic arrays. On a static array, **ERASE** sets all array elements to zero, or to null strings. On a dynamic array, **ERASE** deallocates the array. The dynamic array must be redimensioned with a **DIM** statement before it can be referenced again.

The syntax of the **ERASE** and **REDIM** statements is described in Chapter 10, "Statement and Function Reference."

The following rules apply when dimensioning arrays with **DIM**:

1. Static arrays can only be dimensioned once. Dynamic arrays can be redimensioned using the **REDIM** statement, or with an **ERASE** *array* **DIM** *array* sequence.

2. **DIM** statements that allocate dynamic arrays are considered executable statements, and must appear after all **COMMON** statements in the program.

3. When a **DIM** statement that allocates a dynamic array is executed, the array must be currently unallocated, otherwise a "Redimensioned Array" error message appears.

Warning

Bounds checking is performed on arrays only when the program is compiled with the **-D** option. Severe run-time errors may result when the bounds of an array are exceeded in programs compiled without this option. The upper bound of an implicitly dimensioned static array defaults to 10.

Examples

The following example allocates space for a static two-dimensional array:

```
REM $STATIC
DIM A(10,20)
```

This example allocates space for dynamic arrays C and D:

```
10 INPUT "how many?"; N
20 REM $DYNAMIC
30 DIM C(2,3,4)
40 DIM D(N)
```

Line 50 sets all elements to 0 in static array A; line 60 erases dynamic arrays C and D:

```
50 ERASE A
60 ERASE C,D
```

Line 70 redimensions array C:

```
70 REDIM C(4,5,6)
```

9.2 Using Metacommands

Metacommands tell the compiler to perform certain actions while it is compiling the source file. XENIX BASIC metacommands can do the following:

- Read in and compile other BASIC source files at specific points during compilation (**$INCLUDE**)
- Control the allocation of dimensioned arrays (**$STATIC** and **$DYNAMIC**)
- Change internal module names (**$MODULE**)
- Control the format of listing files
- Control what parts of the source file are included in a listing file

This section describes the metacommands available with the Microsoft XENIX BASIC Compiler, and how to use them. Table 9.2 summarizes the XENIX BASIC metacommands and their functions. Metacommands with on/off switches are on by default.

Table 9.2
XENIX BASIC Metacommands

Name	Function
$DYNAMIC	Causes dynamic allocation of arrays
$INCLUDE:'*file***'**	Switches compilation from the current source file to *file*.
$LINESIZE:*size*	Sets the width of the source code listing, in columns.
$LIST[+|-]	Turns on or off source listing. Errors are always listed.
$MODULE:'*name***'**	Changes an internal module name passed to the linker.
$OCODE[+|-]	Turns on or off listing of disassembled object code.
$PAGE	Skips to next page.
$PAGEIF:*number*	Skips to next page if *number* lines or less left on the listing page.
$PAGESIZE:*number*	Sets length of listing, in lines.
$SKIP[:*number*]	Skips *number* lines or to end of page.
$STATIC	Causes static allocation of arrays.
$TITLE'*title*	Sets the source listing page title.

9.2.1 Metacommand Syntax

Metacommands begin with a dollar sign ($) and are always enclosed in a program comment. More than one metacommand can be given in one comment. Multiple metacommands are separated by the white-space characters space, tab, or line feed. Metacommands that take arguments have a colon between the metacommand and the argument:

REM $*METACOMMAND*: *argument*

String arguments must be enclosed in single quotation marks. White space between the elements of a metacommand is ignored. The following are all valid forms for metacommands:

```
REM $METACOMMAND1 $METACOMMAND2
REM $METACOMMAND1 : 'string argument'  $METACOMMAND2
```

Note that *no space* may appear between the dollar sign and the rest of the metacommand.

To put metacommands in comments, place a character that is not a tab or space before the *first* dollar sign on the line. For example, on the following line both metacommands are ignored:

```
REM x$METACOMMAND1 $METACOMMAND2
```

9.2.2 Processing Additional Source Files: $INCLUDE

The **$INCLUDE** metacommand instructs the compiler to switch processing from the current source file to the BASIC file named in the argument. When end-of-file of the included source is reached, the compiler continues processing the original file. Because compilation begins with the line immediately following the line in which **$INCLUDE** occurred, **$INCLUDE** should be the last statement on the line. The following statement is correct:

```
999 DEFINT I-N : REM $INCLUDE: 'COMMON.BAS'
```

The following restrictions must be observed:

1. Variables in included files must match variables in the main program.

2. Included files must not contain **END** statements, or **GOTO** statements to nonexistent lines.
3. Included files created with the interpreter must be saved with the ,A option.

9.2.3 Dimensioned Array Allocation: $STATIC and $DYNAMIC

The **$STATIC** and **$DYNAMIC** metacommands tell the compiler how to allocate memory for arrays. If a **$DYNAMIC** or **$STATIC** metacommand is present, all subsequent array declarations in a **DIM** statement are allocated accordingly, with one exception: implicitly dimensioned arrays (arrays *not* dimensioned with the **DIM** statement) are considered static even when the **$DYNAMIC** metacommand is present.

If the **$STATIC** metacommand is present, all subsequent arrays in a source file are statically allocated.

Using **$STATIC** and **$DYNAMIC** with the **DIM**, **REDIM**, **ERASE**, and **COMMON** statements is discussed in detail in Section 10.1.2, "Static and Dynamic Arrays."

9.2.4 Source Listing Format: $LIST

The $LIST metacommand turns on and off source listing. The source listing gives you the hexadecimal address of each line in the program relative to the start of the .EXE file, the hexadecimal offset from the start of the data segment for any data values generated by the line, the program line itself, and any warning and error messages generated during compilation. Source code listing is turned on by default. To turn source code listing off at any point in the program, add the following line:

```
REM $LIST-
```

To turn source code listing back on, insert the following line:

```
REM $LIST+
```

The **$LIST** metacommand has no effect on whether or not a source code listing is produced; it only affects what parts of the source code are placed in the listing. A source code listing is produced only if you request it when you start the compiler.

9.2.5 Object Code Listing Format: $OCODE

The **$OCODE** metacommand turns on and off disassembled object code listing. In addition to the source code listing information, the listing produced by **$OCODE** contains the assembly-language code that corresponds to the program. The listing is produced only for the portion of the source file between the **$OCODE+** and **$OCODE-** metacommands.

The **$OCODE** metacommand has no effect on whether or not a source code listing is produced; it only affects what parts of the source code a disassembly listing is produced for. To produce a listing you still must give the **-L** option on the **bascom** command line.

9.2.6 Controlling the Listing Format

The metacommands listed in Table 9.3 control the listing file format. Note that there must be a colon between the metacommand and its argument.

Table 9.3

Listing Format Commands

Metacommand	Effect
$TITLE:*title*	Sets listing title
$SUBTITLE:*subtitle*	Sets listing subtitle
$LINESIZE:*size*	Sets width of listing, in columns
$PAGESIZE:*size*	Sets length of listing, in lines
$PAGE	Skips to next page
$PAGEIF:*number*	Skips to next page if there are *number* lines or less left on listing page
$SKIP[**:***number*]	Skips *number* lines or to end of page

9.2.7 Changing Internal Module Names: $MODULE

The **$MODULE** metacommand allows you to change the internal module name that is passed to the linker. This is useful when you want the module name to be different from that of the source file.

If you use the **$MODULE** metacommand, it must appear before the first executable statement.

If each of the BASIC modules you link together does not have a unique module name, the results will be unpredictable.

Note

The segment names generated by this compiler are different from previous XENIX BASIC Compiler releases. These changes should not affect any assembly-language routines linked with a BASIC application, but if any problems arise, consult the memory map in Chapter 7, "Interfacing With Other Languages."

9.3 New BASIC Statements and Functions

The statements and functions in Table 9.4 are new to the XENIX BASIC Compiler. They are described completely in Chapter 10, "Statement and

Function Reference."

Table 9.4
New BASIC Statements and Functions

Statement/Function	Description
COMMAND$	Returns elements of the command line used to invoke the program, and the XENIX environment.
LBOUND/UBOUND	Returns the lower and upper bounds for a specified array dimension.
LOCK/UNLOCK	Controls access by other processes to all or part of an opened file.
REDIM	Changes the space allocated to a dynamic array.
SHARED	Gives global access to variables.
STATIC	Designates multiline function or subprogram variables as local to the function or subprogram.
SUB...END SUB	Marks the beginning and end of a subprogram.

9.4 Compiler-Interpreter Differences

The BASIC language understood by the XENIX BASIC compiler is slightly different from, but generally compatible with, the Microsoft XENIX BASIC Interpreter. Effort has been made to preserve as much compatibility as possible, but some changes are required because of internal differences between the compiler and the interpreter. If you wish to compile programs originally developed for the interpreter, you may need to modify the source code in your interpreted BASIC program, or use a special compiler option, for one of the following reasons:

1. Your program would work more efficiently with some of the new or enhanced statements and functions supported by the XENIX BASIC Compiler.
2. Your program may contain statements and functions that work differently in the compiler and the interpreter.

3. Your program may contain statements and functions, such as **ON ERROR GOTO**, that require it be compiled with a special option. (See Section 9.4.3, "Operational Differences," for a list of when special options are required.)
4. Your interpreted BASIC program may use one of the statements or functions not supported by the compiler.

The following sections describe these language differences and some rules you should keep in mind when preparing source files for the compiler.

9.4.1 A Comparison of Compilation and Interpretation

A microprocessor can execute only its own machine language instructions; it cannot directly execute BASIC language statements. Before a program can be executed, the BASIC program statements must be translated into the machine language of the microprocessor. Compilers and interpreters are two types of programs that perform this translation. The following discussion explains the difference between the two translation processes.

When you run an interpreted BASIC program, the interpreter analyzes each BASIC statement, translates it into machine language, and executes the statement immediately. Statements that are executed repeatedly (inside a **FOR...NEXT** loop, for example) are translated before each execution of the statement. The interpreter cannot look ahead in the program to see what's coming up, or analyze the whole program and perform actions related to such an analysis, such as size and speed optimizations. The compiler, on the other hand, translates the entire source program into machine code *before* any individual program instruction is executed. It can examine the entire program and generate machine code that executes as fast as possible.

Compiled programs execute faster than interpreted programs. The execution time you save with the compiler depends on the content of your program, but in most cases, XENIX BASIC compiler programs execute 3 to 10 times faster than interpreted programs. If you make maximum use of integer variables, program execution can be up to 30 times faster. Programs with many input or output operations will not benefit as much because of hardware speed limitations.

9.4.2 Source File Format

The compiler expects the source file to be in ASCII (American Standard Code for Information Interchange) format. If you create a file with the BASIC Interpreter, it must be saved with the ,A option; otherwise, the interpreter encodes the text of your BASIC program in a special format that the compiler cannot read. If this happens, reload the BASIC interpreter and resave the file in ASCII format, using the **,A** option. For example, the following interpreter command saves the file "MYPROG.BAS" in ASCII format:

```
SAVE "MYPROG.BAS",A
```

9.4.3 Operational Differences

If your program contains any of the statements listed in Table 9.5, you must compile it with the options indicated.

Table 9.5

Operational Differences

Statement	Option
ON ERROR GOTO	-E
ON *event* GOSUB	-E
RESUME	-X
RESUME NEXT	-X
RESUME 0	-X
TRON/TROFF	-D

9.4.4 Implementation Differences

There are minor implementation differences between the compiler and the interpreter. Differences related to the items listed below are described in Sections 9.4.4.1 through 9.4.4.8:

- Floating-point calculations
- Expression evaluation

- Use of integer variables
- Double precision arithmetic functions
- Double precision loop variables
- String size
- String space implementation
- Newline character
- Program line extension character

9.4.4.1 Expression Evaluation

During expression evaluation, the Microsoft XENIX BASIC Compiler converts operands of different types to the type of the more precise operand.

For example, the following expression causes J% to be converted to single precision and added (in single precision) to A!:

```
X=J%+A!+Q#
```

The sum of J% and A! is converted to double precision and added (in double precision) to Q#. The sum is then converted *back* to single precision and assigned to X.

The BASIC compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter the following statements yield 40000 for A!:

```
I%=20000
J%=20000
A!=I%+J%
```

The Microsoft BASIC Compiler, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode and arithmetic overflow occurs. If the **-D** option is set, the error is detected. Otherwise, an incorrect answer is produced.

With the compiler, I% + J% yields the integer value -25536 which is then converted to a floating-point value and saved in A!.

Besides the above type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

```
I%=20000
J%=-18000
K%=20000
M%=I%+J%+K%
```

If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, overflow does occur. The compiler follows the rules of operator precedence, and parentheses can be used to direct the order of evaluation. No other guarantee of evaluation order can be made.

9.4.4.2 Integer Variables

To produce the fastest and most compact object code possible, you should make maximum use of integer variables. (Integer variables occupy less space than other types.) For example, the following program executes approximately 30 times faster by replacing `I`, the loop control variable, with `I%` or by declaring `I` an integer variable with **DEFINT**:

```
FOR I = 1 TO 10
A(I) = 0
NEXT I
```

It is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

9.4.4.3 Double-Precision Arithmetic Functions

The XENIX BASIC Compiler allows you to use double precision floating-point numbers as operands for arithmetic functions, including all of the transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, SQR). Only single-precision arithmetic functions are supported by the interpreter.

If you are using the interpreter to develop a program with double-precision arithmetic variables, but you plan to compile the final version, your program development strategy should be the following:

1. Implement your Microsoft BASIC program using single precision operands for all functions that you later intend to be double precision.

2. Debug your program with the interpreter to determine the soundness of your algorithm before converting variables to double precision.
3. Declare all desired variables as double precision. Your algorithm should be sound at this point.
4. Compile and link your program. It should implement the algorithm that you have already debugged with the interpreter, but with double the precision in your arithmetic functions.

9.4.4.4 Double Precision Loop Control Variables

The compiler, unlike the interpreter, allows the use of double precision loop control variables. This allows you to increase the precision of the increment or to increase the range of loops.

9.4.4.5 String Size

The compiler supports strings of up to 32767 characters. (The interpreter supports strings of up to 128 characters.) Each string descriptor requires 4 bytes of memory.

9.4.4.6 String Space Implementation

The compiler and interpreter differ in their implementation and maintenance of string space. Note that using either **POKE** with **PEEK** and **VARPTR**, or using assembly language routines to change string descriptors may cause a "String Space Corrupt" error message.

9.4.4.7 Newline Character

XENIX BASIC terminates lines with the newline (linefeed) character (ASCII code 10), instead of the carriage return (ASCII code 13).

9.4.4.8 Program Line Extension Character

The complier uses the underscore character (_) to create "logical" lines of more than 254 characters. The interpreter uses the backslash (\). The underscore removes the significance of the line feed. Be sure to include a space before the underscore, or as the first character on the next line.

9.4.5 Statements and Functions Not Accepted by the Compiler

The statements and functions listed in Table 9.6 cannot be used in a compiled program because they perform editing operations on the source file or they interfere with program execution.

Table 9.6

Statements and Functions Not Accepted by the Compiler

AUTO	LIST	MKSBCD$
CONT	LLIST	NEW
CVDBCD	LOAD	RENUM
CVSBCD	MERGE	SAVE
DELETE	MKDBCD$	USR
EDIT		

9.4.6 Statements Requiring Modification

If your interpreter program contains any of the statements listed in Table 9.7, you will probably need to modify your source code before you can compile the program. These statements are described in detail in Chapter 10, "Statement and Function Reference."

Table 9.7
Statements Requiring Modification

Statement	Modification
CALL/CALLS	The *name* argument is the name of the subroutine or subprogram being called.
CHAIN	The compiler does not support the **ALL, MERGE, DELETE** or *line-number* options.
COMMON	**COMMON** statements must appear before any executable statements.
DEF*type*	**DEF***type* statements should be moved to the beginning of compiled programs.
DIM	All **DIM** statements declaring static arrays must appear at the beginning of compiled programs.
RESUME	If an error occurs in a single-line function, the compiler attempts to resume program execution at the line containing the function.
RUN	The object of a **RUN** statement cannot be a **.BAS** file; it must be an executable (**.EXE**) program. The interpreter **R** option is not supported. However, **RUN** {*linenumber*|*linelabel*} is supported in the compiler; this restarts the program at the specified line.

9.5 Enhanced Statements and Functions

Increased functionality has been added to the functions and statements listed in Table 9.8. The enhancements are described in detail in Chapter 10, "Statement and Function Reference."

Table 9.8
Enhanced Statements and Functions

Statement/Function	Enhancement
COMMON	Data can be shared between the main program and subprograms.
DEF FN	The compiler allows multiline functions as well as single-line functions.
END	Ends multiline function definitions and subroutines as well as BASIC programs.
ERASE	For static arrays, ERASE resets array elements to null strings or zeros; for dynamic arrays, ERASE clears arrays from memory.
FOR...NEXT	The compiler supports double-precision control values in FOR...NEXT loops.
FRE	FRE(-1) returns the size of the largest free block of large numeric array space; FRE(*numeric-expression*) (where *numeric-expression* ≠ -1) returns the size of the next free block of string space; FRE(*string-expression*) returns the size of the next free block of string space, *after* first compacting free string space.
GOSUB...RETURN	The compiler supports line labels as well as line numbers.
GOTO	The compiler supports branching to line labels as well as line numbers.
OPEN	Controls access to opened files in a multiuser environment.
RESUME	The compiler supports resuming at line labels as well as line numbers.

Chapter 10
Statement and Function Reference

Reference Format

Each statement or statement in the alphabetical reference that follows is described using the following format:

Heading	Function
Syntax	Gives the correct syntax for the statement or function. Functions return a value of a particular type and can be used wherever an expression can be used; functions cannot appear by themselves on a BASIC line. Statements do not return a value, but they can appear by themselves on a BASIC program line.
Action	Summarizes what the statement or function does.
Remarks	Describes arguments and options in detail, and explains how to use the statement or function.
See Also	Cross-references related statements and functions. (This is an optional section that does not appear with every reference entry.)
Example	Gives sample commands, programs, and program segments that illustrate the use of the given statement or function. (This is an optional section that does not appear with every reference entry.)
Compiler/Interpreter Differences	Tells whether or not the compiler statement or function is enhanced or different from the same statement in the interpreter. (This is an optional section that does not appear with every reference entry.)

ABS Function

Syntax

ABS(*x*)

Action

Returns the absolute value of the expression *x*.

Example

```
PRINT ABS(7*(-5))
```

Output:

```
35
```

ASC Function

Syntax

ASC(*x$*)

Action

Returns a numerical value that is the ASCII code for the first character of the string *x$*.

See Also

CHR$

Example

The following example prints out the ASCII code of the first character in X$ ("T"):

```
10 X$="TEST"
20 PRINT ASC(X$)
```

Output:

```
84
```

ATN Function

Syntax

ATN(*z*)

Action

Returns the arctangent of x, or the angle whose tangent is x.

Remarks

The argument "x" is a rational number.

The result of ATN is in the range $-\pi/2$ to $\pi/2$ radians, where $\pi = 3.141593$.

This function is evaluated in double precision in the decimal version. In the binary version, results are given in single precision when the argument "x" is in single precision and in double precision when the argument is in double precision.

Example

```
10 LET X = 3
20 PRINT ATN(X)
```

Output (in the decimal version):

```
1.2490457723983
```

In other words, an angle of 1.2490457723983 radians has a tangent of 3.

CALL Statement

Syntax

CALL *variable-name* [[(*argumentlist*)]]

Action

Calls an assembly language subroutine or a compiled routine written in another high-level language.

Remarks

The CALL statement is the only way to transfer program flow to an external subroutine.

To call an ISAM subroutine, you must use the −i option when invoking MS-BASIC.

The *variable-name* is a double-precision variable that names the subroutine being called. This variable must have the value 0, and contain an address that is the starting point in memory of the subroutine. On the first invocation of the CALL statement, MS-BASIC obtains the 32-bit entry point address from the name list of the executing version of MS-BASIC. MS-BASIC then stores the 32-bit address in *variable-name*. (Since 32-bits of precision require a double precision variable, "Type mismatch error" is displayed if *variable-name* is any other variable type.)

The *variable-name* must be the name of an existing subroutine.

Note that certain language processors (such as the C compiler) create names beginning with the underscore character (_). These are not legal MS-BASIC variable names. Therefore, MS-BASIC searches for two entry points. The preferred entry point is the same as *variable-name*. If this entry point is not found, MS-BASIC will search for the entry point constructed by using the underscore character. For example, if the *variable-name* called is XYZ, MS-BASIC will first search for XYZ. If it doesn't find this, it will search for _XYZ, a C compiler-style name. If it finds this, it will start execution at _XYZ. The value of this address is saved in *variable-name*. Subsequent calls using the same *variable-name* will skip the search task and start execution at the address in *variable-name*.

MS-BASIC does not preserve case. Therefore, all entry point names must be entirely in upper case letters.

The *argumentlist* contains the variables that are passed to the external subroutine. No previously unreferenced scalar variable may follow an array element in the *argumentlist.* If this happens, an "Illegal function call" error will result.

Invocation of the CALL statement causes the following results:

- For each argument in the *argumentlist,* the 2-byte offset of the argument's location within the data segment (DS) is pushed onto the stack.
- MS-BASIC's return address code segment (CS), and offset (IP) are pushed onto the stack.
- Control is transferred to the user's routine.

If the argument is a variable, the address of that variable is pushed on the stack.

If the argument is a constant or an expression, BASIC evaluates it and the result is stored in a temporary location. The address of the temporary is pushed on the stack. After the routine returns control to the BASIC program, the temporary is thrown away. From the point of view of the BASIC program, this process is identical to pass "by value." That is, changes to the copy given to the called routine do not affect the original in the BASIC program. Strings are handled somewhat differently (see information that follows on passing strings).

If the argument is numeric, the address of the data is passed. If the argument is a string, the address of the *string descriptor* is passed. The first two bytes of the string descriptor contain the length of the string (0 to 32,767 alphanumeric characters), while the second two bytes are a pointer to the

actual characters.

Warning

The subroutine must change neither the contents of any of the four bytes of the string descriptor, nor any area outside of the string.

If the argument BASIC passes in *argumentlist* is a string variable or constant, problems can arise. Wherever possible, BASIC copies string descriptors instead of entire strings. If a string literal is assigned to a variable or passed to an assembly language routine, the string descriptor points to the program text. Look at an example:

```
10 LET A$ = "some text"
20 CALL OOF#(A$, " or other")
```

If A$ is modified by OOF#, then *both the program text on line 10 and the value of A$* are changed. This happens because A$'s string descriptor points to the program text. If OOF# modifies its second parameter, the program text is modified. The next time MS-BASIC executes line 20, the second argument to OOF# will be the changed value.

To avoid these problems, make a string expression of a string literal you plan to pass. You can do this by concatenating "" to the string literal in your program and assigning it to the variable, as in the following example:

```
10  LET A$ = "MYSTRING" + ""
20  CALL SUBRT#(A$)
```

This causes the string literal "MYSTRING" to be copied into string space, where the subroutine may safely change A$ without affecting the program text.

Alternatively, if you have the following code:

```
10 LET A$ = "MYSTRING"
20 CALL SUBRT#(A$ + "")
```

a copy of A$ is passed to the routine. Changes made by the subroutine are made to the local copy only. A$ is unchanged by the called routine.

See Also

CALLS, VARPTR

Example

This example calls the subroutine MYROUT and passes the variables I, J, and K to it:

```
120 CALL MYROUT#(I,J,K)
```

Compiler/Interpreter Differences

The **CALL** statement can be used with the compiler to transfer program control to one of the following:

1. A preassembled assembly language, C or FORTRAN '77 program that was compiled or assembled in large model.
2. A compiled Microsoft Pascal procedure.
3. A compiled Microsoft FORTRAN procedure without any parameters.

The syntax of the CALL statement is identical in the interpreter and the compiler. However, the parameters have slightly in different meanings in the compiler. The syntax of the CALL statement is:

CALL *name* [[(*argument-list*)]]

In programs compiled with the XENIX BASIC compiler, the CALL statement takes the following arguments:

Argument	Description
name	is the name of the subroutine being called. It is limited to 31 characters. Assembly-language subroutine names must be PUBLIC symbols within the subroutine (i.e., they must be recognized as global symbols by the linker).
argument-list	contains the variables or constants that **CALL** passes to the subroutine.

If you are calling an assembly-language subroutine that uses string arguments, you may need to recode it. Because the compiler allows strings of up to 32767 characters in length, the string descriptor requires 4 bytes rather than 3 as in the interpreter. The bytes are the low byte and high byte of the length, followed by the low byte and high byte of the address.

Assembly language subroutines are FAR procedures; return to Microsoft BASIC using an intersegment RET instruction. It is the responsibility of the assembly language subroutine to preserve the value in the BP register.

The CALLS statement is used to call an assembly-language subroutine, or a Microsoft FORTRAN procedure that has parameters. It has the same syntax as CALL, but CALLS passes the segmented addresses of arguments in the *argument-list*, while CALL passes unsegmented addresses.

Example

The following program, "REC.BAS", calls an assembly-language subprogram named "SORT", which sorts the 50 integers stored in array a().

```
OPTION BASE 1
DEFINT a
DIM a(50)
OPEN "receipts" FOR INPUT AS #1
FOR i = 1 TO 50
   INPUT #1,a(i)
NEXT
CALL SORT(a(),1,50)
.
.
.
END
```

In the CALL statement, the addresses of the first and last integers to be sorted are passed to SORT. SORT was declared PUBLIC in the assembly language source file by including the file name on the **bascom** command line:

```
bascom rec.bas sort.s
```

The **bascom** automatically calls the assembler (**as**) to assemble files with a ".s" extension.

CALLS Statement

Syntax

CALLS *variablename* [[(*argumentlist*)]]

Action

Calls an assembly language subroutine or a compiled routine written in another high-level language.

Remarks

The CALLS statement works just like CALL, except that it passes the segmented addresses of all arguments. (CALL passes unsegmented addresses.)

You should use CALLS when accessing routines written with the FORTRAN calling convention. All FORTRAN parameters are call-by-reference segmented addresses.

CDBL Function

Syntax

CDBL(*x*)

Action

Converts *x* to a double precision number.

Example

This example converts the single precision variable RADIUS! to a double precision number, then uses that value in a predefined formula to calculate the area of a circle:

```
10 RADIUS! = 13/7
20 RAD = CDBL(RADIUS!)
30 PRINT "SINGLE PRECISION = ";RADIUS!
40 PRINT "DOUBLE PRECISION = ";RAD
50 DEF FNAREA(R) = 3.14159283# * R^2
60 ROUND = FNAREA(RAD)
70 PRINT: PRINT "AREA = "; ROUND
```

Output:

```
SINGLE PRECISION = 1.857143
DOUBLE PRECISION = 1.857142806053162

AREA = 10.83528900146484
```

CHAIN Statement

Syntax

CHAIN [MERGE]*filespec*[,[*expression*][,ALL][,DELETE *range*]]

Action

Invokes another BASIC program and optionally passes variables to it from the current program.

Remarks

The string expression *filespec* is the name of the program to be executed.

You may also use the COMMON statement to pass variables.

The numeric *expression* is a line number, or an expression that evaluates to a line number, in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. A RENUM command does not affect *expression*.

With the ALL option, every variable in the current program is passed to the called program. If you omit the ALL option, the current program must contain COMMON statements to list the variables that are passed.

If you use the ALL option, but do not use *expression*, a comma must hold the place of *expression*. For example, the first example below is correct and the second is incorrect:

```
CHAIN "NEXTPROG",,ALL
CHAIN "NEXTPROG",ALL
```

In the latter case, BASIC assumes that ALL is a variable name and evaluates it as a line number expression.

The MERGE option allows a subroutine to be brought into the BASIC program as an overlay. That is, the current program and the called program are merged. If you want to merge the two programs, the called program must be an ASCII file.

After using an overlay, you should usually delete it so you can bring in a new overlay. To do this, use the DELETE option.

The line numbers in *range* are affected by the RENUM command.

Note

The CHAIN statement with the MERGE option preserves the current OPTION BASE setting.

CHAIN leaves files opened.

If you omit the MERGE option, CHAIN does not preserve variable types or user-defined functions for use by the chained program; that is, you would have to restate in the chained program any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables.

When using the MERGE option, you should place user-defined functions in program lines that occur prior to

- any CHAIN MERGE statements
- the lowest-numbered lines that will be merged into this program from another program
- the lowest-numbered lines that will be deleted from this program using the DELETE option

Otherwise, the user-defined functions will be undefined after the merge is complete.

See Also

COMMON, MERGE

Example 1

This program demonstrates chaining using COMMON to pass variables. When the main program, PROG1, gets to line 80, it chains to the chained program, PROG2, which loads the string in B$. At line 85 of PROG2, control chains back to the main program at line 90.

You can observe this process by the text that is printed as the programs execute.

Main Program:

```
10  REM THIS PROGRAM DEMONSTRATES CHAINING
15  REM USING COMMON TO PASS VARIABLES.
20  REM SAVE THIS MODULE ON DISK
25  REM AS "PROG1" USING THE A OPTION.
30  DIM A$(2),B$(2)
40  COMMON A$(),B$()
50  A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED"
60  A$(2)="VALUES BEFORE CHAINING."
70  B$(1)=": B$(2) ="
80  CHAIN "PROG2"
90  PRINT: PRINT B$(1): PRINT: PRINT B$(2): PRINT
100 END
```

Chained Program:

```
10  REM THE STATEMENT "DIM A$(2),B$(3)"
15  REM MAY ONLY BE EXECUTED ONCE.
20  REM HENCE, IT DOES NOT APPEAR IN THIS MODULE.
30  REM SAVE THIS MODULE ON THE DISK AS "PROG2"
35  REM USING THE A OPTION.
40  COMMON A$(),B$()
50  PRINT: PRINT A$(1);A$(2)
60  B$(1)="NOTE THAT SPECIFYING A STARTING LINE NUMBER"
70  B$(2)="WHEN CHAINING AVOIDS THE DIMENSION STATEMENT"
80  B$(3)="NEEDED IN PROG1"
85  CHAIN "PROG1",90
90  END
```

Example 2

This second example illustrates the MERGE, ALL, and DELETE options. After the first program loads A$, control chains to line 1010 of the second program. At the second program's line 1040 there is a chain to the third program's line 1010. This passes all variables, but deletes the second program's lines.

You can observe this process through the descriptive text that prints as the programs execute.

```
10  REM THIS PROGRAM DEMONSTRATES CHAINING USING
15  REM THE MERGE, ALL, AND DELETE OPTIONS.
20  REM SAVE THIS MODULE ON THE DISK AS "MAINPRG".
30  A$="MAINPRG"
40  CHAIN MERGE "OVRLAY1",1010,ALL
50  END


1000  REM SAVE THIS MODULE ON THE DISK AS
1005  REM "OVRLAY1" USING THE A OPTION.
1010  PRINT A$; " HAS CHAINED TO OVRLAY1."
1020  A$="OVRLAY1"
1030  B$="OVRLAY2"
1040  CHAIN MERGE "OVRLAY2",1010,ALL, DELETE 1000-1050
1050  END


1000  REM SAVE THIS MODULE ON THE DISK AS
1005  REM "OVRLAY2" USING THE A OPTION.
1010  PRINT A$; " HAS CHAINED TO ";B$;"."
1020  END
```

Compiler/Interpreter Differences

The XENIX BASIC compiler does not support the ALL, MERGE, DELETE, or *line-number* options to the CHAIN statement that are available in the interpreter. For this reason you should use COMMON to pass variables from one program to another. Note also that files are left open during chaining.

Example

The following example converts a number in any base to a decimal number. The base and the number are input in the main program, "main.bas", which then chains to a second program, "digit.bas". This second program splits the number from the main program into separate digits, converts those string values to numeric values, and stores the numeric values in an array, a(). Control then chains to a third program, "dec.bas", which actually changes the number to a decimal number, deallocates (with **ERASE**) the array in which the digits were stored, and prints the decimal number. Control then chains back to the main program, which repeats the process, as long as the value input for the base b is not zero. Note the use of **COMMON** in all three programs to share variables.

```
REM ** This program is main.bas **
COMMON a(1),n$,b,ln
```

```
INPUT "base,number:  ",b,n$
PRINT
WHILE b
   ln = LEN(n$)
   DIM a(ln)
   CHAIN "digit"
WEND
END


REM ** This program is digit.bas **
COMMON a(1),n$,b,ln
m = ln -1
FOR j = 0 TO m
   a$ = MID$(n$,j+1,1)
   IF a$ < "A" THEN a(m-j) = VAL(a$) _
      ELSE  a(m-j) = ASC(a$) - 55
NEXT
CHAIN "dec"


REM ** This program is dec.bas **
COMMON a(1),n$,b,ln
dec = 0
FOR i = 0 TO (ln-1)
   dec = dec + a(i)*b^i
NEXT
ERASE a
PRINT "Decimal # = ";dec : PRINT
PRINT "Input 0 for base to end program."
CHAIN "main"
```

Sample output:

```
base,number:  16,43E

Decimal # =  1086

Input 0 for base to end program.
base,number:  0,
```

CHDIR Statement

Syntax

CHDIR "*pathname*"

Action

Changes the current operating directory.

Remarks

The string expression *pathname* specifies the name of the directory which is to be the current directory. The *pathname* must be a string of less than 128 characters.

See Also

MKDIR, RMDIR

Examples

The following makes SALES the current directory:

```
CHDIR "SALES"
```

The following makes the directory "pteshoes" (which is in the directory "../dancewear/shoes") the current directory:

```
CHDIR "../dancewear/shoes/pteshoes"
```

CHR$ Function

Syntax

CHR$(*code*)

Action

Returns a string whose one character has the ASCII value given by *code*.

Remarks

CHR$ is commonly used when sending a special character to the screen or to a device. For instance, you could send the ASCII code for the bell character, CHR$(7), to cause the speaker to beep. You could also send a form feed, CHR$(12), to clear the screen and return the cursor to the home position.

See Also

ASC

Example

The command

```
PRINT CHR$(77)
```

displays the following character:

```
M
```

CINT Function

Syntax

CINT(*x*)

Action

Converts *x* to an integer by rounding the fractional portion.

Remarks

If *x* is not in the range -32768 to 32767, an "Overflow" error message is generated.

CINT differs from FIX and INT in that CINT *rounds* the fractional part of *x*, while FIX and INT *truncate* the fractional part.

The CDBL and CSNG functions are related to CINT. They convert numbers to the double precision and single precision data types, respectively.

See Also

CDBL, CSNG, FIX, INT

Example

The command

```
PRINT CINT(45.67), CINT(-45.5)
```

displays the following:

```
46          -46
```

CLEAR Statement

Syntax

CLEAR [,[*variable-space*][,*stack-size*]][,*total-memory*]

Action

The CLEAR statement performs the following actions:

- Allocates available memory for specified work areas.
- Closes all files.
- Clears all COMMON variables.
- Resets numeric variables and arrays to zero.
- Resets the stack and string space.
- Resets all string variables to null.
- Releases all disk buffers.
- Resets to null all DEF FN and DEFINT/SNG/DBL/STR statements.

Remarks

The *total-memory* parameter, the last one in the syntax, allocates how much memory is to be reserved for BASIC, the stack, user code and data space, and block storage that you can use to BLOAD, BSAVE or PEEK and POKE assembly language routines.

The *stack* parameter allocates memory for use by BASIC's stack.

The *variable-space* argument allocates *total-memory* space minus the assembly language block storage space. If the CLEAR statement does not explicitly allocate *variable-space*, BASIC allocates the same amount of space that it did for *total-memory*.

The default data segment size will vary, as ISAM and user data (from general CALL code) may or may not be in the data segment.

Example

This sets all numeric variables to zero, all string variables to null and closes all files opened by BASIC:

```
CLEAR
```

Compiler/Interpreter Differences

In the compiler, the *stack-size* and *total-memory* parameters are ignored. Also, the compiler's CLEAR statement does not clear DEF*type* statements. These declarations are fixed at compile time and cannot vary.

CLOSE Statement

Syntax

CLOSE [[#][*filenumber1*[, [#][*filenumber2* ...]]]

Action

Concludes I/O to a file.

Remarks

The CLOSE statement is complementary to the OPEN statement.

The integer argument *filenumber* is the number with which the file was opened. A CLOSE with no arguments closes all open files.

The association between a particular file and *filenumber* terminates upon execution of a CLOSE statement. The file may then be reopened using the same or a different *filenumber*; likewise, that *filenumber* can be reused to open any file.

A CLOSE of a sequential output file writes the final buffer of output.

The END, SYSTEM, CLEAR, and RESET statements and the NEW command always close all files automatically. (STOP does not close files.)

Example

```
CLOSE #1,#2
```

CLS Statement

Syntax

CLS

Action

Erases contents of the screen and sets the cursor position to the upper left hand corner of the screen.

Example

```
10 FOR I% = 1 TO 10
20   PRINT "Line Number ";I%
30 NEXT I%
40 INPUT "Press return when ready", READY$
50 CLS
60 PRINT "The cursor is back at the top of the screen."
```

COMMAND$ Function

Syntax

string-expression=COMMAND$

Action

Returns the command line used to invoke the program, then elements of the XENIX environment.

Remarks

Only the main program in a chain is allowed to call COMMAND$.

COMMAND$ returns one command line parameter per call. At the end of the parameter list, the next COMMAND$ function returns an empty string. Subsequent COMMAND$ functions return the elements of the XENIX environment in the order displayed by the **set** command.

COMMON Statement

Syntax

COMMON *variable-list*

Action

Passes variables to a chained program.

Remarks

The COMMON statement is used in conjunction with the CHAIN statement. You can put COMMON statements anywhere in a program, but it is a good idea to put them at the beginning; otherwise, they may not be read due to changes in flow of control, and chained programs will then not work.

A particular variable can appear in only one COMMON statement. Array variables are specified by appending "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and leave out the COMMON statement.

Some versions of BASIC allow the number of dimensions in the array to be included in the COMMON statement. This implementation of BASIC accepts that syntax, but ignores the numeric expression itself. For example, the following statements are both valid and are considered equivalent:

```
COMMON A()          COMMON A(3)
```

The number in parentheses is the number of dimensions, not the dimensions themselves. For example, the variable A(3) in the example above might correspond to a DIM statement of DIM A(5,8,4).

Example

This fragment passes numeric variables (A, B, C), a numeric array (D), and a string variable (G$) to the chained program, PROG3:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
```

Compiler/Interpreter Differences

In the compiler, the COMMON statement has been enhanced to allow data to be shared among program modules. The enhanced syntax is:

COMMON [SHARED] */blockname/ variable-list*

SHARED is an optional attribute. Use SHARED to pass variables from a main program to a subprogram within the same module; this way, you don't need the SHARED statement, or another COMMON statement, within the subprogram. You also need the SHARED attribute with COMMON when the subprogram being called is in a separately compiled module.

The *blockname* is any valid BASIC identifier up to 31 characters long. Use *blockname* when not every subprogram shares all variables in the *variable-list*. Items in a named COMMON statement are not preserved across a chain to a new program.

The *variable-list* is a list of variables and arrays used by subprograms or chained-to programs. The same variable cannot appear in more than one COMMON statement. *Static* array variables are specified by appending "()" to the variable name; *dynamic* array variables are specified by appending "(*num*)" to the variable name, where *num* is an integer constant indicating the number of dimensions in the array. Using COMMON with arrays is described in Section X.X, "Referencing Arrays in Common Statements."

With the BASIC interpreter, you may put **COMMON** statements anywhere in a program. With the compiler, however, the COMMON statement must appear in a program before any executable statements. All statements are executable, except the following:

- COMMON
- DEF*type*
- DIM (for static arrays)
- OPTION BASE
- REM
- All metacommands

If you use static array variables in a COMMON statement, then you must declare them in a preceding DIM statement. If an array is dynamic, the DIM statement follows the COMMON statement, since a DIM statement is an executable statement when the array it dimensions is dynamic.

When you use COMMON with CHAIN, both the chaining program and the chained-to program require a COMMON statement. With the XENIX BASIC Compiler, the *order* of variables must be the same for all COMMON statements communicating between chaining and chained-to programs. If the *size* of the common region in the chained-to program is smaller than the region in the chaining program, the extra COMMON variables in the chaining program are ignored. If the size of the common region in the chained-to program is larger, the additional COMMON variables are initialized to zeros and null strings.

To ensure that programs can share common areas, place COMMON declarations in a single "include" file and use the $INCLUDE metacommand in each program. (See Chapter 6, "Using Metacommands," for a discussion of the $INCLUDE statement.)

Example

This program fragment shows the use of an "include" file to share COMMON statements among programs:

```
REM ** This file is menu.bas **
REM $INCLUDE:'comdef.bas'
.
.
.
CHAIN "progl"
END


REM ** This file is progl.bas **
REM $INCLUDE:'comdef.bas'
.
.
.
END


REM ** This file is comdef.bas **
DIM A(100),B$(200)
COMMON I,J,K,A()
COMMON A$,B$(),X,Y,Z
REM ** End comdef.bas **
```

COS Statement

Syntax

COS(x)

Action

Returns the cosine of x, where x is an angle measured in radians.

Remarks

To convert degrees to radians, multiply the degrees by $\pi/180$, where $\pi = 3.141593$ (approximately).

The calculation of COS(x) is performed in double precision in the decimal version, and in single precision in the binary version.

Example

```
10 X=2*COS(.4)
20 PRINT X
```

Output (in the decimal version):

```
1.8421219800 58
```

CSNG Function

Syntax

CSNG(*x*)

Action

Converts *x* to a single precision number.

See Also

CDBL, CINT

Example

The following example converts the double precision value in A# to a single precision value:

```
10 A# = 975.3421114#
20 PRINT A#: CSNG(A#)
```

Output:

```
975.3421114       975.3421
```

CVD, CVI, CVS Functions

Syntax

CVI(*2-byte string***)**

CVS(*4-byte string***)**

CVD(*8-byte string***)**

Action

Converts random file buffer string values to numeric values.

Remarks

Numeric values read in from a random file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Note

Don't use these functions to convert program strings to numeric strings. For that purpose, use the STR$ function.

See Also

MKI$, MKS$, MKD$, STR$

Example

In the following example, the field widths from a random access file, #1, are declared, and the first record obtained. Then the first field, N$, is converted to a single precision number and loaded into a variable, Y!, that can be used in the processing of data within a program:

```
70 FIELD #1,4 AS N$, 12 AS B$
80 GET #1, 1
90 Y!=CVS(N$)
```

DATA Statement

Syntax

DATA *constant-list*

Action

Stores numeric and string constants accessed by the READ statement.

Remarks

DATA statements are nonexecutable. You may place them anywhere in the program. The READ statement will locate the first unused DATA statement, regardless of its location in the program.

A DATA statement may contain as many constants as will fit on a line (separated by commas).

You may use any number of DATA statements in a program. READ statements access DATA statements in order (by line number). You may think of the data contained in DATA statements as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

The *constant-list* parameter may contain numeric constants in any format; i.e., fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) Items are separated by commas.

You needn't to put double quotation marks around string constants in DATA statements, unless they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed, but may be used.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement.

See Also

READ, RESTORE

Example

This example reads 12 numeric values into an array named A and then three string expressions into string variables:

```
120 DIM A(15)
140 FOR I = 1 TO 12
160   READ A(I)
180 NEXT I
200 DATA 1,2,3,4,5,4,3,2,1,9,7,1,Acme Inc.,"Fort Dix, NJ" "22222"
220 READ COMPANY$, CITY$, ZIP$
```

DATE$ Function

Syntax

DATE$

Action

Retrieves the current date.

Remarks

The DATE$ function returns a ten-character string in the form *mm-dd-yyyy*, where *mm* is the month (01 through 12), *dd* is the day (01 through 31), and *yyyy* is the year.

See Also

TIME$

Example

In the following example, the current date is retrieved with the DATE$ function and assigned to the string variable TODAY$:

```
10 LET TODAY$ = DATE$
20 PRINT "Today's Date is "; TODAY$
```

Possible output:

```
Today's Date is 11-24-1985
```

DEF FN Statement

Syntax

DEF FN*name* [[(*parameter-list*)]]=*function-definition*

Action

Defines and names a function written by the user.

Remarks

The *name* parameter must be a legal variable name. This name, preceded by FN, becomes the name of the function.

The *parameter-list* is a list of parameter names, through which arguments are passed to the function when it is called. The items in the list are separated by commas.

The *function-definition* is an expression that performs the operation of the function. It is limited to one logical line. A logical line may be extended over more than one physical line by using the line continuation character (\) at the end of a physical line. A variable name used in a function definition may or may not appear in the *parameter-list*. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

Argument variables or values given in the function call replace the variables in the *parameter-list* on a one-to-one basis.

DEF FN may define either numeric or string functions. If a numeric type (e.g., double precision) is specified in the function *name*, the value of the numeric expression in *function-definition* is forced to that type before it is returned to the calling statement.

If the function type and the argument type do not match (i.e., one is a string variable and the other is numeric), a "Type mismatch" error message is generated.

You must define a function with a DEF FN statement before calling the function. If a function is called before it has been defined, an "Undefined user function" error message is generated. DEF FN is illegal in direct mode.

Example

Line 10 defines the function FNAB. The function is called in line 20, with the variables I and J replacing the parameters X and Y. The result of the evaluation is assigned to the variable T:

```
5 LET I=SQR(10):J=22/7
10 DEF FNAB(X,Y)=X^3/Y^2
20 T=FNAB(I,J)
30 PRINT I,J,T
```

Output:

```
 3.162278          3.142857          3.20148
```

Compiler/Interpreter Differences

The XENIX BASIC Compiler supports both single-line functions and multiline functions. The syntax of compiled single-line functions is identical to the interpreter syntax. The syntax of a multiline function is as follows:

DEF FN*name* [[(*parameter-list*)]]
.
.
.
FN*name* = *expression*
.
.
.
[[**EXIT DEF**]]
.
.
.
END DEF

The *name* must be a legal variable name. This name, preceded by FN, becomes the name of the function.

The *parameter-list* is a list of variable names, separated by commas. When the function is called, it replaces these variables on a one-to-one basis with the values the program supplies.

The *expression* defines the value returned by the function. A multiline function ends with an END DEF statement.

The statement exits the function and returns to the calling program, but does not define the end of the function. is used to exit the function if an abnormal condition, such as an error, occurs.

Argument variables or values that appear in the function call replace the variables in the parameter list on a one-to-one basis.

Warning

Your program must define a function with a statement before it can call the function. If your program calls a function before it is defined, an "Undefined user function" error occurs.

User-defined functions cannot appear inside other multiline functions, nor can they appear inside blocks.

Your program cannot contain recursive function definitions; that is, a function cannot be defined in terms of itself.

The statement is *not* equivalent to Using a statement to exit a multiline function will cause a severe unrecoverable error.

Example

The following example contains a function definition that converts an angle measure in degrees, minutes, and seconds to an angle measure in radians. (An angle must be given in radians for the trigonometric functions of BASIC to return a meaningful answer.)

```
DEF FNdegrad(d,m,s)
   pi = 3.14159263
   d = d + m/60 + s/3600
   FNdegrad = d * (pi/180)
END DEF
deg = 45 : min = 10
PRINT TAB(5);"Angle measurement";TAB(38);"SINE"
PRINT
FOR sec = 10 TO 50 STEP 10
   PRINT TAB(5) deg chr$(248) min "'" sec chr$(34);
   rad = FNdegrad(deg,min,sec)
   PRINT TAB(35) SIN(rad)
```

```
NEXT
END
```

Output:

```
     Angle measurement                SINE

      45° 10' 10"                   .7091949
      45° 10' 20"                   .7092291
      45° 10' 30"                   .7092632
      45° 10' 40"                   .7092974
      45° 10' 50"                   .7093316
```

DEFINT, DEFSNG, DEFDBL, DEFSTR Statements

Syntax

DEFINT *letter-range*

DEFSNG *letter-range*

DEFDBL *letter-range*

DEFSTR *letter-range*

Action

Declares variables as integer, single precision, double precision, or string.

Remarks

Any variable names beginning with the letters specified in *letter-range* are the type of variable specified by the last three letters of the statement, that is, either INT (integer), SNG (single precision), DBL (double precision), or STR (string). However, a type declaration character always takes precedence over a DEF *type* statement.

If no type declaration statements are encountered, Microsoft BASIC assumes that all numeric variables without declaration characters are double precision variables (for the decimal version) or single precision for the binary version.

Examples

In the following example, all variables beginning with the letter A are designated string variables:

```
10 DEFSTR A
```

In the following example, all variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z are designated integer variables:

```
10 DEFINT I-N,W-Z
```

Compiler/Interpreter Differences

The interpreter and the compiler process **DEF***type* statements somewhat differently. The interpreter must scan a statement each time before it executes it. If the statement contains a variable that does not have an explicit type (signified by !, #, $, or %), the interpreter determines the current default type and uses it. In the example below, when the interpreter encounters line 20, it determines that the current default type for variables beginning with "i" is integer, based on the **DEFINT** statement in line 10. Line 30 then changes this default type to single precision and loops back to line 20. The interpreter must rescan line 20 in order to execute it and this time "iflag" becomes a single-precision variable:

```
10  DEFINT i
20  PRINT iflag
30  DEFSNG i : GOTO 20
```

In contrast, the compiler scans the text only once. Therefore, once a variable occurs in a program line, its type cannot be changed. The compiler, unlike the interpreter, does not allow you to circumvent a **DEF***type* statement by directing program flow around it.

You can see these differences in the output from the program in the example in this section.

Note

I!, I#, I$, and I% are all separate and distinct variables; each one can hold a different value. The effects of this are illustrated in the following example.

See Also

DEF FN

Example

The following program gives different results when you run it with the interpreter and with the compiler. The interpreter assigns variable types each time it scans a statement during program execution, so it allows the program to redeclare the variable type inside the **FOR...NEXT** loop. On the other hand, the compiler statically scans **DEF***type* statements, assigning variable types at compile time, so line 160 applies only to occurrences of "t" variables in program lines after line 160.

```
100  test% = 1            ' integer type
110  test! = 10           ' single-precision type
120  DEFINT t
130  FOR i = 1 TO 3
140     PRINT test
150     test = test + 20
160     DEFSNG t
170  NEXT
180  PRINT
190  test = test + 100
200  PRINT "test =";test
210  PRINT "test% =";test%
220  PRINT "test! =";test!
```

Interpreter output:

```
 1
 10
 30

test = 150
test% = 21
test! = 150
```

Compiler output:

```
 1
 21
 41

test = 110
test% = 61
test! = 110
```

DELETE Statement

Syntax

DELETE *linenumber*[-]

DELETE [*linenumber1*][-*linenumber2*]

Action

Deletes program lines.

Remarks

Microsoft BASIC always returns to command level after executing a DELETE. If *linenumber* does not exist, an "Illegal function call" error message is generated.

Examples

The following statement deletes line 40:

```
DELETE 40
```

The following statement deletes all lines from 20 through 30, inclusive:

```
DELETE 20-30
```

The following statement deletes all lines up to and including line 10:

```
DELETE -10
```

The following statement deletes all lines from 15 to the end of the program:

```
DELETE 15-
```

DIM Statement

Syntax

DIM *subscripted-variable-list*

Action

Specifies the maximum values for array variable subscripts and allocates storage accordingly.

Remarks

If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript greater than the specified maximum is used, a "Subscript out of range" error message is generated. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets the values of all elements of the specified arrays to an initial value of zero. The maximum number of dimensions allowed in a DIM statement is 255.

If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, a "Redimensioned array" error message is generated. Therefore, it is good programming practice to place DIM statements at the top of a program where they will be executed before any references are made to the dimensioned variable.

See Also

ERASE, OPTION BASE

Example

This example shows an array, A, being dimensioned to accept subscript values up to ten. These subscripted variables are then assigned values with a READ statement in the loop from statements 20 to 40.

```
10 DIM A(10)
20 FOR I=0 TO 10
30   READ A(I)
40 NEXT I
     .
     .
     .
1010 DATA 17,22,7.5,3.1416,9,9,7,45,3,2,2
```

Compiler/Interpreter Differences

The DIM statement is similar to the DEF*type* statement in that it is scanned rather than executed. That is, DIM takes effect when it is encountered at compile time and remains in effect until the end of the program; it cannot be re-executed at runtime. Therefore, the practice of putting DIM statements in a subroutine at the end of a program generates severe errors. The compiler sees the DIM statement only after it has assigned the default dimension to arrays declared earlier in the program. If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, an "Array Already Dimensioned" error results.

DIM statements should be placed at the top of a program where they will be read before any references are made to the dimensioned variable.

END Statement

Syntax

END

Action

Terminates program execution, closes all files, and returns control to command level.

Remarks

You may place END statements anywhere in the program to stop program execution.

Unlike the STOP statement, END does not cause a "Break in line *nnnnn*" error message to be printed.

An END statement at the end of a program is optional. Microsoft BASIC returns to command level after executing an END.

See Also

CONT, STOP

Example

In this example, program execution terminates if K is greater than 1,000; otherwise, the program branches to line 20:

```
520 IF K>1000 THEN 900 ELSE 20
      .
      .
      .
900 END
```

Compiler/Interpreter Differences

The statement has been enhanced to end multiline function definitions and subroutines, as well as a BASIC program. The enhanced syntax is:

END [[DEF | SUB]]

ends a multiline function definition; you must use with **END SUB** ends a BASIC subroutine; you must use with

The compiler always assumes an statement at the conclusion of any program, so omitting an statement at the end of a program still produces proper program termination.

EOF Function

Syntax

EOF(*filenumber*)

Action

Tests for the end-of-file condition.

Remarks

Returns −1 (true) if the end of a sequential input file has been reached. While reading data from a sequential input file, use EOF to test for end-of-file to avoid "Input past end" error messages.

When EOF is used with a random access file, it returns true (−1) if the last GET statement was unable to read an entire record.

A file opened to KYBD: is at its end when you type a CONTROL-D.

For files opened to PIPE:, EOF returns −1 (true) if no processes have the pipe opened for output and no data is available to be read from the pipe. If a child process is still active, EOF returns 0 (false).

See Also

OPEN

Example

This example opens the sequential file named DATA for input (that is, records from DATA are to be read into program variables), and assigns it the number 1. The WHILE...WEND loop in lines 30-60 reads records from DATA into the array M, until it reaches the end of the file:

```
5  DIM M(100)
10 OPEN "I",1,"DATA"
20 C=0
30 WHILE NOT EOF(1)
40    INPUT #1, M(C)
50    C= C + 1
60 WEND
```

ERASE Statement

Syntax

ERASE *array-list*

Action

Eliminates arrays from memory.

Remarks

The *array-list* consists of array variable names, separated by commas.

Arrays may be redimensioned after they are erased, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first erasing it, a "Redimensioned array" error message is generated.

Example

```
10 DIM A(10)
20 DIM B(10)
   .
   .
   .
50 ERASE A,B
60 DIM B(99)
   .
   .
   .
```

Compiler/Interpreter Differences

In the XENIX BASIC Compiler, ERASE has the same effect as in the interpreter; namely, it resets the elements of a static array to either zeros or null strings, depending on the type specified by the *arrayname*. The syntax is also the same in the compiler as in the interpreter.

However, executing ERASE on a *dynamic* array causes the array elements to be deallocated. Before your program can refer to the dynamic array again, it must first redimension the array iwth either a DIM or REDIM statement. If you try to redimension an array without first erasing it, a "Duplicate definition" error will occur.

ERR, ERL Functions

Syntax

ERR

ERL

Action

Returns error status.

Remarks

When you are using an error handling routine in your program, the ERR function returns the error code for the error and the ERL function returns the number of the line in which the error was detected. The ERR and ERL functions are usually used in IF...THEN statements to direct program flow in the error handling routine.

See Also

ERROR

Example

This example has an error-handling routine that starts at line 100. If the error is the one the user has defined as 200 (file is empty), then the routine asks for a new file name for input, and resumes execution at line 30. If the error is something else, BASIC prints the message "Unprintable error":

```
10 ON ERROR GOTO 100
20 OPEN "PHONE" FOR INPUT AS 1
30 IF LOF(1) = 0 THEN ERROR 200
      .
      .
      .
90 END
100 REM **ERROR HANDLING FRAGMENT**
110 IF ERR = 200 THEN CONDITION%  = -1
120 IF NOT CONDITION% THEN ERROR ERR
```

```
130 PRINT "Phone number file is empty"
140 INPUT "Name of file with information"; FIL$
150 CLOSE #1
160 OPEN FIL$ FOR INPUT AS 1
170 LET CONDITION% = 0
180 RESUME 30
```

ERROR Statement

Syntax

ERROR *integer-expression*

Action

Simulates the occurrence of a Microsoft BASIC error, or allows user-defined error codes.

Remarks

ERROR can be used as a statement (part of a program source line) or as a command (in direct mode).

The value of *integer-expression* must be greater than 0 and less than 256. If the value of *integer-expression* equals an error code already in use by Microsoft BASIC (see Appendix B, "Error Codes and Error Messages"), the ERROR statement simulates the occurrence of that error and the corresponding error message is printed.

To define your own error code, use a value that is greater than any used by Microsoft BASIC error codes. (Use the highest available values, so compatibility may be maintained when more error codes are added to Microsoft BASIC.) This user-defined error code may then be conveniently handled in an error handling routine.

If an ERROR statement specifies a code for which no error message has been defined, Microsoft BASIC responds with the "Unprintable error" error message. An ERROR statement for which there is no error handling routine prints the error message and halts execution.

See Also

ERR, ERL

Example

This example is part of a game program that allows you to make bets. The program traps the error if you exceed the "house limit." Note that the error code is 210, which Microsoft BASIC doesn't use.

```
        .
        .
        .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET"; B
130 IF B > 5000 THEN ERROR 210
        .
        .
        .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
        .
        .
        .
```

EXP Function

Syntax

EXP(x)

Action

Returns e (the base of natural logarithms) to the power of x, or e^x.

Remarks

If x is greater than 145, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
20 PRINT EXP(4)
```

Output:

```
54.59815
```

FIELD Statement

Syntax

FIELD [[#]]*filenumber*, *fieldwidth* **AS** *string-variable*...

Action

Allocates space for variables in a random file buffer.

Remarks

Before a GET statement or PUT statement can be executed, a FIELD statement must be executed to format the random file buffer.

The *filenumber* parameter is the number under which the file is opened. The *fieldwidth* parameter is the number of characters allocated to *string-variable*.

The total number of bytes that you allocate in a FIELD statement must not exceed the record length that you specified when opening the file. Otherwise, a "Field overflow" error message is generated. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed remain in effect at the same time.

All field definitions for a file are removed when the file is closed.

Note

Do not use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer no longer refers to the random record buffer, but to string space.

See Also

GET, LSET, OPEN, RSET

Example 1

This example allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not place any data in the random file buffer.

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

Example 2

Example 2 illustrates a multiply-defined FIELD statement. In statement 20, the 57-byte field is broken up into 5 separate variables for name, address, city, state and zip code. In statement 30, the same field is assigned entirely to one variable, PLIST$. Statements 60 through 90 check to see if ZIP$, which contains the zip code, falls within a certain range; if it does, the complete address string is printed by lines 75 and 80.

```
10   OPEN "R", #1, "MAILLIST", 57
20   FIELD #1, 15 AS NAM$, 25 AS ADDR$, 10 AS CTY$, _
               2 AS ST$, 5 AS ZIP$
30   FIELD #1, 57 AS PLIST$
40   GET #1, 1
50   WHILE NOT EOF(1)
60      ZCHECK$ = ZIP$
70      IF (ZCHECK$ < "85700" OR ZCHECK$ >"85800") THEN GOTO 90
75        INFO$ = PLIST$
80        PRINT INFO$
90      GET #1
100  WEND
```

Example 3

This example shows the construction of a FIELD statement using an array of elements of equal size:

```
10 FOR LOOP%=0 TO 7
20 FIELD #1,(LOOP%*16) AS OFFSET$,16 AS A$(LOOP%)
30 NEXT LOOP%
```

The result is equivalent to the single declaration:

```
FIELD 1,16 AS A$(0),16 AS A$(1),...,16 AS A$(6),16 AS A$(7)
```

Example 4

This example creates a field in the same manner as Example 3. However, the element size varies with each element:

```
10  DIM SIZE% (4): REM ARRAY OF FIELD SIZES
20  FOR LOOP%=0 TO 4:READ SIZE%(LOOP%): NEXT LOOP%
30  DATA 9,10,12,21,41
120 DIM A$(4): REM ARRAY OF FIELDED VARIABLES
130 OFFSET%=0
140 FOR LOOP%=0 TO 4
150 FIELD #1,OFFSET% AS OFFSET$,SIZE%(LOOP%) AS A$(LOOP%)
160 OFFSET%=OFFSET%+SIZE%(LOOP%)
170 NEXT LOOP%
```

The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),...
SIZE%(4%) AS A$(4%)
```

Compiler/Interpreter Differences

When a FIELD statement is executed, the compiler associates all string variables that reference random file fields with the specified file buffer. Therefore, do not reference these string variables after the file is closed, even if it is reopened later. Instead, issue a new FIELD statement.

FILES Statement

Syntax

FILES [["*filespec*"]]

Action

Prints the names of files residing in a specified directory.

Remarks

The *filespec* parameter is a string expression that includes either the name of a file (with directory pathname, if necessary) or the name of a directory (with pathname). (See Section 4.2, "Filenames and Paths.") If you omit *filespec*, all files in the current directory are listed. If you use *filespec*, all files meeting that specification are listed.

Example 1

This shows all files in the current directory:

```
FILES
```

Example 2

This either shows that the file TEST.BAS in the directory ../PROJ/EXAMPLES exists, or responds with a "File not found" error

```
FILES "../PROJ/EXAMPLES/TEST.BAS"
```

FIX Function

Syntax

FIX(x)

Action

Returns the truncated integer part of x.

Remarks

FIX(x) is equivalent to SGN(x)*INT(ABS(x)). The difference between FIX and INT is that, for negative x, FIX returns the first negative integer greater than x, while INT returns the first negative integer less than x.

See Also

CINT, INT

Example 1

```
PRINT FIX(58.75), INT(58.75)
```

Output:

```
 58             58
```

Example 2

```
PRINT FIX(-58.75), INT(-58.75)
```

Output:

```
 -58            -59
```

FOR...NEXT Statement

Syntax

FOR *counter* = *start* **TO** *end* [**STEP** *increment*]

NEXT [*counter*][,*counter*...]

Action

Allows a series of instructions to be performed in a loop a given number of times.

Remarks

The FOR statement uses *start, end,* and *increment* as fixed numeric expressions, and *counter* as a counter. The expression *start* is the initial value of the counter. The expression *end* is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then *counter* is adjusted by the amount specified by STEP, and is compared with the final value, *end.* If *counter* is still not greater than *end,* then BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is called a FOR...NEXT loop.

If you do not specify STEP, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop. The loop is executed until the counter is less than the final value.

It is a good idea not to change the loop variable within the loop, since doing so can make the program more difficult to debug.

Nested Loops

You may nest FOR...NEXT loops; that is, you may place a FOR...NEXT loop within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop.

A NEXT statement that has the form,

```
NEXT I, J, K
```

is equivalent to the sequence of statements

```
NEXT I
NEXT J
NEXT K
```

If you omit the variable in a NEXT statement, the NEXT statement matches the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is generated and execution is terminated.

Example 1

```
10 FOR I= 5 TO 1 STEP -1
20     FOR J= 1 TO I
30     PRINT "*";
40     NEXT J
50 PRINT
60 NEXT I
```

Output:

```
*****
****
***
**
*
```

Example 2

In this example, the loop does not execute because STEP is positive (+1), and the initial value (1) exceeds the final value (0):

```
10     J=0
20     FOR I=1 TO J
30       PRINT I
40     NEXT I
```

Compiler/Interpreter Differences

The syntax and use of FOR...NEXT is the same as in the interpreter, but the XENIX BASIC compiler supports double-precision control values in its FOR...NEXT loops.

FRE Function

Syntax

FRE(*n*)

FRE(" ")

Action

FRE(*n*) returns the number of bytes in BASIC's memory space that are not being used.

FRE(" ") also returns the number of bytes in BASIC's memory space that are not being used. However, FRE(" ") differs from other forms of FRE in that it compacts string space before it returns the number of bytes that are free.

Example

```
PRINT FRE(0)
```

Possible output:

```
14542
```

Compiler/Interpreter Differences

Although the syntax is the same as in the interpreter, the FRE function has a different use when used with the compiler.

In interpreted BASIC programs, FRE with a numeric argument returns the number of bytes of memory not being used by BASIC. In compiled programs, FRE with a numeric argument returns the size of the next free block of string space.

In both compiled and interpreted BASIC programs, FRE with a string argument returns the number of bytes in BASIC's memory space that are not being used. The string argument also causes FRE to compact free string space into a single block before returning the number of free bytes.

GET Statement

Syntax

GET [[#]]*filenumber*[[,*recordnumber*]]

Action

Reads a record from a random file into a random buffer.

Remarks

The *filenumber* is the number under which the file was opened.

The *recordnum* is the number of the record within the file. The largest possible record number is 4,294,967,295 (or $2^{32}-1$). The smallest record number is 1. If you omit *recordnumber*, the next record (after the last GET) is read into the buffer.

After executing a GET statement, BASIC can use INPUT# and LINE INPUT# to read characters from the random file buffer.

You may use EOF(*filenumber*) after a GET statement to check against

reading beyond the end-of-file.

See Also

INPUT#, LEN, LINE INPUT#, OPEN, PUT

Example

This program section opens a file, declares its field and, so long as the operator answers "yes," gets specified records:

```
10 OPEN "R",#1,"FILE",32
15 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
20 LET ANSWER$ = "YES"
25 WHILE LEFT$(ANSWER$,1) = "Y"
30   INPUT "2-DIGIT CODE";CODE
40   GET #1, CODE
50   PRINT N$
60   PRINT USING "$$###.##";CVS(A$)
70   PRINT P$:PRINT
80   INPUT "More? ", ANSWER$
90 WEND
```

GOSUB...RETURN Statement

Syntax

GOSUB *linenumber1*
.
.
.
RETURN [[*linenumber2*]]

Action

Branches to and returns from a subroutine.

Remarks

The *linenumber1* in the GOSUB statement is the first line of a subroutine, where the program continues execution. A RETURN within the subroutine returns control to the statement just following the GOSUB statement in the program text. A subroutine can be called any number of times in a program. A subroutine can also be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

A subroutine can contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.

The optional *linenumber2* can be included in the RETURN statement to return from the subroutine to a specific line number. You should use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the GOSUB will remain active, and error messages such as "FOR without NEXT" can result.

Subroutines can appear anywhere in the program, but the subroutine should be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example

```
10  GOSUB 400
20  PRINT "BACK FROM SUBROUTINE"
30  END
400 REM**PRINT SUBROUTINE
410 PRINT "SUBROUTINE";
420 PRINT "IN";
430 PRINT "PROGRESS"
440 RETURN
```

Output:

```
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

Compiler/Interpreter Differences

The GOSUB statement has been enhanced to support line labels as well as line numbers. In addition, the compiler RETURN statement supports RETURN *line-number2* (or RETURN *line-label2*). The extended syntax is:

GOSUB { *line-number1* | *line-label1*}
.
.
.
RETURN [[{ *line-number2* | *line-label2*}]]

The extended RETURN statement allows a RETURN from a GOSUB statement to the statement having the specified line number or label, instead of a normal return to the statement following the GOSUB statement. Use this type of return with care, however, because any other GOSUB, WHILE, or FOR statements that were active at the time of the GOSUB will remain active, and errors such as "FOR without NEXT" may result.

The first line of the subroutine is *line-number1* in the GOSUB statement.

You can call a subroutine any number of times in a program. You can also call a subroutine from within another subroutine. Such nesting of subroutines is limited only by available memory.

A subroutine may contain more than one RETURN statement. Simple RETURN statements (without the *line-number2* option) in a subroutine cause Microsoft BASIC to branch back to the statement following the most recent GOSUB statement.

Subroutines may appear anywhere in the program, but it is good programming practice to make them readily distinguishable from the main program. To prevent inadvertent entry into a subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Note

The preceding discussion of subroutines applies only to the targets of GOSUB statements, *not* subroutines delimited by SUB...END SUB.

Example

The following example computes arcsine values for arguments between -1 and 1. This is the inverse of the sine function, so the value returned will be an angle whose measure is between $\pi/2$ radians and $3\pi/2$ radians (90° to 270°).

```
INPUT "sine";x
WHILE (x < -1 OR x > 1)
   PRINT "Illegal sine value."
   PRINT "Sine must be >= -1 and =< 1."
   INPUT "sine";x
WEND
pi = 3.141593
guess2 = pi/2 : guess1 = 3 * (pi/2)
WHILE abs(guess1 - guess2) > .0000005
   GOSUB newval
WEND
PRINT "arcsin ";x;" = "; temp
END



newval:
   temp = (guess1 + guess2)/2
   IF SIN(temp) > x THEN guess2 = temp _
      ELSE guess1 = temp
RETURN
```

Sample output:

```
sine? 3
Illegal sine value.
```

```
Sine must be >= -1 and =< 1.
sine? .43
arcsin  .43 = 2.6971
```

That is, the angle whose sine is .43 has a measure of 2.6971 radians.

GOTO Statement

Syntax

GOTO *linenumber*

Purpose

Branches unconditionally to the specified line number.

Remarks

If the statement that has *linenumber* is an executable statement, execution continues with that statement.

If it is a nonexecutable statement, such as a REM or DATA statement, execution proceeds at the first executable statement encountered after *linenumber*.

It is good programming practice to use structured control (IF...THEN...ELSE, WHILE...WEND, ON...GOTO) instead of GOTO statements as a way of branching, because a program with many GOTO statements can be difficult to read and debug.

Example

The following program fragment prints a two-column list, with circle radius in the first column and circle area in the second:

```
10 READ R
20 PRINT "R =";R,
30 A=3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
```

Output:

```
R = 5    AREA = 78.5
R = 7    AREA = 153.86
R = 12   AREA = 452.16
Out of data in 10
```

Compiler/Interpreter Differences

The syntax of the GOTO statement is the same as in the interpreter, but has been extended to allow branching to line labels, as well as line numbers.

For example, the following program prints the area of the circle that has the input radius:

```
PRINT "Input 0 to end."
start:
   INPUT r
   IF r <= 0 THEN END _
   ELSE _
      a = 3.14 * r^2
      PRINT "Area =";a
GOTO start
```

Sample output:

```
Input 0 to end.
? 5
Area = 78.5
? 7
Area = 153.86
? 12
Area = 452.16
? 0
```

HEX$ Function

Syntax

HEX$(*x*)

Action

Returns a string that represents the hexadecimal value of the positive decimal argument.

Remarks

The argument *x* is any numeric expression. It is rounded to an integer before HEX$(*x*) is evaluated.

See Also

OCT$

Example

```
10 INPUT X
20 A$=HEX$(X)
30 PRINT X "DECIMAL IS " A$ "HEXADECIMAL"
```

Output:

```
? 32

32 DECIMAL IS 20 HEXADECIMAL
```

IF...THEN...ELSE, IF...GOTO Statements

Syntax

IF *expression* **THEN** *target* [**ELSE** *alternative*]

IF *expression* [**THEN**] **GOTO** *target* [**ELSE** *alternative*]

Action

Makes a decision regarding program flow based on the result returned by an expression.

Remarks

The entire IF...THEN...ELSE statement must be on *one* logical line.

If the result of *expression* is true (or is not equal to zero) then *target* is executed. In the first syntax, the *target* consists of either a line number for branching or one or more executable statements. In the second syntax, GOTO is always followed by a line number.

If the result of *expression* is false (or is equal to zero), *target* is ignored and the ELSE *alternative*, if present, is executed. As is the case with *target*, *alternative* is either a line number or one or more statements.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example, the following is a legal statement:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X  _
        THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example,

```
IF A=B THEN IF B=C THEN PRINT "A=C"  _
        ELSE PRINT "A<>C"
```

prints "A<>C" only in the following case:

- A=B
- B<>C

However, it does *not* print "A<>C" when

- A<>B
- B=C

Instead, the program simply continues with the next executable statement. If you wanted the program to print "A<>C" when A<>B and B=C, then the statement would look like this:

```
IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A<>C"    _
     ELSE IF B=C THEN PRINT "A<>C"
```

If you enter a line number after an IF...THEN statement in direct mode, an "Undefined line" error message is generated unless you have previously entered a statement with the specified line number in indirect mode.

Example 1

This statement gets record number I if I is not zero:

```
1200 IF I THEN GET#1,I
```

Example 2

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110:

```
100 IF(I<20) AND (I>10) THEN DB=1979-I:GOTO 300
110 PRINT "OUT OF RANGE"
```

Example 3

This statement causes printed output to go either to the screen or to the printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the printer; otherwise, output goes to the screen:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

Example 4

This last example shows the use of conditional variables. The IF statement is true if the variable BONUS% has the value of +1 and false if that value is 0:

```
100 IF SCORE% >= 90 THEN BONUS% = 1 ELSE BONUS% = 0
120 IF BONUS% THEN PRINT "You received a bonus this week."
140 IF NOT BONUS% THEN PRINT "No bonus this week."
```

INKEY$ Function

Syntax

INKEY$

Action

Returns either a one-character string containing a character read from the keyboard or a null string if no character is pending at the keyboard.

Remarks

No characters are echoed. All characters are passed through to the program except for CONTROL-C, which terminates the program.

Example

This example is a subroutine that, for a given number of counts (TIMELIMIT%), checks to see if there is input from the keyboard. If there isn't, it continues until TIMELIMIT% is exceeded. If there is, it checks to see if the character is ASCII character 13, the carriage return. If it is, then TIMEOUT% is set to zero and control is returned to the main program.

```
1000 REM TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030    A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040    IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050    RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

INPUT# Statement

Syntax

INPUT# *filenumber*, *variable-list*

Action

Reads data items from a sequential file and assigns them to program variables.

Remarks

The *filenumber* is the number used when the file was opened for input. The *variable-list* contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) If a comma is appended to INPUT#, then no question mark is printed, just as with INPUT.

The data items in the file should appear just as they would if you were entering data in response to an INPUT statement. Leading spaces, carriage returns, and linefeeds are ignored with numeric values. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a number. The number terminates on a space, carriage return, linefeed, or comma.

If BASIC is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or linefeed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

If an INPUT# statement attempts to read data from a sequential file to which access has been restricted by a LOCK statement, two options are available. The first is to return control to the program immediately with an accompanying error message. All of BASIC's usual error handling routines can trap and examine this error. If error trapping is not active the error message is:

```
Permission denied
```

See Also

INPUT, LOCK

Example

In this example, the loop opens all the records in a file, printing all the names of employees hired in 1981:

```
10 OPEN "I ",#1,"DATA"
15 WHILE NOT EOF(1)
20      INPUT#1,N$,DEPT$,HIREDATE$
30      IF RIGHT$(HIREDATE$,2) = "81" THEN PRINT N$
40 WEND
```

INPUT$ Function

Syntax

INPUT$(*x*[,[#]*y*])

Action

Returns a string of *x* characters, read from file number *y*.

Remarks

If the file number is not specified, the characters are read from the keyboard.

If the keyboard is used for input, no characters are echoed on the screen. All control characters are passed through except CONTROL-C, which is used to interrupt the execution of the INPUT$ function.

Example 1

The following example lists the contents of a sequential file in hexadecimal:

```
10 OPEN "I",1,"DATA"
20 WHILE NOT EOF(1)
30      PRINT HEX$(ASC(INPUT$(1,#1)));
40 WEND
50 END
```

Example 2

The following example reads input from the keyboard. Depending on which key the user presses ("P" or "S"), the program continues executing at line 500, or branches to line 700 to end:

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
```

INPUT Statement

Syntax

INPUT[*"prompt-string"*{;|,}] *variable-list*

Action

Allows input from the keyboard during program execution.

Remarks

When the program encounters an INPUT statement, it pauses in its execution and prints a question mark to indicate that it is waiting for data. If you include *prompt-string*, the string is printed before the question mark. You then enter the required data at the keyboard.

If INPUT is immediately followed by a semicolon, then no new line is generated after the prompt string is printed. This allows you to continue input on the same line as the prompt.

You can use a comma instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE", B$ prints the prompt with no question mark.

The data that is entered is assigned to the variables given in *variable-list*. The number of data items that you supply must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that you input must agree with the type specified by the variable name. Strings input to an INPUT statement need not be surrounded by quotation marks.

Responding to INPUT with too many or too few items or with the wrong type of value (e.g., string instead of numeric) causes the message

"?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Example 1

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
```

Output

```
? 5

5  SQUARED IS 25
```

Example 2

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 IF R=-1 THEN END
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
```

Output

```
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 171.946
WHAT IS THE RADIUS? -1
```

INSTR Function

Syntax

INSTR([*start*,]*x*$,*y*$)

Action

Searches for the first occurrence of string *y*$ in string *x*$, and returns the position at which the match is found.

Remarks

The optional offset *start* sets the position for starting the search.

If *start* is greater than the number of characters in *x*, that is, if *start* is greater than LEN(*x*$), then INSTR returns 0. INSTR also returns 0 when *x*$ is null or when *y*$ cannot be found.

If *y*$ is null, INSTR returns *start* or 1.

The arguments *x*$ and *y*$ can be string variables, string expressions, or string literals.

Example

This example searches for the first occurrence of the letter "B" in the string "ABCDEB", and then prints the number of that position, first starting at the beginning of the string (no *start*), then starting at the fourth letter of the string (*start* = 4):

```
10 X$="ABCDEB"
20 Y$="B"
30 PRINT INSTR(X$,Y$):INSTR(4,X$,Y$)
```

Output:

```
2  6
```

INT Function

Syntax

INT(*x*)

Action

Returns the largest integer less than or equal to *x*.

Remarks

The argument *x* is a numeric expression.

See Also

CINT, FIX

Example 1

```
PRINT INT(99.89)
```

Output:

```
99
```

Example 2

```
PRINT INT(-12.11)
```

Output (since $-13 < -12.11$):

```
-13
```

KILL Statement

Syntax

KILL "*filespec*"

Action

Deletes a file from disk.

Remarks

If a KILL command is given for a file that is currently OPEN, a "File already open" error message is generated. The *filespec* argument is any legal file name.

Example

This deletes the file named "MailLabels" in the current directory:

```
200 KILL "MailLabels"
```

LBOUND Function

Syntax

LBOUND(*array*[,*dimension*]**)**

Action

Returns the lower bound (smallest available subscript) for the indicated dimension of an array. It is used with the UBOUND function to determine the size of an array.

Remarks

LBOUND takes the following arguments:

Argument	Description
array	The name of the array being dimensioned.
dimension	An integer from 1 to the number of dimensions in *array*.

In the array ACCOUNT(A,B,C,D), A is dimension 1, B is dimension 2, C is dimension 3, and D is dimension 4. So the following statment:

```
LBOUND(ACCOUNT,1)
```

finds the smallest subscript in dimension A,

```
LBOUND(ACCOUNT,2)
```

finds the smallest subscript in dimension B, and so on.

The default lower bound for any dimension is either 0 or 1, depending on the setting of the OPTION BASE statement. If OPTION BASE is 0, the default lower bound is 0, and if OPTION BASE is 1, the default lower bound is 1.

You can use the shortened syntax **LBOUND(***array***)** for one-dimensional arrays, since the default value for *dimension* is 1.

Use the UBOUND function to find the upper limit of an array dimension.

See Also

UBOUND

Example

LBOUND and UBOUND can be used together to determine the size of an array passed to a subprogram, as in the following program fragment:

```
CALL PRNTMAT(ARRAY())
.
.
.
SUB PRNTMAT(A(2)) STATIC
    FOR I% = LBOUND(A,1) TO UBOUND(A,1)
        FOR J% = LBOUND(A,2) TO UBOUND(A,2)
           PRINT A(I%,J%);" ";
        NEXT J%
    PRINT:PRINT
    NEXT I%
END SUB
```

LEFT$ Function

Syntax

LEFT$(*x*$,*n*)

Action

Returns a string containing the leftmost *n* characters of *x*$.

Remarks

The argument *n* is a numeric expression that returns a decimal number that rounds to an integer in the range 0 to 32,767. (This argument is rounded to an integer before LEFT$ is evaluated.) If *n* is greater than the number of characters in *x*$, the entire string (*x*$) is returned. If *n* = 0, the null string (a string with no characters) is returned.

See Also

MID$, RIGHT$

Example

```
10 A$="BASIC"
20 B$=LEFT$(A$,3)
30 PRINT B$
```

Output:

```
BAS
```

LEN Function

Syntax

LEN(*x$*)

Action

Returns the number of characters in *x$*.

Remarks

Nonprinting characters and blanks are counted.

Example

```
10 X$="PORTLAND, OREGON"
20 PRINT LEN(X$)
```

Output:

```
16
```

LET Statement

Syntax

[LET]*variable=expression*

Action

Assigns the value of an expression to a variable.

Remarks

Notice that the word LET is optional; that is, the equal sign is sufficient for assigning an expression to a variable.

Example 1

The following two program fragments work identically:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
150 PRINT D,E,F,SUM
```

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
150 PRINT D,E,F,SUM
```

Output:

```
 12             144            20736          20892
```

Example 2

The following fragment assigns string expressions to string variables, then prints those variables:

```
10 LET T$="THOUGHT FOR THE DAY:"
20 LET A$="Judgment comes from experience."
30 LET B$="  Experience, however, often comes from poor judgment."
40 PRINT T$:PRINT:PRINT A$+B$
```

Output:

```
THOUGHT FOR THE DAY:

Judgment comes from experience.  Experience, however, often comes
from poor judgment.
```

LINE INPUT# Statement

Syntax

LINE INPUT# ***filenumber,string-variable***

Action

Reads an entire line, regardless of delimiters, from a sequential data file to a string variable.

Remarks

The ***filenumber*** is the number under which the file was opened. The ***string-variable*** is the variable name to which the line will be assigned.
LINE INPUT# reads all characters in the sequential file up to a newline. It then skips over the newline. The next LINE INPUT# reads all characters up to the newline. (If a newline is encountered, it is preserved; that is it is preserved as part of the string.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII format is being read as data by another program.

If a LINE INPUT# statement attempts to read data from a sequential file to which access has been restricted by a LOCK statement, two options are available. The first is to return control to the program immediately with an accompanying error message. All of BASIC's usual error handling routines can trap and examine this error. If error trapping is not active the error message is:

```
Permission denied
```

See Also

LINE INPUT, LOCK, SAVE

Example

This example shows the use of both the LINE INPUT and LINE INPUT# statements:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
```

This will first prompt for input with the following:

```
CUSTOMER INFORMATION?
```

If you enter the following data:

```
CUSTOMER INFORMATION? LINDA JONES    234,4    MEMPHIS
```

the output looks like this:

```
LINDA JONES    234,4    MEMPHIS
```

LINE INPUT Statement

Syntax

LINE INPUT[;] [*"prompt-string"*;] *string-variable*

Action

Inputs an entire line to a string variable without the use of delimiters.

Remarks

The *prompt-string* is a string literal that BASIC prints on the screen before input is accepted. Unlike INPUT, LINE INPUT does not print a question mark unless it is part of *prompt-string*. All input from the end of *prompt-string* to the new line is assigned to *string-variable*.

If LINE INPUT is immediately followed by a semicolon, then the newline that you enter to end the input line does not echo on the screen.

You can abort a LINE INPUT statement by typing CONTROL-C, causing BASIC to return to the command level. Entering CONT resumes execution at the LINE INPUT.

Example

See example under LINE INPUT#.

LOC Function

Syntax

LOC(*filenumber*)

Actions

With random files, LOC returns the record number of the last record read or written.

With sequential files, LOC returns the number of records read from or written to the file since it was opened.

Remarks

LOC uses *filenumber* as the number under which the file was opened.

When a file is opened for sequential input, BASIC reads the first record of the file, so LOC will return a 1 even before any input from the file occurs.

With files opened to KYBD:, LOC returns the value 1 if any characters are ready to be read from the standard input. Otherwise it returns 0.

With files opened to PIPE:, LOC returns the value 1 if any characters are ready to be read from the pipe. Otherwise, it returns 0.

See Also

OPEN

Example 1

If # 1 is a sequential file, this example will stop program execution if more than 50 records have been read from or written to since file # 1 was opened:

```
200 IF LOC(1)>50 THEN STOP
```

Example 2

If 3 was opened as a random file, the following language loads to the variable "CURRENT" the record number of the random file record most recently read or written:

```
3010 LET CURRENT = LOC(3)
```

Example 3

The following fragment checks whether the operator has pressed a key before continuing the program:

```
300 OPEN "KYBD:" FOR INPUT AS #1
310    CLS: PRINT TAB(10):"PRESS ANY KEY TO BEGIN"
320 WHILE LOC(1) = 0
330    REM ***  DO NOTHING UNTIL KEY PRESSED ***
340 WEND
```

LOCK Statement

Syntax

LOCK [[#]]*filenum* [[,**READ**]] [[,**WAIT**]] [[, [[*recordnum*]][[**TO** *recordnum*]]]]

Action

Restricts access by other processes to all or part of a file.

Remarks

The *filenum* is the number with which the file was opened.

The *recordnum* is the number of the record within the file. The largest possible record number is 4,294,967,295 (or $2^{32}-1$). The smallest record number is 1.

LOCK will restrict access by other programs to *filenum* in the *recordnum* range, or for the entire file. If you open *filenum* for sequential input or output, then BASIC locks the entire file and any range specifications result in an error. If you open *filenum* in random access mode, then BASIC allows you to specify an inclusive range of records to be locked. If you don't specify a starting *recordnum*", then record number 1 is assumed. You must specify a final *recordnum* whenever requesting a range of locked records.

Locks can be applied in two different ways: partial and total. Total locking is the default for the LOCK statement and is applied to the file by not using the READ keyword. Total locking prevents any access by another program to the locked portion of the file.

Partial locking allows another program to read the file, but prevents that program from modifying the locked portion of the file. Partial locking is applied to the file by specifying the READ keyword in the statement.

In the case where another program has locked a portion of the requested file region, two options are available. The first is to return control to the program immediately with an accompanying error message. All of BASIC's usual error handling routines can trap and examine this error. If error trapping is not active the error message is:

```
Permission denied
```

This is the default option. It is chosen if the WAIT keyword has not been specified in the LOCK statement.

The second option is to wait until the program that issued the original LOCK unlocks the requested region of the file. The presence of the WAIT keyword will force this second option. It is possible to interrupt this wait by pressing CONTROL-C.

A deadlock situation can occur when waiting for a LOCK request. An example of this "deadly embrace" follows:

- A program has File1 open and locked.
- A different program has File2 open and locked.
- The first program executes a LOCK request with the WAIT option on File2.
- The second program executes a LOCK request with the WAIT option on File1.

XENIX will attempt to detect any deadlock situations. If it does detect one, it returns a "Deadlock" error message.

Multiple LOCK statements have a cumulative effect. Locking records 1 through 3 and then locking records 10 through 100 will leave both ranges locked. Locking a record which is already locked will have no effect; the record remains locked and no error is generated. A record locked multiple times with different locking characteristics (READ or WAIT) retains the characteristics of the latest LOCK statement.

It is recommended that a file not be opened simultaneously on multiple channels. However, if it becomes necessary to do so, the following information is important. Locks applied using a different channel number against a single file will act in the same way as multiple LOCK statements issued against the same file. For example,

```
100 OPEN "R", 1, "Employees"
120 OPEN "R", 2, "Employees"
140 LOCK #1, 1 TO 3
160 LOCK #2, READ, 2 TO 5
```

leaves records 1 through 5 of Employees locked. In addition, records 2 through 5 are locked in the READ mode. The first channel closed releases *all* locks against the file. That is, if the next line of the previous program fragment closes #2, then both locks will be released.

Records are locked based on their size and position within the file. If a file is opened multiple times with different specified record sizes, and locks are applied to records, then portions of records can become locked.

The LOCK statement is complementary to the UNLOCK statement.

See Also

UNLOCK

Example 1

The following applies a total lock to records 1 through 32 of file number 4:

```
500 LOCK #4, 1 TO 32
```

Example 2

The following fragment opens a file and locks it. It then allows an operator to update information. When the operator is done, the program unlocks the locked records. (Unlocking the locked records allows other people to work with the file.)

```
1000 OPEN "monitor" AS #1 LEN = 59
1020 FIELD #1, 15 AS PAYER$, 20 AS ADDRESS$, 20 AS PLACE$, 4 AS OWE$
1040 LET CONTINUE$ = "YES"
1060 WHILE LEFT$(CONTINUE$, 1) = "Y"
1080     CLS:LOCATE 10,10
1100     INPUT "Customer Number?    #"; NUMBER%
1120     GET #1, NUMBER%
1140        LOCK #1, READ, NUMBER%
1160        LET DOLLARS! = CVS(OWE$)
1180        LOCATE 11,10: PRINT "Customer: ";PAYER$
1200        LOCATE 12,10: PRINT "Address:  ";ADDRESS$
1220        LOCATE 13,10: PRINT "Currently owes: $";DOLLARS!
1240        LOCATE 15,10: INPUT "Change (+ or -)", CHANGE!
1260        LET DOLLARS! = DOLLARS! + CHANGE!
1280        LSET OWE$ = MKS$(DOLLARS!)
1300     PUT #1, NUMBER%
1320     UNLOCK #1, NUMBER%
1340     LOCATE 17,10: INPUT "Update another? ", CONTINUE$
1360 WEND
```

LOF Function

Syntax

LOF(*filenumber***)**

Action

Returns the length of the named file in bytes.

Remarks

LOF (or the Length-Of-File function) is valid for any file.

Files opened to SCRN:, KYBD:, or LPT1: always return the value 0.

For files opened to PIPE:, LOF returns 1 if any characters are ready to be read from the pipe. If there are no characters, it returns 0.

See Also

OPEN

Example

In this example, the variables REC and RECSIZ contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

```
110 IF REC*RECSIZ>LOF(1) THEN PRINT "INVALID ENTRY"
```

LOG Function

Syntax

LOG(*x*)

Action

Returns the natural logarithm of *x*.

Remarks

The natural logarithm is the logarithm of *x* to the base *e*. (The number *e* is approximately equal to 2.718282.)

The decimal number *x* must be greater than zero.

LOG delivers a double precision result in the decimal version of BASIC, and a single precision result in the binary version.

Example 1

```
PRINT LOG(45/7)
```

Output (in the decimal version):

```
1.860752340149
```

Example 2

```
100 E = 2.718282
110 FOR I = 1 TO 4
120 PRINT LOG(E^I)
130 NEXT I
```

Output:

```
 1         2         3         4
```

LPOS Function

Syntax

LPOS(*x*)

Action

Returns the current column position of the device LPT1:.

Remarks

The argument *x* is a dummy argument.

LPOS does not necessarily give the physical position of the print head.

Example

This example forces a carriage return (CHR$(13)) when the current line printer column exceeds 60:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

LPRINT, LPRINT USING Statements

Syntax

LPRINT [[*expression-list*]]

LPRINT USING *string-exp*; *expression-list*

Action

Prints on the line printer.

Remarks

The *expression-list* contains the string expressions or numeric expressions that are to be printed, separated by semicolons.

The *string-exp* is a string literal (or variable) composed of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.

LPRINT and LPRINT USING are the same as PRINT and PRINT USING, except that output always goes to the line printer with LPRINT and LPRINT USING.

LPRINT causes output to be "spooled", or piped to the lpr command for later printing. Output is piped to the lpr command. No output is printed until one of the END, CLEAR, or SYSTEM statements is executed.

Another way to produce line printer output is to open a file to "LPT1:" and to subsequently PRINT to it. Output to this file is sent to the printer when LPT: is closed.

See Also

LPOS, PRINT, PRINT USING

Examples

See examples in PRINT and PRINT USING.

LSET, RSET Statements

Syntax

LSET *bufferstring=x$*

RSET *bufferstring=x$*

Action

Moves data from memory to a random file buffer in preparation for a PUT statement; also, justifies data in a string.

Remarks

If *x$* requires fewer bytes than were fielded to *bufferstring*, LSET left-justifies *x$* in the field, and RSET right-justifies *x$*. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right end of the string. Numeric values must be converted to strings before they are LSET or RSET.

LSET or RSET can also be used with a nonfielded string variable to left-justify or right-justify a string in a given field.

See Also

MKD$, MKI$, MKS$

Example 1

This example left-justifies three variables into fielded strings preparatory to loading them into a random file:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=MKD$(DESC(4))
170 LSET E$= TOTAL$
```

Example 2

This example illustrates how RSET can be used to format output:

```
100 LET S$ = SPACE$(20)
120 PRINT "12345678901234567890"
140 LET A$ = "3.1416"
160 PRINT A$
180 RSET S$ = A$
200 PRINT S$
```

Output:

```
12345678901234567890
3.1416
                3.1416
```

MID$ Function

Syntax

MID$(*x*$,*n*[,*m*])

Syntax

MID$(*x1*$,*n*[,*m*]) = *x2*$

Action

The function returns a string of length *m* characters from *x*$, beginning with the *n*th character.

The statement replaces a portion of *x1*$ with all or part of *x2*$.

Remarks

The values *n* and *m* are numeric expressions returning decimal numbers that round to integers in the range 1 to 32,768. (These arguments are rounded to integers before MID$ is evaluated.)

In the function syntax, if *m* is omitted or if there are fewer than *m* characters to the right of the *n*th character, all rightmost characters, beginning with the *n*th character are returned. If *n* is greater than the number of characters in *x*$ (that is, if *n* is greater than LEN(*x*$)), then MID$ returns a null string.

In the statement syntax, *n* and *m* are integer expressions, while *x1*$ and *x2*$ are string expressions. The characters in *x1*$, beginning at position *n*, are replaced by the characters in *x2*$. The optional *m* refers to the number of characters from *x2*$ that will be used in the replacement. If *m* is omitted, all of *x2*$ is used. The replacement of characters never exceeds the original length of *x1*$.

Example 1

This is an example of the MID$ function. It returns a string of length 7, starting at the 9th position of B$ (or the first "E" in "EVENING"):

```
10 LET A$="GOOD "
20 LET B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
```

Output:

```
GOOD EVENING
```

Example 2

This is an example of the MID$ statement. It replaces the "MO" in A$ with "KS":

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
```

Output:

```
KANSAS CITY, KS
```

MKD$, MKI$, MKS$ Functions

Syntax

MKI$(*integer-expression*)

MKS$(*single-precision-expression*)

MKD$(*double-precision-expression*)

Action

Converts numeric values to string values for insertion into random file buffers.

Remarks

Any numeric value placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Note

Don't use these functions to convert numbers that are later assigned to string variables. For that purpose, use the STR$ function.

See Also

CVD, CVI, CVS, LSET, RSET

Example 1

The MKD$ statement in line 110 converts a numeric variable, AMT, into a string to be stored in file #1:

```
90 AMT=(K+T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$=MKS$(AMT)
120 LSET N$=A$
130 PUT #1
```

Example 2

This example opens the file "checks" as a random access file, and converts the variables CHECK% (integer), ACCTNO! (single precision), and AMT# (double precision) to string values. These string values are then left-justified in buffer-strings and loaded into the file with a PUT statement:

```
1000 OPEN "checks" AS #1 LEN=34
1020 FIELD #1, 2 AS CHECKNO$, 20 AS PAYEE$, _
               4 AS ACCOUNT$, 8 AS AMOUNT$
        .
        .
        .
1200 LSET CHECKNO$ = MKI$(CHECK%)
1220 LSET PAYEE$ = OWEDTO$
1240 LSET ACCOUNT$ = MKS$(ACCTNO!)
1260 LSET AMOUNT$ = MKD$(AMT#)
1280 PUT #1, CHECK%
```

MKDIR Statement

Syntax

MKDIR "*pathname*"

Action

Creates a new directory.

Remarks

The string expression *pathname* specifies the name of the directory which is to be created. The *pathname* must be a string of less than 128 characters.

See Also

CHDIR, RMDIR

Examples

The following creates the new directory SALES within the current directory:

```
MKDIR "SALES"
```

The following creates the new directory "proofreaders" within the directory "../publications/editorial":

```
MKDIR "../publications/editorial/proofreaders"
```

NAME Statement

Syntax

NAME *old-filename* **AS** *new-filename*

Action

Changes the name of a file.

Remarks

The *old-filename* must exist and *new-filename* must not exist; otherwise, an error will result. These may be any legal filenames.

A file may not be renamed with a new device designation. If this is attempted, an error message is be generated. After a NAME command, the file exists on the same disk, in the same area of disk space, but with the new name.

Example

In this example, the file that was formerly named ACCTS is now named LEDGER:

```
NAME "ACCTS" AS "LEDGER"
```

NEXT Statement

Syntax

NEXT [[*variable1*]][[,*variable2...*]]

Action

Allows a series of instructions to be performed in a loop a given number of times.

Remarks

See "FOR...NEXT" for a discussion of the use of NEXT.

OCT$ Function

Syntax

OCT$(*x*)

Action

Returns a string that represents the octal value of the decimal argument.

Remarks

The argument *x* must be a positive decimal number. It is rounded to an integer before OCT$(*x*) is evaluated.

See Also

HEX$

Example

```
PRINT OCT$(24), OCT$(10.34), OCT$(7.84)
```

Output:

```
30              12              10
```

ON ERROR GOTO Statement

Syntax

ON ERROR GOTO *linenumber*

Action

Enables error handling and specifies the first line of the error handling routine.

Remarks

Once error handling has been enabled, all errors detected, including direct mode errors (for example, syntax errors), will cause a jump to the specified error handling routine. If *linenumber* does not exist, an "Undefined line" error message is given.

The RESUME statement is required to continue program execution.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 if a program encounters an error for which there is no recovery action.

Note

If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does *not* occur within the error handling routine.

See Also

ERL, ERR, ERROR, RESUME

Example

This example shows an error-handling routine. Line 10 directs the program to line 1000 when any errors are detected; if the error is of the type defined by the programmer (ERROR 230, "Null string in INS$"), and the line in which the error occurs is line 110, then line 1000 causes the message "Try Again" to be printed on the screen, and program execution resumes at line 100:

```
10 ON ERROR GOTO 1000
     .
     .
     .
100 INPUT "Insurance Coverage"; INS$
110 IF INS$ = " " THEN ERROR 230
     .
     .
     .
1000 IF ERR = 230 AND ERL = 110 _
     THEN PRINT "Try Again" : RESUME 100
```

ON...GOSUB, ON...GOTO Statements

Syntax

ON *expression* **GOSUB** *linenumber-list*

ON *expression* **GOTO** *linenumber-list*

Action

Branches to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks

The value of *expression* determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list is the destination of the branch. If the value is a noninteger, the fractional portion is rounded.

If the value of *expression* is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of *expression* is negative or greater than 255, an "Illegal function call" error message is generated.

Example

In this example, the operator's response, ITEM, is used to direct program flow to subroutines at lines 150, 300, 320, or 390 depending on whether the user answers 1, 2, 3 or 4:

```
60  LET ITEM = 0
70  WHILE (ITEM < 1) OR (ITEM > 4)
80        INPUT "Which item do you want"; ITEM
90  WEND
100 ON ITEM GOSUB 150, 300, 320, 390
```

OPEN Statement

Syntax

OPEN "*filename***"[FOR** *mode1*] **AS** [#]*filenumber* [**LEN=***record-length*]

OPEN "*mode2***",**[#]*filenumber*,*filename* [,*record-length*]

OPEN "PIPE:*command***"** [**FOR** *mode3*] **AS** [#]*filenumber*

Action

Allows I/O to a file, device, or process.

Remarks

A file must be opened before any I/O operation can be performed on that file.

In the first two syntaxes, OPEN associates a *filenumber* with a *filename*. The *filenumber* is an integer expression whose value is between 1 and 255. The number is used to refer other I/O statements to the file, and is associated with the file for as long as it is OPEN. The *filename* is a string expression containing the name of a file or device.

For the first syntax, *mode1* is one of the following string expressions:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output mode and sets the file pointer at the end of file. A PRINT# or WRITE# statement will then extend (append to) the file.

If *mode1* is omitted, the default random access mode is assumed. Note that, in this statement syntax, you cannot explicitly specify the random input/output mode.

For the second syntax, *mode2* is one of the following:

O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.
A	Specifies sequential output mode and sets the file pointer at the end of file. A PRINT# or WRITE# statement will then extend (append to) the file.

The *record-length* cannot exceed 32,767 bytes. If the *record-length* option is not used, the default length is 128 bytes.

The third syntax, OPEN "PIPE:*command*", allows your BASIC program to open sequential files for output to, or input from, the child process specified by *command*. For example

```
OPEN "PIPE: grep 'Allah' koran" FOR INPUT AS #2
```

executes the XENIX process "grep." The output from this process — all lines containing "Allah" in the file named "koran" — is then stored in file number 2. In this syntax, *mode3* is one of the following:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.

You can also open files to PIPE: for random I/O. This allows your BASIC program to interact with a child process, sending output to the process, and redirecting input from the process back to BASIC. (See Example 4.)

In random I/O to a pipe, there is no field buffer. Instead, characters are sent directly to and from a child process. Since there is no buffer with OPEN PIPE:, the LEN option is ignored in this syntax. In addition, FIELD, PUT, and GET statements are not allowed, and there is no field overflow error.

For files opened to PIPE:, LOC(1) and LOF(1) both return 1 if any characters are ready to be read from the pipe. If there are no characters, they both return 0. EOF returns −1 (true) if no processes have the pipe opened for output and no data is available to be read from the pipe. If the child process is still active, EOF returns 0 (false).

You can also open XENIX pipes with syntax 2; once again, if you try to specify the *record-length* option, it will be ignored, since there is no field buffer to a pipe. For more information on pipes and the OPEN

"PIPE:*command*", see Section 4.1, "Device Independent Input/Output."

Note

A file may be opened under more than one file number if that file is random access or sequential input. However, if a sequential file is being opened for output, it may be opened under only one file number assignment at a time.

Example 1

This example opens the file named MAILING.DAT, assigns it the number 1, and allows data to be added to it without destroying its current contents.

```
10 OPEN "MAILING.DAT" FOR APPEND AS 1
```

Example 2

This example opens the file named INVEN for sequential input as file number 2:

```
10 OPEN "I",2,"INVEN"
```

Example 3

This opens MAILERS.DAT for input from the keyboard:

```
10 OPEN "KYBD:MAILERS.DAT" FOR INPUT AS 6
```

Example 4

The following program fragment opens a PIPE: to the XENIX fgrep command, sends it data from the array PHONELIST$, then stores the output from fgrep in the array CODE$.

```
100 REM * PHONELIST$ IS A LIST OF NAMES AND PHONE NUMBERS.
110 REM * THIS PROGRAM EXTRACTS THE NAME & PHONE NUMBER OF
120 REM * EVERY PERSON LIVING IN AREA CODE 301, AND STORES
130 REM * THIS INFORMATION IN THE ARRAY CODE$.
140 DIM PHONELIST$(1000)
150 OPEN "PIPE: fgrep '(301)'" AS #1
160 J = 0
170 FOR I = 0 TO 1000
180     PRINT#1, PHONELIST$(I)
190     IF LOC(1) = 0 THEN 300
200             LINE INPUT#1, CODE$(J)
210             J = J + 1
220 NEXT I
300 CLOSE #1
```

OPTION BASE Statement

Syntax

OPTION BASE *n*

Action

Declares the minimum value for array subscripts.

Remarks

The argument *n* must be either 0 or 1.

This statement determines whether array subscripts may have values less than one. If *n* is 1, then 1 is the lowest value possible; if *n* is 0, then 0 is the lowest possible value. The default base is 0.

If a program includes an OPTION BASE statement, that statement must execute before any arrays are defined or used.

Frequently, a program is built to find an inserted array item based on the order in which the array was filled. If the option base is 0, the array item that was the *n*th loaded would exist as array item *n*-1. However, if the option base is 1, the *n*th item loaded into the array would exist as array item *n*.

Example 1

Line 10 in the following program fragment overrides the default value of zero, so the lowest value a subscript in an array may have in this program is 1:

```
10 OPTION BASE 1
20 DIM A(3)
30 FOR I%=1 TO 3: A(I%) = I%:NEXT
40 FOR J%=1 TO 3:PRINT "A(":J%;") = ":A(J%):NEXT
```

Output:

```
A( 1 ) = 1
A( 2 ) = 2
A( 3 ) = 3
```

Example 2

The following example is similar to Example 1, except that now the default base, 0, is the lowest array subscript:

```
10 DIM A(3)
20 FOR I%=0 TO 2: A(I%) = I% + 1:NEXT
40 FOR J%=0 TO 2:PRINT "A(";J%;") = ";A(J%):NEXT
```

Output:

```
A( 0 ) = 1
A( 1 ) = 2
A( 2 ) = 3
```

PEEK Function

Syntax

PEEK(*n*)

Action

Returns the byte (a decimal integer in the range 0 to 255) read from the indicated memory location (*n*).

Remarks

The returned value is an integer in the range 0 to 255. The argument (*n*) must be in the range 0 to 65535 (provided you have requested a full segment of memory with CLEAR).

BASIC generates an "Illegal function call" error if (*n*) is outside of BASIC's user-readable data space. This space includes BASIC's variables and stack, the user's program, variables and strings.

PEEK is the complementary function of the POKE statement.

See Also

POKE, VARPTR, CLEAR

Example

This example retrieves the byte stored in memory location H5A00:

```
A=PEEK (&H5A00)
```

POKE Statement

Syntax

POKE *address, byte*

Action

Stores *byte* in memory location *address.*

Remarks

The arguments *address* and *byte* are integer expressions. The expression *address* represents the address of a memory location; *byte* is the data byte, in the range 0-255. The *address* must be in the range 0 to 65535 (provided you have requested a full segment of memory with CLEAR).

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read.

Warning

Use POKE carefully. Altering system memory can cause unpredictable—and frequently fatal—system errors.

See Also

CLEAR, PEEK, VARPTR

Example

This example stores the hexadecimal number HFF in memory location 5A00:

```
10 POKE &H5A00,&HFF
```

POS Function

Syntax

POS(*n*)

Action

Returns the current horizontal (column) position of the pointer for the screen device SCRN:.

Remarks

The leftmost screen position is 1.

The argument *n* is a dummy argument, and has no significance.

See Also

LPOS

Example

This example causes the speaker to beep if the pointer is past column 60:

```
IF POS(X)>60 THEN PRINT CHR$(7)
```

PRINT#, PRINT# USING Statements

Syntax

PRINT# *filenumber*,[[**USING** *string-exp*;]] *expression-list*

Action

Writes data to a sequential file.

Remarks

The *filenumber* is the number used when the file was opened for output. The *string-exp* consists of formatting characters as described in "PRINT USING." The expressions in *expression-list* are the numeric or string expressions that will be written to the file.

PRINT# does not compress data. An image of the data is written to the file, exactly as it would be displayed on the terminal screen with a PRINT statement. For this reason, you should be careful to delimit the data so it will appear correctly in the file.

In the list of expressions, you should delimit numeric expressions with semicolons. For example,

```
PRINT# 1,A;B;C;X;Y;Z
```

(If you use commas as delimiters, the extra blanks that are inserted between print fields will also be written to the file.)

You must separate string expressions in the list. To format the string expressions correctly in the file, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement PRINT# 1,A$;B$ would write CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT statement as follows: PRINT# 1,A$;",";B$ The image written to the file is CAMERA,93604-1. This can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or linefeeds, then write the quotation marks to the file using CHR$(34).

If a PRINT# statement attempts to write data to a sequential file to which access has been restricted by a LOCK statement, two options are available. The first is to return control to the program immediately with an accompanying error message. All usual error handling routines can trap and examine this error. If error trapping is not active the error message is:

```
Permission denied
```

See Also

CHR$, LOCK, PRINT, PRINT USING, WRITE#

Example 1

```
A$="CAMERA, AUTOMATIC"
B$="   93604-1"
PRINT#1,A$;B$
```

Output to file:

```
CAMERA, AUTOMATIC   93604-1
```

The statement

```
INPUT#1,A$,B$
```

would then input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$.

Example 2

As you can see from Example 1, the comma embedded in A$ caused the INPUT statement to read part of A$ into B$. To separate these strings properly in the file, write double quotation marks to the file image using CHR$(34):

```
PRINT# 1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

This writes the following image to the file:

```
"CAMERA, AUTOMATIC"    "93604-1"
```

And the statement

```
INPUT# 1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

Example 3

The PRINT# statement may also be used with the USING option to control the format of the file. The following example writes the variables J, K, and L to file number 1. The variables are written with two decimal places of accuracy, a dollar sign to the immediate left, and a comma at the end (see PRINT USING for an explanation of the symbols):

```
PRINT# 1,USING "$$####.##,";J;K;L
```

PRINT Statement

Syntax

PRINT [[*expression-list*]]

Action

Outputs data on the screen.

Remarks

If *expression-list* is omitted, a blank line is printed. If *expression-list* is included, the values of the expressions are printed on the screen. The expressions in the list may be numeric or string expressions. (String constants must be enclosed in quotation marks.)

Note that when writing program lines, you may use a question mark (?) as a synonym for PRINT. When you list a line where you have used the question marks as a shorthand way of writing "PRINT," BASIC will substitute the word "PRINT." This can be a time-saving shorthand tool, especially when entering long programs with many consecutive PRINT statements.

Punctuation Marks and Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. In the list of expressions, a comma causes the next value to be printed at the beginning of the next print zone, while a semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly.

If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the line width as set by the WIDTH statement, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign.

A single precision number with 7 or fewer digits is represented in the unscaled format, as long as the unscaled format is as accurate as the scaled format. For example, 1E–7 is output as .0000001 (unscaled format), but 1.2E–7 is output as 1.2E–07 (scaled format).

A double precision number with 16 or fewer digits is represented in the unscaled format, as long as the unscaled format is as accurate as the scaled format. For example, 1D–16 is output as .0000000000000001, but 1.3D–16 is output as 1.3D–16.

See Also

PRINT#, PRINT USING, WRITE

Example 1

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone:

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
```

Output:

```
10            0             -25           3125
```

Example 2

In this example, the semicolon at the end of line 20 causes the values in lines 20 and 30 to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
```

Output

```
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729
```

```
? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261
```

Example 3

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (A number is always followed by a space, and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT. When the line is listed, however, the question mark is replaced with the word PRINT:

```
10 J = 0: K = 0
20 FOR X = 1 TO 5
30   J = J+5
40   K = K+10
50   ?J;K;
60 NEXT X
```

Output:

```
 5   10   10   20   15   30   20   40   25   50
```

PRINT USING Statement

Syntax

PRINT USING *string-exp;expression-list*

Action

Prints strings or numbers using a specified format.

Remarks

The *expression-list* contains the string expressions or numeric expressions that are to be printed, separated by semicolons.

The *string-exp* is a string literal (or variable) composed of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.

Formatting Characters: String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

!	Specifies that only the first character in the given string is to be printed.
\ *n spaces*\	Specifies that 2 + *n* characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.
&	Specifies a variable length string field. When the field is specified with the ampersand (&), the string is output without modification.

Here is an example of how the three string formatting characters (!,\ \,&) affect printed output:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
60 PRINT USING "!";A$;
70 PRINT USING "&";B$
```

Output:

```
LO
LOOKOUT
LOOK OUT  !!
LOUT
```

Formatting Characters: Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

#	A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.
.	A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.

Example 1

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20    5.30   66.79    0.23
```

In the last example above, three spaces were inserted at the end of the format string to separate the printed values on the line.

Example 2

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95    +2.40   +55.60    -0.90

PRINT USING "##.##-   ";-68.95,22.449,-7.01
```

```
68.95-    22.45      7.01-
```

`+` A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

`-` A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

Example 3

```
PRINT USING "**#.#    ";12.39,-0.9,765.1
*12.4   *-0.9   765.1

PRINT USING "$$###.##";456.78
 $456.78

PRINT USING "**$##.##";2.34
***$2.34
```

`**` A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The double asterisk also specifies positions for two more digits.

`$$` A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The `$$` specifies two more digit positions, one of which is the dollar sign.

`**$` The double asterisk dollar sign combines the effects of the double asterisk and double dollar sign symbols. Leading spaces are asterisk-filled and a dollar sign is printed before the number. `**$` specifies three more digit positions, one of which is the dollar sign.

The exponential format cannot be used with `**$`. When negative numbers are printed, the minus sign will appear immediately to the left of the dollar sign.

Example 4

```
PRINT USING "####.##,";1234.5
1234.50,

PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "##.##^^^^";234.56
 2.35E+02
```

```
PRINT USING ".####^^^^-";-888888
 .8889E+06-

PRINT USING "+.##^^^^";123
 +.12E+03

PRINT USING "_!##.##_!";12.34
!12.34!

PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

,
: A comma to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with exponential (^^^^) format.

^^^^
: Four carets (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or – is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

_
: An underscore in the format string causes the next character to be output as a literal character.

  The literal character itself may be an underscore by placing "__" in the format string.

%
: If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

  If the number of digits specified exceeds 24, an "Illegal function call" error results.

PUT Statement

Syntax

PUT [[#]]*filenumber*[[,*record-number*]]

Action

Writes a record from a random buffer to a random access file.

Remarks

The *filenumber* is the number under which the file was opened. If *record-number* is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number

is 4,294,967,295 (or $2^{32}-1$). The smallest record number is 1.

You may use PRINT#, PRINT# USING, and WRITE# to put characters in the random file buffer before executing a PUT statement. Generally, this buffer is filled with FIELD, LSET and RSET statements.

In the case of WRITE#, Microsoft BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer

causes a "Field overflow" error.

See Also

GET, PRINT#, WRITE#

Example

This example illustrates the steps needed to prepare data for a PUT statement like the one found in line 100:

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";PERSON$
   .
   .
70 LSET N$=PERSON$
   .
   .
   .
100 PUT #1,CODE%
110 GOTO 30
```

RANDOMIZE Statement

Syntax

RANDOMIZE [[*expression*]]

Action

Reseeds the random number generator.

Remarks

This statement reseeds the random number generator with *expression*, if given, where *expression* is an integer between -32768 and 32767. If *expression* is omitted, Microsoft BASIC suspends program execution and asks for a value before randomizing by printing: "Random Number Seed (-32768 to 32767)?"

To reseed the random number generator, place a RANDOMIZE statement at the beginning of the program and change the argument with each run. If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run.

Different random number seeds produce different sequences of random

numbers.

See Also

RND

Example

```
10 RANDOMIZE
20 FOR I=1 TO 3
30   PRINT RND;
40 NEXT I
```

This displays the following (user enters 3 in response to the prompt):

```
Random Number Seed (-32768 to 32767)? 3
```

Possible output (in the decimal version):

```
 .26334726810455 .88974887132645 .97605919837952
```

READ Statement

Syntax

READ *variable-list*

Action

Reads values from a DATA statement and assigns them to variables.

Remarks

You must always use a READ statement in conjunction with a DATA statement. READ statements assign DATA statement values to variables on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a

"Syntax error" is generated.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in *variable-list* exceeds the number of elements in the DATA statements, an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the DATA statements, subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

See Also

DATA, RESTORE

Example 1

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, the value of A(2) is 5.19, and so on:

```
80  FOR I=1 TO 10
90     READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
```

Example 2

The following program reads string and numeric data from the DATA statement in line 30:

```
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
```

Output:

```
CITY          STATE          ZIP
DENVER,       COLORADO       80211
```

REDIM Statement

Syntax

REDIM [[**SHARED**]] *arrayname*(*subscripts*) [[*arrayname* (*subscripts*)...]]

Action

Changes the space allocated to an array that has been declared dynamic

Remarks

REDIM takes the following arguments:

Argument	Description
SHARED	Allows you to share variables among all the subprograms in a compiland; must appear in a REDIM statement at the main program level.
arrayname	The name of the array you want to redimension
subscripts	The new dimensions of the array

The REDIM statement c nges the space allocated to an array that has been declared dynamic, e er as the result of a $DYNAMIC metacommand or as the result of a variable in a DIM statement. Static arrays cannot appear in a REDIM statement.

When a REDIM statement is compiled, all arrays declared in the statement are treated as dynamic. At run time, when a REDIM statement is executed, the array is deallocated (if it is already allocated) and then reallocated with the new dimensions. Old array element values are lost because all numeric elements are reset to 0, and all string elements are reset to null strings.

Note

While it is possible to change the *size* of an array's dimensions with REDIM , it is not possible to change the *number* of dimensions. For example, the following statements are legal:

```
' $DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(20,15)              'A still has 2 dimensions
```

However, the following statements are *not* legal, and will produce an "Array Already Dimensioned" compile-time error:

```
' $DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(5,5,5)              'Changed number of dimensions from 2 to 3
```

See Also

DIM, ERASE

Example

The following program fragment shows a subroutine that deletes a record from a random-access file; this subroutine uses REDIM to allocate a temporary string array (STORE$) to hold the records from STOCK.DAT. After the records are stored in STORE$, STOCK.DAT is closed, deleted, then reopened, and the values in STORE$ are put back into the file. After this is done, the ERASE statement deallocates STORE$.

```
GOSUB OPENFILE
TOTAL% = LOF(1)/50
MAIN:
    PRINT "1) Add a record"
    PRINT "2) Update a record"
    PRINT "3) Delete a record"
    PRINT "4) End program"
    PRINT : INPUT "What is your choice"; BRANCH
    ON BRANCH GOSUB ADDAREC, UPDATEREC, DELETEREC, ENDPROG
    GOTO MAIN
```

```
OPENFILE:     'Open STOCK.DAT and allocate random-file buffers
    OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN=50
    FIELD #1, 50 AS RECORD$     'Multiply-defined field
    FIELD #1, 10 AS PART$, 35 AS DESC$, 5 AS QTY$
RETURN
.
.
.
DELETEREC:
    INPUT "Record number to delete: ",REC%
    GET #1, REC% : PRINT DESC$
    INPUT "Is this the correct record"; CH$
    IF CH$ <> "y" THEN GOTO DELETEREC
    GET #1, TOTAL%                'Put the last record where
    PUT #1, REC%                  'the deleted record was
    TOTAL% = TOTAL% - 1
    REDIM STORE$(TOTAL%)          'Allocate temporary array
    FOR I% = 1 TO TOTAL%          'Store STOCK.DAT in array
        GET #1, I%
        STORE$(I%) = RECORD$
    NEXT
    CLOSE #1 : KILL "STOCK.DAT" 'Erase STOCK.DAT
    GOSUB OPENFILE                'Reopen STOCK.DAT
    FOR I% = 1 TO TOTAL%          'Put STORE$ array values
        LSET RECORD$ = STORE$(I%)       'in STOCK.DAT
        PUT #1, I%
    NEXT
    ERASE STORE$                  'Erase array STORE$
RETURN
```

REM Statement

Syntax

REM *remark*

Action

Allows explanatory remarks to be inserted in a program.

Remarks

REM statements are not executed but are output exactly as entered when the program is listed.

A GOTO or GOSUB statement can branch into a REM statement. Execution will continue with the first executable statement after the REM statement.

You can add remarks to the end of a line by preceding the remark with a single quotation mark instead of REM.

Warning

REM should not be used in a DATA statement, since it will be considered legal data.

Example

Each of the following program segments shows a different way to add a remark to a program:

```
120 REM **Calculate Average Velocity**
130 FOR I=1 TO 20
140   SUM=SUM + V(I)
150 NEXT I
```

or

```
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130   SUM=SUM+V(I)
140 NEXT I
```

RESTORE statement

Syntax

RESTORE [[*linenumber*]]

Action

Allows DATA statements to be reread from a specified line.

Remarks

After a RESTORE statement with no specified line number is executed, the next READ statement accesses the first item in the first DATA statement in the program. If *linenumber* is specified, the next READ statement accesses the first item in the specified DATA statement.

See Also

DATA, READ

Example

RESTORE in this example allows the two READ statements in lines 10 and 30 to read the same DATA statement:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
50 PRINT A,B,C,D,E,F
```

Output:

```
 57          68          79          57          68          79
```

RESUME Statement

Syntax

RESUME [[*linenumber*]]

RESUME 0

RESUME NEXT

Action

Continues program execution after an error recovery procedure has been performed.

Remarks

Any one of the four syntaxes shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement that caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one that caused the error.
RESUME *linenumber*	Execution resumes at *linenumber*.

A RESUME statement that is not in an error handling routine causes a "RESUME without error" error message to be printed.

Example

In this example, program execution resumes at line 80 after the error-handling routine is executed:

```
10  ON ERROR GOTO 900
    .
    .
    .
900 IF  (ERR=230) AND (ERL=90)
    THEN PRINT "TRY AGAIN":RESUME 80
```

Compiler/Interpreter Differences

Interpreted BASIC programs that contain RESUME statements may require modification before they are compiled. In the compiler, if an error occurs in a single-line function, both RESUME and RESUME NEXT attempt to resume program execution at the line containing the function.

Under the compiler, the RESUME statement accepts both line number and line label arguments.

Note

Statements containing error-handling routines should be compiled with either the -E or -X compiler options.

Considerable extra code is required to support RESUME, RESUME 0 and RESUME NEXT statements. If you can rewrite your programs so they contain RESUME *linenumber* | *linelabel* statements instead, you can compile with the -E option, making the executable file significantly smaller.

In the compiler, if an error occures in a single-line function, both RESUME and RESUME NEXT attempt to resume program execution at the line containing the function. In the interpreter, RESUME NEXT attempts to resume execution at the line *following* the one that caused the error.

RETURN Statement

Syntax

RETURN [[*linenumber*]]

Action

Returns execution control from a subroutine.

Remarks

The *linenumber* in the RETURN statement acts as it does with a GOTO; that is, the program branches unconditionally to *linenumber*. If no *linenumber* is given, execution begins with the statement immediately following the last executed GOSUB statement.

See Also

GOSUB

Example

```
 10 PRINT "The Beginning"
 40 GOSUB 100
 60 PRINT "The end!" :END
100 REM **Program execution will now return to line 60**
110 RETURN
```

RIGHT$ Function

Syntax

RIGHT$(*x*$,*n*)

Action

Returns the rightmost *n* characters of string *x*$.

Remarks

The argument *n* is a numeric expression that returns a decimal number that rounds to an integer in the range 0 to 32,767. (This argument is rounded to an integer before RIGHT$ is evaluated.)

If *n* is greater than or equal to the number of characters in LEN(*x*$), RIGHT$ returns *x*$. If *n* = 0, it returns the null string (length zero).

See Also

LEFT$, MID$

Example

```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,5)
30 PRINT RIGHT$(A$,2)
```

Output:

```
BASIC
IC
```

RMDIR Statement

Syntax

RMDIR "*pathname*"

Action

Removes an existing directory.

Remarks

The string expression *pathname* specifies the name of the directory which is to be deleted. The *pathname* must be a string of less than 128 characters.

The *pathname* to be removed cannot be the working directory (.) or the parent directory (..).

See Also

CHDIR, MKDIR

Examples

The following example deletes the directory SALES within the current directory:

```
RMDIR "SALES"
```

The following example deletes the directory "proofreaders" from the directory "../publications/editorial":

```
RMDIR "../publications/editorial/proofreaders"
```

RND Function

Syntax

RND [(*x*)]

Action

Returns a random number between 0 and 1.

Remarks

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded with RANDOMIZE.

$x < 0$ always restarts the same sequence for any given x.

$x > 0$ or x omitted generates the next random number in the sequence.

$x = 0$ repeats the last number generated.

The values produced by the RND function vary with different implementations of Microsoft BASIC.

See Also

RANDOMIZE

Example

This fragment generates three random 2-digit integers:

```
10 FOR I=1 TO 3
20    PRINT INT(RND*100),
30 NEXT
```

Possible output:

```
24          30          11
```

RSET Statement

Syntax

RSET *string-variable=string-expression*

Action

Moves data from memory to a random file buffer in preparation for a PUT statement.

Remarks

See "LSET" for a discussion of both LSET and RSET.

RUN Statement

Syntax

RUN [*linenumber*]

RUN *filename*[,R]

Action

Executes either the program currently in memory, or the program specified by *filename*.

Remarks

If *linenumber* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC always returns to command level after executing a RUN statement.

With the second form of the syntax, the named file is loaded from a device into memory and run. If there is a program in memory when the command executes, the original program is no longer in memory.

In the second syntax, the *filename* must be that used when the file was saved.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain open.

Example 1

This example executes the program currently in memory:

```
RUN
```

Example 2

This example runs the program "NEWFIL," keeping open any files that were open:

```
RUN "NEWFIL".R
```

Example 3

This example runs the program currently in memory, starting with line 950:

```
RUN 950
```

Compiler/Interpreter Differences

RUN is used to invoke executable files created by the compiler, or by other languages; unlike interpreted programs, compiled programs cannot directly execute ".BAS" source files.

The syntax is the same as in the interpreter, however the XENIX BASIC Compiler does not support the interpreted BASIC "R" option, which allows all opened data files to remain open. If you want to run a new file, yet leave all data files open, use CHAIN instead of RUN .

SADD Function

Syntax

SADD(*x*$)

Action

Returns the address of the specified string expression, *x*$.

Remarks

You should use this function with care, since strings in string space can move at any time if a string garbage collection is performed, such as when you use the FRE function.

See Also

FRE

Example

This example prints the address of the string expression A$:

```
10 A$ = "New Arrivals"
20 PRINT SADD(A$)
```

SGN Function

Syntax

SGN(x)

Action

Indicates the value of x, relative to zero.

Remarks

If $x>0$, SGN(x) returns 1.
If $x=0$, SGN(x) returns 0.
If $x<0$, SGN(x) returns –1.

Example

This fragment causes program execution to branch to line 100 if X is negative, line 200 if X is 0, and line 300 if X is positive:

```
50  ON SGN(X)+2 GOTO 100,200,300
```

SHARED Statement

Syntax

SHARED *variable* [,*variable*...]

Action

Gives a subprogram access to variables declared in the main program without having to pass them as parameters.

Remarks

The argument *variable* is either a variable name or an array name followed by ().

By using either the SHARED statement in a subprogram, or the SHARED attribute with COMMON or DIM at the main program level, you can use variables in a subprogram without passing them as parameters. The SHARED *attribute* shares variables among all subprograms in a module, while the SHARED *statement* affects variables within a single subprogram.

Note

The SHARED statement only shares variables within a single compiled module. It does not share variables with modules compiled separately, then linked.

The SHARED statement must appear only within a named subprogram; if SHARED occurs outside a subprogram, a subroutine error occurs. See "SUB...END SUB Statement."

See Also

COMMON, DIM, SUB...END SUB

Example

The following program calls a subprogram CONVERT that converts the input decimal number to its string representation in the given new base. The string N$ is shared by the subprogram and the main program.

```
DEFINT A-Z
START:
    INPUT "Decimal number (input 0 to quit): ",DECIMAL
    IF DECIMAL <= 0 THEN END
    INPUT "New base: ",NEWBASE
    N$ = ""
    PRINT DECIMAL "base 10 equals ";
    WHILE DECIMAL
        CALL CONVERT (DECIMAL,NEWBASE)
        DECIMAL = DECIMAL\NEWBASE
    WEND
    PRINT N$ " base" NEWBASE
    PRINT
    GOTO START

SUB CONVERT (D,NB) STATIC
    SHARED N$
    R = D MOD NB
    IF R < 10 THEN DIGIT$ = STR$(R) ELSE DIGIT$ = CHR$(R+55)
    N$ = RIGHT$(DIGIT$,1) + N$
END SUB
```

Sample output:

```
Decimal number (input 0 to quit): 238
New base: 15
 238 base 10 equals 10D base 15

Decimal number (input 0 to quit): 88
New base: 3
 88 base 10 equals 10021 base 3

Decimal number (input 0 to quit): 0
```

SHELL Statement and Function

Function Syntax

x = SHELL(*string-expression*)

Statement Syntax

SHELL(*string-expression*)

Action

Starts a child process.

Remarks

A process run under a SHELL command is called a "child process." With the function syntax, this child process is run in background mode and is executed by the XENIX command processor, sh, which executes the command given by *string-expression*. In contrast, the SHELL statement executes *string-expression* and returns control to BASIC only when the child process dies.

With the SHELL statement, if no argument is included, a shell is executed that waits for commands from the standard input. Only a CONTROL-D will terminate this child shell.

In the function, BASIC starts an asynchronous process. The identification number of this system process is assigned to the numeric variable x. This child process executes sh, which in turn executes the command passed in *string-expression*. BASIC resumes execution immediately without waiting for the child process to terminate.

Example 1

In this example, control passes to sh and the sort executes. Control returns to BASIC only when the sort process is complete:

```
10 A$="sort -n deliveries.file >temp.file"
20 SHELL(A$)
```

Example 2

In the following example, the SHELL function starts the execution of a system process, loaded to a string EXTERNAL$. Then it stores the process identification in PROCESS! and prints it.

```
10 PROCESS! = SHELL (EXTERNAL$)
20 PRINT "Process I.D. --> ";PROCESS!
```

SIN Function

Syntax

SIN(*x*)

Action

Returns the sine of *x, where x* is in radians.

Remarks

To change degrees to radians, multiply the number of degrees by 3.14159/180.

COS and SIN are related by this formula:

COS(*x*) = SIN(*x*+3.14159/2).

See Also

COS, TAN

Example

```
PRINT SIN(1.5)
```

Output (in the binary version of MS-BASIC):

```
 .9974951
```

SPACE$ Function

Syntax

SPACE$(*n*)

Action

Returns a string of spaces of length *n*.

Remarks

The numeric expression *n* must return a decimal number that rounds to an integer in the range 0 to 32,767. (The decimal number returned is rounded to an integer before SPACE$ is executed.)

See Also

SPC

Example

```
10 FOR I=1 TO 5
20    X$=SPACE$(I)
30    PRINT X$;I
40 NEXT I
```

Output:

```
 1
   2
     3
       4
         5
```

SPC Function

Syntax

SPC(*n*)

Action

Generates *n* spaces in a PRINT statement.

Remarks

You may use SPC only with PRINT and LPRINT statements.

The numeric expression *n* must return a decimal number that rounds to an integer in the range 0 to 32,767. (The decimal number returned is rounded to an integer before SPC is executed.)

A semicolon (;) suppressing generation of a newline is assumed to follow the SPC(*n*) function.

See Also

SPACE$, TAB

Example

```
PRINT "OVER" SPC(15) "THERE"
```

Output:

```
OVER                THERE
```

SQR Function

Syntax

SQR(*x*)

Action

Returns the square root of *x*.

Remarks

The argument *x* must be a nonnegative number.

SQR returns a double precision value in the decimal version.

Example

```
10 FOR X=5 TO 15 STEP 5
20   PRINT X, SQR(X)
30 NEXT X
```

Output (in the decimal version of BASIC):

```
 5            2.2360679774998
10            3.1622776601684
15            3.8729833462074
```

STATIC Statement

Syntax

STATIC *variable* [[,*variable*...]]

Action

Designates simple variables or arrays as local to a function definition or subprogram, and preserves their values when the subprogram or function is exited and then reentered

Remarks

The argument *variable* is either a variable name or an array name followed by an integer constant in parentheses. This integer constant represents the number of dimensions in the array, not the actual value of the dimensions.

The STATIC statement can appear only within a named subroutine or multiline function definition; if the statement occurs outside either one of these, a subroutine error (SB) occurs (see SUB...END SUB and DEF FN).

Normally, simple variables or arrays that are declared or referred to in a subprogram are considered local to that program, with initial values of zero or null string assumed. However, if the subprogram is exited and then reentered, the values contained in the variables may have changed. By declaring the variables in a STATIC statement, you guarantee that subprogram variables will retain their previous values.

Simple variables or arrays declared within a STATIC statement override any shared variables or arrays with the same name.

Usually, variables used in multiline function definitions (DEF FN) are global; however, you can use the STATIC statement inside DEF FN to declare a variable as local to that function only.

Note

Do not confuse the STATIC statement with the STATIC that is part of the SUB...END SUB syntax or with the $STATIC metacommand. The STATIC attribute in SUB...END SUB shows that the subprogram is nonrecursive, while $STATIC statically allocates memory for arrays.

See Also

DEF FN, SHARED, SUB...END SUB

Example 1

This example contrasts the STATIC and SHARED statements within a subprogram, AUGMENT. The variables R and R are both local to this subprogram, while the variables REP and NUM are shared between the main program and the subprogram. The STATIC R,N statement ensures that R and N retain the last values assigned to them each time the main program calls the subprogram.

```
REP = 0 : NUM = 0
PRINT "Before loop, rep = ";REP;", num = ";NUM;
PRINT ", r = ";R;", n = ";N
FOR I = 1 TO 10
   CALL AUGMENT
NEXT
PRINT "After loop, rep = ";REP;", num = ";NUM;
PRINT ", r = ";R;", n = ";N
END
SUB AUGMENT STATIC
   SHARED REP,NUM      'SHARED WITH MAIN PROGRAM
   STATIC R,N          'NOT SHARED WITH MAIN PROGRAM
   R = R + 1           'BOTH R & N INITIALLY EQUAL 0
   N = N + 2
   REP = R
   NUM = N
END SUB
```

The output from the above example is:

```
Before loop, rep = 0 , num = 0 , r = 0 , n = 0
After loop, rep = 10 , num = 20 , r = 0 , n = 0
```

Example 2

The following program searches for every occurrence of a certain string expression (stored in the variable OLD$) in the specified file and replaces that string with the string stored in NW$. The name of the file with these changes is the old filename with the extension ".NEW".

The program also prints the number of substitutions and the number of lines changed.

```
INPUT "Name of file";F1$
INPUT "String to replace";OLD$
INPUT "Replace with";NW$
REP = 0 : NUM = 0
M = LEN(OLD$)
OPEN F1$ FOR INPUT AS #1
CALL EXTENSION
OPEN F2$ FOR OUTPUT AS #2
WHILE NOT EOF(1)
   LINE INPUT #1, TEMP$
   CALL SEARCH
   PRINT #2, TEMP$
WEND
CLOSE
PRINT "There were ";REP;" substitutions in ";NUM;" lines."
PRINT "Substitutions are in file ";F2$
END

SUB EXTENSION STATIC
   SHARED F1$,F2$
   MARK = INSTR(F1$,".")
   IF MARK = 0 THEN
      F2$ = F1$ + ".NEW"
   ELSE
      F2$ = LEFT$(F1$,MARK - 1) + ".NEW"
   END IF
END SUB

SUB SEARCH STATIC
   SHARED TEMP$,OLD$,NW$,REP,NUM,M
   STATIC R
   MARK = INSTR(TEMP$,OLD$)
   WHILE MARK
      PART1$ = LEFT$(TEMP$,MARK - 1)
      PART2$ = MID$(TEMP$,MARK + M)
      TEMP$ = PART1$ + NW$ + PART2$
      R = R + 1
      MARK = INSTR(TEMP$,OLD$)
   WEND
   IF REP = R THEN
       EXIT SUB
```

```
    ELSE
        REP = R
        NUM = NUM + 1
    END IF
END SUB
```

The output from the above example might look like this:

```
Name of file? CHAP1.S
String to replace? CHAPTER 1
Replace with? INTRODUCTION
There were 23 substitutions in 19 lines.
Substitutions are in file CHAP1.NEW
```

The file CHAP1.NEW now contains every line in CHAP1.S, with each occurrence of the string CHAPTER 1 replaced by INTRODUCTION.

STOP Statement

Syntax

STOP

Action

Terminates program execution and returns to command level.

Remarks

You may use STOP statements anywhere in a program to terminate execution. STOP is often used for debugging.

When a STOP is encountered, the message "Break in line *nnnnn*". is printed.

The STOP statement does not close files.

BASIC always returns to command level after executing a STOP.

Example

In line 130 of this fragment, program execution stops if the value -1 has been assigned to DEBUG:

```
 20 LET DEBUG=1
    .
    .
    .
130 IF DEBUG THEN STOP
```

STR$ Function

Syntax

STR$(*x*)

Action

Returns a string representation of the value of *x*.

Remarks

The argument *x* can be any numeric expression.

If *x* is positive, the string returned by STR$ contains a leading blank (the space reserved for the plus sign).

See Also

VAL

Example

```
10 DATA -5.6,789,1.09,23
15 FOR I = 1 TO 4
20       READ A
30       LET A$ = STR$(A)
40       PRINT A$, LEN(A$)
30 NEXT
```

Output:

```
-5.6            4
 789            4
 1.09           5
 23             3
```

STRING$ Function

Syntax

STRING$(*m*,*n*)

STRING$(*m*,*x*$)

Action

The first syntax returns a string of length *m* whose characters all have ASCII code *n*.

The second syntax returns a string of length *m* whose characters are all the first character of *x*$.

Remarks

The numeric expression *m* must return a decimal number that rounds to an integer in the range 0 to 32,767. (The decimal number returned is rounded to an integer before STRING$ is executed.)

Example

```
10 X$ = STRING$(10,45)
15 Y$ = "ABCDE"
20 Z$ = STRING$(5,Y$)
20 PRINT X$;"MONTHLY REPORT";X$
25 PRINT Z$
```

Output:

```
----------MONTHLY REPORT----------
AAAAA
```

SUB...END SUB Statements

Syntax

SUB *global-name*[(*parameter-list*)] **STATIC**
.
.
.
[**EXIT SUB**]
.
.
.
END SUB

Action

Marks the beginning and end of a subprogram

Remarks

SUB takes the following arguments:

Argument	Description
global-name	A variable name up to 31 characters long. This name cannot appear in any other SUB statement in the same program or the user library. If duplicate names are present, the linker will bind to the local subroutine in the user library, and no error message will appear.
parameter-list	Contains the names of simple variables and arrays passed to the subprogram by a CALL statement in the main program; each name is separated from a preceding name in the list by a comma. Note that these variables and arrays are passed by reference, so any change to an argument's value in the subprogram also changes its value in the calling program. An array name in a SUB statement is followed by an integer constant in parentheses. This integer constant represents the number of dimensions in the array, not the actual value of the dimensions.

STATIC Shows that the subprogram is nonrecursive; that is, it does not contain an instruction that causes the subprogram to call itself, or call a second subprogram, which in turn calls the first subprogram. This version of BASIC supports only nonrecursive subprograms, and generates an error if you omit STATIC.

A subprogram is similar to a multiline function. However, unlike a multiline function, a subprogram does not return a value associated with its name, and therefore cannot appear as part of an expression.

SUB and END SUB mark, respectively, the beginning and end of a subprogram. You can use the EXIT SUB statement to exit a subprogram under abnormal conditions such as an error. Because EXIT SUB does not define the end of the subprogram, it should not be used to exit a subprogram under normal conditions.

Subprograms are called by a CALL statement. When a subprogram is exited and later reentered, the value in a particular subprogram variable may be affected by another part of your program. To guarantee that the variable retains its assigned value upon reentry to the subprogram, use the STATIC statement.

Any subprogram variables or arrays are considered local to that subprogram, unless they are explicitly declared as shared variables in a SHARED statement.

User defined functions are not permitted inside a SUB...END SUB block.

Note

You cannot use GOSUB, GOTO, or RETURN to enter or exit a subprogram

---

See Also

CALL, SHARED, STATIC

Example

In this example, the main program calls a subprogram, LINESEARCH, which searches for the given string, P$, in each line of input from file F$. When the subprogram finds P$ in a line, it prints the line, along with the number of the line.

```
INPUT "File to be searched";F$
INPUT "Pattern to search for";P$
OPEN F$ FOR INPUT AS #1
WHILE NOT EOF(1)
   LINE INPUT #1, TEST$
   CALL LINESEARCH(TEST$,P$)
WEND

SUB LINESEARCH(TEST$,P$) STATIC
   STATIC NUM
   NUM = NUM + 1
   X = INSTR(TEST$,P$)
   IF X = 0 THEN
      EXIT SUB
   ELSEIF X > 0 THEN
      PRINT "Line #";NUM;": ";TEST$
   END IF
END SUB
```

The output from the above example might look like this:

```
File to be searched? search.bas
Pattern to search for? sub
```

Output:

```
Line # 9: SUB LINESEARCH(TEST$,P$) STATIC
Line # 14:    EXIT SUB
Line # 18: END SUB
```

SWAP Statement

Syntax

SWAP *variable1,variable2*

Action

Exchanges the values of two variables.

Remarks

Any type variable (integer, single precision, double precision, string) may be swapped, but the two variables must be of the same type or a "Type mismatch" error results.

If the second variable is not already defined when SWAP is executed, an "Illegal function call" error will be generated.

This statement is quite useful and efficient in sorting routines. Instead of storing a lesser value to a temporary variable, transferring the greater value to the lesser's old variable, and then storing the temporary variable's value to the greater's old variable, one can simply SWAP the two variables' values. This turns a three-step process into a one-step process.

Example

```
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
```

Output:

```
 ONE FOR ALL
 ALL FOR ONE
```

SYSTEM Statement

Statement Syntax

SYSTEM

Action

Closes all open files, and exits BASIC.

Remarks

When a SYSTEM command is executed, all open files are closed and control returns to the operating system.

TAB Function

Syntax

TAB(*n*)

Action

Moves the print position to column *n*.

Remarks

The numeric expression *n* must return a decimal number that rounds to an integer in the range 0 to 32,767. (The decimal number returned is rounded to an integer before TAB is executed.)

If the current print position is already beyond space *n*, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one.

You may use TAB only in PRINT and LPRINT statements.

A semicolon (;) suppressing generation of a newline is assumed to follow the TAB(*n*) function.

See Also

PRINT, SPC

Example

```
10 PRINT "NAME";TAB(25);"AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$;TAB(25);B$
40 DATA "G. T. JONES","$25.00"
```

Output:

```
NAME                      AMOUNT

G. T. JONES               $25.00
```

TAN Function

Syntax

TAN(x)

Action

Returns the tangent of x.

Remarks

The argument x should be given in radians, where 1 radian = 180 degrees. To convert degrees to radians, multiply degrees by 3.14159/180.

If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

See Also

COS, SIN

Example

```
10 PRINT TAN(9)
```

Output (in the decimal version of BASIC):

```
- .45231565944189
```

TIME$ Function

Function Syntax

TIME$

Action

Retrieves the current time.

Remarks

The TIME$ function returns an eight-character string in the form where *hh* is the hour (00 through 23), *mm* is minutes (00 through 59), and *ss* is seconds (00 through 59).

Example

This example puts the value of the clock in a variable, then prints that variable:

```
10 CHRONOS$ = TIME$
20 PRINT CHRONOS$
```

Possible output:

```
00:23:46
```

TRON, TROFF Statement

Syntax

TRON

TROFF

Action

Traces the execution of program statements.

Remarks

As an aid in debugging, the TRON statement (executed in either direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Examples

These two examples achieve similar results: the first one enables TRON with a statement in direct mode; the second includes TRON and TROFF in the body of the program.

```
10 K=10
20 FOR J=1 TO 2
30     L= K + 10
40     PRINT J;K;L
50     K= K + 10
60 NEXT J
70 END
```

In direct mode, enter

```
TRON
RUN
```

Output (the three lines of numbers enclosed in brackets show the order of program execution - twice through the loop - and the two lines to the right are the output from the PRINT statement in line 40):

```
[10] [20] [30] [40]  1   10   20
[50] [60] [30] [40]  2   20   30
[50] [60] [70]
```

TRON and TROFF statements inside a program give the same result:

```
5  TRON
10 K=10
20 FOR J=1 TO 2
30     L= K + 10
40     PRINT J;K;L
50     K= K + 10
60 NEXT J
65 TROFF
70 END
```

Output:

```
[10] [20] [30] [40]  1   10   20
[50] [60] [30] [40]  2   20   30
[50] [60] [65]
```

UBOUND Function

Syntax

UBOUND(*array*[,*dimension*])

Action

Returns the upper bound (largest available subscript) for the indicated dimension of an array.

Remarks

The argument *dimension* is an integer from 1 to the number of dimensions in *array*.

In the array "ACCOUNT(A, B, C, D)," A is dimension 1, B is dimension 2, C is dimension 3, and D is dimension 4. So the function

```
UBOUND(ACCOUNT, 1)
```

finds the largest subscript in dimension A, the function

```
UBOUND(ACCOUNT, 2)
```

finds the largest subscript in dimension B, and so on.

You can use the shortened syntax **UBOUND(*array*)** for one-dimensional arrays, since the default value for *dimension* is 1.

Use the LBOUND function to find the lower limit of an array dimension.

See Also

LBOUND

Example

LBOUND and UBOUND can be used together to determine the size of an array passed to a subprogram, as in the following program fragment:

```
CALL PRNTMAT(ARRAY())
.
.
.
SUB PRNTMAT(A(2)) STATIC
    FOR I% = LBOUND(A,1) TO UBOUND(A,1)
        FOR J% = LBOUND(A,2) TO UBOUND(A,2)
           PRINT A(I%,J%);" ";
        NEXT J%
    PRINT:PRINT
    NEXT I%
END SUB
```

UNLOCK Statement

Syntax

UNLOCK [[#]]*filenumber* [[*recnum1*]] [[**TO** *recnum2*]]

Action

Releases access restrictions to specified portions of a file.

Remarks

The *filenumber* is the number with which the file was opened.

The expressions *recnum1* and *recnum2* are record numbers in the file. The largest possible record number is 4,294,967,295 (or $2^{32}-1$). The smallest record number is 1.

UNLOCK releases locks applied to the specified file. If *recnum1* or a range of record numbers are specified and the file is concurrently open in random mode, only the records in the range are unlocked.

Note

The UNLOCK statement is complementary to the LOCK statement.

See Also

LOCK

Example

This unlocks records 1 through 32 in file number 4:

```
650 UNLOCK #4, 1 TO 32
```

VAL Function

Syntax

VAL(*x*$)

Action

Returns the numerical value of string *x*$.

The VAL function strips leading blanks, tabs, and linefeeds from the argument string. For example, VAL(" -3") returns the numeric value –3.

Do not use VAL to convert random file strings into numbers. For that purpose, use the CVI, CVS and CVD functions.

See Also

STR$, CVI, CVS, CVD

Example

This example shows string zip code values being turned into numbers so they can be tested for location:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699 _
   THEN PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$) >= 90801 AND VAL(ZIP$) <= 90815 _
   THEN PRINT NAME$ TAB(25) "LONG BEACH"
```

VARPTR Function

Syntax

VARPTR(*variable-name*)

VARPTR(*filenumber*)

Action

Returns the address of the first byte of data identified by *variable-name* or the start of a file data buffer.

Syntax 1 Remarks

The program must assign a value to *variable-name* before it executes VARPTR; otherwise, BASIC generates an *Illegal function call* error message.

You can use any type of variable (numeric, string, or array) within a VARPTR function. For string variables, the address of the first byte of the string descriptor is returned. (See Section 5.2.3, "Assembly Language Coding Rules," for more information about the string descriptor.) The address returned is a number in the range 0 to 4,294,967,295.

This form of VARPTR is most often used to obtain the address of a variable or array so that it can be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually used when passing an array, so that the element with the lowest address in the array is returned.

Note

All simple variables should be assigned before calling VARPTR for an array element because the addresses of the arrays change whenever a new simple variable is assigned.

Syntax 2 Remarks

This second form of VARPTR is generally used to obtain the address of a file data buffer so that it may be passed to an assembly language subroutine.

This function should immediately precede the place in a program where its resulting value is to be used. This is because closing another file may move the memory location of the file data buffer associated with *filenumber*.

See Also

PEEK, POKE

Example

This obtains the starting address of the data buffer for the file opened as number 1, then assigns that address to the variable X:

```
100 X = VARPTR(1)
```

WHILE...WEND Statement

Syntax

WHILE *expression*

[[*statements*]]

WEND

Action

Executes a series of statements in a loop as long as a given condition is true.

Remarks

If the *expression* is true (or it evaluates to a non-zero number), then *statements* are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks *expression*. If it is still true, the process is repeated. If it is not true (or if it is equal to zero), execution resumes with the statement following the WEND statement.

WHILE...WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error message to be generated, and an unmatched WEND statement causes a "WEND without WHILE" error message to be generated.

Warning

Do not direct program flow into a WHILE...WEND loop without entering through the WHILE statement, as this will confuse BASIC's structuring of control flow.

Example

The following fragment performs a bubble sort on the array A$. Line 100 makes FLIPS true by assigning it a non-zero value; this forces one pass through the WHILE...WEND loop. When there are finally no more swaps, then all the elements of A$ will be sorted, FLIPS will be false (that is, equal to zero), and the program will continue execution with the line following 150:

```
90 REM **BUBBLE SORT ARRAY A$ WHICH HAS J ELEMENTS**
100 FLIPS=1 'FORCE ONE PASS THROUGH LOOP
110 WHILE FLIPS
115  FLIPS=0
120    FOR I=1 TO J-1
130      IF A$(I)>A$(I+1) THEN SWAP A$(I),A$(I+1):FLIPS=1
140    NEXT I
150 WEND
```

WIDTH Statement

Syntax

WIDTH [[*output-device,*]] [[*n*]]

WIDTH # *filenumber*[[,*n*]]

Action

Sets the printed line width in number of characters for any output device.

Remarks

The *output-device* may be SCRN:, COM1: or LPT1:, and if not specified is assumed to be SCRN:.

The integer *n* is the length of the output line for the given device. However, the position of the pointer or the print head, as given by the POS or LPOS function, returns to zero after position 255. The default line width for the screen is 255 (infinite).

A WIDTH statement for a device that is already open will have no effect on the line width for that device until that device is closed and subsequently reopened.

If *n* is 255, the line width is "infinite"; that is, BASIC *never* inserts a newline.

Valid width for all devices is 1 to 255 characters, inclusive. Any value outside these ranges results in an "Illegal function call" message, and the previous value is retained.

The *filenumber* is a numeric expression assigned to a file. This is the number of the file that is to have a new width assignment.

If the device is specified as SCRN:, the line width is set at the screen.

If the output device is specified LPT1:, the line width is set for the line printer.

When files are first opened, they take the device width as their default width. The width of opened files may be altered by using the second WIDTH syntax shown above.

See Also

LPOS, LPRINT, POS, PRINT, TAB

Example 1

In this example, the width of the line printer is set to 120 columns:

```
100 WIDTH "LPT1:", 120
```

Example 2

In this example, the width of the standard output device is set to 60:

```
100 WIDTH, 60
```

Example 3

In this example, the record width in file #1 is set to 40 columns:

```
100 WIDTH #1, 40
```

WRITE# Statement

Syntax

WRITE# *filenumber,expression-list*

Action

Writes data to a sequential file.

Remarks

The *filenumber* is the number under which the file was opened for OUTPUT or APPEND in the OPEN statement. The expressions in the *expression-list* are string or numeric expressions. They must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary for you to put explicit delimiters in the list. A newline is inserted, once the last item in the list has been written to the file.

If a WRITE# statement attempts to write data to a sequential file to which access has been restricted by a LOCK statement, two options are available. The first is to return control to the program immediately with an accompanying error message. All of BASIC's usual error handling routines can trap and examine this error. If error trapping is not active the error message is:

```
Permission denied
```

See Also

OPEN, LOCK, PRINT#, WRITE

Examples

These two short programs, and their output, illustrate the difference between WRITE# and PRINT# statements:

```
10 A$="TELEVISION, COLOR":B$="$599.00"
20 OPEN "O",1,"PRICES"
```

```
30 PRINT #1,A$,B$
40 CLOSE #1
50 OPEN "I",1,"PRICES"
60 INPUT #1,A$,B$
70 PRINT A$;TAB(25);B$
```

Output:

```
TELEVISION                COLOR           $599.00
```

Substituting the following line

```
30 WRITE #1,A$,B$
```

in the previous program gives this output:

```
TELEVISION,COLOR          $599.00
```

WRITE Statement

Syntax

WRITE [*expression-list*]

Action

Outputs data to the screen.

Remarks

If *expression-list* is omitted, a blank line is output. If *expression-list* is included, the values of the expressions are output to the screen. The expressions in the list may be numeric or string expressions, and must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, BASIC inserts a newline.

WRITE outputs numeric values using the same format as the PRINT statement.

Example

This example shows the difference between PRINT and WRITE statements:

```
10 A = 80 :B = 90 :C$ = "THAT'S ALL"
20 WRITE A,B,C$
30 PRINT A,B,C$
```

Output:

```
80, 90,"THAT'S ALL"
 80        90        THAT'S ALL
```

Part 3
Appendixes

A ASCII Character Codes 347

B Microsoft BASIC Reserved Words 349

C Summary of Commands 351

D ISAM Reference 357

E Rebuild 2.0 385

F Error Messages 407

Appendix A
ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	031	1FH	US
001	01H	SOH	032	20H	SPACE
002	02H	STX	033	21H	!
003	03H	ETX	034	22H	"
004	04H	EOT	034	22H	"
005	05H	ENQ	035	23H	#
006	06H	ACK	036	24H	
007	07H	BEL	037	25H	%
008	08H	BS	038	26H	&
009	09H	HT	039	27H	,
010	0AH	LF	040	28H	(
011	0BH	VT	041	29H	)
012	0CH	FF	042	2AH	*
013	0DH	CR	043	2BH	+
014	0EH	SO	044	2CH	,
015	0FH	SI	045	2DH	-
016	10H	DLE	046	2EH	.
017	11H	DC1	047	2FH	/
018	12H	DC2	048	30H	0
019	13H	DC3	049	31H	1
020	14H	DC4	050	32H	2
021	15H	NAK	051	33H	3
022	16H	SYN	052	34H	4
023	17H	ETB	053	35H	5
024	18H	CAN	054	36H	6
025	19H	EM	055	37H	7
026	1AH	SUB	056	38H	8
027	1BH	ESCAPE	057	39H	9
028	1CH	FS	058	3AH	:
029	1DH	GS	059	3BH	;
030	1EH	RS	060	3CH	<

Dec	Hex	CHR	Dec	Hex	CHR
061	3DH	=	094	5EH	^
062	3EH	>	095	5FH	_
063	3FH	?	096	60H	`
064	40H	@	097	61H	a
065	41H	A	098	62H	b
066	42H	B	099	63H	c
067	43H	C	100	64H	d
068	44H	D	101	65H	e
069	45H	E	102	66H	f
070	46H	F	103	67H	g
071	47H	G	104	68H	h
072	48H	H	105	69H	i
073	49H	I	106	6AH	j
074	4AH	J	107	6BH	k
075	4BH	K	108	6CH	l
076	4CH	L	109	6DH	m
077	4DH	M	110	6EH	n
078	4EH	N	111	6FH	o
079	4FH	O	112	70H	p
080	50H	P	113	71H	q
081	51H	Q	114	72H	r
082	52H	R	115	73H	s
083	53H	S	116	74H	t
084	54H	T	117	75H	u
085	55H	U	118	76H	v
086	56H	V	119	77H	w
087	57H	W	120	78H	x
088	58H	X	121	79H	y
089	59H	Y	122	7AH	z
090	5AH	Z	123	7BH	{
091	5BH	[	124	7CH	|
092	5CH	\	125	7DH	}
093	5DH	]	126	7EH	~
			128	7FH	DEL

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

Appendix B
Microsoft BASIC Reserved Words

The following reserved words are used in Microsoft BASIC. If you attempt to use these words as variable names, a syntax error is generated.

ABS
AND
APPEND
AS
ASC
ATN
BASE
CALL
CALLS
CDBL
CHAIN
CHDIR
CHR$
CINT
CLEAR
CLOSE
CLS
COMMON
COS
CSNG
CVD
CVI
CVS
DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INPUT
INPUT#
INPUT$
INSTR
INT
KILL
LEFT$
LEN
LET
LINE
LOC
LOCATE
LOCK
LOF
LOG
LPOS
LPRINT
LSET
MID$
MKD$
MKDIR
MKI$
MKS$
MOD
NAME
NEXT
NOT
OCT$
ON
OPEN
OPTION
OR
OUTPUT
PEEK
POKE
POS
PRINT
PRINT#
PUT
RANDOMIZE
READ
REM
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RUN
SADD
SGN
SHELL
SIN
SPACE
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TIME$
TIMER
TO

TROFF	USR	WEND	WRITE#
TRON	VAL	WHILE	XOR
UNLOCK	VARPTR	WIDTH	
USING	WAIT	WRITE	

Appendix C
Summary of Commands

C.1 Compiler Options 353
C.2 Linker (ld) Options 354
C.3 XENIX BASIC Metacommands 355

C.1 Compiler Options

The **bascom** command compiles and links BASIC source and assembly language source files to create an executable program. The **bascom** command has the following form:

bascom [[*options*]] *files*

Table C.1 summarizes the options available on the **bascom** command line. The options are described in detail in Chapter 3, "Compiling a BASIC Program."

Table C.1
XENIX BASIC Compiler Options

Option	Description
-A	Includes a listing of the disassembled object code in the source listing.
-c	Suppresses linking.
-D	Generates debugging code for run-time error checking and enables the DELETE key.
-E	Indicates the presence of **ON ERROR GOTO** with **RESUME** *linenumber* statement.
-i	Links with the ISAM library as well as the BASIC library.
-L	Generates a source listing file.
-m	Generates a linker map listing file.
-o	Allows you to specify the name of the executable file.
-R	Stores arrays in row order.
-s	Causes the linker to strip local symbols out of the symbol table when linking user object files with object files generated by the compiler.
-S	Writes quoted strings to .OBJ file instead of symbol table.
-X	Indicates presence of **ON ERROR GOTO** with **RESUME**, **RESUME NEXT**, or **RESUME 0**.

C.2 Linker (ld) Options

The XENIX linker can be invoked separately with the **ld** command. The command has the following form:

ld -P [**-Ml**[**e**]] [**-F** *num*] [**-C**] [**-S**] [**-s**] [**-o***outputfile*] [**-m***mapfile*] *objectfiles* [**-l** *library*] [**-l** *library* ...]

Table C.2 summarizes the **ld** options. The options are described in detail in Chapter 4, "Linking Object Modules."

Table C.2
XENIX Linker Options

Option	Description
-C	Causes the linker to ignore the case of symbols.
-F *num*	Sets the size of the stack to *num* bytes.
-l *library*	Specifies other libraries to be linked.
-m *mapfile*	Produces a linker map file named *mapfile*.
-Ml[**e**]	The **-Ml** portion of the option creates a large model program. The **e** portion of the option is required when using ISAM libraries; it permits mixed model linking. You must use **-Mle** with ISAM libraries.
-o *outputfile*	Assigns the name *outputfile* to the executable file.
-P	Keeps segments defined in an assembly language or compiled BASIC program separate. This argument is required or the program will not run.
-S	Sets the maximum number of data segments.
-s	Strips local symbols from the symbol table.

C.3 XENIX BASIC Metacommands

Metacommands tell the compiler to perform certain actions while it is compiling the source file. Table C.3 summarizes the XENIX BASIC compiler metacommands. The metacommands are described in detail in Section 9.2, "Using Metacommands"

Table C.3
XENIX BASIC Metacommands

Name	Function
$DYNAMIC	Causes dynamic allocation of arrays
$INCLUDE:'*file***'**	Switches compilation from the current source file to *file*.
$LINESIZE:*size*	Sets the width of the source code listing, in columns.
$LIST[+|-]	Turns on or off source listing. Errors are always listed.
$MODULE:'*name***'**	Changes an internal module name passed to the linker.
$OCODE[+|-]	Turns on or off listing of disassembled object code.
$PAGE	Skips to next page.
$PAGEIF:*number*	Skips to next page if *number* lines or less left on the listing page.
$PAGESIZE:*number*	Sets length of listing, in lines.
$SKIP[:*number*]	Skips*number* lines or to end of page.
$STATIC	Causes static allocation of arrays.
$TITLE'*title*	Sets the source listing page title.

Appendix D
ISAM Reference

D.1 Introduction 359
D.2 Writing an MS-ISAM Application 360
D.3 Parameters 363
D.4 MS-ISAM Subroutines 368
D.5 MS-ISAM Codes 381

D.1 Introduction

Microsoft MS-ISAM is a library of subroutines that allows access to files, both sequentially and by index. You can use MS-ISAM whenever you want to access data based on the contents of records. If, for example, you want to read the record that contains information about product number 34056-J, delete any records with information on employee T. R. James, or update the record containing information on the price and availability of marble, MS-ISAM provides a fast way to do so.

Each MS-ISAM file is physically two files: a data file and a key file. A key is a data field that has been identified and described to MS-ISAM. Both files must be present to use MS-ISAM.

The maximum length of a file name (without extension) is 10 characters; the maximum length with the extension is 14 characters. Conventionally, data file names end with a ".dat" extension. Key file names, which are created by MS-ISAM, always end with a ".key" extension.

The data file consists of data records and, usually, a data dictionary. The data dictionary, which resides at the beginning of the data file, contains binary descriptions of records in the file. Whenever you create an MS-ISAM file, you will give MS-ISAM information about how your data is formatted, such as where data fields start and end, what type of data is contained in a field, and whether it is acceptable to have the same value in a given field of more than one record. This information is stored in the data dictionary.

The key file contains the indexing information that MS-ISAM uses to access the data in the data file. MS-ISAM gets this information (where the fields are, what type of data they contain) from the data dictionary in the data file. The indexing is in the form of B-trees. B-trees are a special kind of index that point to the records in the data file. There is one tree in the key file for each key that you specify.

Fields containing employee numbers or names, product codes, or zip codes are examples of fields that you might want to use as key fields.

There is a special type of key, called a split key, that contains more than one field. Components of a split key can be adjacent or non-adjacent fields, of the same or different data types, and may or may not be keys themselves. Split keys are explained further under "Split Keys" in Section D.3, "Parameters."

There are two types of MS-ISAM data records: non-segmented and segmented. Non-segmented records contain key fields that have fixed sizes. They may, however, contain one field which is variable in length, as long as that field is the last field in the record.

The other type of record is called a segmented record. It supports key fields that can vary in size, and is usually used to contain strings. You can also use variable length strings in non-segmented records by making sure that your fixed-length field is long enough to hold the longest string that you will use. It is strongly recommended that you use only segmented records if it is very important to minimize the amount of storage space used for variable-length fields. Segmented records are described in Section D.3, "Parameters."

Record types cannot be mixed in one file.

D.2 Writing an MS-ISAM Application

The MS-ISAM interface has been designed to make access to MS-ISAM files as simple as possible. In general, MS-ISAM file access is similar to random I/O procedures. Specific MS-ISAM subroutines are called to open and close MS-ISAM files, to find records within a file, and to read, write, delete, or rewrite data.

There are four basic steps in MS-ISAM applications:

1. open an MS-ISAM file
2. seek to (search for) some location in the file
3. operate upon the data
4. close the MS-ISAM file

The next four sections briefly describe each step. For detailed information on each MS-ISAM subroutine, refer to the alphabetical list of subroutines in Section D.5, "MS-ISAM Subroutines."

Opening a File

Before reading or writing to an existing MS-ISAM file, or creating a new MS-ISAM file, you must open the file. Use the MS-ISAM subroutine IOPEN to open files.

Seeking a Record

For any specified key, the MS-ISAM subroutine ISEEK can find the first record, the last record, the first record with a key value equal to a specified value, or the first record with a key value greater than a specified value.

The current record is generally the record pointed to by the last seek. (Certain other subroutines, such as IWRITE, INEXT, and IPREV, can also change which record is the current one.)

Two other subroutines that are useful for finding a location in a file are ISAVEFP and IRESTOREFP. With these routines, you can mark and return to any specified file position.

Operating on a Record

MS-ISAM can read or write a record, rewrite or delete a record, locate a related record, or find the size of a record. The following list describes these operations briefly. The predefined ISAM status variable, IXSTAT, must be used after each operation to see if the operation was successful. For complete information on all MS-ISAM subroutines, refer to Section D.5, "MS-ISAM Subroutines."

- Read a record

  Once the current record has been established, often by ISEEK, that data record can be fetched by IREAD.

- Write a record

  To insert a record into the data file, use IWRITE. The newly inserted record becomes the current record.

- Rewrite a record

  The IREWRITE procedure differs from IWRITE in that it deletes the old record from the data file and inserts the modified record. The keys are updated to reflect this change. The modified record becomes the current record.

- Delete a record

  Removing a record from the data file is done by establishing the current record and using IDELETE. The next record then becomes the current record.

- Move to the next or previous record

  The data file can be examined sequentially by using INEXT and IPREV. The records are ordered (alphabetically or numerically) by the value in the key field that was used in the last seek operation. The record that you move to becomes the current record.

- Find the size of a record

  ISIZEOF finds the size of the current record, in bytes. This can be useful when using variable-length records; to set the size of a buffer, for example, before reading a variable-length record into it.

Closing a File

It is important to use ICLOSE to close every MS-ISAM file that you open. If an MS-ISAM file is not closed by ICLOSE, its key file may be corrupted. In this case, you must use the Rebuild Utility to insure that the key file is valid. Rebuild is an MS-ISAM application program that is included in your MS-ISAM package. Its primary use is in building key files. For complete information on rebuild, refer to Appendix E.

Warning

Do not rely on BASIC's CLOSE statement to close ISAM files. This will not work. ISAM files must be explicitly closed with ICLOSE.

Other MS-ISAM Subroutines

The other five MS-ISAM subroutines are

ICONTROL	Writes all data held in memory to the data file.
IDREAD	Reads a record. The difference between IDREAD and IREAD is that you give IDREAD a pointer to the record you want to read. IREAD reads the current record. IDREAD can also use the tag file generated by SORT to read a data file directly.
IGETKD	Gets the key description.
IGETDP	Gets a pointer to a record.
ILOCK	Controls record locking.

D.3 Parameters

Some parameters, such as file handles and key handles, have unique meaning to MS-ISAM. You will also use parameters that are specific to MS-ISAM when describing fields and records. The rest of this appendix explains these parameters. More information on each parameter can be found in Section D.4, "MS-ISAM Subroutines," where each MS-ISAM subroutine and its arguments are listed and explained. This section also explains split keys and segmented records, and some of the parameters you will have to supply if you are using them.

File Handles

The file handle is a number used by MS-ISAM to refer to a specific MS-ISAM file. Although an MS-ISAM file is physically two files, there is only one file handle for each data file/key file pair. MS-ISAM will return the file handle each time you open a file.

Key Handles

The key handle is a number used by MS-ISAM to refer to a specific key in an MS-ISAM file. The key handle is assigned by the programmer in the field description when the MS-ISAM file is created.

Key handle values range from 1 to *n*, where *n* is the number of keys. There needn't be a physical relationship between the key handle values and the record layout, but it is a good idea to assign key values from the lowest to highest part of the record.

Split Keys

A split key is a key that is made up of more than one field. The component fields of a split key may or may not be adjacent, may be the same or different data types, and may be non-key fields, keys, or split keys. All the components of one split key must have the same key handle.

If a component field of a split key is also a key, that field's description must be given twice: once to describe it as a key field, and once to group it with the other components of the split key. This type of field will also have more than one key handle: one handle of its own, and one handle that is the same as the other components of the split key.

When key values are compared (to determine the order of records, or to determine if values are equal) the split key components are compared according to the order in which they were declared in the key description.

Split keys cannot be used with segmented records.

Segmented Records

MS-ISAM data files contain either segmented or non-segmented records. These record types cannot be mixed in one file.

The address of a key field is given by a segment number and an offset. For non-segmented records, the segment number is 1. In segmented records, the segment number acts as an index to a segment table, which must be inserted in front of each record. The segment table is an array of 16-bit offsets; this offset is the number of bytes from the start of the record to the start of the segment. For a given key *n*, the address of the key is the address in the *n*th entry in the segment table, plus any offset within the segment itself. A field length of zero indicates that the field length equals the length of the entire segment.

The segment table and offsets within segments must be supplied in the record and field descriptions when the file is created. The segment table must be maintained by the application programmer. It is recommended that segmented records be used only if variable-length key fields are needed. Often, all fixed-size record fields are placed in the first segment, and each

string field is placed in its own segment.

If a data file contains segmented records, it is not necessary for each record to contain the same number of segments. All segments that contain keys, however, must be present in each record. If a segment that contains a key is missing from a record, IXSTAT will return the status code *ixstat%* =10 (key not found.)

Segmented records cannot contain split keys.

Record Description

The record description tells how many keys are in the record, and if the record is segmented or non-segmented. This information is given to MS-ISAM as an array, Rdes%.

Rdes%(1) = *# keys*

The number of keys in the record.

Rdes%(2) = *segment-flag*

If the record is non-segmented, *segment-flag* =0. If the record is segmented, *segment-flag* =1.

Rdes%(3) = *minimum-allocation*

The minimum record allocation defaults to 8 bytes: 5 bytes of data and 3 bytes of overhead. The 8-byte minimum was chosen because of the way MS-ISAM handles one problem of variable length records: what happens when you rewrite a large record over a small record?

When a record is rewritten over a record that is too small to contain the new record, MS-ISAM makes the old record into an "indirection record." The indirection record points to the location of the new, larger record. By using indirection records, MS-ISAM avoids having to change every key that pointed to the old record location. To make sure that every record is large enough to hold an indirection record, MS-ISAM sets the minimum record allocation to 8 bytes.

If you know your records will be larger than 8 bytes, it is recommended that you raise the minimum record allocation. If, for example, your records are all 100 bytes long, you will generate an indirection record nearly every time you rewrite a record. Extensive rewriting of variable-length records can product a large number of indirection records. You can use rebuild, as described in Appendix E, to compress a data file, removing the indirection

records.

Field Description

Whenever you create an MS-ISAM file, you must describe each field that you will use as a key field; this information is used to build key files. You can also describe non-key fields. MS-ISAM will put this information in the data dictionary, at the beginning of the data file. Whenever a file is opened, its data dictionary is loaded into memory from the data file.

If you are using files created by SORT, you should be aware that some of these files do not have a data dictionary. When using these files, you must specify the field description each time you open the file.

It is recommended that you describe each field in the record when you create an MS-ISAM file. This provides an easy way to identify each file and its contents. Complete field descriptions can also be used by other utilities to access field information.

Field descriptions are given to MS-ISAM as a nine-integer array, Kdes. The required parameters for each field that you describe are as follows:

Kdes(1) = *varptr(field-name$)*

A pointer to a buffer that contains the name of the field. The field name must be less than or equal to 40 characters. If no field name is supplied, this pointer must be null. Field names can be used by utilities, such as general file dump utilities, to access fields in a data file.

Kdes(3) [high byte] = *subtype*

A subclassification of the data type. It is present for future MS-ISAM expansion, and must be initialized to zero.

Kdes(3) [low byte] = *data-type*

The data type of the field. Section D.5, "MS-ISAM Codes," contains a complete list of supported data types. For certain data types, like integer, the length is implied, so any specified length is ignored.

Kdes(4) = *segment-number*

For segmented records, the number of the segment containing the field. For non-segmented records, this number must be 1.

Segments are numbered from 1 to *n*, where *n* is the number of segments. Segment 1 is the first segment in the record and segment *n* is the last.

Each segment can contain many fields but no field can span more then one segment.

Kdes(5) = *field-position*

This is the position from the beginning of the segment to the beginning of the key field. The first byte in the segment is numbered 1.

Kdes(4) and Kdes(5) make the field address. The *segment number* tells what segment the field is in, and *field-position* tells the distance from the beginning of the segment to the beginning of the field.

Kdes(6) = *key-length*

The length of the field in bytes. A zero length field indicates that the field size is from the field segment position to the end of the segment. If the field length is variable then this number should always be zero.

Kdes(7) = *key-handle*

Any value between 1 and *n*, where *n* is the number of keys. The convention is to assign key handles beginning with 1 and starting with the leftmost byte in the record. Using this convention makes it easier to remember key handles. Key handles can be determined at run time by using the IGETKD procedure to fetch key field descriptions.

Kdes(8) [high byte] = *duplicates-allowed flag, descending flag, and case-insensitive flag*

The duplicates-allowed flag = 1, the descending flag = 2, and the case-insensitive flag = 4.

Add the values of the desired flags together, and enter that number as the high byte. For example, to switch the duplicates-allowed and case-insensitive flags, use

Kdes(8) = (256 * (1 + 4)) + *field-mode*

duplicates-allowed flag

Indicates whether duplicate keys are allowed in the data file.

descending-flag

Inverts the meaning of comparisons performed on this field. The result is that the records are inserted into the key set in descending instead of ascending order. It is most useful with split keys where the ordering of the different components might need to be inverted.

case-insensitive-flag

Causes string-based data types to ignore differences in case (e.g., the values 'FiRst' and 'first' would be equal).

Kdes(8) [low byte] = *field-mode*

Tells if the field is a key. If it is a key, it tells if it is a split key.

If the field is a non-key field, *field-mode* = 0. If the field is a non-split field, *field-mode* = 1. If the field is a component of a split key, *field-mode* = 2.

Kdes(9) = *filler*

A reserved word area. It must be initialized to zero.

D.4 MS-ISAM Subroutines

MS-ISAM is made up of 17 subroutines. Your BASIC programs can call these routines with the CALL statement. The following section will describe the syntax of the subroutine and describe each parameter. If the subroutine returns a value (e.g., IOPEN returns a file handle) then the syntax section will be followed by the word **Returns:** and the value that is returned. Where pointer values are indicated in the text, they should be obtained by using the VARPTR function to return a pointer to a variable of the the proper size and data type.

The interface to assembly language is also provided in this appendix, including an example of record and field descriptions, and a list of codes used in MS-ISAM.

ICLOSE

ICLOSE closes the key and data files, and empties and frees any buffers for the named file.

While the key file is open, portions of the B tree are kept in core. If the MS-IS M subroutine ICLOSE is not used to close the file, these sections of the B tree may not be written to the key file. When this happens, the file is marked damaged or "corrupted" (IXSTAT will return *ixstat%* =5), and IOPEN will not open the file. If this occurs, the Rebuild Utility must be run to regenerate a good key file.

ICLOSE (*fileno%*)

fileno%	The file handle (integer value returned by IOPEN) identifying the file to be closed.

ICONTROL

ICONTROL requests the check point function. This function writes all data held in memory to the data file. It then closes and reopens the file used to hold the data. These two operations guarantee that the data file is updated. The key file is still held in memory and needs to be written to disk. Should you not do an ICLOSE, rebuild can completely restore the key file.

ICONTROL (*request%*, *checkpoint*)

request%	This designates the ICONTROL check point function. Its value is two.
checkpoint	Pointer to an array: the first element of the array holds the file handle (integer returned by IOPEN) identifying the MS-IS M file that you want to check point; the second element of the array holds a pointer to the string used to name the file in IOPEN; the third element is a reserved space.

Example:

```
DIM CHECKPOINT(3)
.
.
.
CHECKPOINT(1)=FILENO%   'FILENO% is file handle
CHECKPOINT(2)=VARPTR(FILENAME$) 'FILENAME$ is name used by IOPEN
CALL ICONTROL(2,VARPTR(CHECKPOINT(1)))
```

IDELETE

IDELETE deletes the current record and all key values associated with that record.

IDELETE (*fileno%*)

fileno% — The file handle (integer returned by IOPEN) identifying the file from which the record will be deleted.

IDREAD

This data read routine is a special routine that will read directly from the file. Given a record pointer, this routine will determine if the pointer points directly to a data record or to an indirection record. In either case, it will return the data record. Use IGETDP to get the data pointer. Other

methods are not reliable if the file has been overwritten.

IDREAD (*fileno%, pbuffer, length%, plocation*)

fileno%	The file handle (integer returned by IOPEN) identifying the file to use.
pbuffer	The pointer to the buffer to receive the record.
length%	This is the number of bytes that the record buffer *(pbuffer)* can hold. For fixed length records this is the record size. For variable length records it is the maximum size. IXSTAT will return an error (*ixstat%*=18) if the record was greater than *length%*. ISIZEOF can be used to determine the size of any record. IDREAD uses *length%* to return the actual length of the record.
plocation	A pointer to a 4-byte buffer that holds the pointer to the data. This number is set in IGETDP.

Returns:

length%	The number of bytes read into the record buffer.

IGETDP

This routine fetches a pointer to a record. (The data pointer may not point directly at the record.) You can accumulate several data pointers and do the data reads later. It is a good idea to use IGETDP with IDREAD. IGETDP sometimes returns a pointer to an indirection record, but IDREAD compensates for this, and will read the associated data record. All data pointers fetched by IGETDP will be correct until the record is deleted or the file is reorganized by rebuild. If you are using the SORT utility, remember that data pointers returned by SORT will probably be wrong if the file has been modified.

IGETDP (*fileno%, plocation*)

fileno%	The file handle (integer returned by IOPEN) identifying the file to be used.

Returns:

plocation — The address of a 4-byte buffer in which ISAM puts the pointer to the record. A single precision number is big enough to be a data pointer.

IGETKD

This routine returns key descriptions that were given when the file was created. (Even if other fields were described when the file was created, they cannot be accessed through this routine.) In addition, for split keys, IGETKD returns the number of components in the split key.

If you want ISAM to return the current field name, you must set Kdes(1) to point to a string of 40 bytes; otherwise, set Kdes(1) to zero.

For example,

```
field.name$=string$(40,0)
Kdes(1)=VARPTR(field.name$)
```

A CHR$(0) will appear at the end of the field name returned by ISAM. To extract just the name, use the following BASIC statement:

LEFT$(*field.name*$,INSTR(*field.name*$,CHR$(0)) - 1)

IGETKD (*fileno%*, *keyno%*, *componentno%*, *pkeydes*)

fileno% — The file handle (integer returned by IOPEN) identifying the file to be used.

keyno% — The key handle that identifies the key that you want information about. Key handles range from 1 to *n*, where *n* is the number of keys in the file. The convention is to assign key handles beginning with 1, starting with the left-most byte in the record.

componentno% — The desired key component number. The key component number is a value from 1 to *n*, where *n* is the number of components of a split key. For keys that are not split keys, the key component number is 1. IGETKD uses *componentno%* to return the actual number of key components. If IGETKD returns -1 for the value of *componentno%*, there was an error.

pkeydes — This is a pointer to a buffer to hold the key description. For a split key, this is a pointer to a 9-integer array that will hold the key components.

Returns:

componentno% This is the number of fields in the key. For a non-split key, *componentno%* =1. If *componentno%* =-1, there was an error.

ILOCK

ILOCK controls record locking on a single file. This subroutine is particularly useful if you have opened a file in any one of the multiuser modes. For example, if you are updating a record, a lock on that record will prevent another user from modifying it between the time you read the record and then rewrite it. Without record locking, any changes the other user made would be lost.

There are two kinds of record locking: automatic and manual. Automatic mode is the default: every time you establish a new file position, the current record is locked, and the last record that was locked is released.

In manual mode, no record is locked unless a lock is specifically requested. The L_LOCK request locks the current record; it remains locked until the program calls the L_RELEASE function, ILOCK(fileno%,4). As a result, more than one record can be locked at one time in manual mode.

In both modes, a record must be unlocked before it can be read, written, rewritten or deleted. Any ISAM call that seeks or moves to a locked record results in a locked warning.

If a record is locked, there are two options: WAIT mode or NOWAIT mode. WAIT, which is the default mode, pauses until the required record is unlocked; with WAIT, your program will never return a locked error or warning. The NOWAIT mode, on the other hand, returns a locked error immediately.

ILOCK(*fileno%,request%*)

fileno% The file handle (integer returned by IOPEN) identifying

the file to access

request% This integer (0 to 5) specifies the desired locking function.

0: L_AUTO	Sets locking to Automatic (default)
1: L_MANUAL	Sets locking to Manual
2: L_WAIT	Causes program to wait if lock conflict (default)
3: L_NOWAIT	Lock conflicts returned to program
4: L_RELEASE	All locks on file released (Auto/Manual mode)
5: L_LOCK	Locks the current record

INEXT

For a given key, INEXT finds the record in the file with the next higher key value than the current record, and makes that record the current record. If INEXT finds duplicate key values, it determines what the next record will be, based on the order in which the records were entered. For records with identical key values, the "next" record will be the younger record (the record entered most recently). If there are duplicate key values, IXSTAT will return *ixstat%* =12 for "duplicate key warning" if the "duplicates allowed" switch was not set in the key description.

INEXT (*fileno%*, *keynum%*)

fileno% The file handle (integer returned by IOPEN) identifying what file to access.

keynum% The key handle of the keyset that you are sequentially reading. Each keyset has a position associated with it.

IOPEN

IOPEN opens an MS-ISAM file. IOPEN returns a handle to be used to identify the file in other MS-ISAM operations. When creating a new file (*mode%* = 1, 3, or 6), you must include a record description. The record description must at least contain descriptions for all key fields, and may contain descriptions for non-key fields. The record description is not neces-

sary when you are reading or updating a file (*mode%* =0, 2, 4, or 5).

IOPEN (*filename$*, *mode%*, *precdes*, *pkeydes*, *fileno%*)

filename$* This character string is the MS-ISAM data file name. Data file names conventionally have the suffix ".dat". *Filename$ must be less than 64 characters long.

mode% This integer specifies the mode in which to open the file. The seven modes are:

0: Read Only

Data file can only be read.

1: Write Only

New data file is created and can only be written to.

2: Update

Data file can be written to and read.

3: New Update

New data file is created and can be written to and read.

4: Multiuser Read Only

Data file can be read by more than one user.

5: Multiuser Update

Data file can be written to or read by more than one user.

6: Multiuser New Update

New data file is created which can be written to or read by more than one user.

precdes Integer pointing to start of record description. The record description contains three integers: the number of field descriptions (this may be more than the number of keys if split keys are used), the record type (segmented or nonsegmented), and the minimum record allocation (the default minimum allocation being 8 bytes).

pkeydes Integer pointing to start of key description.

Returns:

fileno% This is an integer used by MS-ISAM to identify the file given in *filename$*.

IPREV

For a given key, IPREV finds the record in the file with the next lower key value than the current record, and makes that record the current record. If IPREV finds duplicate key values, it determines what the next record will be, based on the order in which the records were entered. For records with identical key values, the "previous" record will be the older record (the record entered earlier). If there are duplicate key values, IXSTAT will return *ixstat%*=12.

IPREV (*fileno%*, *keynum%*)

fileno% The file handle (integer returned by IOPEN) identifying what file to access.

keynum% The key handle.

IREAD

IREAD reads the current record into the buffer. This routine does not change the file position.

IREAD (*fileno%, pbuffer, length%*)

fileno%	The file handle (integer returned by IOPEN) identifying the file in which to read.
pbuffer	Pointer to buffer to receive current record.
length%	This is the number of bytes that the record buffer *(pbuffer)* can hold. For fixed length records this is the record size. For variable length records it is the maximum size. IXSTAT will return an error (*ixstat%* =18) if the record was greater than *length%*. ISIZEOF can be used to determine the size of any record. IREAD uses *length%* to return the actual length of the record.

Returns:

length%	This is the number of bytes read into the record buffer.

IRESTOREFP

IRESTOREFP restores the file position and the key of reference that was saved by the last ISAVEFP call.

IRESTOREFP (*fileno%*)

fileno%	The file handle (integer returned by IOPEN) identifying what file to access.

IREWRITE

This subroutine rewrites an MS-ISAM record and updates the appropriate keys. IREWRITE is similar to IDELETE followed by IWRITE.

IREWRITE (*fileno%, pbuffer, length%*)

fileno%	The file handle (integer returned by IOPEN) identifying the file in which to rewrite.
pbuffer	

Pointer to buffer that holds the record to be rewritten. This pointer is returned by VARPTR().

length% This is the number of bytes to be written (the length of pbuffer).

ISAVEFP

ISAVEFP saves the current file position. The saved file position can then be restored by IRESTOREFP. Use ISAVEFP to place a "book mark" in your file and use IRESTOREFP to find it. This is faster than using ISEEK to find your place again because ISAVEFP saves the most recent key of reference established by ISEEK.

ISAVEFP (*fileno%*)

fileno% The file handle (integer returned by IOPEN) identifying what file to access.

ISEEK

This subroutine searches the file for a specified key, and, optionally, a specified value of that key. ISEEK can locate the first or last record, or the first record with a value equal to or a value greater than the specified key value.

ISEEK determines what record is the "first" or "last" record based upon the value of the specified key field in that record. Records are arranged alphabetically and numerically (numbers come before letters). When there is more than one record with the same value in the specified key field, order is determined by the order in which the records were originally entered, with older entries coming before younger entries.

If there are duplicate key values, IXSTAT will return *ixstat%*=12 for "duplicate key warning."

ISEEK (*fileno%, keynum%, pkey, keylen%, mode%*)

fileno% The file handle (integer returned by IOPEN) identifying the file in which to seek.

keynum% The key handle. This value can be found by IGETKD.

pkey This is a pointer to a buffer containing the key value to search for.

keylen% Integer length of the key value, in bytes.

mode% Integer indicating what to search for. The search modes are:

0	First record in a given key set.
1	Last record in a given key set.
2	Key equal to this value. This positions the file at the first record matching the key value.
3	Key greater than this value. This positions the file at the first record with a key value greater than the specified key value.
4	Key greater than or equal to this value. This positions the file at the first record with a key value equal to or greater than the specified key value.
5	First Equal Key. Use this when the seek key is less than the size of the keyfield. This positions the file at the first record that has a key with a matching prefix.

ISIZEOF

Computes the number of bytes in the current record. This can be useful when an MS-ISAM file is opened for reading and was created by another program or when the record has variable length fields.

ISIZEOF (*fileno%,fillsize%*);

fileno% The file handle (integer returned by IOPEN) identifying what file to access.

Returns:

recordsize%
Integer size of current record.

IWRITE

IWRITE writes a new record to the MS-ISAM file. The newly created record becomes the current record. The keys associated with the record are inserted into the key file.

IWRITE (*fileno%, pbuffer, length%*)

fileno%	The file handle (integer returned by IOPEN) identifying the file in which to write.
pbuffer	Pointer to buffer that holds the record.
length%	This is the number of bytes to be written (the length of pbuffer).

D.5 MS-ISAM Codes

Table D.1

Values Returned by Function IXSTAT

Symbol Name	Return Value	Description
IXS_OK	0	Successful completion
IXS_FN	1	Invalid MS-ISAM file number.
IXS_KD	2	Invalid key descriptor.
IXS_KN	3	Invalid key number.
IXS_VER	4	File and MS-ISAM version mismatch
IXS_DAM	5	File damaged
IXS_MEM	6	Memory full
IXS_NAM	7	File or directory not found, or invalid name.
IXS_ACC	8	Permission to access the file denied.
IXS_FUL	9	Disk is full or file is too large.
IXS_KEY	10	Key not found
IXS_EOK	11	Keyset boundary reached (start or end).
IXS_DKW	12	Duplicate key warning
IXS_DKE	13	Duplicate key error
IXS_RLK	14	Record locked
IXS_FLK	15	File locked
IXS_DSK	16	Fatal error: Disk I/O operation failure.
IXS_POS	17	File position (location of current record) has been lost.
IXS_OVR	18	Buffer overflow (buffer too small).
IXS_DD	19	Data dictionary error.
IXS_SEG	20	Invalid segment number in field descriptor.
IXS_TYP	21	Invalid field type in field descriptor.
IXS_RES	22	Reserved value in field descriptor non-zero.
IXS_KL	23	Key Length out of range in field descriptor.
IXS_MU	24	Multiuser failure.
IXS_BOK	25	Beginning Of Keyset boundary reached.
IXS_CRIT	26	Critical operating system error has aborted MS-ISAM.
IXS_LOAD	27	MS-ISAM has not been loaded in memory.
IXS_MODE	28	Invalid Open mode

Table D.2
Open Mode Definitions

Symbol Name	Return Value	Description
IO_READ	0	Read Only
IO_NEW_WRITE	1	Clear file, and open for Write only
IO_UPDATE	2	Read and Write
IO_NEW_UPDATE	3	Clear file, and open for Read and Write
IO_MU_READ	4	Clear file, and open for multiuser Read only
IO_MU_UPDATE	5	Multiuser Read and Write
IO_MU_NEW_UPDATE	6	Clear file, and open for multiuser Read and Write

Table D.3
Seek Modes

Symbol Name	Return Value	Description
IS_FIRST	0	First key in key set
IS_LAST	1	Last key in key set
IS_EQUAL	2	key equal to this value
IS_GREATER	3	key greater than this value
IS_NOT_LESS	4	key greater than or equal to this value
IS_EQ_FIRST	5	First equal key, seek on short length

Table D.4
ICONTROL Requests

Symbol Name	Return Value	Description
IC_CHECK	2	Write the data file to disk

Table D.5

Data Types

Symbol Name	Return Value	Description
IK_WORD	0	Unsigned integer - 2
IK_INTEGER	1	Signed integer - 2
IK_ALPHA	2	Character string
IK_CHAR	3	Character string
IK_LSTRING	4	Pascal Lstring type
IK_STRING	5	Character string
IK_NUMERIC	6	Free format, integer/real ascii number
IK_SMICRO	7	Single precision, Microsoft real number - 4
IK_SIEEE	8	Single precision, IEEE real number - 4
IK_SDEC	9	Single precision, Decimal real number - 4
IK_DMICRO	10	Double precision, Microsoft real number - 4
IK_DIEEE	11	Double precision, IEEE real number - 4
IK_DDEC	12	Double precision, Decimal real number - 4
IK_SHORT	13	Signed integer - 1
IK_LONG	14	Signed integer -4
IK_BYTE	15	Unsigned 1 byte integer - 1

Data types shown in Table D.5 that are followed by a number have the given fixed size in bytes. You must always give a value when referring to these types. The IEEE numbers refer to the floating point numbers that appear in the binary version of MS-BASIC. The decimal numbers refer to the floating point numbers that appear in the decimal version of MS-BASIC.

Table D.6

ILOCK Requests

Symbol Name	Return Value	Description
L_AUTO	0	Set record locking to Automatic
L_MANUAL	1	Set record locking to Manual
L_WAIT	2	Wait for locked record
L_NOWAIT	3	Don't wait for locked record
L_RELEASE	4	Release all locked records
L_LOCK	5	Lock current record

Appendix E
Rebuild 2.0

E.1 Introduction 387
E.2 Using Rebuild With Your Data Files 388
E.3 Invoking Rebuild 389
E.4 Definitions of Command Line Arguments 389
E.5 Using Rebuild as a Tool 391
E.5.1 Fixing a Corrupted Key File 391
E.5.2 Compressing the Data File 392
E.5.3 Converting Microsoft COBOL Indexed Files 392
E.6 Data Loss After a System Crash 394
E.7 Adding and Deleting Indexes 395
E.7.1 Updating a Key File: ASCII Input File 395
E.7.2 Updating a Key File: Interactive Mode 398
E.8 Creating and Using a dd (ASCII) Text File 401
E.8.1 Syntax Considerations 401
E.8.2 Statement Directory 402

E.1 Introduction

The Rebuild Utility (rebuild), version 2.0 and later, is a tool provided with the XENIX BASIC Compiler for use with files created with MS-ISAM. Rebuild allows you to:

1. Generate new key files from MS-ISAM data files.

   This is probably the most common use of rebuild. Rebuild must be run to regenerate a corrupted key file. The data file is the input to rebuild for this task. See Section E.5.1, "Fixing a Corrupted Key File," for more information on this task.

2. Compress the data file.

   Free space is created during normal record processing using IDELETE and IREWRITE statements, especially when variable length records are used. Data file compression removes this free space from the data file.

3. View the data dictionary.

   Whenever MS-ISAM or rebuild create a data file, they also build a "data dictionary, " a binary record description located at the beginning of the data file. This useful feature allows you to examine the Indexed file structure. It also allows you to save the data dictionary as an ASCII file by redirecting the output as shown in Section E.8, "Creating and Using a dd (ASCII) Text File."

   Rebuild can use a data dictionary to rebuild key files.

Note

Rebuild supports an interactive mode for entering new data dictionary information. You switch to this mode by starting rebuild with the −i switch as explained in Section E.7.2, "Updating a Key File: Interactive Mode."

E.2 Using Rebuild With Your Data Files

You can damage your key file by erasing all or part of it, or by failing to use ICLOSE to close an MS-ISAM file. Key files can also be damaged if your system goes down while an MS-ISAM file is open. If you know or suspect that your key file has been damaged, you can give a rebuild command. Rebuild erases the old key file and builds a new key file by using the information in your data file.

You may want to use rebuild when you change or add keys for a particular file. Suppose, for example, you have a file EMPLOYEE.DAT that you created for making mailing lists. EMPLOYEE.DAT contains the names and addresses of employees, their social security numbers, and dates of employment. Assume that EMPLOYEE.DAT has only one key field: the "name" field. Now, you have to send information on the pension plan to any employees who started working before a certain date. This would be easier to do if the "date-of-employment" field were a key field. You can do this in one of the following two ways: 1) use IOPEN to create new data and key files, describing the date-of-employment field as a key field in the "keydes" array; or 2) use rebuild to edit the data dictionary in EMPLOYEE.DAT and create a new key file. It is usually faster to use rebuild than to create a new MS-ISAM file.

Rebuild changes only the indexing information, not the actual data in the data file. You have to make sure the descriptions you give rebuild accurately describe the format and contents of your data file. If you use rebuild to change the name field to an integer data type, for example, rebuild trusts you to make sure the name field contains integers. It doesn't check or change any of the data. Similarly, if you tell rebuild the date-of-employment field is now the address field, and the address field is now the date-of-employment field, rebuild changes all the indexes and pointers accordingly, but will not move any of the data from one column to another.

Rebuild is also useful if you have an MS-ISAM file that has been extensively updated. The more an MS-ISAM file has been rewritten, the more likely it is that there is wasted space inside the file. This is especially likely if you are using variable length records, because rewriting can generate indirection records. (See Appendix D, "ISAM Reference," for a discussion of indirection records.) Whenever you tell rebuild to copy a file, it compresses the input data file by removing free space and indirection records.

E.3 Invoking Rebuild

Use the following syntax to provide arguments on the rebuild command line:

```
rebuild [– switch(es)] {source-file | - } [target-file]
        [-k "key description"] [-d dd-file]
```

Blanks are allowed in the command line between the arguments. An empty argument generates default values for the target-file and key description.

E.4 Definitions of Command Line Arguments

Rebuild 2.0 supports the following command line arguments:

1. *switches* (optional)

   a. –i(nteractive) switch

      The –i switch turns on rebuild's interactive mode (see Section E.7.2, "Updating a Key File: Interactive Mode," for details). If you use this switch, rebuild ignores all other arguments except source-file and target-file.

   b. –t(erse) switch

      The –t switch sets rebuild in the terse mode. If there are no fatal errors, there is no output to the screen.

   c. –p(rint) switch

      The –p switch brings an ASCII version of the data dictionary to the screen. The output can be redirected to a file or device. (See Section E.8, "Creating and Using a dd (ASCII) Text File," for information on redirection.)

   d. –f(orce) switch

      The –f switch allows rebuild to exit or perform other routines without waiting for your confirmation.

   e. –s(ingle key) switch

      The –s switch copies the data file, removes the data dictionary and free space; no key file is built. Generates a data file that is

compatible with the single-key Rebuild Utility, version 1.1.

2. *source-file* or the hyphen (-)

   The source-file is the name of the data file from which you want to generate a key file. Source-file is the only required element of the command line. Using the hyphen instead of a file name indicates you want to build an empty data file using an ASCII dd-file or you want to interactively enter the record structure.

3. *target-file* (optional)

   If target-file is present, the data file is copied and a key file (*target-file*.key) is generated. All free space is removed from the file. A target-file must be specified if a data dictionary is to be added to a data file.

4. *key description* (optional)

   Key description describes a single key of the format "*integer-1:integer-2* data type". Data type integer-1 and integer-2 refer to the key field location and key field length.

   The key description can be used only when the MS-ISAM file contains a single key. The description contains the starting position and size of the key.

5. *dd-file* (optional)

   The dd-file is an ASCII file prepared with a text editor. The dd-file contains an ASCII description of the data dictionary for a corresponding MS-ISAM data file. The presence of dd-file preceded by the −d flag tells rebuild that a file in ASCII format must be processed. A description of the dd-file is given in Section E.8, "Creating and Using a dd (ASCII) Text File."

   The data dictionary is a binary record description located at the beginning of the data file. Rebuild generates this binary record automatically during a data file copy, but will also generate this binary data dictionary from the ASCII record description contained in dd-file if you give both the dd-file and target-file arguments. If no target-file is specified, rebuild uses the data dictionary in source-file, rather than the dd-file, to create the new key file.

See the FIELD statement syntax in Section E.8.1, "Syntax Considerations," for the details about defining the record fields with rebuild 2.0.

E.5 Using Rebuild as a Tool

The following sections describe rebuild's major uses as a tool.

See Section E.3, "Invoking Rebuild," for a description of rebuild's command line arguments.

E.5.1 Fixing a Corrupted Key File

Assume an ISAM file named emprec.dat was open for writing when a power failure occurred. Since it was not closed by ISAM, it was "corrupted" and must have its key file rebuilt. Assume that the data file contains a data dictionary (this will be true in most cases). The command line to rebuild would be

```
rebuild emprec.dat
```

Rebuild will read the data file description from the data dictionary in data file emprec.dat, and use that to build a key file. The key file will be called emprec.key.

If the data file emprec.dat had not contained a data dictionary as assumed above (for example, if it had been created by COBOL 1.0 or MS-SORT), a description of the data would have to be provided by entering a key description on the rebuild command line, by providing a dd file (see Section E.8, "Creating and Using a dd (ASCII) Text File"), or by using rebuild's interactive mode (see Section E.7.2, "Updating a Key File: Interactive Mode").

If there is only a single key in the record, the command line itself can contain this data description. Use the command

```
rebuild emprec.dat -k "1:10 string"
```

This will also build the key file called emprec.key. It will contain a single key that is a fixed-length string, ten characters in length, which starts on the first character of the record. Since no target-file was specified to initiate the data file copy, your data file still won't contain a data dictionary and never will until a data file copy using a target-file specification is initiated.

E.5.2 Compressing the Data File

When a target file is specified on the command line, a copy of the source data file is produced in addition to a new key file. This data file is compressed (all free space records are removed). Free space records are "empty" records resulting from deletions and rewrites which have yet to be reused by MS-ISAM.

Whenever a data file copy is performed without the −s switch, rebuild will put a data dictionary into the data file. If one did not exist before, it will after rebuild is run. For example, using the indexed file emprec.dat on the rebuild command line

```
rebuild emprec.dat emprec2.dat
```

will create the new file emprec2.dat, which will contain the data records and the data dictionary from emprec.dat. This new file will contain no free space records. The key file will be called emprec2.key.

Note

If you have indexed files created by MS-COBOL that you want to use in your BASIC applications, read the following section on converting MS-COBOL indexed files.

E.5.3 Converting Microsoft COBOL Indexed Files

The key file formats for Indexed files created by MS-COBOL or rebuild versions prior to 2.0 (1.0 format files) and Indexed files created by MS-COBOL or rebuild 2.0 or later, or the Microsoft ISAM Facility (2.0 format files) are not identical and are not compatible. The conversion process from 1.0 to 2.0 follows:

1. 1.0 Format to 2.0 Format

   This is straightforward since rebuild is able to support both formats. If the 1.0 format data file is called file1.dat, a valid rebuild command line could be

   ```
   rebuild file1.dat file2.dat -k "1:10 ALPHA"
   ```

Note

Since Microsoft 1.0 format Indexed files do not produce a data dictionary, you must provide a key field description on the command line; in this case, −k "1:10 ALPHA". Rebuild will build the new Indexed data file with an appropriate data dictionary. This new data file (file2.dat) is now fully compatible with MS-COBOL version 2.0 and later.

2. 1.0 Format to Multi-Key 2.0 Format

If you plan to simultaneously upgrade this single-keyed data file to include alternate record keys, you'll need to revise your source program and create a new key file. Use one of the following methods to make this conversion:

New Data Dictionary Method

a. Extract a copy of the data dictionary from the created data (empty) file by invoking rebuild 2.0 with the −p switch and redirecting the data dictionary to a file. Delete the empty data file.

```
rebuild -p empty.dat >multi.dd
```

b. Update the single-key file of single.dat to a multi-key file and create your targeted multi-key data file by invoking rebuild again with a target-file for the data file copy and the data dictionary filename preceded by the −d switch.

```
rebuild single.dat multi.dat -d multi.dd
```

Rebuild Key Manipulation Method

Use the key manipulating functions of rebuild as described in Section E.7.1, "Updating a Key File: ASCII Input File" and Section E.7.2, "Updating a Key File: Interactive Mode" to update the keyfile and data dictionary by adding the described new key fields.

E.6 Data Loss After a System Crash

A data file can be damaged when electrical power to the computer system is interrupted or the operating system is rebooted while an ISAM file is open in WRITE or UPDATE mode.

A system failure may leave the data file with partially written data records because of the high degree of disk file buffering in memory. This may cause rebuild to fail to completely recover an indexed file for either or both of the following reasons:

1. If the system failure occurred during a file update job, when 512 bytes of the file are kept in memory, the file may contain records with both original and new information. Rebuild cannot determine which part of the data was written during the terminated job, and therefore cannot exclude the new, incomplete data from the rebuilt file. Adding a current date field to data records may help discriminate between original and new data.
2. If the system failure occurred while records were being added to the indexed file, the last 512 bytes of data will not be written to disk. Rebuild will detect that information is missing from the end of the file but will not be able to add it to the recovered file.

A data file can also be damaged when space is exhausted during a WRITE operation to the disk on which the indexed file resides.

You will know that space was exhausted during the WRITE operation when WRITE produces a boundary error (ixstat% = 9), indicating that the disk is full. When this happens, you should perform a CLOSE in order to write as much information as possible to disk.

It is likely, however, that the CLOSE will also return with a boundary error. As in the case of a system failure during the addition of records, the last 512 bytes of information will not be present within the data file and is therefore not recoverable by rebuild.

E.7 Adding and Deleting Indexes

When indexes need to be added, deleted, or otherwise modified, you can use two methods to update the key file: an ASCII input file, or rebuild's interactive mode.

E.7.1 Updating a Key File: ASCII Input File

Use a command line of the form

```
rebuild sample.dat sample2.dat -d new.dd
```

This directs rebuild to create sample2.dat and sample2.key, using new.dd as the data dictionary (record description) for sample2.dat. Rebuild also uses the file new.dd to update the index information in the key file when sample2.key is generated.

Note

Rebuild will ignore the input file new.dd if you have not specified sample2.dat. Without an indication that you want a data file copy, rebuild will assume that the existing data dictionary in sample.dat is acceptable.

If you use a dd-file that changes the record description for an ISAM file, you will need to add the same index and field information contained in the dd-file to the programs that access the data file.

For example, suppose that the dd-file, new.dd, contains the following record description, and is designed to update the key file for a multi-key data file called 'buyer.dat.' New.dd contains this information (*n:n* represents beginning byte:length):

```
Field "Process-Code" IS 59 : 1 String
  Keyed By 1 ;

Field "Number" IS 1 Integer
  Keyed By 2 ;

Field "Name" IS 3 : 20 String
  Keyed By 3 Duplicates Allowed ;

Field "City" IS 23 : 20 String ;

Field "Zip-Code" IS 43 Single IEEE ;

Split Keyset 4 is "Process-Code" "Zip-Code" Duplicates Allowed ;
```

You should change all MS-BASIC programs that used the buyer.dat key file with its previous index to reflect its new index structure. In the following source fragment from an application report.int, a split key containing the two italicized fields is added to the existing field statement in line 20:

```
10   Open "/dev/null" as #9 len=59   ' Open needed for FIELD
20   Field #9, \
               2 as Number$, \
              20 as Name$, \
              20 as City$, \
               4 as Zip.Code$, \
              12 as Description$, \
               1 as Process.Code$
```

Lines 500 to 680 in the following fragment indicate the corresponding fields in the KDES array that will make up the new split key, 59:

```
 10 DIM RDES(3), KDES(54)
 30 '
 60 RDES(1) = 6                      '6 keys in record
 70 RDES(2) = 0                      'Non-segmented record
 80 RDES(3) = 0                      'No minimum record allocation
 90 '
100 NAMES1$ = "Process-Code"   'Field Names
110 '
120 KDES(1) = VARPTR(NAMES1$)            'Field name
130 KDES(3) = 5                          'String Type
140 KDES(4) = 1                          'In segment 1
150 KDES(5) = 59                         'Position 1 of record -- 59:1
160 KDES(6) = 1                          'Length of field
170 KDES(7) = 1                          'Key number 1
180 KDES(8) = 1                  'Duplicates allowed
190 '
200 NAMES2$ = "Number"
210 '
```

```
220 KDES(1+9) = VARPTR(NAMES2$)
230 KDES(3+9) = 1                          'Integer
240 KDES(4+9) = 1
250 KDES(5+9) = 1                          '1:2
260 KDES(6+9) = 2
270 KDES(7+9) = 2                          'Key number 2
280 KDES(8+9) = 1
290 '
300 NAMES3$ = "Name"
310 '
320 KDES(1+18) = VARPTR(NAMES3$)
330 KDES(3+18) = 5                         'String
340 KDES(4+18) = 1
350 KDES(5+18) = 3                         '3:20
360 KDES(6+18) = 20
370 KDES(7+18) = 3                         'Key number 3
380 KDES(8+18) = (256*1) + 1               'Duplicates
390 '
400 NAMES4$ = "City"
410 '
420 KDES(1+27) = VARPTR(NAMES4$)
430 KDES(3+27) = 5                         'String
440 KDES(4+27) = 1
450 KDES(5+27) = 23                        '23:20
460 KDES(6+27) = 20
470 KDES(7+27) = 0                         'Not a key field
480 KDES(8+27) = 0
490 '
500 '     Split key
510 '
520 KDES(1+36) = 0                         'Process code already has name
530 KDES(3+36) = 5                         'string
540 KDES(4+36) = 1
550 KDES(5+36) = 59                        '59:1
560 KDES(6+36) = 1
570 KDES(7+36) = 4                         'Split Key #4, first component
580 KDES(8+36) = (256*1) + 2
590 '
600 NAMES5$ = "Zip-Code"
610 '
620 KDES(1+45) = VARPTR(NAMES5$)
630 KDES(3+45) = 8                         'Single precision
640 KDES(4+45) = 1
650 KDES(5+45) = 43                        '43:4
660 KDES(6+45) = 4
670 KDES(7+45) = 4                         'Split Key #4, second component
680 KDES(8+45) = (256*1) + 2
```

When you invoke rebuild with the command line

```
rebuild buyer.dat newbuyer.dat -d new.dd
```

the data dictionary and the key file for newbuyer.dat are updated, making newbuyer.dat compatible with new versions of report.int.

E.7.2 Updating a Key File: Interactive Mode

Rebuild also has an interactive mode that lets you modify the record description in an MS-ISAM file's data dictionary.

To invoke rebuild in this mode, enter a command line of the form

```
rebuild -i sample.dat sample2.dat
```

These arguments place the table of commands on your terminal screen (see Figure E.1, "Rebuild Interactive Mode Main Menu"), and place rebuild into the interactive mode. The record description can then be examined or modified.

List <field>:	List a field, or with no args, all fields.
Delete <field>:	Delete a field and/or split key.
Replace <field>:	Replace a field or split key.
Add <field>:	Add a new field or another field to a split key.
File options:	Change the file options.
Help:	List this display.
Exit:	Cancel rebuild.
Quit:	Quit editing and perform rebuild operations.

Figure E.1 Rebuild Interactive Mode Main Menu

Target-file sample2.dat is specified in the command line so that changes made to the record description will become part of the data dictionary once you complete the edit, and exit rebuild.

See Section E.8.2, "Statement Dictionary," for descriptions of the FIELD and RECORD statements.

Menu Descriptions

The following commands may be typed in full or represented by their first letter. You will receive a prompt if additional information is required. If <field> is more than one word, enclose the name in quotation marks ("). Most of the commands take one or more arguments. You may enter all arguments at once or as prompted by rebuild. Multiple arguments are separated by spaces.

List <field>:

Displays the current record description in ASCII file format. All field descriptions are printed if <field> is omitted.

Delete <field>:

Deletes a field description.

Add <field><statement>:

Adds a field description to the data dictionary. If <statement> is omitted, the Add statement issues the prompt

```
Specify a complete statement for Add
```

A FIELD/SPLIT statement, in the ASCII format described in Section E.8.1, "Syntax Considerations," should then be entered.

File <options>:

Modifies the file options that have a global effect. If <options> is omitted, the prompt returned is

```
N)ame S)egmented M)inimum Allocation
```

Each option corresponds to one of the the options on the RECORD statement from the ASCII format. Enter the first letter for the desired option. The RECORD statement is described in Section E.8.2, "Statement Directory." The Name and Minimum Allocation options will prompt for further input.

Replace <field><subcommand>:

Replaces all or part of a field description.

If the <subcommand> option is omitted, rebuild issues the following prompts:

```
N)ame L)oc T)ype K)ey# DE)scending D)uplicates I)nsensitive

R)emove-Split A)dd-Split DES)cending-Split
```

To select subcommands enter the capitalized letter(s) to the left of the right parenthesis ")". The subcommands N)ame through I)nsensitive deal with fields and keys. The subcommands R)emove-Split through DES)cending-Split deal with split keys.

a. N)ame ["]*field name*["]

   Replaces the field name description. The quotation marks are needed only when the name has more than one word separated by spaces.

b. L)oc [*segment#*]*start position*[:*length*]

   Changes the field position.

c. T)ype *field type*

   Changes the data type.

d. K)ey# *number*

   Assigns a key number to the field. You must select the same key number that your MS-BASIC programs are going to use. Specifying a key number of 0 makes the field non-keyed.

e. DE)scending

   This switches the internal "Descending" flag on or off each time it is executed.

f. D)uplicates

   This switches "duplicates allowed" on or off. This subcommand also works with split keys.

g. I)nsensitive

   This switches the insensitive switch on or off.

h. R)emove-Split *field*

   Allows you to delete the components which make up a split key.

i. A)dd-Split *field*

   Allows you to add the components which make up a split key.

j. DES)cending-Split *field*

Allows individual components of a split key to have the descending attribute independent of the descending attributes in the referenced field. This switches Descending on or off each time it is executed.

E.8 Creating and Using a dd (ASCII) Text File

To bring an ASCII version of sample.dat's data dictionary to the screen, use the −p switch on the command line

```
rebuild -p sample.dat.
```

If you want to redirect this output to another file (e.g., text.dd), and then edit it to new specifications, enter the command line

```
rebuild -p sample.dat >text.dd
```

You could then edit text.dd, and add the new version of the data dictionary to sample2.dat, with the command line

```
rebuild sample.dat sample2.dat text.dd
```

The ASCII files acceptable to rebuild lend themselves to easy maintenance. The following sections describe the syntax and language recognized by rebuild, and provide practical examples of record descriptions in data dictionary format.

E.8.1 Syntax Considerations

You can describe an MS-ISAM file record by using the RECORD, FIELD, and SPLIT KEYSET statements. The RECORD statement is optional (only one RECORD statement may be given for any file), but you must use one or more FIELD statements. SPLIT KEYSET statement(s) are optional. All statements must end with a semicolon.

In the syntax diagrams that follow, optional material is indicated by square brackets ([ ]). Braces ({ }) are used two ways: with the vertical bar (|) to indicate a choice of two options; or with ellipses (...) to indicate a repeated group. The optional quoted square brackets (["["] and ["]"]) in the

FIELD statement indicate that an actual bracket character may be entered. If the optional opening bracket ([) is chosen, the closing bracket (]) must also be used, and vice versa.

```
[RECORD  [record-name] [SEGMENTED]
    [MINIMUM ALLOCATION IS integer]] :

FIELD [field-name] IS ["["] position [:size]
    datatype [INSENSITIVE] ["]"]

   [KEYED [BY] key-number [DUPLICATES ALLOWED]
   [DESCENDING]] :

[SPLIT KEYSET key-number IS
   {{field-name | key-number} [DESCENDING]} ...
      [DUPLICATES ALLOWED]] :
```

E.8.2 Statement Directory

Rebuild 2.0 supports the following keywords to create an ASCII file that describes data file records:

RECORD Statement

The optional RECORD statement sets global file options. These options can also be set with the F(ile) options command in the interactive mode.

The arguments of the RECORD statement are

record-name	an optional identifying name.
SEGMENTED	the file will have a special segmented structure.
MINIMUM ALLOCATION	the minimum record size allocated from the data file. MINIMUM ALLOCATION can be used to minimize data file fragmentation when IREWRITEing variable length records.

FIELD Statement

The FIELD statement describes each field in the record. We recommend that all fields in a record be defined even though rebuild requires that only key fields be defined. This additional effort will allow future utilities to access desired data items. The interactive mode commands A(dd), L(ist), P(rint), R(eplace), and D(elete) use field names or entire FIELD statements as input.

The arguments of the FIELD statement are

field-name	Name of field; maximum 40 characters. If more than one word, enclose the name in double quotation marks (").
position	Position from start of the record (1 being the first position).
size	Size of data field in characters.
datatype	The data type of the contents of the field. Rebuild supports the following data types:

ALPHANUMERIC

BOOLEAN {1-, 2-, 4-byte lengths only}

BYTE {1}

CHARACTER

DOUBLE DECIMAL {8}

DOUBLE IEEE {8}

DOUBLE [MICROSOFT] {8}

INTEGER {1-, 2-, 4-byte lengths only}

LOGICAL {1-, 2-, 4-byte lengths only}

LSTRING

NUMERIC

SINGLE DECIMAL {4}

SINGLE IEEE {4}

SINGLE [MICROSOFT] {4}

STRING

WORD {2}

Data types followed by numbers in braces ({ }) have a fixed *size* value, which must be specified as noted in *size*. Neither the braces themselves, nor the numbers they contain, are actually part of the FIELD statement syntax. If a *size* argument is given for a data type with an implied length, *size* will be ignored.

When using the binary version of BASIC, use the IEEE floating point data types. For the decimal version, use the DECIMAL floating point data types.

INSENSITIVE — Characters which differ by case only are considered equal.

KEYED BY Clause

The optional KEYED BY clause makes a specific field value into a key to the record. During revisions of the key file using rebuild, the KEYED BY clause is used to make a field into a record key. The arguments are:

key-number — Identifies a key and is used in the MS-ISAM file system. The number can range from 1 to (n), where (n) is the number of keys.

DUPLICATES ALLOWED — Allows two or more records to have the same key value.

DESCENDING — Reverses the ordering of keys. DESCENDING key files should be used only with programs that expect this feature.

SPLIT KEYSET Statement

The SPLIT KEYSET statement defines split keys. The parameters are

key-number — Serves the same purpose as the key number in the KEYED BY clause. Identifies a single key composed of several concatenated record fields.

field-name or *key-number* — *field-name* is the same field name used to identify the field in the FIELD statement.

	If no name was given in the FIELD statement, *key-number* can be substituted for *field-name.*
DESCENDING	Reverses the ordering of the preceding component of the split key.

Appendix F
Error Messages

F.1 Invocation Errors 409
F.2 Compile-Time Errors 410
F.3 Run-Time Errors 416

During development of a Microsoft BASIC program with the XENIX Compiler, four different kinds of errors may occur:

1. Invocation errors
2. Compile-time errors and warnings
3. Linker errors
4. Run-time errors

Each type of error is associated with a particular step in the program development process. Invocation errors occur when you invoke the compiler; compile-time errors and warnings occur during compilation; linker errors occur during linking, and run-time errors occur when the compiled XENIX program is running.

This appendix lists error codes and error messages for invocation, compile-time and run-time errors, along with any error numbers that are assigned. Error messages are organized in the following sections:

Section	Errors
F.1	Invocation errors
F.2	Compile-timer errors
F.3	Run-time errors

F.1 Invocation Errors

Invocation errors occur when you enter illegal input on the command line or in response to prompts during invocation. The messages that may occur when the compiler is invoked are listed below:

`Bad filename`

Improper file specification entered.

`Can't create file`

Disk is write protected or disk is full.

`Command error: 'c'`

An error has occurred at character *c*. If no *drivename* appears, the disk in the default drive is full.

`File not found`

The file does not exist, or you do not have permission to access the file.

`Unknown switch: /s`

Illegal compiler option *s*.

F.2 Compile-Time Errors

When errors occur while your program is compiling, the compiler displays the line containing the error, and a two-character code for the error. An arrow beneath that line points to the place in the line where the error occurred. In some cases, the compiler reads ahead on a line to determine whether an error has really occurred, and the arrow will point a few characters beyond where the error actually took place.

The messages listed below describe both severe errors and warning errors. When a severe error occurs, the compiler will attempt to continue. However, the resulting object file will not be correct. You must correct the error and recompile the source file before linking.

On the other hand, when a warning error occurs, compilation continues, but warning errors are displayed to point out poorly constructed program statements. The resulting object file can be linked to form an executable program, but the program may not perform as intended. Errors and warnings are indicated either by a long message or by a two-letter code. Long error messages describe general conditions that are not associated with a particular line number. Two-letter codes indicate errors on specific lines.

`Binary source file`

The file you have attempted to compile is not an ASCII file. All source files saved from within the BASIC interpreter should be saved with the ",A" option.

BS

Bad subscript. The following situations cause this error:

- Illegal array dimension value
- Wrong number of subscripts

CD

Duplicate COMMON variable.

CN

COMMON array not dimensioned.

CO

COMMON out of order. COMMON must appear before any executable statements.

DD

Array already dimensioned. This error can be caused by the following:

- More than one DIM statement for same array
- DIM statement after initial use of array
- OPTION BASE after array dimensioned

FD

Function definition error. This error occurs when a previously defined function is redefined or when DEF FN...EXIT DEF/END DEF statements are incorrectly nested.

FN

FOR...NEXT error. This can be caused by the following conditions:

- FOR loop index variable already in use
- FOR without NEXT
- NEXT without FOR

IN

$INCLUDE file not found.

Internal error

An internal error has occurred in the XENIX Compiler. Please note precisely what actions preceded this message and report the problem to Microsoft immediately.

`Line` ***label*** `is undefined`

A statement refers to a nonexistent line label.

`Line` ***n*** `is undefined`

A statement refers to a nonexistent line number.

`LL`

Line too long. Lines are limited to 254 characters.

`LS`

String constant too long. Strings are limited to 32767 characters.

`MC`

Warning error. A metacommand is incorrect.

`Memory overflow`

Available memory has been exhausted. Try compiling without the debug options. If memory is still exhausted, break your program into parts and use the CHAIN command or separate compilation facilities.

`Missing NEXT for variable`

No NEXT was found to match a FOR statement.

`ND`

Warning error. An array is declared but not dimensioned.

`OM`

Out of memory. This error can be caused by the following conditions:

- Array too big
- Data-memory overflow
- Too many statement numbers
- Program-memory overflow

`OV`

Math overflow. The result of a calculation is too large to be represented in BASIC number format.

SB

This is a subprogram definition error and is usually caused by one of the following:

- The subprogram is already defined, or a subprogram of that name is already defined.
- The program contains incorrectly nested SUB...EXIT SUB/END SUB statements.

SI

Warning error. Statement ignored. This warning often results when an unimplemented command is used in a program.

SN

Syntax error. This error can be caused by the following conditions:

- Illegal argument name
- Illegal assignment target
- Illegal constant format
- Illegal debug request
- Illegal DEF*xxx* character specification
- Illegal expression syntax
- Illegal function name
- Illegal function formal parameter
- Illegal separator
- Illegal format for statement number
- Invalid character
- Missing AS
- Missing equal sign
- Missing GOTO or GOSUB
- Missing comma
- Missing INPUT
- Missing line number

- Missing left or right parenthesis
- Missing minus sign
- Missing operand in expression
- Missing semicolon
- Missing TO
- Missing THEN
- Missing BASE
- Name too long
- Expected GOTO or GOSUB
- String assignment required
- String expression required
- String variable required
- Illegal syntax
- Illegal FOR loop index variable
- Illegal COMMON name
- Illegal subroutine name
- Variable required
- Wrong number of arguments
- Formal parameters not unique
- Single variable only allowed

SQ

This is a sequence error and is usually caused by one of the following conditions:

- Duplicate statement number
- Statement out of sequence

ST

Warning error. There is a missing STATIC or SUB statement.

TC

This message is caused by one of the following conditions:

- Expression too complex
- Too many arguments in function call (limit of 60)
- Too many dimensions (limit of 255)
- Too many variables for LINE INPUT (limit of 1)
- Too many variables for INPUT (limit of 60)

TM

Type mismatches are caused by the following conditions:

- Data type conflict
- Variables of different types

UC

Unrecognizable command.

UF

Function not defined. Functions must be defined before they are used.

WE

This is a WHILE...WEND error, caused by either a WHILE statement without a corresponding WEND, or a WEND statement without a corresponding WHILE.

/O

This message is caused by division by zero, division of the integer -32768 by 1 or -1, or moding of the integer -32768 by 1 or -1.

/E

Missing "/E" option. Programs that contain ON ERROR GOTO statements must be compiled with the -E option.

/X

Missing "/X" option. Programs that contain RESUME, RESUME NEXT, and RESUME 0 statements must be compiled with the /X option.

F.3 Run-Time Errors

The coded errors listed below may occur while the program is running. The compiler run-time system prints long error messages followed by an address, unless a **-D**, **-E**, or **-X** option is specified in the compiler command line. In those cases, the error message is also followed by the number of the line in which the error occurred. The standard forms of the error messages are as follows:

`Error` *X* `in module` *zzzzzzzz* `at address` *nnnn:nnnn*.

and

`Error` *X* `in line` *yyyy* `of module` *zzzzzzzz* `at address` *nnnn:nnnn*.

A description of the uncoded (unnumbered) run-time error messages follows this list.

Code	Message
2	`Syntax Error` A line is encountered that contains an incorrect sequence of characters in a DATA statement.
3	`RETURN without GOSUB` A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	`Out of Data` A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	`Illegal Function Call` A parameter that is out of range is passed to a math or string function. A function call error may also occur for the following reasons: • A negative or unreasonably large subscript is used. • A negative number is raised to a power that is not an integer.

Code	Message
	• A USR function is called that has an undefined starting address. • A negative record number is given when using GET *file* or PUT *file*. • An improper or out-of-range argument is given to a function. • Strings are concatenated to create a string greater than 32767 characters in length.
6	Floating Overflow or Integer Overflow The result of a calculation is too large to be represented within the range allowed for floating-point numbers.
7	Out of memory Not enough memory is available to allocate a file buffer.
9	Subscript Out of Range An array element was referenced with a subscript that was outside the dimensions of the array; or an element of an undimensioned dynamic array was accessed. This message is generated only if the **-D** option was specified at compile time.
10	Redimensioned Array A second DIM statement was executed for an already dimensioned dynamic array.
11	Division by Zero A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Also occurs if the integer -32768 is divided by 1 or -1, or if -32768 is moded by 1 or -1.
13	Type mismatch The argument types are not compatible.

Code	Message
14	Out of String Space String variables exceed the allocated amount of string space.
16	String formula too complex A string formula is too long, or an INPUT statement requests more than 15 string variables. Break the formula or INPUT statement into parts for correct execution.
19	No RESUME The end of the program was encountered while the program was in an error-handling routine. A RESUME statement is needed to remedy this situation.
20	RESUME without Error A RESUME statement is encountered before an error-trapping routine is entered.
50	Field Overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal Error An internal malfunction occurred in the XENIX Compiler. Report to Microsoft the conditions under which the message appeared.
52	Bad File Number A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
53	File Not Found A KILL, NAMES, FILES, or OPEN statement references a file that does not exist on the current disk.

Code	Message
54	Bad File Mode
	An attempt is made to use PUT or GET with a sequential file, or to execute an OPEN with a file mode other than I, O, or R. This error also occurs when an attempt is made to read from a file opened for output or appending.
55	File Already Open
	A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.
57	Disk I/O Error
	An I/O error occurred on a disk I/O operation. The operating system cannot recover from the error.
58	File Already Exists
	The filename specified in a NAME statement is identical to a filename already in use on the disk.
61	Disk Full
	All disk space has been allocated.
62	Input Past End
	An INPUT statement reads from a null (empty) file, or from a file in which all data has already been read. To avoid this error, use the EOF function to detect the end of file.
63	Bad Record Number
	In a PUT or GET statement, the record number is equal to zero.
64	Bad File Name
	An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).
68	Device unavailable
	The device you are attempting to access is not on-line or does not exist.

Code	Message
70	Permission Denied An attempt was made to write to a write-protected file.
71	Unprintable error There is no message for this error.
72	Disk media error Disk-drive hardware has detected a physical flaw on the disk.
73-76	Unprintable error There are no message for these errors.
77	Lock attempt failed An attempt to lock a file or section of a file has failed because the section is already locked by another user.
78-87	Unprintable error There are no messages for these errors.
88	Child process error The child process set in motion by a **SHELL** statement failed, either as the result of a XENIX error, or as the result of a shell termination signal (e.g, Control-D).
89-255	Unprintable error There are no messages for these errors.

The following is a list of uncoded run-time error messages:

Error in an executable file

This error occurs when a file is not of the correct type or does not have the correct permissions. A file must be an executable file if it is to be executed with RUN or CHAIN. For chaining, the user must have both read and write permission on the chained-to file. This error is severe and cannot be trapped.

No Line Number in *modnam* at address *nnnn:nnnn*

The error address cannot be found in the line-number table during error trapping. This occurs when you have forgotten to use either the **-X** or **-E** compiler options in programs that contain **RESUME** and **ON ERROR GOTO** statements. It can also occur if the line-number table has been accidentally overwritten by the user program. This error is severe and cannot be trapped.

String Space Corrupt in [*line number* of] *modnam* at address *nnnn:nnnn*

This error occurs when an invalid string in string space is being deallocated, usually in a string assignment statement. See the listing following the error message "String Space Corrupt during G.C." below for additional causes. This error is severe and cannot be trapped.

String Space Corrupt during G.C. in [*line number* of] *modnam* at address *nnnn:nnnn*

This error occurs when an invalid string in string space is being deleted during garbage collection. The probable causes for either of the "String Space Corrupt" errors are as follows:

- A string descriptor or string backpointer has been improperly modified. This may occur if you use an assembly-language subroutine to modify strings.
- Out-of-range array subscripts are used and string space is inadvertently modified. The **-D** option may be used to ensure that array subscripts do not exceed the array bounds.
- Incorrect use of the POKE and/or DEF SEG statements may modify string space improperly.
- Mismatched COMMON declarations may occur between two chained programs.

Glossary

The definitions in this glossary are intended for use with this manual. Neither the individual definitions nor the list of terms is comprehensive.

Basename

The portion of the filename that precedes the filename extension. For example, SAMPLE is the basename of the file SAMPLE.BAS.

Call by reference

See "Pass by reference."

Call by value

See "Pass by value."

Compile time

The time during which the compiler is executing, compiling a BASIC source file, and creating a relocatable object file.

Compiler

A set of programs that translates BASIC programs into a language understood by the computer.

Disassembly listing

A listing that shows assembled code, instructions, and addresses relative to the start of the program or module.

Double precision

A real value that is allocated 8 bytes of memory.

Executable program

A file that contains executable program code. When the name of the file is typed at the system prompt, the statements in the file are executed.

External symbol

A symbol referenced in one assembly-language module but defined (made PUBLIC) in another module. References to the symbol are resolved (filled in with the correct addresses) by the linker.

Heap

An area of random access memory that is used by BASIC as the data management area. Variables and arrays are stored here.

Library

A directory that stores related modules of compiled code.

Link map file

A file that shows the address of every code and data segment in a program, relative to the start of the program.

Link time

The time during which the linker is executing, and during which it collects and links relocatable object files and library files. *See also* Compile time, Run time.

Linking

The process in which the linker loads modules into memory, computes absolute offset addresses for routines and variables in relocatable modules, and resolves all external references by searching the run-time library. After loading and linking, the linker saves the modules it has loaded into memory as a single executable file.

Machine code

Instructions that a microprocessor can execute.

Main program

In a program that calls subprograms or subroutines, the calling program is the main program.

Memory map

A representation of where in memory the computer expects to find certain types of information.

Metacommands

Metacommands are special commands enclosed in comments in the source file that tell the compiler to perform certain actions while it is compiling the program; for example, to produce a listing file in a certain format.

Module

A general term for a discrete unit of code. There are several types of modules, including relocatable and executable modules. The compiler creates relocatable modules that are later processed by the linker. Your final executable program is an executable module.

Object file

A file that contains relocatable machine code.

Offset

The number of bytes from the beginning of a segment to a particular byte in that segment.

Pass by reference

A method of passing parameters in which the calling routine provides the called routine with the memory addresses of the parameters. This permits subroutines to change the values of the parameters.

Pass by value

A method of passing parameters in which the calling routine provides the called routine with the current value of the parameters. This prevents the possibility of the subroutine changing the value of the parameter; the subroutine only changes its copy of the value.

Relocatable

The term applied to a module when the code within it can be placed and run at different locations in memory. The relocatable modules created by the compiler are an intermediate stage between source code and executable code; they are changed into executable modules by the linker and have .o extensions.

Routine

Executable code residing in a module, and usually representing a particular feature or procedure. More than one routine may reside in a module.

Run time

The time during which a compiled and linked program is executing. Run time refers to the execution time of a program rather than to the execution time of the compiler or the linker.

Subprogram

A separately compilable module of code delimited by the SUB and END SUB or EXIT SUB statements. *See also* Main program.

Unbound External

A symbol that is referenced in one assembly-language module but is not made PUBLIC in another module that is linked with it. Unbound external references are usually caused by misspellings, or by omitting from the link command line the name of the module containing the desired symbol.

Index

ABS function, 150
Absolute value, 150
ALL, 173
Alphanumeric line labels. *See* Line labels
a.out file
  executable program file name, 19
  naming, 21, 30
Apostrophe, entering, 6
Arctangent, 152
Arithmetic operators, 118
Arithmetic overflow, 23, 126
Arrays, 197
  bounds check, 23
  dimensioning, 115
  $DYNAMIC metacommand, 130
  dynamic
    defined, 129
    ERASE statement, effect, 130
    memory space allocation, 130
    REDIM statement, 130, 291
  elements, 115
  LBOUND function, 79, 234
  memory allocation, 134
  passing with CALL, 77
  storing, 25
  subprograms, parameter list form, 74
  $STATIC metacommand, 130
  static
    bounds checking, 131
    defined, 129
    ERASE, effect, 130
    REDIM, effect, 130
    redimensioning, 130
  subscripts, 116
  UBOUND function, 79, 333
  unallocated, default to static, 132
  variables, 173, 191
ASC function, 151
ASCII
  codes, 166, 347
  file, 390
  format, 151, 160
Assembler error messages, 26
Assembly language
Assembly language *(continued)*
  coding rules, 97
  interfacing, 93
  loading files, 93
  source file, using, 25
  subroutines, 93, 153, 158, 273, 337
ATN function, 152
AUTO command, 143

.BAS file, execution by compiler, 144
bascom command
  -A option, 22
  -c option, 20
  disassembly listing, creating. 22
  error messages, 26
  executable file naming option, 21
  form, 17
  link map file option, 22
  -m option, 22
  -o option, 21
  object file, creating, 20
  options,form, 18
  with assembly language source, 25
Basename, defined, 422
BASIC run-time errors, 416
Boundary error indicator, 394
Bounds checking, arrays, 131
Built-in functions, 124

C language modules, 99
CALL statement, 153, 158
  assembly language subroutine, 93
  BASIC subprograms, 75
  compiler/interpreter differences, 144
Calling C language modules, 99
CALLS statement, 99
  compiler/interpreter differences, 144
Carriage return, 229, 242, 341, 343, 345
Case, line labels, 107
CDBL function, 159
CHAIN statement, 144, 160, 173
Characters recognized by BASIC, 105
CHDIR, 165

Child process, 35
CHR$ function, 166
CINT function, 167
CLEAR statement, 168
Clearing output window, 171
CLOSE statement, 170
CLS statement, 171
Code segment address listing, 22, 32
COMMAND$ function, 137
Commands
  See also Specific Command
  bascom, 17
Comments, introducing, 108
COMMON statement, 173
  compiler/interpreter differences, 144, 145
  described, 131
  DIM, restriction with, 130
  order of variables, 175
  SHARED attribute, 80
Compile time
  defined, 422
  error messages, 410
Compiler/interpreter comparison, 138
Compiling
  bascom command, 17
  multiple source files, 20
  single source file, 19
Compressing ISAM data files, 392
CONT command, 143, 242
Converting indexed file format, 392
Converting interpreted programs, 137
COS function, 176
CSNG function, 177
CVD function, 178
CVDBCD function, 143
CVI function, 178
CVS function, 178
CVSBCD function, 143

-d flag, 390
-D option, 23, 24
Damaged data file, 394
Data segment address listing, 22, 32
DATA statement, 180, 296
Data types
  Double-precision floating-point numeric, 110
  integer numeric, 109
  Single-precision floating-point numeric, 109
Data types *(continued)*
  string, 109
DATE$ function, 182
dd-file, 390
Debug option, static array bounds checking, 131
Debugging messages, 23
Declaring variable types, 114
DEF FN statement, 145, 183, 184
DEF USR statement, 143
DEFDBL statement, 143, 187
DEFINT statement, 115, 143, 187
DEFSNG statement, 143, 187
DEFSTR statement, 143, 187
DEF*type* statements
  See also Specific Statement.
  compiler/interpreter differences, 144
DELETE command, 143
DELETE statement, 190
Device-independent I/O, 35
DIM statement, 191
  array dimensioning, 130
  COMMON, restriction with, 130
  compiler/interpreter differences, 144
  SHARED attribute, 80
  use, 130
Direct mode, 263
Directories, hierarchical, 37
Disabling metacommands, 133
Disassembled object code listing, 22, 135
Disk storage error, 394
Division by zero, 119
Double precision, 111, 159, 187, 279
$DYNAMIC, 134
Dynamic arrays
  See also Arrays
  $DYNAMIC metacommand, 134
  REDIM Statement, 291
$DYNAMIC metacommand
See also Metacommands
  array memory allocation, 130, 134

-E option, 24, 139
EDIT command, 143
END SUB, 73
END statement, 193, 215, 216
  compiler/interpreter differences, 145
  use with $INCLUDE, 134

EOF function, 195
ERASE statement, 197
  arrays, effect on, 130
  compiler/interpreter differences, 145
ERL function, 108, 199
ERR function, 199
Error codes, 201
Error handling, 23, 24, 199, 263
Error messages, 409
  assembler, 26
  bascom command, 26
  linker, 26
ERROR statement/command, 201
Error trapping, 201, 297
  ISAM files, 47
  line 0, 107
Errors
  run-time, 416
  severe, 410
Executable file
  creating from object file, 30
  name, 19, 21, 30
Executing a program, 19
Execution time. *See* Program execution
EXIT statement, use in subprograms, 324
EXIT SUB, 73
EXP function, 203
Expressions
  conversion of operands, 126
  definition, 116
  passing to subprograms, 79
External symbol, 422

FIELD statement, 109, 204
File naming conventions, 36
FILES statement, 207
Files
  ASCII format, 139, 390
  data, 37
  dd-file, 390
  EOF function, 195
  $INCLUDE, 133
  LOC function, 243
  LOCK statement, 245
  LOF function, 248
  multi-key Indexed file, 392
  multi-user, 65
  naming, 36
  object See Object file, 20
Files *(continued)*
  OPEN statement, 266
  random, 204, 213, 233, 253, 257, 266, 285, 303
    accessing, 43
    creating, 41
  sequential, 37, 226, 233, 240, 266, 275, 343
    accessing, 39
    appending, 40
    creating, 38
  single-key Indexed file, 392
FIX function, 208
Fixed-point constants, 111
Floating-point constants
  allowable range
    double precision, 110, 111
    single precision, 109, 111
  exponent, 109
  mantissa, 109
  most significant bit, 109
FOR...NEXT statement, 145, 209, 261
FORTRAN subroutines, 99
FRE function, 145, 212
Functional operators, 124
Functions, 183
  COMMAND$, 137
  contrasted with statements, 149
  enhanced in compiler, 144
  FRE, 145
  interpreter
    CVDBCD, 143
    CVSBCD, 143
    MKDBCD$, 143
    MKSBCD$, 143
    USR, 143
  intrinsic, 124
  LBOUND, 79, 137
  new. *See* New statements/functions
  not accepted by compiler, 143
  UBOUND, 79, 137
  user-defined, 124, 184

GET statement, 204, 213
Global symbols
  address listing, 22, 32
  listing, 22, 32
GOSUB statement, 215, 216, 299
GOSUB...RETURN, 145
GOTO statement, 215, 219

GOTO statement *(continued)*
compiler/interpreter differences, 145
$INCLUDE, use with, 134
required with line label, 108
subroutines, use with, 216

Heap, 423
HEX$ function, 221
Hexadecimal, 221
Hierarchical directories, 37
Hierarchy of operations, 117

IF...GOTO statement, 222
IF...THEN statement, 199, 222
IF...THEN...ELSE statement, 222
GOTO, when required, 108
line labels, 108
$INCLUDE metacommand, 133
Indexes, input to rebuild, 395
INKEY$ function, 225
INPUT$ function, 228
INPUT statement, 204
INPUT# statement, 226
INPUT statement, 229
INSTR function, 231
INT function, 232
Integer constants, 167, 208, 232
decimal, 110
hexadecimal, 111
octal, 111
Integer division, 119
Internal module names, changing, 136
Interpreter programs, converting, 137
Interpreter/compiler comparison, 138
Intrinsic functions, 124
Invocation errors, 409
Invoking rebuild, 389
ISAM files, 47
adding a record, 59
advantages, 47
automatic locking, 66
changing a record, 61
closing a file, 362
codes, 1, 381
comparison with sequential, 48
compressing, 392
construction, 47
creating, 50
currency, 361

ISAM files *(continued)*
current record, 361
data dictionary, 47
data file
non-segmented records, 48
record types, 48
data files, 47, 359
data file
segmented records, 48
definition, 359
deleting a record, 60
error codes, 1, 381
error trapping, 47
field description, 366
file handles, 363
ICLOSE, 369
ICONTROL, 363, 369
IDELETE, 370
IDREAD, 363, 370
IGETDP, 363, 371
IGETKD, 363, 372
ILOCK, 66, 363, 373
in BASIC programs, 50
INEXT, 374
IOPEN, 374
IPREV, 376
IREAD, 377
IRESTOREFP, 377
IREWRITE, 377
ISAVEFP, 378
ISEEK, 378
ISIZEOF, 379
IWRITE, 380
key, 359
key description array, 53
key files, 47
key handles, 50, 363, 364
keys, 49
locking, 66
manual locking, 66
modified with rebuild, 398
multi user applications, 67
opening a file, 361
operating system concerns, 67
ownership, 67
parameters, 363
permissions, 67
reading a record, 361
record description, 365
record description array, 53
records, 360

records *(continued)*
non-segmented, 360
segmented, 360, 364
rewriting a record, 361, 362
searching, 62
segment table, 364
segmented records, 364
sharing in XENIX, 67
sizing a record, 362
split key, 359, 364
split keys, 49
status codes, 1, 381
subroutines, 368
updating, 56
writing a record, 361
writing an application, 360
XENIX considerations, 67
ISAM file modes, 375
ishare utility, 68

-k flag, 390
KILL command, 233
KYBD:, 35, 195, 243, 248

LBOUND function, 79, 234
LBOUND/UBOUND, 137
ld command
executable file naming option, 30
-l option, 31
link map file option, 32
-m option, 32
-o option, 30
LEFT$ function, 236
LEN function, 237
LET statement, 204, 238
Library
search order, 31
special, linking option, 31
Line 0, effect on error trapping, 107
LINE INPUT# statement, 240
LINE INPUT statement, 242
Line labels
alphanumeric, 106, 107
case, significance, 107
RESUME statement, 108
use, 106, 108
use with GOTO, 108
Line length restrictions, 108
Line number check, 23
Line numbers
example of, 107
restrictions, 107
RESUME statement, 108
use of, 106
Line printer, 250, 251, 341
Linefeed, 229, 242, 343, 345
$LINESIZE, 135
Link map file
described, 22, 32
producing, 22, 32
Link time, 423
Linker error messages, 26
Linking
bascom command, 17
special libraries, 31
$LIST, 134
LIST command, 143
LLIST command, 143
LOAD command, 143
LOC, 40
LOC function, 243
LOCK Statement, 65
LOCK statement, 245
LOCK...UNLOCK, 137
LOF function, 248
LOG function, 249
Logical operators, 121
Loops, 209, 261, 339
LPOS function, 250, 341
LPRINT statement, 251, 341
LPRINT USING statement, 251
LPT1:, 35, 248
LSET statement, 253, 303

Main program, defined, 73
MERGE command, 143
Metacommands, 132
See also Specific Metacommand
defined, 132
disabling, 133
$DYNAMIC, 130, 134
$INCLUDE, 133
$LINESIZE, 135
$LIST, 134
listing format, 135
$MODULE, 136
$OCODE, 135
$PAGE, 135
$PAGEIF, 135

Metacommands *(continued)*
$PAGESIZE, 135
purpose, 132
$STATIC, 130, 134
$SUBTITLE, 135
syntax, 133
$TITLE, 135
MID$ function, 255
MKD$ function, 257
MKDBCD$ function, 143
MKDIR statement, 259
MKI$ function, 257
MKS$ function, 257
MKSBCD$ function, 143
Modular programming, defined, 73
$MODULE metacommand, 136
Module, relocatable, 424
Modulo arithmetic, 119
Multi-user files, 65

NAME command, 260
NEW command, 143
New statements/functions
COMMAND$, 137
LBOUND/UBOUND, 137
LOCK...UNLOCK, 137
REDIM, 137
SHARED, 137
STATIC, 137
SUB...END SUB, 137
Newline, 240
NEXT, 261
Notational conventions, 6
Numeric constants, 110

Object file
creating, 20, 21
filename extension, 20
linking, 30
$OCODE, 135
OCT$ function, 262
Octal, 262
ON ERROR GOTO statement, 263
-E option required, 139
line 0, 107
ON ERROR GOTO...RESUME, -E option, 24
ON ERROR GOTO
-X option, 24
ON event GOSUB, -E option required, 139
ON event GOTO line 0, 107
ON GOSUB statement, 265
ON GOTO statement, 265
OPEN statement, 145, 204, 266
Operators, 116, 117
arithmetic, 118
functional, 124
logical, 121
relational, 120
string, 124
OPTION BASE statement, 270
Options
array storage, -R, 25
-D, 23, 24
Debug, 131
-E, 24
error handling See Error handling options. *See*
-R, 25
-X, 24
Output file, naming, 19
Overflow, 119, 203, 329

$PAGE, 135
$PAGEIF, 135
$PAGESIZE, 135
Parameter passing
by reference, 77
by value, 77
single module
SHARED attribute, 80
SHARED statement, 80, 82
Pass by value, 424
Pathnames defined, 37
PEEK function, 272, 273
PIPE:, 35, 195, 243, 248, 266
POKE statement, 272, 273
POS function, 274, 341
PRINT# statement, 275
PRINT statement, 278
PRINT USING statement, 281
PRINT# USING statement, 275
Program
converting to compiled, 137
executing, 19
execution, speeding up, 138
naming, 19
PUT statement, 204, 285

-R option, 25
Random files, 41, 204, 213, 233, 253, 257, 266, 285, 303
Random numbers, 287, 302
RANDOMIZE statement, 287, 302
READ statement, 289, 296
rebuild, 362, 387
  changing key information
    example, 388
  command line syntax, 389
  compressing data files, 388
  damaged key files, 388
  FIELD statement, 403
  KEYED BY Clause, 404
  redirect output, 401
  SPLIT KEYSET statement, 404
  switches
    -f, 389
    -i, 389
    -p, 389
    -s, 389
    -t, 389
Records, 40
REDIM statement, 137, 291
  arrays, use with, 130
  SHARED attribute, 80
Relational operators, 120
Relocatable, 424
REM statement, 294
RENUM command, 143, 160, 199
Reserved words, 349
RESTORE statement, 296
RESUME 0
  line 0, 107
  -X option, 139
RESUME NEXT statement, 108
RESUME NEXT
  -X option required, 139
RESUME statement, 297
  alphanumeric line labels, 108
  compiler/interpreter differences, 144, 145
  -X option required, 24, 139
RETURN statement, 215, 299
  check for GOSUB, 23
  not equivalent to EXIT statement, 324
  syntax, 216
RIGHT$ function, 300
RMDIR statement, 301
RND function, 287, 302
Routine, 424
RSET statement, 253, 303
RUN command, 304
RUN statement, 144, 304
Running a program
  See Executing a program
Run-time
  defined, 424
  error messages, 416

-S option, 24
SADD function, 306
SAVE command, 139, 143
Saving a program
  ASCII format, 139
SCRN:, 35, 248
Sequential files, 226, 233, 240, 266, 275, 343
SGN function, 307
SHARED statement, 137
SHARED Statement, 308
SHARED statement, form, 82
SHARED statement
  single module, 82
SHELL function, 69, 310
SHELL statement, 69
SIN function, 312
Single precision, 177, 187, 279
Single-precision numbers, 111
$SKIP, 135
Source file
  compiling, 19
  format, 139
  linking, 20
  multiple, compiling, 20
  name extension, 19
Source listing
  format metacommands, 135
  turning off, 134
SPACE$ function, 313
SPC function, 314
Special characters, 105
SQR function, 315
Statements enhanced in compiler
  COMMON, 145
  DEF FN, 145
  END, 145
  ERASE, 145
  FOR...NEXT, 145
  GOSUB...RETURN, 145

Statements enhanced in compiler *(continued)*
GOTO, 145
OPEN, 145
RESUME, 145
Statements
CALL, 144
CHAIN, 144
COMMON, 131, 144, 145
contrasted with functions, 149
DEF FN, 145
DEF type, 144
DEFDBL, 143
DEFINT, 143
DEFSNG, 143
DEFSTR, 143
DIM, 130, 144
END, 145
END SUB, 323
enhanced in compiler, 144
ERASE, 130, 145
executable, 107
execution
compiler, 138
interpreter, 138
FIELD, 109
FOR...NEXT, 145
GOSUB...RETURN, 145
GOTO, 108, 145
IF...THEN...ELSE, 108
interpreter
AUTO, 143
CONT, 143
DELETE, 143
EDIT, 143
LIST, 143
LLIST, 143
LOAD, 143
MERGE, 143
NEW, 143
RENUM, 143
SAVE, 143
LOCK, 137
new. *See* New statements/functions
nonexecutable, 107
not accepted by compiler, 143
ON ERROR GOTO, 107
OPEN, 145
prohibited, 143
REDIM, 130, 137
requiring modification in compiler, 143
Statements *(continued)*
RESUME 0, 107
RESUME, 144, 145
RESUME NEXT, 108
RUN, 144
SHARED, 137
STATIC, 137
SUB, 323
SUB...END SUB, 137
UNLOCK, 137
STATIC, 74
$STATIC, 134
Static arrays
See also Array
STATIC statement, 137, 316
$STATIC metacommand, 130, 134
STOP statement, 193, 215, 216, 320
STR$ function, 321
String constants, 110
String function, 237, 255, 310
STRING$ function, 322
String functions, 178, 231, 236, 300, 321, 336
String literal, 98
String operators, 124
String space, 212
String variables, 187, 240, 242
Strings
concatenation, 124
writing to disk, 24
Structured programs
See also Subprograms.
defined, 73
single module
passing parameters, 80
SHARED attribute, 80
SHARED statement, 80
SUB...END SUB statement, 137, 323
SUB statement, formal parameter list, 74
Subprograms, 323
argument, parameter list errors, 89
defined, 73
format, 73
GOSUB in, 86
invoking with CALL, 75
passing
arrays, 74
expressions, 79
prohibited expressions, 74
RETURN in, 86

Subprograms *(continued)*
variable aliasing errors, 89
variables, 76, 308, 316
Subroutines, 153, 158, 215, 216, 265, 299
Subscripts, 191, 270
$SUBTITLE, 135
SWAP statement, 326
Switches *See* Options
Syntax notation, 6
System maintenance mode, installing compiler in, 11
SYSTEM statement/command, 327

TAB function, 328
TAN function, 329
TIME$ function, 330
$TITLE, 135
TROFF, 23, 139, 331
TRON command, 23,139, 331
Type conversion, 125
Type mismatch error message, 126

UBOUND function, 79, 137, 333
Unbound, 425
UNLOCK statement, 137, 335
Unresolved external, 425
User-defined functions, 124
USR function, 143

VAL function, 336
Variables, 112
array, 173, 191
global
function definitions, 137, 316
subprograms, 137, 308, 316
local
function definitions, 137, 316
subprograms, 137, 308, 316
passing to subprograms
by reference, 76
single module, 80, 83
passing with COMMON, 160
reinitializing, 168
SHARED attribute, 80
SHARED statement, 83
string, 187, 240, 242
subprogram aliasing errors, 89
VARPTR function, 337

WAIT, 65
WEND statement, 339
WHILE statement, 339
WIDTH LPRINT statement, 341
WIDTH statement, 341
WRITE# statement, 343
WRITE statement, 345

-X option, 24

MICROSOFT®
16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem (Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken: