IBM RT PC BASIC Interpreter and Compiler

BASIC Language Handbook

Programming Family

Personal
Computer
Software

59X9289

International Business Machines Corporation Armonk, New York 10504

IBM Program License Agreement

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THIS PACKAGE. OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED AND YOUR MONEY WILL BE REFUNDED.

IBM provides this program and licenses its use in the United States and Puerto Rico. Title to the media on which this copy of the program is recorded and to the enclosed copy of the documentation is transferred to you, but title to the copy of the program is retained by IBM or its supplier, as applicable. You assume responsibility for the selection of the program to achieve your intended results, and for the installation, use and results obtained from the program.

LICENSE

You may:

a. use the program on only one machine at any one time except as otherwise specified by IBM in the enclosed Program Specifications (available for your inspection prior to your acceptance of this Agreement);

b. copy the program into machine readable or printed form for backup or modification purposes only in support of such use. (Certain programs, however, may include mechanisms to limit or inhibit copying. They are marked "copy protected");

c. modify the program and/or merge it into another program for your use on the single machine. (Any portion of this program merged into another program will continue to be subject to the terms and conditions of this Agreement.); and,

d. transfer the program with a copy of this Agreement to another party only if the other party agrees to accept from IBM the terms and conditions of this Agreement. If you transfer the program, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copies not transferred; this includes all modifications and portions of the program contained or merged into other programs. IBM will grant a license to such other party under this Agreement and the other party will accept such license by its initial use of the program. If you transfer possession of any copy, modification or merged portion of the program, in whole or in part, to another party, your license is automatically terminated.

You must reproduce and include the copyright notice on any copy, modification, or portion merged into another program.

You may not reverse assemble or reverse compile the program without IBM's prior written consent.

You may not use, copy, modify, or transfer the program, or any copy, modification or merged portion, in whole or in part, except as expressly provided for in this Agreement.

You may not sublicense, assign, rent or lease this program.

TERM

The license is effective until terminated. You may terminate it at any other time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the program together with all copies, modifications and merged portions in any form.

LIMITED WARRANTY AND DISCLAIMER OF WARRANTY

IBM warrants the media on which the program is furnished to be free from defects in materials and workmanship under normal use for a period of 90 days from the date of IBM's delivery to you as evidenced by a copy of your receipt.

IBM warrants that each program which is designated by IBM as warranted in its Program Specifications, supplied with the program, will conform to such specifications provided that the program is properly used on the IBM machine for which it was designed. If you believe that there is a defect in a warranted program such that it does not meet its specifications, you must notify IBM within the warranty period set forth in the Program Specifications.

ALL OTHER PROGRAMS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT IBM OR AN IBM AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IBM does not warrant that the functions contained in any program will meet your requirements or that the operation of the program will be uninterrupted or error free or that all program defects will be corrected.

THE FOREGOING WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

LIMITATIONS OF REMEDIES

IBM's entire liability and your exclusive remedy shall be as follows:

1. With respect to defective media during the warranty period:
   a. IBM will replace media not meeting IBM's "Limited Warranty" which is returned to IBM or an IBM authorized representative with a copy of your receipt.
   b. In the alternative, if IBM or such IBM authorized representative is unable to deliver replacement media which is free of defects in materials and workmanship, you may terminate this Agreement by returning the program and your money will be refunded.
2. With respect to warranted programs, in all situations involving performance or nonperformance during the warranty period, your remedy is (a) the correction by IBM of program defects, or (b) if, after repeated efforts, IBM is unable to make the program operate as warranted, you shall be entitled to a refund of the money paid or to recover actual damages to the limits set forth in this section.

   For any other claim concerning performance or nonperformance by IBM pursuant to, or in any other way related to, the warranted programs under this Agreement, you shall be entitled to recover actual damages to the limits set forth in this section.

IBM's liability to you for actual damages for any cause whatsoever, and regardless of the form of action, shall be limited to the greater of $5,000 or the money paid for the program that caused the damages or that is the subject matter of, or is directly related to, the cause of action.

In no event will IBM be liable to you for any lost profits, lost savings or other incidental or consequential damages arising out of the use of or inability to use such program even if IBM or an IBM authorized representative has been advised of the possibility of such damages, or for any claim by any other party.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

SERVICE

Service from IBM, if any, will be described in Program Specifications or in the statement of service, supplied with the program, if there are no Program Specifications.

IBM may also offer separate services under separate agreement for a fee.

GENERAL

Any attempt to sublicense, assign, rent or lease, or, except as expressly provided for in this Agreement, to transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be construed under the Uniform Commercial Code of the State of New York.

Program Specification

IBM RT Personal Computer BASIC Interpreter and Compiler Licensed Program Product (55X8905)

Statement of Limited Warranty

The IBM RT PC BASIC Interpreter and Compiler Licensed Program Product is warranted to conform to this Program Specification when properly used in its designated operating environments.

Any other documentation with respect to this licensed program is provided for information purposes only and does not extend or modify this IBM RT PC BASIC Interpreter and Compiler Licensed Program Product Program Specification.

The IBM RT PC BASIC Interpreter and Compiler Licensed Program Product Program Specification may be updated from time to time. Such updates may constitute a change to these specifications.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

This limited warranty and the 90-day program media warranty are contained in the IBM Program License Agreement supplied with this product. These warranties are available to all licensees of the IBM RT PC BASIC Interpreter and Compiler Licensed Program Product. The limited warranty period is until February 1, 1988, or until six months after written notice by IBM that the warranty period has been terminated, whichever is sooner.

Statement of Function Warranted

The IBM RT Personal Computer BASIC Interpreter and Compiler Licensed Program Product (55X8905) provides a level of function comparable to the IBM PC BASIC 1.1 Interpreter, Advanced Version Licensed Program Product (IBM PC BASIC), plus extensions. The user can select the "PC Mode," which provides similarity to IBM PC BASIC, or the "Native Mode," which provides additional IBM RT PC capabilities.

The highlights of this licensed program are:

- Includes both a Compiler and an Interpreter
- Support for both the IBM PC mode and the IBM RT PC Native mode
  - PC mode provides similarity to IBM PC BASIC
  - Native mode provides additional capability such as IEEE format floating-point numbers, 4-byte integers, AIX Operating System standard interfaces, and programs larger than 64K.
- Use of both the Compiler and Interpreter is facilitated by common syntax and semantics
- IBM RT PC AIX Operating System Licensed Program Product shell commands can be invoked in both PC and Native modes
- Character and graphics modes supported for IBM RT PC local work stations
- Character mode supported for ASCII work stations that are supported by the IBM RT PC AIX Operating System

- In native mode, support for subroutines written in the C language provided with the IBM RT PC AIX Operating System Licensed Program Product (55X8994).

Specified Operating Environment

Machine Requirements

The minimum machine requirement is an IBM RT Personal Computer with a display (for example, the IBM RT Personal Computer Advanced Color Graphics Display, IBM RT Personal Computer Advanced Monochrome Graphics Display, the IBM Personal Computer Display, or equivalent display).

Notes:

1. The number of users on the IBM RT PC AIX Operating System Licensed Program Product (55X8994), the number and type of tasks, and the application requirements may expand the requirements beyond these minimums.
2. The IBM RT PC Floating-Point Accelerator, designed to improve performance of floating-point operations, may be optionally installed.

Programming Requirements

The IBM RT PC AIX Operating System Licensed Program Product (55X8994) is a prerequisite for program execution.

Statement of Service

Program service for valid program-related defects in the IBM RT PC BASIC Interpreter and Compiler Licensed Program Product is available to all IBM RT PC BASIC Interpreter and Compiler Licensed Program Product licensees, at no additional charge, until February 1, 1988, or until six months after written notice by IBM that the warranty period has been terminated, whichever is sooner. However, service will be provided only for the current update level and for the prior release for ninety (90) days following release of the current level of update.

Each licensee's access to program service is determined by the marketing channel through which the license was obtained. For example, in the United States and Puerto Rico, if the IBM RT PC BASIC Interpreter and Compiler Licensed Program Product license was obtained through:

- An authorized IBM personal computer dealer.

  Requests for program service should be made through your dealer.

- The IBM North-Central Marketing Division or the IBM South-West Marketing Division.

  Your company will have established a technical support location to interface to IBM central service through an IBM Support Center, and your request for program service should be made through your company's technical support location.

If the IBM RT PC BASIC Interpreter and Compiler Licensed Program Product is obtained through transfer of license from another party under the conditions of the IBM Program License Agreement supplied with this product, the new licensee may obtain program service through the access arrangement provided for the original licensee.

When a license is transferred, if the original license was obtained through the IBM North-Central Marketing Division or the IBM South-West Marketing Division, the old licensee is responsible for contacting their IBM marketing representative to make arrangements to transfer service entitlement to the new licensee; the new licensee must also establish a qualified technical support location to interface to IBM central service.

IBM does not guarantee service results or that the program will be error free, or that all program defects will be corrected.

IBM will respond to a reported defect in an unaltered portion of a supported release of the licensed program by issuing: defect correction information such as correction documentation,

corrected code, or notice of availability of corrected code; a restriction; or a bypass.

Corrected code is provided on a cumulative basis on diskettes; no source code is provided. Only one copy of the corrections with supporting documentation will be issued to the licensee, or the agent of the licensee, reporting the defect. IBM will authorize various agents such as the IBM Personal Computer dealers and the IBM North-Central Marketing Division or IBM South-West Marketing Division customer's technical support locations to make and distribute a copy of the corrections if needed, to each IBM RT PC BASIC Interpreter and Compiler Licensed Program Product licensee which they serve.

IBM will notify authorized IBM Personal Computer dealers, IBM marketing and service representatives, and IBM North-Central Marketing Division and IBM South-West Marketing Division customer's technical support locations if and when an update is made available. Program updates contain all currently available changes for the licensed program.

Licensees may request available updates to this licensed program, if any, prior to the program service termination date. As with defect corrections, IBM will authorize various agents such as IBM Personal Computer dealers and the IBM North-Central Marketing Division and IBM South-West Marketing Division customer's technical support locations to make and distribute a copy of the update, if needed, to each IBM RT PC BASIC Interpreter and Compiler Licensed Program Product licensee which they serve.

The total number of copies of an update distributed to IBM RT PC BASIC Interpreter and Compiler Licensed Program Product licensees within a customer's location may not exceed the number of copies of the IBM RT PC BASIC Interpreter and Compiler Licensed Program Product licensed to the customer.

IBM does not plan to release updates of IBM RT PC BASIC Interpreter and Compiler Licensed Program Product code on a routine basis for preventative service purposes. However, should IBM determine that there is a general need for a preventative service update, it will be made available to all licensees through the same process that is utilized to distribute general IBM RT PC BASIC Interpreter and Compiler Licensed Program Product updates, as described above.

Following the discontinuance of all program services, this program will be distributed on an "As Is" basis without warranty of any kind either express or implied.

January 1986 Printed in U.S.A.
(GC23-0897-0)

IBM Corporation
Engineering Systems Products
472 Wheelers Farms Road
Milford, Connecticut 06460

IBM RT PC BASIC Interpreter and Compiler

BASIC Language Handbook

Programming Family

Personal
Computer
Software

First Edition (AUGUST 1985)

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

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

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Book

IBM RT PC (Personal Computer) BASIC offers both the BASIC Interpreter and the BASIC Compiler in one licensed product. You can use the Interpreter and Compiler in two *Modes:* PC Mode and Native Mode. See Chapter 1, "The Environments of BASIC," for more information about PC Mode and Native Mode.

This *IBM RT PC BASIC Language Handbook* (cited as the *BASIC Handbook*) and the companion volume, the *IBM RT PC BASIC Language Reference*, (cited as the *BASIC Reference*) are references for IBM RT PC BASIC.

This manual describes IBM RT PC BASIC in PC Mode. The Native Mode is identical to PC Mode except where noted in this manual. This manual describes the Interpreter as well as the Compiler. All descriptions in this manual apply equally to the Compiler BASIC and the Interpreter BASIC unless otherwise specified. The name BASIC, as used in this manual, refers to IBM RT PC BASIC.

This book contains general information about using BASIC. Chapters 2, 3, 4, and 5 help you get started using BASIC. Chapter 6 contains information about graphics. The appendixes contain information on advanced subjects for the experienced programmer. Appendix F describes how to install BASIC.

The *BASIC Reference* is an encyclopedia-type manual. It contains, in alphabetic order, the syntax and semantics of commands, statements, and functions in BASIC. It also contains appendixes for error messages and special codes.

The two BASIC books are indexed and cross-referenced.

It is important for you to know that this book is written *only as a reference* for the BASIC programming language, not as a textbook that teaches you how to program. If you need step-by-step instruction to program in BASIC, we suggest that you ask for introductory materials at your library, bookstore, or computer store.

Related Publications

The following books contain related information that you might find useful.

IBM RT PC Guide to Operations

This guide describes procedures for operating the hardware. It includes information on the system unit, the display, the keyboard, and other devices that can be attached.

IBM RT PC Problem Determination Guide

This guide describes procedures for locating, identifying, and correcting problems with the system. It includes information on running the hardware diagnostic utilities to gather information on software problems. Included is a high-capacity (1.2MB) diskette containing the system diagnostic routines.

IBM RT PC Messages Reference

This manual lists the errors you may see on your display, along with how to respond to the messages.

IBM RT PC Using and Managing the AIX Operation System

This manual contains information on using the AIX Operating System commands, working with the file system, developing Shell procedures, and performing such system management tasks as creating and mounting file systems, backing up the system, and repairing file system damage.

IBM RT PC AIX Operating System: Commands Reference

This manual describes the AIX Operating System commands, including the following:

- The proper syntax for each command with the acceptable flags and arguments
- Examples on proper usage

IBM RT PC AIX Operating System: Technical Reference

This manual gives details about the AIX Operating System, the file system, files, special files, miscellaneous files, and writing device drivers.

IBM RT PC BASIC Language Reference

This book is an encyclopedia-type manual. It contains, in alphabetical order, the syntax and format of every command, statement, and function in BASIC.

Ordering Additional Copies of This Book

To order additional copies of this publication (without program diskettes), use either of the following sources:

- To order from your IBM representative, use Order Number SV21-8019.
- To order from your IBM dealer, use Part Number 55X8907.

A binder is included with the order.

Contents

Chapter 1. The Environments of BASIC

CONTENTS

An Overview of RT PC BASIC

IBM RT PC BASIC offers both the BASIC Interpreter and the BASIC Compiler in one licensed product. You can use the Interpreter and the Compiler in each of two *Modes:* PC Mode and Native Mode. Thus, four different environments are supported, as seen below:

IBM RT PC BASIC			
PC MODE		NATIVE MODE	
Interpreter Environment	Compiler Environment	Interpreter Environment	Compiler Environment

Programs written for one mode might not run in the other mode without modification. You should keep a record of the mode intended for each program. This can be done in a comment statement early in the program.

When you write a BASIC program you choose either PC Mode or Native Mode. Within either Mode, you can write the program for the Interpreter environment or the Compiler environment.

Both Modes of BASIC support the following features:

- Extended character set. You can display 256 different characters: the usual letters, numbers, and symbols, plus international characters and other symbols such as Greek letters that are used in scientific and mathematical applications.
- Graphics capability. You can draw points, lines, objects, and entire pictures. The screen is *all points addressable* in the graphics environments.
- Special input/output device support. You can use sound and locator devices to make your programs more interesting and more productive.

The *BASIC Reference* describes in detail the commands, statements, and functions of BASIC. This manual describes BASIC in PC Mode. The Native Mode is identical to PC Mode except where noted in this manual. This manual also describes the Interpreter as well as the Compiler for BASIC. Descriptions in this manual apply equally to the Compiler and Interpreter environments unless otherwise specified.

When you install BASIC, the installation process loads a number of files into the system. One of these files (**/usr/basic/basic.info**) contains a list identifying the files in the BASIC Interpreter and Compiler. This file may also contain other information about the product.

PC Mode

PC Mode is similar to Advanced BASIC (BASICA) on the IBM Personal Computer. Numbers are represented in the same way as in BASICA, with 16-bit integers and PC BASIC floating point format.

In PC Mode, a graphics terminal is made to look like the IBM Personal Computer graphics display with two graphics environments and a screen buffer at an apparent segment address of B8000 hex. The user's address space is limited to 64K bytes.

Native Mode

Native Mode BASIC is designed to let a programmer use the power of the IBM RT PC computer and operating system.

Arithmetic in Native Mode differs from PC Mode. Integers are 32 bits, giving an integer range of -2^{31} to $2^{31}-1$. Integer expressions are evaluated using integer calculations. This provides some performance benefits compared to converting to floating point form, but the possibility of overflow exists. (Overflow is trappable by the programmer.) Floating point uses the IEEE standard formats for increased dynamic range in double precision and allowing optional floating-point hardware to be used without changing the programming interface.

The user's address space is limited only by the virtual address space of the computer. As a result, you can manipulate very large quantities of data.

Interpreter

The Interpreter translates a program from BASIC into an intermediate code. When you enter a new line of BASIC code, the Interpreter immediately translates that line of code into the same intermediate code. During this translation, the Interpreter checks for syntax errors. If there is an error, the Interpreter lets you correct it before going on to the next line.

After the Interpreter has translated the entire program and there are no syntax errors, you can RUN the program. When you type the command RUN, the BASIC Interpreter reads the intermediate code, and carries out the instructions one line at a time.

An interpreted program is usually faster to write and debug than a compiled program.

Compiler

The BASIC Compiler translates an entire BASIC program at one time and creates a new file called an object file. The object file contains machine code. All translation is performed before you actually run your program.

A compiled program usually runs faster than an interpreted program.

Chapter 2. How to Use the BASIC Interpreter

CONTENTS

Introduction

This chapter explains how to use the BASIC Interpreter. First, there is a description of how to get the Interpreter started and how to stop it. Then you are shown how to give commands to BASIC and how to load and run a program called **samples.bas**. Finally, you can read about more advanced features such as command line options. Actual use of the Editor (how to type in and change a BASIC program) is explained in Chapter 3.

Note: For the sake of clarity, many of the explanations given in this chapter are brief. For more detailed information about commands such as SYSTEM, RUN, SAVE, LIST and LOAD, refer to their entries in the *BASIC Reference.*

Logging On to the System

To use the BASIC Interpreter, you must first log on to the system by following the instructions in the *IBM RT PC Using and Managing the AIX Operating System* manual.

Note: BASIC must be installed on your system. Appendix F contains instructions on how to install BASIC.

Starting the Interpreter

To start the BASIC Interpreter enter one of the following:

PC Mode: `$ basic`
Native Mode: `$ basicn`

Returning from BASIC to the Operating System

When you are done using the BASIC Interpreter, you can return to the operating system by entering the command SYSTEM. The operating system prompt ($) then appears on your screen. For example,

```
Ok SYSTEM
$
```

Note: When BASIC encounters the statement SYSTEM in a program, it stops the program and returns you directly to the operating system.

Executing Commands and Programs

After the BASIC Interpreter starts, it displays the **Ok** prompt. You are at the *command level* and BASIC is ready to receive instructions from you. You can give instructions using either the *direct method* or the *indirect method.* You can only issue commands using the *direct method.*

Direct Method

When you use the *direct method*, BASIC executes a command immediately after it is entered (that is, just after you have typed the command and pressed the **Enter** key). In the direct method you leave out the line number from the statement. With no line number, the instructions are not saved after they are performed. However, the direct method allows you to issue BASIC commands such as RUN, LOAD, SAVE, REPLACE, and RENUM. You can also quickly determine the results of arithmetic and logical operations, store them for later use, and look at the values currently assigned to variables in memory. This is very useful for writing and debugging programs, as well as for quick computations that do not require a complete program. For example,

```
Ok PRINT 20 + 2
 22
Ok PRINT X
 0
Ok
```

Indirect Method

You use the *indirect method* to enter program lines into a BASIC program in memory. A line must begin with a line number in order for it to become part of the program. The program does not start until you enter the command RUN. For example,

```
Ok 10 X = 20 + 2
Ok 20 PRINT X  Ok RUN
 22
Ok
```

To learn more about how to type in and change programs, see Chapter 3, "How to Use the BASIC Editor."

Working With Complete Programs

You can save a program in memory to a file by using the SAVE or REPLACE command:

```
Ok SAVE "myprogram"
Ok
```

To load a program from a file into memory, use the LOAD command:

```
Ok LOAD "myprogram"
Ok
```

Note: All programs loaded into the BASIC Interpreter must be in ASCII format only. Programs written using BASICA must be SAVEd with the ,A option.

To see what instructions are in the program, use the LIST command:

```
Ok LIST
   10 X = 20 + 2
   20 PRINT X
Ok
```

Finally, to run the program now in memory, use the RUN command:

```
Ok RUN
 22
Ok
```

Note: You can load and run a program with one command by giving the file name with the RUN command:

```
Ok RUN "myprogram"
```

Running the SAMPLES Program

This section shows you how to start BASIC, and load in and run an example program called **samples**. To run the program:

1. At the operating system level, type

   ```
   $ cd /usr/basic
   ```

   and press **Enter**. The BASIC demonstration programs are installed in **/usr/basic**. Next, type

   ```
   $ basicn
   ```

   and press **Enter**. This loads Native Mode BASIC into the computer's memory. You will see the BASIC copyright, the BASIC prompt (**Ok**), and, across the bottom of the screen, the ten BASIC function keys.

2. Type

   ```
   OK run "samples"
   ```

   and press **Enter**. This loads the program and runs it.

3. When the **samples** title screen is displayed, press the space bar. You will then see the menu screen.

4. Select the item you want from the menu screen. Note the remarks next to the menu items. They tell you what you need to run the program you select: whether you need a graphics display and the amount of memory you need to run these sample programs.

5. Try a program such as H, COLORBAR. Type

   ```
   h
   ```

   You do not have to press **Enter**. If you have a monochrome display, you will see two rows of bars in different shades of one color. If you have a color display, you will see bars of various colors. Adjust the monitor controls as necessary until you achieve a pleasing and appropriate display.

6. After you have entered your program selection, follow the instructions on the screen. Press the **Esc** key to stop a program and return to the menu.

7. When you have tried all the programs you want to see and have returned to the menu, press the **Esc** key. This leaves the program and returns you to the command level **Ok** prompt.

Interpreter Command Line Format

You can include options in the BASIC command line for the Interpreter when you start BASIC. These options specify the amount of storage you want BASIC to use for programs, data, and buffer areas. You can also tell the Interpreter to immediately load and run a program. You can use the Interpreter without the options.

The format of the BASIC command for the Interpreter is shown below:

```
basic[n] [<stdin] [>[>]stdout] [-r progname]
         [-p name1 [name2 [name3]]] [-w workspace]
         [-s sourcespace]
```

In this syntax, the following conventions are used:

- You must type exactly all words that are not in italics.
- Lowercase *italic* letters stand for names or numbers that you must supply.

- Items in square brackets ([ ]) are optional.
- All punctuation except square brackets (such as the greater-than symbol (>), the less-than symbol (<), and hyphens (-)) must be included where shown.
- The command line must start with basic[n], but you may place the options in any order.
- All options must be separated by at least one space.

An explanation of the options follows.

n — This option starts Native Mode BASIC. If you do not use the n option, the PC Mode Interpreter starts running.

<stdin — *Stdin* is a pathname. A BASIC program normally receives its input from the keyboard (standard input device). Using *<stdin* allows BASIC to receive input from a file instead of the keyboard. Refer to "Redirection of Standard Input and Output" for more information.

>[>]stdout — *Stdout* is a pathname. A BASIC program normally writes its output to the screen (standard output device). Using *>stdout* causes BASIC to write output to the file designated by *stdout* instead of to the screen. If the file already exists, BASIC writes over the file. Using *>>stdout* causes BASIC to append its output to the file you specify. You cannot use graphics or sound statements if output has been redirected to a file. Refer to "Redirection of Standard Input and Output" for more information.

-r progname This option tells BASIC to immediately load and run the program named in *progname*. BASIC will proceed as if you had given it a RUN *progname* command. *Progname* is a pathname and it must follow the rules for naming files described under "Files" in Chapter 5. The program in the file must be in standard ASCII format. Note that when you specify -r *progname*, the program starts immediately, the BASIC heading with the copyright notices does not appear.

Note: You can run BASIC programs as a batch process by putting the basic command line in a shell procedure. If you do so, your program should finish with the SYSTEM statement so that the shell procedure can perform its next command. Also, when BASIC encounters a STOP, END, or untrapped error, it returns to the operating system.

-p name1 [name2 [name3]]
The -p option designates the output files for LPT1:, LPT2:, and LPT3:. They are associated with *name1*, *name2*, and *name3*, respectively.

For example, the LLIST command and the LPRINT statement write to LPT1:, and the default file for LPT1: is **lpt1.lst**. This means that when you use LLIST or LPRINT in BASIC, your output goes (by default) to **lpt1.lst**. If, however, you use the -p *name1* option when you start BASIC, BASIC sends LPT1: output to the file given as *name1* and not to **lpt1.lst**.

Like LPT1:, using LPT2: and LPT3: in your program is optional. However, LPT2: and LPT3: have no default files. If your program uses LPT2: and LPT3:, you must specify files (*name2* and *name3*) for them.

-w workspace *Workspace* is an integer that specifies the size (in K bytes) of the user area reserved for BASIC program variables, strings, and arrays. One K byte is equal to 1024 bytes. For example, -w 64 reserves 64K bytes of workspace. The default workspace size is 64K bytes. In PC Mode, the maximum workspace size you can reserve is 64K bytes; if you specify more than 64K, the default value of 64K is used.

-s sourcespace *Sourcespace* is an integer that specifies the size (in K bytes) of the area reserved by the translator for program text, variable names, and the intermediate code generated by the translator. This space is separate from the user workspace.

The default size for *sourcespace* is 256K bytes. The minimum size for *sourcespace* is 32K bytes, and if you specify less than 32K bytes, BASIC allocates 32K bytes.

Command Line Examples

Some examples of the BASIC command line for the Interpreter:

```
$ basic -r payroll
```

This example uses 64K of memory in PC Mode; it loads and starts **payroll.bas** running.

```
$ basicn -r invent -w 512 -p invent.spool
```

This example starts the Native Mode Interpreter with 512K bytes of workspace; it loads and runs **invent.bas**, with LPRINT output sent to the file **invent.spool**.

Interpreter Debugging Commands

The Interpreter contains commands very useful for debugging a program. Some of them are as follows:

BREAK	UNBREAK
FOLLOW	UNFOLLOW
TRACE	UNTRACE
TRON	TROFF

These commands are described in the *BASIC Reference.*

Using a Shell Procedure

Shell procedures provide a convenient way to reuse options that you use often. For example, to invoke the BASIC Native Mode Interpreter, run a program, and print the output requires the following command lines:

```
basicn -r payroll -p payroll.spl cheques.spl -w 512
print payroll.spl
```

These command lines could be put in a text file called **payroll** and given execution status with

```
chmod +x payroll
```

Then, the single command line

```
payroll
```

can be used to execute all the steps in the shell procedure file.

Recovering from Errors

If the BASIC Interpreter or a BASIC program terminates abnormally and returns to the system prompt ($), you may be able to recover the keyboard/screen environment that existed while the program was running. There are two ways you can attempt recovery:

1. Enter the following operating system command.

   ```
   $ stty sane
   ```

2. Invoke and then leave the Interpreter.

   ```
   $ basic(n)
   OK system
   ```

   (OK is the Interpreter prompt.)

Chapter 3. How to Use the BASIC Editor

CONTENTS

Introduction

This chapter describes the BASIC Editor, which is the primary interface between you and the BASIC Interpreter or Compiler. The BASIC Editor allows you to create new program text and change or delete existing program text.

First, this chapter describes the general characteristics of the console keyboard, and then describes how to use the BASIC Editor to manipulate program text.

The Console Keyboard

The console keyboard is divided into four areas:

- The function keys and other special keys are on the top row of the keyboard.
- The typewriter area is in the middle. This is where you find the regular letter and number keys.
- The four arrow keys for moving the cursor are to the right of the typewriter area.
- The numeric keypad, similar to a calculator keypad, is on the far right side of the keyboard.

The console keyboard is shown on the following page. General use of the keyboard is described in the *IBM RT PC Guide to Operations*. The following paragraphs describe specific keyboard use for BASIC.

Esc
F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
Print Screen
Scroll Lock
Pause
Num Lock
Caps Lock
Scroll Lock
~ `
! 1
@ 2
# 3
$ 4
% 5
^ 6
& 7
* 8
(9
) 0
_ -
+ =
← Backspace
Insert
Home
Page Up
Num Lock
/
*
-
Tab
Q
W
E
R
T
Y
U
I
O
P
{ [
}]
| \
Delete
End
Page Down
7
8
9
+
Caps Lock
A
S
D
F
G
H
J
K
L
: ;
" '
← Enter
4
5
6
Shift
Z
X
C
V
B
N
M
< ,
> .
? /
Shift
1
2
3
Enter
Ctrl
Alt
Alt
Action
0
.

Function Keys

The first ten function keys labeled **F1** through **F10**, can be used as follows:

- As typing keys. You can set each key to automatically type any sequence of up to 15 characters. Some frequently used commands have already been pre-assigned to these keys. You can use the KEY statement to change them if you wish.
- As program interrupts, through use of the ON KEY statement.

Typewriter Keys

The typewriter area of the keyboard behaves much like a standard keyboard. The **shift**, **caps lock**, and **tab** key work like they do on a typewriter.

The **Enter** key is on the right side of the typewriter area. You use this key in BASIC to enter information or a command. Pressing the **Enter** key tells BASIC you have finished typing.

The Arrow Keys

The *cursor* is the little blinking line or box on the screen. The four arrow keys just to the right of the typewriter area move the cursor around the screen. The section on "Special Program Editor Keys" will describe how to do this.

The Numeric Keypad

You can enter numbers using the numbered keys either on the top row of the typewriter area, or on the keypad on the right side of the keyboard.

Special Key Combinations

The section that follows describes some key combinations that are available in BASIC.

Alt Key

The **Alt** (Alternate) keys are on both sides of the space bar. You can use them to type an entire BASIC keyword with a single keystroke.

To do this, hold down one of the **Alt** keys while pressing one of the alphabetic keys, **a–z**. Keywords associated with the letters are listed below. Letters that do not have reserved words associated with them are noted by "(no word)."

A	AUTO	**N**	NEXT
B	BSAVE	**O**	OPEN
C	COLOR	**P**	PRINT
D	DELETE	**Q**	(no word)
E	ELSE	**R**	(no word)
F	FOR	**S**	SCREEN
G	GOTO	**T**	THEN
H	HEX$	**U**	USING
I	INPUT	**V**	VAL
J	(no word)	**W**	WIDTH
K	KEY	**X**	XOR
L	LOCATE	**Y**	(no word)
M	(no word)	**Z**	(no word)

You can also use the **Alt** key with keys on the numeric keypad to enter characters not found on the keys. To do this, hold down the **Alt** key and type the three-digit ASCII code for the character. For example, typing 168 on the numeric key pad while holding down the **Alt** key prints an upside-down question mark on the screen. See Appendix B in the *BASIC Reference* for a complete list of ASCII codes.

Ctrl Key

The **Ctrl** key is also used to enter certain codes and characters not otherwise available from the keyboard.

Ctrl-A: Ctrl-A followed by a–z has the same effect as **Alt** (a–z) described in the **Alt** key section.

Ctrl-B: Ctrl-B interrupts the program at the next BASIC instruction and returns to BASIC command level. It is also used to turn off AUTO line numbering. To use **Ctrl-B**, hold down the **Ctrl** key while pressing the **B** key.

Ctrl-G: Ctrl-G is the bell character. When this character is printed, the speaker beeps. To enter the bell character, hold down the **Ctrl** key and press the **G** key.

You also use the **Ctrl** key together with other keys when you edit programs with the BASIC Program Editor. These operations are explained in the section that follows. The numeric keypad keys and the **Alt** key cannot be used with the following values to produce special characters: 0 to 31, 130, and 160 to 164.

The BASIC Program Editor

Any line of typed text is processed by the BASIC Program Editor. There are two types of editing, command-level editing and full-screen editing.

Note: It is important to make a distinction between a screen line and a BASIC line. A *screen line* in the Editor is one horizontal row of 39 or 79 character positions on the screen. A *BASIC line* is a line of BASIC code. It may be a line in a program, or a direct method command. It may run several screen lines long. For example,

```
20 X$ = "This is an example"; _
      + "of one program line"; _
      + "on three screen lines"
```

Command-level Editing

You can also think of this as single-line editing. With command-level editing, you enter and edit a line of BASIC freely within a single screen line. The line is not completed until you press the **Enter** key. With command-level editing, you cannot move the cursor up or down on the screen. You are at the command level when you see the **Ok** prompt.

Full-screen Editing

With the full-screen editor, you can enter text anywhere on the screen. You must use the EDIT command to start full-screen editing. You can do this by pressing the **F10** function key (provided that you have not changed F10's meaning from "EDIT ." using the KEY statement).

You normally leave full-screen editing with the CONT command. This takes you back to the command level (**Ok**). However, using any of the following commands also takes you out of full-screen editing:

CONT	RUN
LOAD, R	STEP

Issuing any other BASIC command does not take you out of full-screen editing.

Any direct method BASIC statement (those without initial line numbers) also returns you to the command level (**Ok**).

In full-screen editing, a new or changed line of text is not read by the Interpreter until after you have pressed the **Enter** key somewhere on that line.

If that line of text is a line-numbered BASIC statement, the line becomes a part of your BASIC program when you press the **Enter** key. Error checking occurs when you type RUN. If the line of text does not have a line number with it, the Interpreter tries to perform it as a command immediately. If the line does not contain a valid command or statement, an error message appears immediately.

A command or single BASIC statement may run several screen lines long. Pressing the **Enter** key with the cursor anywhere on any of the screen lines of the command or statement tells the Interpreter to read the entire command or statement.

During full-screen editing, you can issue the EDIT command again. This keeps you in full-screen editing, clears the screen, moves the cursor to the top of the screen, and lists the lines designated by your EDIT command. You can also use the LIST command to display program lines for changing. With LIST, the listing starts at the current cursor position, not at the top of the screen.

To become familiar with the capabilities of the Program Editor, type in a sample program and practice using the edit keys as described on the pages that follow.

Special Program Editor Keys

You can use cursor control keys, the **Backspace** key, and the **Ctrl** key to move the cursor on the screen, to insert characters, or to delete characters.

There are two kinds of Program Editor keys. You can use all of the keys during full-screen editing. However, command-level editing does not allow you to move the cursor up and down on the screen; cursor movement is restricted to the current line.

In the following list, two key combinations are shown for some functions (such as **Ctrl-Home** or **Ctrl-O**). Some keyboards do not support both combinations.

Keys	Function
Home	Moves the cursor to the upper left-hand corner of the screen. (Full-screen editing only.)
Ctrl-Home **or** **Ctrl-O**	Clears the screen and positions the or cursor in the original postion. (Full-screen editing only.)
Cursor Up (↑)	Moves the cursor **up** one line. (Full-screen editing only.)
Cursor Down (↓)	Moves the cursor **down** one line. (Full-screen editing only.)
Cursor Left (←)	Moves the cursor **left** one character position. During full-screen editing, if the cursor moves beyond the left edge of the screen, it wraps around to the right side of the screen on the line above.
Cursor Right (→)	Moves the cursor **right** one position. During full-screen editing, if the cursor moves beyond the right edge of the screen, it wraps around to the left side of the screen on the next line down.
Next Word (Ctrl-→) **or** **(Ctrl-R)**	Moves the cursor right to the beginning of the next word. A *word* is or any continuous string of alphanumeric characters (letters and digits). During full-screen editing, if the next word is at the beginning of the next line down, then the cursor moves down to the beginning of that word.

Previous Word (Ctrl-←) or (Ctrl-L)	Moves the cursor left to the beginning of the previous word. During full-screen editing, if the previous word is at the end of the next line up, the cursor moves up to the beginning of that word.
End	Moves the cursor to the end of the BASIC line.
Ctrl-End or Ctrl-E	Erases characters from the current cursor position to the **end** of the BASIC line.
Ins	Switches Insert Mode on and off. When Insert Mode is on, the cursor forms a box covering the lower half of a character position. Insert Mode allows you to type in new characters without erasing characters already on the line. It pushes towards the end of the line characters that are at and to the right of the current cursor position, and inserts a newly-typed character into the empty space. When Insert Mode is off, any characters typed replace the existing characters at the current cursor position. Initially Insert Mode is off. **Note:** You also turn off Insert Mode when you press any of the cursor movement keys or the **Enter** key.
Del	Deletes the character at the current cursor position. All characters to the right of the deleted character move one position left to fill in the empty space. Line wrapping can occur.

Backspace	Deletes the character to the left of the cursor. All characters to the right of the deleted character move left one position to fill in the space. Line wrapping can occur.
Ctrl-Delete or **Ctrl-F**	When pressed anywhere on a BASIC line, **Ctrl-Delete** erases (blank fills) the entire BASIC line from the screen. The resulting blank screen lines stay on the screen. The cursor goes to the beginning of the blanked line. The erased line is not passed to BASIC for processing. However, if it is an existing program line (begins with a line number), it is not deleted from the program in memory.
Ctrl-B	This key combination provides the **break** function. If you press **Ctrl-B** during command-level editing, BASIC ignores changes made to the line, moves the cursor to the next screen line, and waits for a new command. BASIC does not save any changes made to the line that the cursor was on, and it does not erase the line from the screen. During full-screen editing, **Ctrl-B** first turns off AUTO line generation. It then cancels any changes made to the line and moves the cursor to the beginning of the next BASIC line. If your program is scrolling out on the screen after a LIST command, pressing **Ctrl-B** stops the listing.
Ctrl-S	If your program is scrolling on the screen after a LIST command, pressing **Ctrl-S** pauses (**stalls**) the listing temporarily. Pressing **Ctrl-Q** restarts it.

Ctrl-Z	Sometimes the operating system allows you to put information on the screen from outside the BASIC Editor. **Ctrl-Z** clears that information from the screen, refreshes the **text** and clears any extraneous information from the screen.
Tab	Moves the cursor right to the next tab stop. Tab stops occur every eight character positions (1, 9, 17, ... , 73). If Insert Mode is on, BASIC inserts spaces into the text as the cursor moves right. If Insert Mode is off, pressing the **Tab** key just moves the cursor right to the next tab stop without inserting spaces.
Ctrl-C	When pressed anywhere on a screen line, the **Ctrl-C** key combination concatenates (joins) the line to the following screen line, thereby creating a new BASIC line.
Ctrl-N	The **Ctrl-N** key combination inserts a line-continuation underline into the program text.

Entering or Changing a BASIC Line

Since any line of text typed while BASIC is at the command level is processed by the Program Editor, you can use most of the keys described in the previous section to make corrections on the current line. (BASIC is at the command level when you see the prompt **Ok**.)

You can extend a BASIC line over more than one screen line by simply typing beyond the edge of the screen. The cursor wraps down to the next screen line.

You can also use **Ctrl-C** to enter subsequent text on the next screen line. The **Ctrl-C** actually fills the remainder of the screen line with blank characters and concatenates it to the next line. These blanks are included in the 1896 characters allowed for a BASIC line.

The **Ctrl-N** key combination inserts a line-continuation underline into the program text to allow the Compiler to translate long program lines written in the full-screen editor. This line-continuation underline becomes an underline and newline combination when the program is LISTed or written to a file.

When the **Enter** key is finally pressed, the entire BASIC line is passed to BASIC for processing.

Changing Characters

If you are typing a line and make an error, you can correct it. Use the **Cursor Left** or other cursor movement keys (explained in the previous section) to move the cursor to the position where the mistake occurred, and type the correct letters over the wrong ones. Then, move the cursor back to the end of the line using the **Cursor Right** or **End** keys, and continue typing.

Erasing Characters

If you notice that you have typed an extra character in the line you are typing, you can erase (delete) it using the **Del** key. Use the **Cursor Left** or other cursor movement keys to move the cursor to the character you want to erase; press the **Del** key, and the unwanted character is deleted. Then use the **Cursor Right** or **End** keys to move the cursor back to the end of the line, and continue typing.

Adding Characters

If you see that you have left out characters in the line you are typing, move the cursor to the position where you want to put the new characters; press the **Ins** key to get into Insert Mode; type the characters you want to add. The characters you type appear at the cursor position. The characters above and following the cursor are pushed to the right. As before, when you are ready to continue typing at the end of the line, use the **Cursor Right** or **End** keys to move the cursor there and just continue typing. Insert Mode automatically turns off when you use either of these keys.

Erasing Part of a Line

To erase to the end of a line on the screen, press **Ctrl-E**.

For example, suppose you have typed the following:

```
Ok 10 REM *** this is a remark_
```

You decide to change the line. You move the cursor to under the **t** in the first word **this**, to erase the rest of the line, press **Ctrl-E**:

```
Ok 10 REM *** _
```

Canceling a Line

To cancel a line that is currently being typed, press the **Ctrl-F** key anywhere in the line. You do not have to press **Enter.** The entire line beginning with the previous **Enter** will be erased. For example, suppose you have this line:

```
Ok THIS IS A LINE THAT HAS NO MEANING_
```

Even though the cursor is at the end of the line, the entire line is erased when you press **Ctrl-F**, and the cursor goes to the beginning of the line.

Entering or Changing a BASIC Program

An Interpreter BASIC program line always begins with a line number, and ends with an **Enter** key. The entire BASIC program line must be on the screen when you press the **Enter** key. It can contain a maximum of 1896 characters, including the line number and **Enter** (1896 characters is one screen full of text). If a line contains more than 1896 characters, the extra characters are truncated (removed) when **Enter** is pressed. Even if the extra characters still appear on the screen, they are not processed by BASIC.

You can enter BASIC keywords and variable names in any combination of uppercase (capital) and lowercase (small) letters.

Editing a Program Line on the Screen

You can edit any program line on the screen. First, move the cursor to where you want to make the change. Then use any or all of the techniques described in the previous section to change, delete, or add characters to the line. The changes do not become part of your program until you press **Enter**.

If you want to modify program lines that are not displayed at the moment, you can use the EDIT or LIST command to display them. EDIT clears the screen before listing the lines. LIST does not.

Note that when you are making corrections to a BASIC line you have already entered, you do not have to move the cursor to the end of that BASIC line before pressing **Enter**. The Program Editor knows where each BASIC line ends and it processes the whole line even if **Enter** is pressed at the beginning of the line.

Note: Use of the AUTO command for automatic line numbering can be very helpful when you are entering your program. However, you must turn AUTO off, by pressing **Ctrl-B**, before changing any lines other than the current one.

Copying a Line

You can copy a line in a program by moving the cursor to the beginning of the line to be copied and changing its line number to the new line number (by typing over the old number). When you press **Enter**, both the old line and the new line are in the program.

Adding a New Line to a Program

To add a new line to a program, type a valid line number (1 through 65529) that has not already been used in the program, then at least one nonblank character, then **Enter**. The line is saved as part of the BASIC program in storage.

If a line already exists that has the same line number as the line you have just entered, the old line is erased and replaced with the new one.

If you try to add a line to a program when there is no more room in storage, an **Out of memory** error occurs and the line is not added.

Replacing a Program Line

You can replace an existing line by typing the number of the line already in the program, the new text as you want it to appear, and then **Enter**.

Deleting a Program Line

To delete an existing program line, type only the number of the line and then press **Enter**. For example, if you just enter

```
Ok 10
```

line 10 is deleted from the program.

To delete a group of program lines, use the DELETE command.

Do not use the **Ctrl-F** key to delete program lines. **Ctrl-F** causes a line to be erased only from the screen, not from the BASIC program.

Deleting an Entire Program

To delete the entire program that currently resides in memory, enter the NEW command.

Saving the Program

Remember, changes made using these techniques change only the program in memory. To permanently save the program with the new changes, use the SAVE or REPLACE command before entering a NEW command or leaving BASIC.

Syntax Errors

During command-line editing, the BASIC Interpreter shows most of your errors immediately. Program lines entered during full-screen editing have their errors shown after you type RUN. See Appendix A in the *BASIC Reference* for a list of error messages and what they mean.

When you edit a line and store it in the program while the program is in Break Mode (you see the **Br** prompt) you cannot use CONT to continue the program.

Chapter 4. How to Use the BASIC Compiler

CONTENTS

Introduction

This chapter first shows you how to start the Compiler. Then it describes the steps of compiling and running, using a demonstration program as an example. Finally, it lists command line options, and describes program development in more detail.

Logging On to the System

To use the BASIC Compiler, you must first log on to the system by following the instructions in the *IBM RT PC Using and Managing the AIX Operating System* manual.

Starting the Compiler

To start the BASIC Compiler enter one of the following:

PC Mode: `$ basic -c progname`
Native Mode: `$ basicn -c progname`

Note: For both PC Mode and Native Mode the –c option specifies use of the Compiler. *Progname* is a pathname designating the file containing the BASIC program. It should follow the rules for naming files described in "Files" in Chapter 5. If you do not specify a *progname* or if the file is not found, BASIC returns an error.

A Session with demo.bas

This section uses a demonstration program to illustrate the step-by-step instructions for using the BASIC Compiler.

We recommend compiling the demonstration program before compiling any other programs, because this sample session gives you an overview of the compilation process. Also, you should read all the following sections. They contain information that is important to successful development of a program.

If you enter commands exactly as described in this section, you should have a successful session with the BASIC Compiler. If a problem does arise, check and redo each step carefully.

The steps involved in developing a program with the BASIC Compiler are

- Creating and debugging a source file
- Compiling and linking the code
- Running the program

We will follow these steps on the following pages, using the **demo.bas** program as an example.

Creating and Debugging demo.bas

We have prepared a debugged demonstration program called **demo.bas.** Because it is already prepared, you can start compiling it. (Please note that you can also run this program with the Interpreter.)

Before you compile the demonstration program, you should copy it to your own directory. To do so, enter the following commands:

```
$ cd
$ cp /usr/basic/demo.bas.
```

Compiling demo.bas

To compile the demonstration program, type

```
$ basic -c demo -l
```

As soon as you press the **Enter** key, the Compiler begins its work. When it is done, the $ prompt appears.

The Compiler generates an object file named **demo.o**. At the same time, it writes a listing file to the file **demo.lst** because we used the –l option. The Compiler should not find any errors in **demo.bas**, so there should not be any error messages on the screen. If you enter the operating system command **li**, you should see the two new files listed in your directory: **demo.o** and **demo.lst**.

At this point in the demonstration, you can view or print out the source listing file (**demo.lst**). To print the listing, enter:

```
$ print demo.lst
```

When you finish looking at the listing file, you can delete it. To do this, enter:

```
$ rm demo.lst
```

Linking demo.o

To LINK the demonstration program, type

```
$ baslink demo.o
```

the baslink shell procedure links **demo.o** with standard system libraries to create an executable file named **demo**.

Running demo

To run the demonstration program, enter:

```
$ demo
```

When you are done with the program, press the **Esc** key to return to the operating system prompt.

Creating and Debugging a Source Program

You can create a BASIC source file using any general purpose text editor that produces a standard ASCII file. Files saved by the IBM PC's BASICA without its A option do not compile.

Perhaps the best way to create a Compiler program is to use the editing and debugging facilities of the BASIC Interpreter. Compiler BASIC is similar to Interpreter BASIC. (The few differences are mentioned in notes in this handbook and in the *BASIC Reference.*) Because your programs will usually run in the same way with both versions, you can use the BASIC Interpreter as a powerful writing and debugging tool. This will save you from doing time consuming compilations and links while trying to find a bug in your program.

Note: The Compiler *metacommands* ($INCLUDE and others) and its *-n option* (relaxed line numbering) are not supported by the Interpreter.

Compiling

After you have created a BASIC source program, the next step is to compile it. This converts your program into a relocatable object code. If there are any BASIC errors, the Compiler will list them.

Compiler Command Line Format

You can include options in the BASIC command for the Compiler when you start BASIC. You can use these options to select Native Mode or PC Mode, redirect standard output, and specify a name for your object file.

The format of the BASIC command line for the Compiler is

```
basic[n] [>[>]stdout] -c progname [-d] [-l [pathname]] [-n]
[-o pathname] [-p name1 [name2 [name3]]][-v] [-w workspace]
[-s sourcespace]
```

This syntax uses the following conventions:

- You must type exactly all words that are not in italics.
- You must supply any items shown in lowercase *italic* letters.
- Items in square brackets ([ ]) are optional.
- All punctuation except square brackets must be included where shown.
- The command line must start with basic[n], but you may place the options in any order.
- All options must be separated by at least one space.

An explanation of the options follows.

n — This option starts Native Mode BASIC. If you do not use the n option, the PC Mode Compiler starts running.

>[>]stdout — *Stdout* is a pathname. The Compiler normally writes its output to the screen (standard output device). Using *>stdout* causes BASIC to redirect error messages and any statistics to a file instead of the screen. See "Redirection of Standard Input and Output" for more information.

-c progname	This parameter causes BASIC to compile the file specified by *progname*. The *progname* is a pathname and it must conform to the rules for specifying files described in Chapter 5. The program in the file must be in standard ASCII format.
-d	Normally, the generated object code includes line numbers so that runtime error messages can indicate where the error occurred. The -d option turns this feature off.
-l pathname	This option generates a Compiler listing to the file specified by *pathname*. The *pathname* is optional. If you do not specify a *pathname*, the name will be the same as the program, but with an extension of .lst (**progname.lst**). If the -l option is not used, then the Compiler does not generate a Compiler listing. If you have metacommands in your source program, the listing is produced under control of the metacommands.
-n	The -n option tells the Compiler to relax line numbering constraints. When -n is specified, line numbers in your source file may be in any order, or they may be eliminated entirely. Any line numbers which exist have nothing to do with the sequence of the lines; they serve as necessary target labels for GOSUBs, GOTOs, and any other statements which use line numbers as references for branching.
-o pathname	The -o option specifies a name for your object file. If you do not use the -o option, the name of the object file is the same as that of the program, but with an extension of .o (**progname.o**).

-p name1 [name2 [name3]]

The -p option designates the output files for LPT1:, LPT2:, and LPT3:. They are associated with *name1*, *name2*, and *name3*, respectively.

For example, the LPRINT statement writes to LPT1:, and the default file for LPT1: is **lpt1.lst**. This means that when you use the LPRINT statement in BASIC, your output goes (by default) to **lpt1.lst**. If, however, you use the -p *name1* option when you compile your program, then LPRINT's output goes to the file given as *name1* and not to **lpt1.lst**.

Like LPT1:, using LPT2: and LPT3: in your program is optional. However, LPT2: and LPT3: have no default files. If your program uses LPT2: and LPT3:, you must specify files (*name2* and *name3*) for them.

-v

If your BASIC source program uses event traps, the Compiler inserts an event trap check between each statement. This option inserts an event trap check at each line number instead of at each statement. If your program uses event traps, the -v option can produce a smaller program that may take less time to run.

-w workspace

Workspace is an integer that specifies the size of the user area reserved for BASIC program variables, strings, and arrays. The integer specifies the number of K bytes reserved. One K byte is equal to 1024 bytes. For example, -w 64 reserves 64K bytes of workspace. The default workspace size is 64K bytes. In PC Mode, the maximum workspace size you can reserve is 64K bytes; if you specify more than 64K, the default value of 64K is used.

–s sourcespace

Sourcespace is an integer that specifies the size (in K bytes) of the area reserved by the translator for program text, variable names, and the intermediate code generated by the translator. This space is separate from the user workspace.

The default size for *sourcespace* is 256K bytes. The minimum size for *sourcespace* is 32K bytes, and if you specify less than 32K bytes, BASIC allocates 32K bytes.

Command Line Example

Some examples of the BASIC command line for the Compiler:

```
$ basic -c prog1
```

This example tells the PC Mode Compiler to compile **prog1.bas** into **prog1.o**. No print file is specified.

```
$ basicn >error -c prog2 -d -p prog2.spl -w512
```

This example starts the Native Mode Compiler. It compiles **prog2.bas** into **prog2.o**. The >error option sends any compile time error information to the file named error. The –d option suppresses debugging code. The –p option sends LPRINT output to **prog2.spl**. The –w option establishes a workspace of 512K bytes.

Compiler Metacommands

A feature of the BASIC Compiler are the *metacommands.* They are called Compiler metacommands rather than BASIC statements because they are really not a part of the BASIC language, but rather they are commands to the Compiler. The metacommands for the Compiler are as follows:

$INCLUDE
$LINESIZE
$LIST
$PAGE
$PAGEIF
$PAGESIZE
$SKIP
$SUBTITLE
$TITLE

Note the distinctive $ prefix on the Compiler metacommands.

The metacommands are included in your source file as part of a remark, after the keyword REM or the single quote ('). (Because they are embedded in a remark, the metacommands do not cause a syntax error in the BASIC Interpreter.) You can use more than one metacommand on a line. All the metacommands are discussed in more detail in the *BASIC Reference.*

Most of the metacommands control the format of the listing file created by the Compiler. The $INCLUDE metacommand lets you combine several program files into one.

The List File and Metacommands

The Compiler listing is a list of your source program with any error messages that occurred during compilation. To get a listing, you must use the Compiler's -l *pathname* option. The format of the listing can be affected by the BASIC Compiler metacommands listed earlier.

Every page of the BASIC Compiler source listing has a header at the top. In the upper left-hand portion of the page, the first two lines contain your choice of title and subtitle, set with the $TITLE and $SUBTITLE metacommands, respectively.

The first three lines in the upper right-hand portion of the page contain the page number, the date, and the time. The name and version number of the Compiler appear on the line below the time, aligned with the right margin. The column labels also appear on that line.

You can turn the source code listing off and on with the $LIST− and $LIST+ metacommands. Errors are always listed. For example, you can get a listing of a program without the contents of $INCLUDE files by putting a $LIST− metacommand at the first line of each $INCLUDE file and a $LIST+ at the end of each $INCLUDE file.

$INCLUDE Metacommand

The $INCLUDE metacommand allows you to combine files for your source file. The $INCLUDE metacommand looks like this:

```
REM $INCLUDE: "pathname"
```

The Compiler includes the specified file in the source file at the point where it encounters this metacommand. That is, the contents of *pathname*, known as the *included file*, are read and processed as though the included file were inserted in your source file immediately following the $INCLUDE metacommand. When the Compiler finishes processing the included file, it goes back to the original BASIC source file and resumes processing the source file.

This process may be thought of as embedding *pathname* into your source file at the location of the $INCLUDE metacommand.

Included files can themselves include other files. You can nest up to eight levels of files in this way. If you go beyond that, you will get a **Too many $INCLUDE levels** error.

If you use a text editor other than the BASIC Program Editor, you can create a file of lines without line numbers. The Compiler allows lines without line numbers if you use its –n option. The $INCLUDE metacommand makes it very easy to include the same file in many different programs.

Included files can be very useful for COMMON declarations existing in programs that are linked or chained together, or for useful subroutines that you might have in an external library of subroutines.

When the Compiler Finishes

As soon as you enter the command line, the Compiler begins its work. If there are no errors in your program, the Compiler translates your program and sends the object code to a file with the name **progname.o** (unless you specify a different name using the –o option). When the Compiler is finished processing, the operating system prompt ($) appears. You can interrupt the Compiler run prior to its normal completion by pressing the **Delete** key.

Warnings and Error Messages

There are two kinds of error messages, warnings and severe errors. Warnings do not have to be corrected. Severe errors must be corrected, which means that you must rewrite part of your source program and start the compilation process again. Appendix A in the *BASIC Reference* lists and explains error messages.

After the Compiler has listed the errors, it then reports the number of errors found. The error report takes the form:

```
nnnnn Warning Error(s)
nnnnn Severe  Error(s)
```

If you use the Compiler's –l option it includes the error messages in the listing.

When the Compiler is done, it returns you to the operating system ($). If a severe error was encountered during compilation, the Compiler returns a nonzero exit value.

Linking

The .o file created by the Compiler is not executable and needs to be linked to the appropriate library. Linking is the process of

- Combining separately produced object (.o) modules
- Searching the appropriate library files for definitions of unresolved external references
- Resolving external cross-references
- Computing absolute addresses for local references within modules
- Producing an executable file

To link a compiled program, use the shell procedure baslink (for a PC Mode program) or baslinkn (for a Native Mode program):

```
baslink[n] progname [mod1.0 [mod2.0 ...]]
```

where *progname.bas* is the source file, *progname.o* is your object file, and *progname* is the name of the executable file. *mod1.0 mod2.0 ...* are the names of any object modules you intend to include in the program. Typically, modules are written in the C programming language. The module names on the command line must be separated by spaces.

The shell procedure invokes the AIX linker, **ld**. For more information on the **ld** command, refer to the *IBM RT PC AIX Operating System: Commands Reference* manual.

Compiling and Linking in One Step

Use the shell procedure basicc (for a PC Mode program) or basicnc (for a Native Mode program). The format of the command line for the shell procedure is:

```
basic[n]c progname [compiler options
[ -- mod1.0 [mod2.0 ... ]]]
```

An explanation of the options follows:

progname

This is the name of the program to be compiled and linked. The source must be in **progname.bas**; the executable program will be written to *progname*.

compiler options

These are any of the *compiler options* documented in the Compiler Command Line Format section presented earlier in this chapter.

-- (two hyphens)

These separate Compiler options from the names of object modules passed to **ld**. There should be at least one blank space before and after the hyphens.

mod1.0, mod2.0 ...

These are the names of the object modules you intend to include in the executable program.

Running the Program

The executable object file can be run by entering the file's name at the system prompt. The following command loads and starts the program.

```
$ demo
```

The executable file can also be started from within another program, as in the following statement:

```
10 CHAIN "demo"
```

If the program terminates abnormally, you may be able to recover the keyboard/screen environment. See *Recovering from Errors* in Chapter 2.

Chapter 5. General Information about BASIC

CONTENTS

BASIC Program Lines

Program lines in a BASIC program have the following format:

```
nnnnn BASIC statement[:BASIC statement...]['comment]
```

Program lines begin with a line number and end with **Enter**.

Line Numbers

"nnnnn" is a line number from 1 to 65529. Line numbers show the order in which the program lines are stored and serve as reference points for branching and editing.

Note: If you are using the Compiler with the relaxed line number option (–n), you can put line numbers in any order or even leave them out. See "Compiler Command Line Format" in Chapter 4.

Long Lines

You can, if you wish, have more than one BASIC statement on a program line, but each statement must be separated from the one before it by a colon. For example,

```
10 FOR I = 1 TO 3 : PRINT I : NEXT
```

A program line can be up to 1896 characters long.

If you want to continue a program line onto the next line, the underline character (_) must be at the end of the continued line. For example,

```
10 REM This is a BASIC remark _
   that continues on to the next line, _
   and on to the next one too.
20 PRINT A, B, _
         C, D
30 PRINT "This is a string constant _
          continued on the next line."
```

The underline characters must not break up any BASIC keywords, line numbers, numeric constants, or variable names. For example, the next three lines are illegal:

```
100 IF SumOfAB =  100 THEN SumOf _
    AB =  0 : GO_
    SUB 500
```

The Interpreter accepts ASCII BASIC files using the underline character for line continuation. To type a line-continuation underline in the BASIC full-screen Editor, use the **Ctrl-N** key combination. This inserts an underline into the text onto the screen, which is interpreted as an underline/newline combination when the program is LISTed or written to a file.

Comments

Comments are statements that you write to describe how the program works. They improve your understanding of the program, but are not executed by BASIC. Comments can be included at the end of a line. The single quote (') or the keyword REM separates the comment from the rest of the line. For example,

```
10 INPUT "Type your name" ; N$  REM comment
20 PRINT "Thank you, "; N$      'also a comment
```

Character Set

The BASIC character set consists of alphabetic (a–z, A–Z), numeric (0–9), and special characters.

The following characters have specific meanings in BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign or concatenation symbol
−	Minus sign
*	Asterisk or multiplication symbol
/	Slash, division symbol, or path separator
\	Backslash; integer-division symbol
^	Caret or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent sign or integer type-declaration character
#	Number (or pound) sign, or double-precision type-declaration character
$	Dollar sign or string type-declaration character
!	Exclamation point or single-precision type-declaration character
&	Ampersand
,	Comma
.	Period
'	Single quotation mark, apostrophe, or remark delimiter
;	Semicolon
:	Colon or statement separator
?	Question mark (PRINT abbreviation)
<	Less than
>	Greater than
"	Double quotation mark or string delimiter
_	Underline

Many characters can be printed or displayed even though they have no particular meaning to BASIC. See Appendix B, "ASCII Character Codes," in the *BASIC Reference* for a complete list of these characters.

Reserved Words

Certain words and letter combinations have special meaning in BASIC. They are called *reserved words.* Reserved words include all BASIC commands, statements, function names, and operator names. Reserved words cannot be used as variable names.

Reserved words must be separated from data or other parts of a BASIC statement by blanks or special characters as allowed by the syntax described in the *BASIC Reference.*

You can enter reserved words in uppercase or lowercase letters, or in any combination thereof.

The following are reserved words in BASIC:

ABS	CIRCLE	DRAW	GOSUB
APPEND	CLEAR	EDIT	GOTO
AS	CLOSE	ELSE	HEX$
ASC	CLS	END	IF
ASK	COLOR	EOF	$INCLUDE
ATN	COM	EQU	INKEY$
AUTO	COMMON	ERASE	INP
BASE	CONT	ERL	INPUT
BEEP	COS	ERR	INPUT#
BLOAD	CSNG	ERROR	INPUT$
BREAK	CSRLIN	EXP	INSTR
BSAVE	CVD	EXTERNAL	INT
CALL	CVI	FIELD	KEY
CCHAR	CVP$	FILES	KILL
CDBL	CVS	FIX	LEFT$
CHAIN	DATA	FOLLOW	LEN
CHANGE	DATE$	FOR	LET
CHDIR	DEF	FN	LINE
CHR$	DELETE	FRE	$LINESIZE
CINT	DIM	GET	LIST

$LIST	OPTION	RESUME	SYSTEM
LLIST	OR	RETURN	TAB
LOAD	OUT	RIGHT$	TAN
LOC	OUTPUT	RMDIR	THEN
LOCATE	$PAGE	RND	$TITLE
LOF	$PAGEIF	RSET	TIME$
LOG	$PAGESIZE	RUN	TIMER
LOG10	PAINT	SAVE	TO
LPOS	PEEK	SCREEN	TRACE
LPRINT	PLAY	SEG	TROFF
LSET	PMAP	SGN	TRON
MERGE	POINT	SHELL	UNBREAK
MID$	POKE	SIN	UNFOLLOW
MKDIR	POS	$SKIP	UNTRACE
MKD$	PRESET	SOUND	USING
MKI$	PRINT	SPACE$	VAL
MKS$	PRINT#	SPC	VARPTR
MOD	PSET	SQR	VARPTR$
NAME	PUT	STEP	VIEW
NEW	RANDOMIZE	STICK	WEND
NEXT	READ	STOP	WHILE
NOT	REM	STR$	WIDTH
OCT$	RENUM	STRIG	WINDOW
OFF	REPLACE	STRING$	WRITE
ON	RESET	$SUBTITLE	WRITE#
OPEN	RESTORE	SWAP	

BASIC Statements

A BASIC statement is either executable or nonexecutable. *Executable statements* are instructions that tell BASIC what to do next while running a program. For example, PRINT X is an executable statement. *Nonexecutable statements*, such as COMMON, DATA, DEFtype, or REM, contain information only and do not cause any program action when BASIC sees them. All the BASIC statements are explained in detail in the *BASIC Reference.*

BASIC Data Types

BASIC allows you to work with the following types of data:

Data Type	Short Definition	Example
String	Sequence of characters	Dale
Integer	Whole number	1234
Real	Number with fractional part	12.34

BASIC has two types of real numbers, single precision and double precision. Double-precision real numbers can have a larger numeric range and be more than twice as accurate as single-precision ones. See "Numeric Precision," later in this chapter.

Data in BASIC can be represented as constants or as variables. We shall discuss constants and then variables.

Constants

Constants are values that you supply in the BASIC program and that do not change when the program is running. There are two types of constants: string (character) constants and numeric (integer and real) constants. Examples of constants are Hello, 12, and 58797873673.5.

String Constants

A *string* constant is a sequence of characters enclosed in double quotation marks. In PC Mode, a string constant may have up to 255 characters. In Native Mode, a string constant may have up to 1896 characters. The characters can be letters, numbers, or symbols. For example,

```
HELLO
$25,000.00
Number of Employees
!@#$%^&*()_+
```

Note: If you start a string with a quotation mark but do not finish it with the second quotation mark, BASIC assumes that the string continues to the end of the screen line. Be careful when you start a string constant, to end it with a quotation mark; otherwise, any statements following on the same line are also part of that string. For example, in the following line, **PRINT 1 + 2** is assumed to be part of the constant starting with "Fred.

```
Ok PRINT "Fred; : PRINT 1 + 2
Fred; : PRINT 1 + 2
Ok
```

There are a few cases where BASIC knows that a particular sequence of characters must be a string constant, and the quotation marks are not required. These cases are noted where appropriate in the *BASIC Reference.*

Numeric Constants

Numeric constants are positive or negative numbers. Negative numbers must be preceded by a minus sign (−), but a plus sign (+) is optional on a positive number. Numeric constants in BASIC do *NOT* contain commas or dollar signs ($). Numeric constants can take the following forms:

Real Number Constants: Decimal Notation

This common notation includes numbers with decimal points and very large integers. Real numbers using this notation are accurate up to 16 digits.

BASIC considers decimal constants ending with a number sign (#) to be double-precision numbers (accurate to 16 digits). For example, 123.45# is a double-precision number, equivalent to 123.4500000000000.

BASIC considers decimal constants ending with an exclamation point (!) to be single-precision numbers. For example, 123.45! is a single-precision number, equivalent to 123.450 in PC Mode.

A number with no trailing character is assumed to be an integer, single-precision real, or double-precision real, depending on how large the number is, and whether or not it has a decimal point.

Real Number Constants: Exponential Notation

Exponential notation is similar to scientific notation. It consists of an optionally signed integer or decimal number (the mantissa) followed by the letter E or D and an optionally signed integer (the exponent). The E or D means *times ten to the power of*. It takes one of the following formats:

```
[sign] number E [sign] integer
```

or

```
[sign] number D [sign] integer
```

For example, in the number 23E–2, 23 is the mantissa, and –2 is the exponent. This number is read as 23 times 10 to the power of minus two. You can write it as 0.23 in decimal notation.

Single-precision numbers use the letter E. For example, –1.234E2 is equivalent to –123.4!, 567.89E–2 is equivalent to 5.6789!, and 235.988E–7 is equivalent to .0000235988!.

Double-precision floating point numbers use the letter D instead of the letter E. For instance, –1.234D2 is equivalent to –123.4#, 567.89D–2 is equivalent to 5.6789#, and 2359D6 is equivalent to 2359000000#.

Remember, E indicates a single-precision number and D indicates a double-precision number.

Integer Constants

Integers, being whole numbers, cannot have decimal points. They can range from -2^{15} through $2^{15}-1$ in PC Mode, and from -2^{31} through $2^{31}-1$ in Native Mode. BASIC allows five different integer notations:

Decimal: This is the most common notation. For example, 1, 567, and –10. Decimal integers range from –32768 through +32767 in PC Mode; and from –2147483648 through +2147483647 in Native Mode.

Hexadecimal: Hexadecimal integers use the digits 0 through 9, and the letters A through F. They must have a prefix of &H. For example, &H76, &H32F, and -&HA. Hexadecimal integers can be up to four digits in length in PC Mode (&HFFFF), and up to eight digits in Native Mode (&HFFFFFFFF).

Octal: Octal integers use the digits 0 through 7, and a prefix of &O or just &. For example, &O347, &362535 and -&12. Octal integers can be up to 6 digits in length in PC Mode, and up to 11 digits in Native Mode. The largest PC Mode octal integer is &O177777. For Native Mode it is &O37777777777.

Binary: Binary integers use the digits 0 and 1, and a prefix of &B. For example, &B01, &B11001010, and -&B1010. Binary integers can be up to 16 digits long in PC Mode, and up to 32 digits long in Native Mode.

ASCII: ASCII notation uses a prefix of &" followed by a single character and optionally a second ". Some examples are: &"J", &"d", and -&"&". This gives you the ASCII value of the character. You can only use printable characters. See Appendix B in the *BASIC Reference* for a list of ASCII characters and their numeric values.

Variables

Variables are like electronic boxes used to store data in the computer. Each variable is given a name, such as Denise$, and Phone. Variable names may not change, but the numbers or characters stored in a variable may change while the program is running.

As with constants, there are two general types of variables: numeric and string. A numeric variable always has a numeric value. A string variable can only have a character string value. A string variable can have up to 255 characters in PC Mode and 65535 characters in Native Mode.

You can give a variable an unchanging value (such as salary equals a certain amount) or you can set its value to be the result of calculations or data input statements in the program (such as salary equals 10% of sales). In either case, the variable type (string or numeric) must match the type of data assigned to it.

If you use a numeric variable before you assign a value to it, its value is assumed to be zero. String variables are initially assumed to be null; that is, they have no characters in them and have a length of zero.

How to Name a Variable

BASIC allows a variable name to be up to 39 characters long plus an optional type character (for example, $ or %). If a variable name is longer than 39 characters, BASIC ignores the characters between the 39th character and the type character.

The characters can be letters (uppercase or lowercase), numbers, and decimal points. The first character must be a letter. BASIC treats uppercase and lowercase letters in variable names or keywords as equivalent. For example, the variable AA$ is the same as Aa$, aa$, or aA$. Special characters that identify the type of variable are also allowed as the last character of the name. For more information about types, see the next section, "How to Declare Variable Types."

A variable name cannot be a reserved word, but it can contain embedded reserved words. See "Reserved Words," earlier in this chapter, for a list of reserved words. For example,

```
10 EXP = 5
```

is invalid, because EXP is a reserved word. However,

```
10 EXPONENT = 5
```

is okay, because EXP is embedded in the variable name.

Note: A variable beginning with FN*l* (where *l* is a letter) is assumed to be a call to a user-defined function. See the DEF FN statement in the *BASIC Reference.*

How to Declare Variable Types

A variable name determines its type (string or numeric, and if numeric, its precision).

String variable names finish with a dollar sign ($). For example,

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

The dollar sign is a variable type-declaration character. It *declares* that the variable will represent a string.

A string variable with a $ character occupies four bytes of storage in PC Mode and eight bytes in Native Mode, plus one byte for each character in the string.

Numeric variable names can declare integer, single-, or double-precision values.

The type-declaration characters for numeric variables are

% Integer variable

! Single-precision variable

Double-precision variable

Note: If a variable name does not end with a $, %, !, or #, BASIC assumes the variable is a single-precision number.

Examples of variable names with type-declaration characters:

PI#	declares a double-precision value
MINIMUM!	declares a single-precision value
N$	declares a string value
ABC	defaults to a single-precision value

Variable types can also be declared in another way. The BASIC statements DEFINT, DEFSNG, DEFDBL, and DEFSTR can be included in a program to declare the types for certain variable names. These statements are described under "DEFtype Statements" in the *BASIC Reference.* All the examples in this book assume that no DEFtype statements are used unless they are explicitly shown in the examples.

Arrays

An *array* is a list or table of values that is referred to by a single name. Each value in the array is called an *element.* Elements are string or numeric variables and can be used in expressions and in BASIC statements.

The *subscript*, which is the number in parentheses, indicates the position of an element in an array. Zero is the first position unless you explicitly change it. See the OPTION BASE statement in the *BASIC Reference.*

Declaring the name and type of an array and setting the number of elements and their arrangement within it is known as *defining*, or *dimensioning*, the array. The maximum number of dimensions for an array is 15.

To define an array, use the DIM statement. For example,

```
DIM B$(5)
```

This statement creates a one-dimensional, string-variable array named B$ with a maximum of six elements. Array B$ can be thought of as a list of character strings.

B$(0)
B$(1)
B$(2)
B$(3)
B$(4)
B$(5)

The statement below creates a two-dimensional, numeric-variable array named A. Since the array does not include a type-declaration character, the array by default consists of single-precision values.

```
DIM A(2,3)
```

Array A can be thought of as a table of rows and columns.

A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)

The element in the second row, first column, is called A(1,0).

If you use an array element before you define or dimension the array, BASIC dimensions the array with subscripts 0 through 10 for each dimension you declare. For example, the following statement,

```
10 X( 5 ) = 2356.88
```

creates an array, X, with a subscript range 0 through 10.

Here is a sample program:

```
Ok LIST
   10 DIM YEARS(3,4)
   20 YEARS(2,3)=84
   30 FOR ROW=0 TO 3
   40 FOR COLUMN=0 TO 4
   50 PRINT YEARS(ROW,COLUMN);
   60 NEXT COLUMN
   70 PRINT
   80 NEXT ROW
Ok RUN
 0  0  0  0  0
 0  0  0  0  0
 0  0  0 84  0
 0  0  0  0  0
Ok
```

In this program, line 10 dimensions an array with 20 elements (4 rows and 5 columns). Line 20 assigns the value of 84 to the array element at position 2,3. The nested loops in lines 30-80 print the array as a 4-by-5 table.

Note: A simple variable can have the same name as an array variable because A$ is different from any of the elements in array A$( ).

Numeric Expressions and Operators

A numeric expression can be simply a numeric constant or variable. It can also be an operator, combining other numeric expressions to produce a single numeric value.

Numeric operators perform mathematical or logical operations on numeric values to produce a value that is a number. BASIC numeric operators can be divided into the following categories:

- Arithmetic
- Relational
- Logical
- Functions

Arithmetic Operators

The arithmetic operators perform the usual arithmetic operations in the standard mathematical order of preference; that is, when an expression contains more than one operation, they are carried out in the following order:

Operator	Operation	Sample Expression
^ or **	Exponentiation	X^ Y or X**Y
-	Negation	-X
* /	Multiplication, Floating Point Division	X*Y
\	Integer Division	X\ Y
MOD	Modulo Arithmetic	X MOD Y
+ -	Addition, Subtraction	X+Y X-Y

Although most of these operations probably look familiar to you, integer division and modulo arithmetic might need some explanation.

Integer Division: A backslash (\) denotes integer division. It rounds its two operands to integers before dividing (unless the operand is already an integer). Then it truncates the quotient to an integer. For example,

```
Ok LIST
   10 A =  10\4
   20 B =  25.68\6.99
   30 PRINT A;B
Ok RUN
 2  3
Ok
```

Modulo Arithmetic: Modulo arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example,

```
Ok LIST
   10 A =  7 MOD 4
   20 PRINT A
Ok RUN
 3
Ok
```

This result occurs because 7\4 is 1, with a remainder of 3.

MOD rounds its two operands to integers before dividing (unless the operand is already an integer). For example, in the following statement,

```
Ok PRINT 25.68 MOD 6.99
 5
Ok
```

the result is 5 because MOD rounds 25.88 and 6.99, and 26\7 is 3, with the remainder 5.

Numeric Functions

A function is used like a variable in an expression to call a predetermined operation that is to be performed on one or more operands. BASIC has some built-in numeric functions, such as SQR (square root) and SIN (sine).

You can also define your own numeric functions using the DEF FN statement.

Numeric Precision

BASIC stores numbers as integers, single-precision reals, or double-precision reals. The following table summarizes the range and accuracy of these numbers.

PC Mode

	Integer	Single Precision	Double Precision
Range	−32768 to 32767	2.9E−39 to 1.7E+38	2.9D−39 to 1.7D+38
Storage	2 bytes	4 bytes 7 digits	8 bytes 17 digits
Accuracy	Exact	6 digits	16 digits
Printed	All digits	7 digits	16 digits

Native Mode

	Integer	Single Precision	Double Precision
Range	-2^{31} to $2^{31}-1$	1.41E−45 to 3.40E+38	4.95D−324 to 1.79D+308
Storage	4 bytes	4 bytes 7 digits	8 bytes 17 digits
Accuracy	Exact	7 digits	16 digits
Printed	All digits	7 digits	16 digits

Note: The values given for single- and double-precision reals apply to both positive and negative numbers.

For more information about storage of numbers, see Appendix B, "How Variables are Stored," in this handbook.

How BASIC Converts Numbers from One Precision to Another

When necessary, BASIC converts a number from one precision to another according to the following rules:

1. If a numeric value of one precision is assigned to a numeric variable of a different precision, BASIC converts the number to the precision declared in the target variable name. For example,

```
Ok LIST
   10 A% =  23.42
   20 PRINT A%
Ok RUN
 23
Ok
```

   prints A% as an integer, not as a real number.

2. Rounding as opposed to truncation, occurs when converting a higher precision value to a lower precision value (for example, changing from double- to single-precision values). For example,

```
Ok LIST
   10 C =  55.8834667#
   20 PRINT C
Ok RUN
 55.88347
Ok
```

   This affects not only assignment statements (for example, I%=2.6 results in I%=3), but also function, expression, and statement evaluations. For instance, TAB(4.2) goes to the fourth position; A(1.8) is the same as A(2); and X=11.6 MOD 4 results in a value of 0 for X.

Note: When the digit to be rounded is 5, Native Mode rounds the number down and PC Mode rounds it up. For example, the following program,

```
Ok 10 a% = 2.5
Ok 20 PRINT a%
```

prints 2 in Native Mode, and 3 in PC Mode.

3. If you convert from a lower precision to a higher precision number, the resulting higher precision number cannot be any more accurate than the lower precision number. For example, in PC Mode, if you assign a single-precision value (A) to a double-precision variable (B#), only the first six digits of B# will be accurate because only six digits of accuracy were supplied with A. The error can be bounded using the following formula:

   ```
   ABS(B#-A) < 6.3E-8 * A
   ```

   That is, the absolute value of the difference between the printed double-precision number and the original single-precision value is less than 6.3E–8 times the original single-precision value. For example,

   ```
   Ok LIST
      10 A = 2.04
      20 B# = A
      30 PRINT A; B#
   Ok RUN
    2.04 2.039999961853027
   Ok
   ```

   Note: Be careful when using both single- and double-precision numbers in the same expression because it might reduce accuracy.

Numeric Precision and Expression Evaluation

The precision in which an expression is evaluated depends on the numeric precision of the operands, and on the type of operator used. The following concepts apply.

Operand Precision

- In PC Mode, single-precision real numbers have greater precision (accuracy) than integers.
- In Native Mode, the opposite is true.
- In both Modes, double-precision real numbers have greater precision than single-precision real numbers and integers. (See the table under "Numeric Precision.")

Operator Precision

- The +,−,*,^ and ** operators work in Native Mode in the minimum precision necessary and in PC Mode in the minimum *floating-point* precision necessary.
- The floating-point division operator (/) uses the minimum *floating-point* precision necessary.
- The MOD and \ operators work in integer.

These concepts translate into the following rules:

- The operators MOD and \ convert their two operands to integers before dividing and returning an integer.
- For the +,−,*,/,^, and ** operators, if their two operands are of the same numeric type, then the operation is evaluated in the precision of that type.

  Exception 1: In PC Mode, these operators convert two integer operands to single-precision reals before evaluating the expression and returning a single-precision real number. For example,

```
Ok PRINT 32000 * 32000
 1.024E+09
Ok
```

Exception 2: In Native Mode, the floating-point division operator (/) converts two integer operands to double-precision reals before evaluating the expression and returning a double-precision real number. For example,

```
Ok PRINT 2/3
 .6666666666666667
Ok
```

- For the +,−,*,/,^, and ** operators, if their two operands are of mixed types, then BASIC converts the operands to double-precision real numbers before evaluating the expression and returning a double-precision real number.

  In the following program, BASIC converts the 7 to a double-precision real number, performs the division in double precision and returns a double-precision result, D#.

```
Ok LIST
   10 D# = 6#/7
   20 PRINT D#
Ok RUN
 .8571428571428571
Ok
```

  Exception: In PC Mode, if one operand is an integer and the other is a single-precision real number, then BASIC converts the integer to a single-precision real number before evaluating the expression. For example,

```
Ok LIST
   10 D = 6.0/7
   20 PRINT D
Ok RUN
 .8571429
Ok
```

Techniques for Formatting Numeric Output

BASIC has built-in statements and functions that you can use in your programs to display numbers in the desired format and with the desired accuracy.

To display program results in decimal notation, use the PRINT USING and LPRINT USING statements. These statements let you choose the format in which the results will be printed or displayed. For example,

```
Ok LIST
   10 FOR I=4 to 5 STEP .1
   20 PRINT USING "#.# ";I;
   30 NEXT
Ok RUN
4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0
Ok
```

Relational Expressions and Operators

Relational operators compare two values. You can use the relational operators to compare two numeric values or two string values. The result of the comparison is either true (–1) or false (0). This result is usually used to make a decision regarding program flow. See the IF statement in the *BASIC Reference.*

Operator	Relation Tested	Sample Expression
=	Equality	X=Y, X$=Y$
<> or ><	Inequality	X<>Y X><Y
<	Less than	X<Y
>	Greater than	X>Y
<= or =<	Less than or equal to	X<=Y X=<Y
>= or =>	Greater than or equal to	X=>Y X=>Y

Numeric Comparisons

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. The expression (X+Y < (T−1)/Z) is true (−1) if the value of X plus Y is less than the value of T−1 divided by Z.

BASIC makes sure that the two relational operands are of the same type. If the operands are of different (mixed) numeric types, BASIC converts both operands to double-precision real values before comparing them. (Exception: In PC Mode, if BASIC compares an integer with a single-precision real value, it converts the integer to single precision.) For example, in Native Mode, (X% = Y) is a double-precision floating point comparison and is the same as (CDBL(X%) = CDBL(Y)). In PC Mode, (X% = Y) is a single-precision floating-point comparison and is exactly the same as (CSNG(X%) = Y). If the two relational operands are already of the same numeric type, BASIC does not convert them before comparison.

String Comparisons

String comparisons can be thought of as alphabetical. That is, a string is *less than* another if the first letter of the string comes before the other one alphabetically. Lowercase letters are *greater than* their uppercase counterparts. Numbers are *less than* letters.

The way two strings are actually compared is by taking one character at a time from each string and comparing the ASCII codes. See Appendix B, "ASCII Character Codes," in the *BASIC Reference.* If all the ASCII codes are the same, the strings are equal. Otherwise, as soon as the ASCII codes differ, the string with the lower code number is less than the string with the higher code number. If the end of one string is reached during string comparison, the shorter string is said to be less. Leading and trailing blanks are significant. For example, all the following relational expressions are true (that is, the result of the relational operation is –1):

```
"AA" < "AB"
"filename" = "filename"
"X&" > "X#"
"kg" > "KG"
"RICH" < "RICHB"
B$ < "718"  (where B$ = "12343")
```

All string constants used in comparison expressions must be enclosed in quotation marks.

Logical Expressions and Operators

Logical operators perform logical, or *Boolean*, operations on integer values. Just as the relational operators are usually used to make decisions regarding program flow, logical operators are usually used to connect two or more relations and return a true or false value to be used in a decision. See the IF statement in the *BASIC Reference.*

Logical Operators

A logical operator takes a combination of true-false values and returns a true or false result. An operand of a logical operator is considered to be true if it is not equal to zero (like the –1 returned by a relational operator), or false if it is equal to zero. The number is calculated by performing the operation bit by bit.

The logical operators are

- NOT (logical complement)
- AND (conjunction)
- OR (disjunction)
- XOR (exclusive OR)
- IMP (implication)
- EQV (equivalence)

Each operator returns results as indicated in the following table. T indicates a true, or nonzero value. F indicates a false, or zero value. The operators are listed in order of precedence.

NOT

X	NOT X
True	False
False	True

AND

X	Y	X AND Y
True	True	True
True	False	False
False	True	False
False	False	False

OR

X	Y	X OR Y
True	True	True
True	False	True
False	True	True
False	False	False

XOR

X	Y	X XOR Y
True	True	False
True	False	True
False	True	True
False	False	False

EQV

X	Y	X EQV Y
True	True	True
True	False	False
False	True	False
False	False	True

IMP

X	Y	X IMP Y
True	True	True
True	False	False
False	True	True
False	False	True

The following examples demonstrate ways to use logical operators in decisions.

```
IF HE>60 AND SHE<20 THEN 1000
```

In this example, the result is true if the value of the variable HE is more than 60 and the value of SHE is less that 20.

```
50 IF NOT (P=-1) THEN 100
```

In this example, the program branches to line 100 if P is not equal to -1. Note the NOT (P=-1) does not produce the same result as NOT P. See the next section, "How Logical Operators Work," for an explanation.

```
100 FLAG% = NOT FLAG%
```

This example switches a value back and forth from true (-1) to false (0).

How Logical Operators Work

Logical operators convert their operands to integers and return an integer result. (Operands must be in the integer range or an **Overflow** error occurs.) If the operand is negative, the twos complement form is used. This turns each operand into a sequence of 16 bits. The operation is performed on these sequences. That is, each bit of the result is determined by the corresponding bits in the two operands, according to the tables for the operator listed previously. A 1 bit is considered true and a 0 bit is false.

Thus, you can use logical operators to test for a particular bit pattern. For instance, the AND operator might be used to mask all but one of the bits of a status flag.

The following examples show how the logical operators work.

```
A% = 63 AND 16
```

Here, A% is set to 16. Because 63 is binary 111111 and 16 is binary 10000, 63 AND 16 equals 010000 in binary, which is equal to 16.

In the following example, B% is set to 8. Because –1 is binary 11111111 11111111 and 8 is binary 1000, –1 AND 8 equals binary 00000000 00001000, or 8.

```
B% = -1 AND 8
```

In the next example, C% equals 6. Because 4 is binary 100 and 2 is binary 010, 4 OR 2 is binary 110, which is equal to 6.

```
C% = 4 OR 2
```

The next example shows how to form the twos complement of a number. X% is 2, which is 10 binary. NOT X% is then binary 11111111 111111101, which is –3 in decimal; –3 plus 1 is –2, the complement of 2. That is, the twos complement of any integer is the bit complement plus one.

```
X% = 2
TWOSCOMP% = (NOT X%) + 1
```

Note that if both operands are equal to either 0 or –1, a logical operator returns either 0 or –1.

String Expressions and Operators

A string expression can simply be a string constant or variable, or it can combine constants and variables by using operators to produce a single string value.

String operators are used to arrange character strings in different ways. The two categories of string operators are

- Concatenation
- Function

Note that although you can use the relational operators =, < >, <, >, <=, and >= to compare two strings, these are not considered to be string operators because they produce an integer result, rather than a string result. Read through "Relational Operators" earlier in this chapter for an explanation of how you can compare strings using relational operators.

Concatenation

Joining two strings together is called concatenation. Strings are concatenated using the plus symbol (+). For example,

```
Ok LIST
   10 COMPANY$ =  "IBM"
   20 TYPE$ =  " Computer"
   30 FULLNAME$ =  TYPE$ +  " System"
   40 PRINT COMPANY$+FULLNAME$
Ok RUN
IBM Computer System
Ok
```

String Functions

A string function is like a numeric function except that it returns a string result. A string function can be used in an expression to call a predetermined operation that is to be performed on one or more operands. BASIC has built-in string functions, such as MID$, which returns a string from the middle of another string, and CHR$, which returns the character with the specified ASCII code.

You can also define your own string functions using the DEF FN statement.

Order of Evaluation

In the previous sections, the categories of numeric operations have been discussed in their order of precedence, and the precedence of each operation within a category was indicated in the discussion of the category. In summary:

1. Operations within parentheses ( ) are performed first.
2. Function calls are evaluated.
3. Arithmetic operations are performed in this order:
   a. ^or **
   b. unary –
   c. *, /
   d. \
   e. MOD
   f. +,–
4. Relational operations are evaluated.
5. Logical operations are performed last in this order:
   a. NOT
   b. AND
   c. OR
   d. XOR
   e. EQV
   f. IMP

For example, in the following statement, because * is higher on the list than +, BASIC evaluates the expression 3 * 4 (=12), before adding it to 2.

```
Ok PRINT 2 +  3 * 4
 14
Ok
```

The order in which operations at the same level in the list are performed might vary between PC Mode and Native Mode (but not between the Compiler and the Interpreter in the same Mode). To control the order in which the operations are performed, use parentheses. For example,

```
Ok PRINT ( 2 +  3 ) * 4
 20
Ok
```

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
$X + 2Y$	X + Y*2
$X - \frac{Y}{Z}$	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)

Note: If you have two operators next to each other, the second operator can only be +, −, or NOT. For example, the expression X*-Y is legal; BASIC understands it to be X*(-Y). However, the expression X-*Y is not legal.

Files

A *file* is a collection of information that is kept somewhere other than in the random access memory (usually on disk). To access the information, you must open the file with the OPEN statement. Then you can use the file for input and/or output.

BASIC supports the concept of general device I/O files. This means that any type of input/output can be treated as I/O to a file, whether you are using a disk file or you are using your computer to communicate with another computer.

File Number

BASIC performs I/O operations using a *file number.* You assign the number to a file when you open it with the OPEN statement. A file number can be any number, variable, or expression from 1 to 15.

Filename

The filename must conform to the operating system conventions.

- The *name* can be from 1 to 14 characters.

  If *name* is longer than 14 characters, a **Bad filename** error occurs.

- Only the following characters are allowed in a *name*:

  a through z
  A through Z
  0 through 9
  ( ) { }
  @ # $ % ^& !
  - _ ' ` ~

  Note: We strongly suggest that you use only lowercase letters (a – z), digits, and the period in filenames because other characters may have special meaning to BASIC and the operating system.

- For BASIC program files:

  If the filename does not already end with .bas, BASIC appends .bas to the filename. If adding .bas makes the filename longer than 14 characters, a **Bad Filename** error occurs.

Some examples of filenames for BASIC are as follows:

```
27hal
vdl
program1.bas
```

BASIC appends .bas to the first two filenames because the resultant names are still within the 14-character limit.

Device Names

The operating system does not support device names, but BASIC recognizes some names as having special meaning.

The device name consists of up to four alphameric characters followed by a colon (:). The name can be in UPPERCASE or lowercase. It is a name assigned to input/output devices and device files. Device names and what they apply to are as follows:

KYBD: or kybd:	Keyboard. STDIN when BASIC is invoked. Input only.
SCRN: or scrn:	Screen. STDOUT when BASIC is invoked. Output only.
LPT1: or lpt1:	First print file. Output.
LPT2: or lpt2:	Second print file. Output.
LPT3: or lpt3:	Third print file. Output.
COM1: or com1:	Operating system device driver linked to /dev/com1. Input and output.
COM2: or com2:	Operating system device driver linked to /dev/com2. Input and output.

Naming Files

A file is described by its *pathname*. The *pathname* is a string expression in the form:

```
[directory]filename
```

The directory tells BASIC which directory contains the file. The filename tells BASIC which file to look for.

If you specify a device name, BASIC ignores the directory and filename.

All device names end with a colon (:). The colon is part of the device name and you must include it whenever that device is specified.

Note: File specifications for communications files are different. The filename is replaced with a list of options specifying such things as line speed. See the OPEN "COM statement in the *BASIC Reference* for details.

If you use a string constant for the *pathname*, you must enclose it in quotation marks. For example,

```
KILL "data"
```

The directory is a list of directory names separated by slashes (/).

You can use pathnames for the following commands and statements:

BLOAD	MERGE
BSAVE	NAME
CHAIN	NEW
CHAIN MERGE	OPEN
KILL	RUN
LOAD	SAVE
REPLACE	STEP

Remember:

1. A complete *pathname* including a directory cannot contain more than 63 characters.
2. If you specify a device, BASIC ignores the directory and filename.
3. If you use a string constant for the path, you must enclose it in quotation marks. For example,

   ```
   "/sales/june/report/"
   ```

4. If you specify a file that is not in your current directory, you must supply BASIC with a path of directory names so it can locate the file.

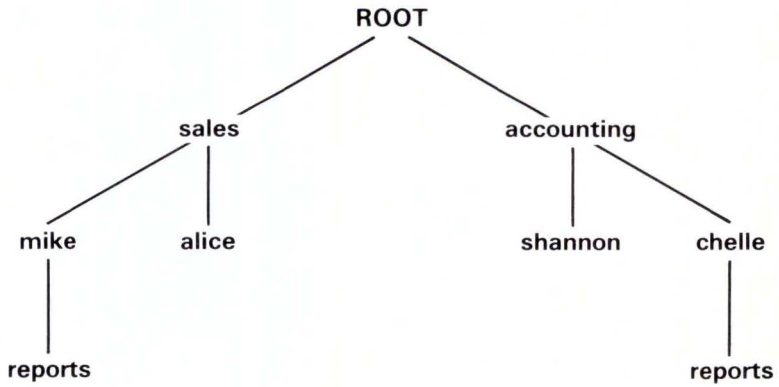
Tree-structured Directories

A *directory* is a collection of files; each *file* contains a packet of information such as a program, a letter, or a list of names and addresses. You can find out what files are in a directory from within the BASIC Interpreter by using the FILES Command.

The operating system uses a special kind of file called the *subdirectory*. Each subdirectory is itself a directory, containing any number of additional files and subdirectories.

The operating system allows you to manage files with directories known as *tree-structured directories*. These directories begin with the root directory (named with a /). In addition to containing the names of files, the *root* directory also contains the names of other directories called *subdirectories*. Unlike the root directory, these subdirectories are actually files. Each of them can contain any number of additional files and subdirectories–limited only by the amount of available space on the disk. You can find all available directories and files by searching from the root directory.

For example, if a company has two departments, sales and accounting, that share the computer, all the company's files can be kept on the computer's fixed disk. The organization of the files might look like this:



Naming Directories

Directory names are in the same format as filenames. All characters that are valid for filenames are valid for directory names. Each directory can contain file and directory names that also appear in other directories.

Current Directory

BASIC remembers which directory you are in when you start BASIC. This is the *current directory*. If you enter a filename without specifying the directory that contains the file, BASIC assumes that it is in the current directory. You can change the current directory by using the CHDIR statement.

If a filename is included in a path, it must be separated from the previous directory name by a slash (/). If a path begins with a slash, BASIC starts its search from the root directory; otherwise, the search begins at the current directory.

Accessing Another Device

You can access another device by using the operating system command called MOUNT. For example, imagine that you have four subdirectories called a, b, c, and d under your root directory and you want to access subdirectory c on another drive. You can mount the other drive under subdirectory d on the current drive. To access subdirectory c on the other drive, you type the following:

```
/d/c/filename
```

For more information on the MOUNT command, see the *IBM RT PC AIX Operating System: Commands Reference* manual.

Redirection of Standard Input and Output

In BASIC, you can redirect BASIC input and output (I/O). Standard input, normally read from the keyboard, can be redirected to any file you specify in the BASIC command line. Standard output, normally written to the screen, can be redirected to any file you specify in the BASIC command line.

```
basic[n] [>stdin][>][>stdout] [-r pathname]
```

When redirecting standard input and output, keep in mind the following:

- When input is redirected, all INPUT, INPUT$, INKEY$, and LINE INPUT statements are read from the specified input file, instead of from the keyboard.
- When output is redirected, all PRINT and WRITE statements and error messages write to the specified output file, instead of to the screen.
- Graphics statements are not allowed in a program with output redirected to a file.
- File output to SCRN: still goes to STDOUT.
- If standard input is redirected and there is no pathname on the command line, the input file must contain valid BASIC commands.

Some examples of redirection of I/O follow.

```
basic >data.out -r myprog
```

In this example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will continue to come from the keyboard. All screen output will go into the **data.out** file. This includes all data printed to the screen with the PRINT or WRITE statements, any error messages that BASIC prints to the screen, and any data entered as part of an input statement that is normally echoed to the screen.

```
basic <data.in
```

Here, commands and data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the **data.in** file. Data written by PRINT will continue to go to the screen.

```
basic <myinput.dat >myoutput.dat -r myprog
```

In this example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the **myinput.dat** file and data written by PRINT will go in the **myoutput.dat** file

```
basic </sales/john/trans >>/sales/sales.dat -r myprog
```

In the last example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the **/sales/john/trans** file. Data written by PRINT will be *appended* to the **/sales/sales.dat** file.

Note that these examples apply to both PC and Native Modes.

Calling C Functions from BASIC

BASIC lets you call C language functions that have been linked either with the Interpreter or with your BASIC Compiler program. See the **ld** command in the *IBM RT PC AIX Operating System: Commands Reference* manual for instructions on how to link.

To call a C language function, you must first declare the function in the BASIC program using the following syntax:

```
EXTERNAL function name[typech] [(parameter list)]
```

This EXTERNAL declaration must precede any executable statements in the program.

typech is optional. It declares the type of data returned by the function. *typech* is one of the following characters:

% function returns an integer value
! function returns a single-precision value
# function returns a double-precision value
$ function returns a pointer to char.

You can declare the data type of *function name* using the DEFtype statement. The DEFtype statement must precede the EXTERNAL declaration.

The default type for *function name* is single-precision.

(parameter list) must be of the form (var, var, var, ...).

The parameter list defines the type of data that BASIC passes to the C function. All elements in the parameter list must be simple integer or simple double-precision variables only. Integers will always suffice for passing addresses.

Calling C Functions

You can call a C function either as a single statement or as part of a larger statement. The following are examples of valid function calls:

```
50  EXTERNAL func%( x, y% )
100 func%( 34, v% )
120 v% = x% + func%( COS( y ) - z, 9 )
```

If necessary, BASIC converts the arguments you use to conform to the types declared in the parameter list.

You cannot use strings or arrays as arguments to C functions. You can, however, pass addresses. For arrays, you can use the VARPTR function to determine the address of the first element of the array, and pass that address along with the dimensions of the array to the C function. See Appendix B, "How Variables Are Stored," for information on how BASIC stores variables. See the following section for information on passing and returning strings.

Note: Event trapping is disabled while in external subroutines.

Passing Strings to C Functions

To pass a string to a C function, you must first use the CCHAR function. The format is as follows:

```
adr% = CCHAR( b$,c%() )
```

b$ represents the string you want to pass to the C function. CCHAR takes the string, converts it to a C format, null-terminated string, and stores it in a BASIC integer array, represented in this case by c%( ). After CCHAR has stored the string in the integer array, it returns the address of the first element of the array, c%(0). This is the address that you pass to the C function when you call it. Calling the function takes the following form:

```
100 Cfunc( adr% )        or        100 x = Cfunc( adr% )
```

For example, to use the C function strlen to determine the length of string xyz$, your BASIC program would read as follows:

```
10 EXTERNAL strlen%( v$ )          'Declare strlen%() as
                                   'an external function.

50 adr% = CCHAR( xyz$, xyz%()) 'Store xyz$ into xyz%()
                                   'and put the address of
                                   'xyz%() into adr%.

80 LenXYZ% = strlen%( adr% )       'Call strlen
```

Note that you can use the DIM statement to dimension the integer array prior to using it as an argument in CCHAR. If you do so, the size of the one-dimension array must be large enough to hold any strings assigned to it, including the terminating null character. If not, an **Array bounds exceeded** error occurs. If you decide not to dimension the integer array, CCHAR does so automatically, creating a one-dimension array just large enough to write the string to it.

Passing Strings from C Functions

If you type your C function as a pointer to char both in the BASIC program (with $) and in the C routine, the C function can return a string directly to BASIC. The format looks like this:

```
EXTERNAL Cfunc$() 'Cfunc$() declared a pointer to char
X$ = Cfunc$()     'Call Cfunc$(), returning a string
```

You can use the CVP$ function to convert any null-terminated C string into a BASIC string given the starting address of the C string as input. This takes the following form:

```
x$ = CVP$( StringAddress% )
```

If you have used CCHAR to store a BASIC string into a BASIC integer array, as explained earlier in this section, you can use the CHANGE statement to convert the integer array back to a BASIC string. Note that the last character of the integer array must be null. For example, the following two lines of BASIC are a roundabout way of writing the statement abc$ = xyz$.

```
100 adr% = CCHAR( xyz$, xyz%() )  'Copy xyz$ into xyz%
200 CHANGE xyz%() to abc$         'Copy xyz% into abc$
```

If you want to use the operating system subroutine strtok( ), your program might look something like this:

```
10   EXTERNAL strtok$( a$, b$ )
:
:
90   'This routine finds the tokens in an expression
91   'such as "a = b + c"
100  Seprtr% = CCHAR( "=+-*/ ", S%() )
110  i% = 0
120  Token$(i%) = strtok$(CCHAR(Exprsn$,E%()), Seprtr%)
130  WHILE Token$( i% ) <> ""
140        i% = i% + 1
150        Token$( i% ) = strtok$( 0, Seprtr% )
160  WEND
```

Demonstration Program

A demonstration program in BASIC and a C program are included when BASIC is installed to provide an example of linking external functions into the BASIC Interpreter. Use the following steps to run the demonstration program.

1. Change to the login directory and copy the necessary files to your directory:

```
$ cd
$ cp /usr/basic/clink.bas .
$ cp /usr/basic/strtok2.c .
```

2. Compile the C program to generate a **.o** file:

```
$ cc -c strtok2.c
```

3. Use the **fgrep** system command to create a file containing just the external statements, one per line:

```
$ fgrep external clink.bas > externals
```

4. Run the BASIC External Referencing Tool to produce **exfn.o**:

```
$ bert externals
```

5. Run the Interpreter Linker (select either PC Mode or Native Mode). Specify the object files or archives that contain the external references. Use the *−o* option to specify a name for the new BASIC Interpreter that you are creating:

   PC Mode

```
$ intlink strtok2.o -o mybasic
```

   Native Mode

```
$ intlinkn strtok2.o -o mybasicn
```

   You can use the strip command to reduce the size of the output files.

6. Run the new BASIC Interpreter:

```
$ mybasic
LOAD" CLINK
RUN
```

Chapter 6. Graphics

CONTENTS

Graphics

BASIC can display text, special characters, points, lines, and more complex shapes in one color or in full color. How much of this you can do depends on what hardware you have.

Text and Graphics Environments

BASIC offers two basic kinds of screen display environments, Text and Graphics.

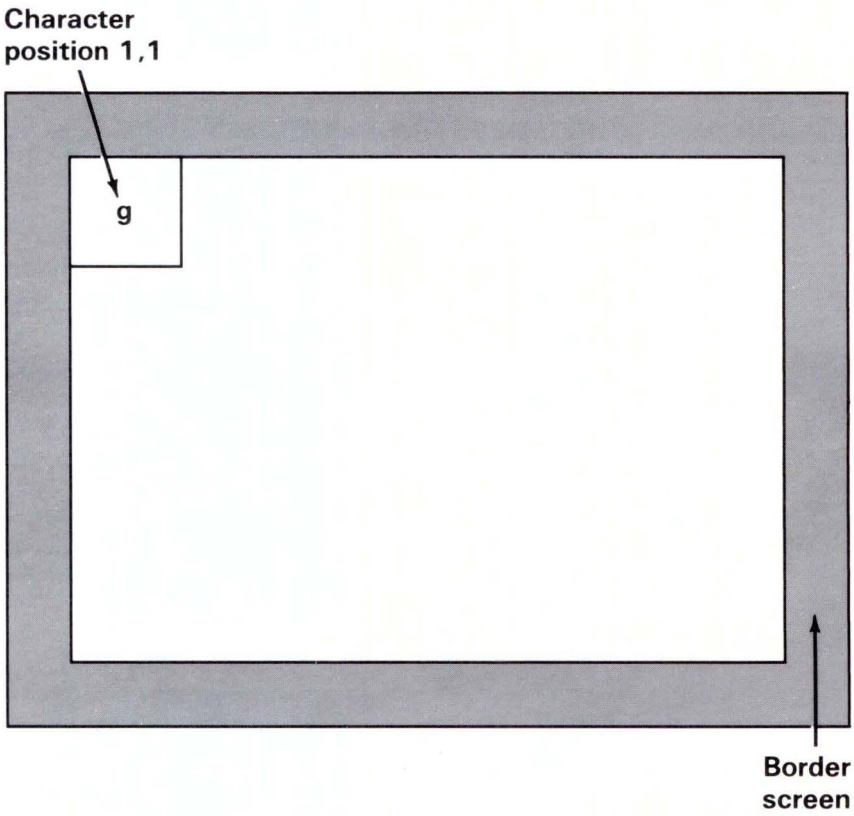
Text is the minimum environment provided for all displays. It is the default environment for BASIC. In it you can print letters, numbers, and all the special characters listed in Appendix B of the *BASIC Reference.* You can draw pictures using the special characters. You can also create blinking, reverse image, invisible, and highlighted characters by setting options in the COLOR statement.

If you have a color display and it is connected to a color adapter, you can vary the colors that appear on the screen. If you do not have color hardware, your screen can only show two colors: one dark, and one light.

The *graphics* environments require that you have the proper hardware, including a graphics adapter. The graphics environments let you draw complex pictures anywhere on the screen. They are more versatile for creating pictures than the text environment. For purposes of discussion in this handbook and in the *BASIC Reference*, the term *graphics* refers only to screen displays in a graphics environment. The use of the special characters listed in Appendix B of the *BASIC Reference* are not called graphics.

Text Environment

The text environment is the initial environment. The BASIC statement for it is SCREEN 0. In the text environment, the screen can be pictured like this:



Characters are shown in 25 horizontal lines across the screen. Line 1 is at the top; line 25 at the bottom. Each line has 80 character positions. These are numbered 1 to 80 from left to right. The line and position numbers are used in the LOCATE statement, and in the TAB, SCREEN, POS(0), and CSRLIN functions. For example, the character g in the upper left corner of the screen is on line 1, position 1. The VIEW and WINDOW statements, (used in graphics) do not change these line and position numbers.

Function Key Display

Line 25 is usually used for the function key display, but you can write text in this area of the screen if you turn off the display. See the KEY statement in the *BASIC Reference*. The 25th line is never scrolled by BASIC.

The function key display should not be used for an attached work station with a 24-line by 80-column screen. Applications designed for such terminals should restrict the use of the 25th line because the results are unpredictable.

Text Statements, Functions, and Variables

Statements you can use to display text are as follows:

CLS	PRINT
COLOR	WRITE
LOCATE	

The following functions and system variables can be used with text.

CSRLIN	SPC
POS	TAB
SCREEN	

These statements and functions also work the same in the graphics environments. They are, in general, not affected by the SCREEN, WINDOW, and VIEW Statements.

Text Colors

In the text environment, if you have color hardware, you may have access to the following 16 colors:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Yellow	14	High-intensity Yellow
7	White	15	High-intensity White

Note: The actual colors might vary depending on your hardware.

Character and Screen Colors

You can set the color for each text character, and for the box on the screen around the character. To do this, use the color numbers (listed with each color in the table above) in the COLOR *character%, screen%* statement. For example, the statement

```
Ok COLOR 6,4
```

will get you yellow (6) characters on a red (4) background. You can use all 16 colors for the characters. Only colors 0 through 7 are available for the screen. You can also make the characters blink by adding 16 to the color number.

Graphics Environments

The graphics environments let you draw lines, circles, graphs, and other complex figures on graphics displays. Graphics are available only if you have a graphics adapter.

BASIC offers several different graphics environments with varying degrees of resolution and numbers of colors available. These environments are set using the SCREEN statement (see "Setting Graphics Environments" later on in this chapter for more information).

Note: Graphics statements are not allowed when output is redirected to a file.

Graphics Statements and Functions

The BASIC statements used for graphics are as follows:

ASK	PRESET
CIRCLE	PSET
COLOR	PUT
DRAW	SCREEN
GET	VIEW
LINE	WINDOW
PAINT	

The graphics functions are as follows:

PMAP
POINT

Specifying Coordinates

BASIC graphics offers the flexibility to write powerful graphics routines expressed directly in the numbers you are working with. It also allows you to have your routines automatically adapt to a variety of terminals.

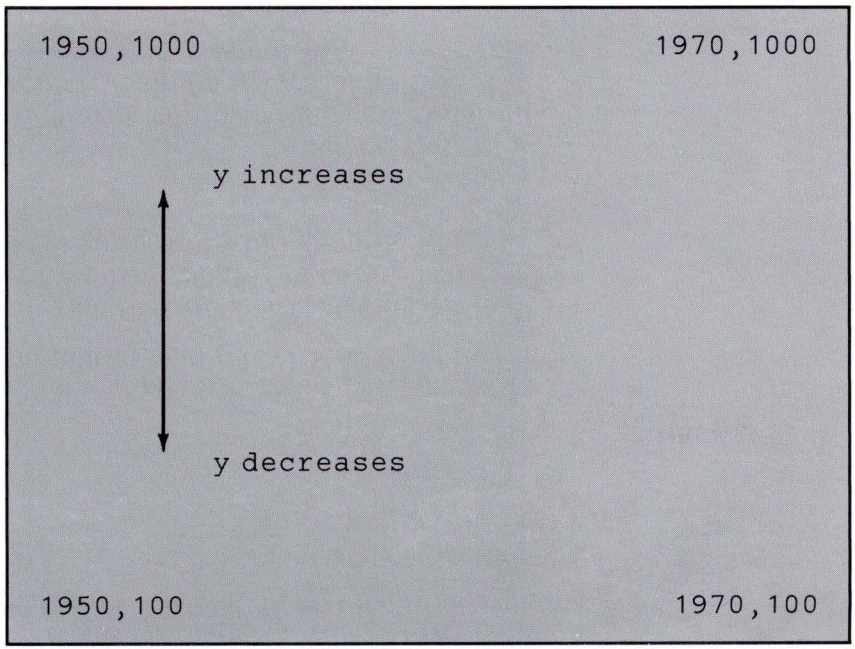
In order to do this, you should think not in terms of physical points on a screen, but of points on a plane that extends to infinity in all directions. This plane is called a *problem space* or a *world coordinate space.* BASIC references points on this plane using a *world coordinate system.* Conceptually, this system has an x and a y axis, each running from − infinity to + infinity.

Windows

Since the graphics in any one program need to refer only to a relatively small number of points on the infinite plane, it is useful to focus our attention on a rectangular section on the plane. This rectangular section, called a *window*, contains the points relevant to our program that appear on the screen. The dimensions of the window vary from program to program.

When you begin using graphics commands in a program, you should first define the x and y dimensions of the window you are using. You can also think of it as setting the scales of the x and y axes that you use to refer to points in the window. These window points may or may not correspond on a one-to-one basis to the physical points on your screen. BASIC converts them for you.

You define the x and y dimensions by using the WINDOW statement. For example, if your routine were to plot points representing stock market averages for the years 1950 to 1970, you might want to have a window extending from 1950 to 1970 on the x axis, and from 100 to 1000 on the y axis:



The statement to define the window would read:

```
Ok WINDOW (1950,100)-(1970,1000)
```

You can then plot points in the window using coordinates defined in terms of years and stock averages. For example, the statement

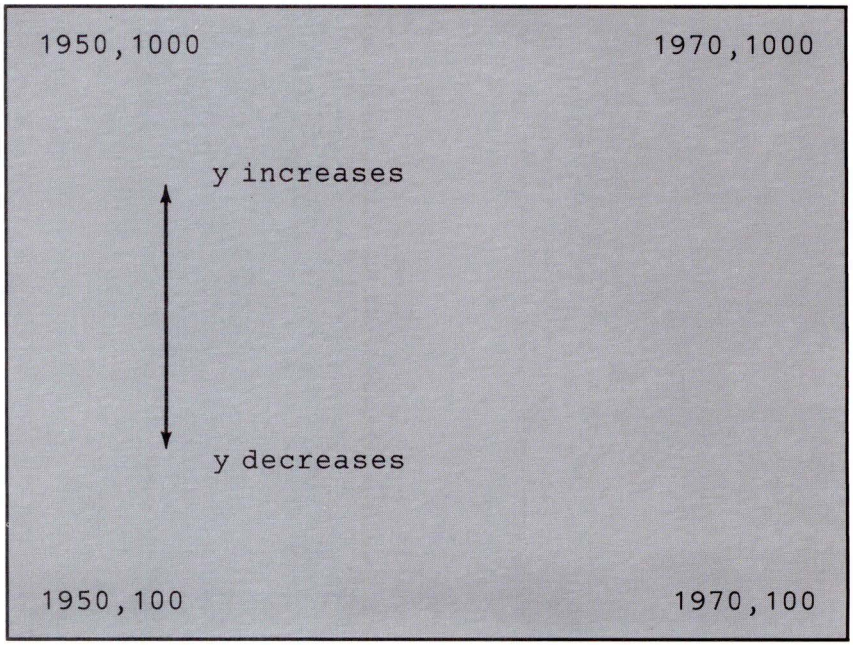
```
Ok PSET (1960,550)
```

would draw a point in the center of the window. Any points drawn beyond the edges of the window do not appear on the screen.

Adding the word SCREEN to the WINDOW statement sets the numerically lower ends of the X and Y axes in the upper left corner instead of the lower left corner. For example, the statement

```
Ok WINDOW SCREEN (1950,100)-(1970,1000)
```

defines this window setting:



Viewports

Normally, when you define a window, it fills the entire screen. It is possible, however, to place the window within a specific area of the screen. This area is called the viewport. It lets you place and use several windows on the screen, one after the other, without disturbing the rest of the screen.

Points plotted outside the viewport are *clipped*, that is they do not appear on the screen, but BASIC remembers what they were. hey may appear later on the screen if you zoom out by changing the WINDOW settings and redrawing the object. You may get an **Overflow** error if a WINDOW coordinate translates into a viewport coordinate exceeding plus or minus 32768.

You use the VIEW statement to set up a viewport and to tell BASIC where to place it on the screen. The (*x1,y1*) and (*x2,y2*) options of VIEW designate the upper left and lower right-hand corners of the viewport. They use the coordinates of the screen's current graphics environment, with the numbering starting at 0,0 in the upper left corner of the screen.

PC Mode: In PC Mode there are 320-by-200 points on the screen in medium resolution, and 640-by-200 points in high resolution. So, to place the previously mentioned stock average window in the right half of a medium resolution PC Mode screen, you would need the following statements:

```
10 SCREEN 1                          'Medium resolution
20 VIEW (160,0)-(319,199)            'Viewport on right
30 WINDOW (1950,100)-(1970,1000) 'Window in viewport
```

You can later reset the viewport to the left half of the screen, VIEW (0,0)-(159,199), to place and use the same or another window there.

Native Mode: In Native Mode, the number of points available on the screen is not predefined. It is a function of the number of physical points on the screen, and of the current graphics environment. Therefore, before you use the VIEW statement you need to find out how many points are actually available. You can do this by using an ASK statement, as shown in the following example.

```
10 SCREEN 1                  'Set graphics environment 1
20 ASK VIEW x%,y%            'Number of points available
30 VIEW (x%/2,0)-(x%-1,y%-1)     'Viewport on right
40 WINDOW (1950,100)-(1970,1000) 'Window in viewport
```

Note: You cannot use the ASK VIEW statement in the text environment (SCREEN 0).

Initial Coordinates

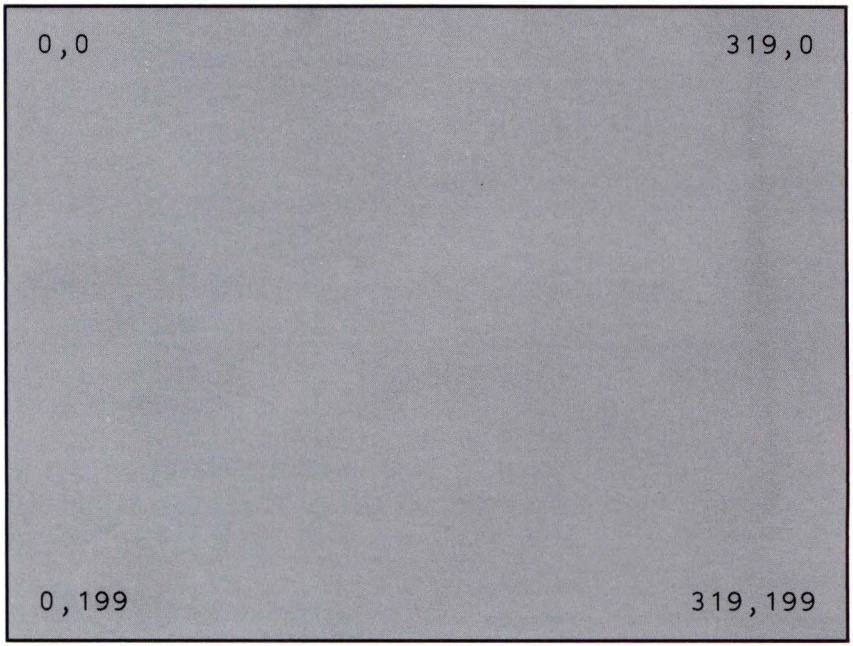
Whenever BASIC executes a SCREEN statement to put the screen in a graphics environment, it establishes the following initial conditions:

Viewport: The starting viewport is the entire screen.

Windows: The initial dimensions of the window are those of the screen in the current graphics environment. Numbering starts at (0,0) in the upper left corner of the screen.

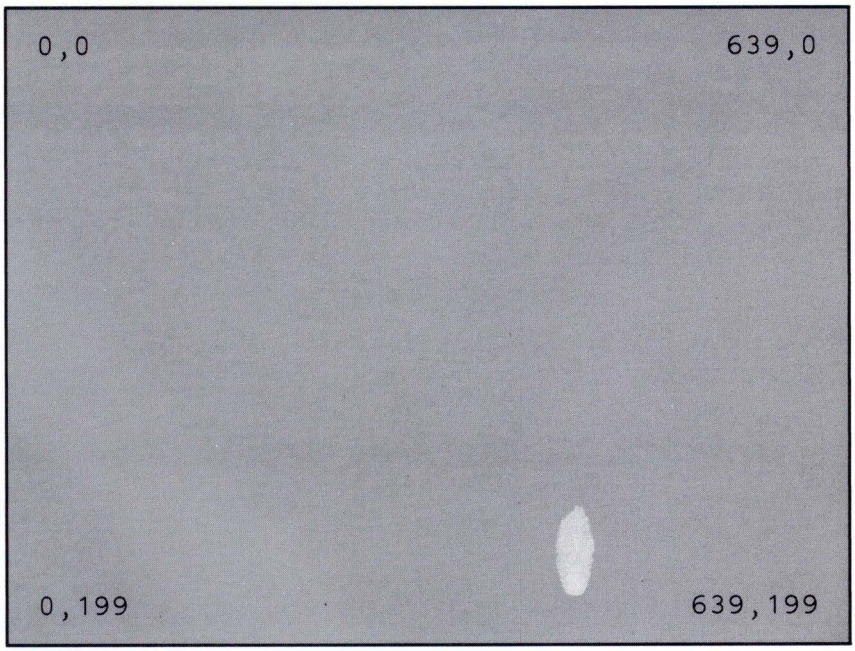
In **PC Mode**, the initial dimensions for medium resolution (SCREEN 1) are

```
Ok WINDOW SCREEN (0,0)-(319,199) 'initial window
```



In **PC Mode**, for high resolution (SCREEN 1), and

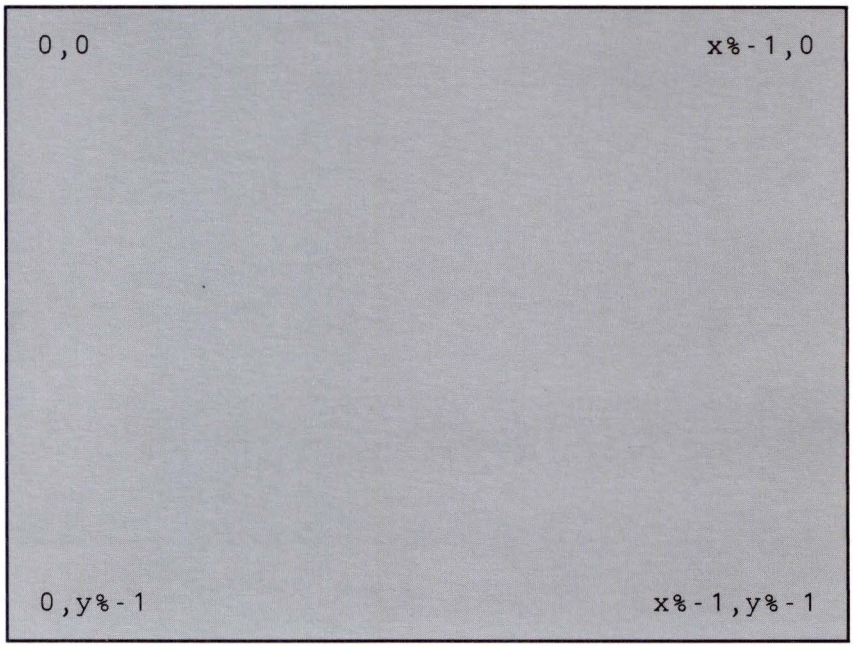
```
Ok WINDOW SCREEN (0,0)-(639,199) 'initial window
```



for high resolution (SCREEN 2).

In **Native Mode** the initial dimensions depend on your hardware configuration. Use the ASK VIEW statement to find out what they are. For example,

```
ASK VIEW x%,y%  'ASK gets the size of the screen
WINDOW SCREEN (0,0)-(x%-1,y%-1) 'initial window
```



Relative Coordinates (STEP)

Normally, when you specify a point on the screen, you give the coordinates in the form (x,y), where x is the horizontal position, and y is the vertical position. This form is known as the *absolute form* and refers to the actual coordinates of the point in the window.

There is another way to show coordinates, known as *relative form.* Using this form you tell BASIC where the point is, relative to the *last point referenced.* The last point referenced is the last point plotted by a graphics statement. Initially, after a SCREEN statement, the last point referenced is the center of the screen. After a CLS statement, it is the center of the viewport. Later statements change the last point referenced. What each graphics statement sets as the last point referenced is indicated in the discussion of that statement in the *BASIC Reference.* The last point referenced may fall outside the viewport.

To designate a point relative to the last point referenced, you use the keyword STEP:

```
STEP (xoffset,yoffset)
```

You indicate inside the parentheses the *offset* (distance) in the horizontal and vertical directions from the last point referenced.

The following example shows use of both forms of coordinates:

```
Ok List
   100 SCREEN 1
   110 WINDOW (1,1)-(100,100)
   120 PSET (20,70)                    'absolute form
   130 PSET STEP (10,-20)              'relative form
Ok
```

In this example, PSET (20,70) draws a point at window coordinates (20,70), which becomes the last point referenced. PSET STEP (10,-20) draws a point 10 points to the right, and 20 points down from the last point referenced, at coordinates (30,50).

Setting Graphics Environments

PC Mode and Native Mode use different methods to set the current graphics environment (that is, the number of points on the screen, and the colors available).

Before you run a BASIC program that uses a graphics terminal in text mode (SCREEN 0) you should set the system variable **$TERM** to the graphics terminal type to be used. For example:

```
$ TERM=ibm5154
$ export TERM
```

PC Mode Graphics

PC Mode graphics offers two graphics environments, medium resolution (SCREEN 1) and high resolution (SCREEN 2).

Medium Resolution: There are 320 horizontal and 200 vertical points on the screen in the PC Mode, medium resolution environment. Medium resolution is turned on using the SCREEN 1 statement as follows:

```
Ok SCREEN 1
```

If your hardware supports it, the medium resolution environment lets you work with four colors, numbers 0 through 3. The colors on your hardware might be as follows:

Number	Color
0	Black
1	Blue
2	Red
3	White

For example, the following statement sets the background color to red (2).

```
Ok COLOR 2
```

To draw a blue (1) circle, you can use the circle statement as follows:

```
Ok CIRCLE (150,100), 50, 1
```

You can still display text characters on the screen when you are in a graphics environment. The medium resolution environment can display 25 lines of 40 characters.

High Resolution: In high resolution there are 640 horizontal and 200 vertical points. High resolution is turned on by the SCREEN 2 statement.

High resolution has only two colors: black (0) and white (1). When you print text (characters) in high resolution, you get 80 characters per line.

Native Mode Graphics

Graphics Environments: A display with graphics capability may be able to handle several graphics environments in Native Mode. The environments available depend on your hardware. To find out what is possible with your hardware, use the ASK statement. For example, the statement

```
ASK SCREEN environ%
```

assigns to the variable environ% the number of graphics environments your hardware can handle. If environ% is 0, then the hardware can only show text. If, for example, environ% is 3, then the hardware can handle text, and three graphics environments: 2 color, 4 color, and 16 color.

You can then tell BASIC which graphics environment you want by using the SCREEN *n* statement. For example, if your hardware handles three graphics environments, (environ% = 3), the use of one of the following screen statements switches you into the corresponding environment:

```
SCREEN 0  :  Text environment
SCREEN 1  :  2  color graphics environment
SCREEN 2  :  4  color graphics environment
SCREEN 3  :  16 color graphics environment
```

The actual resolution available in each environment depends on your hardware. Use the ASK VIEW x%,y% statement to find out what it is. See "Viewports" earlier in this chapter.

Color: To find out how many colors are actually available in the current graphics environment, use the ASK COLOR statement. For example, the statement

```
ASK COLOR color%
```

assigns to the variable color% the number of colors available in the current graphics environment. If it is available, graphics environment 1 (SCREEN 1) has two colors available, black (0), and white (1). Graphics environment 2 (SCREEN 2), may have four colors available, black (0), blue (1), red (2), and white (3). Graphics environment 3 (SCREEN 3), may have available the 16 colors listed earlier under "Text Colors." What the actual colors are will depend on your hardware.

Viewports and Coordinates: Using the statement

```
ASK VIEW x%,y%
```

returns the number of points available on the x and y axes in the current graphics environment. You can use this information to set up the viewport, as explained earlier in "Viewports."

Text: Native Mode Graphics prints characters eight points high and eight points wide. As in the text environment, there are 80 character positions on 24 (25) text lines.

Note: You cannot use ASK VIEW in the text environment (SCREEN 0). If you do, an **Illegal Function Call** error occurs.

The following program prints out information about your hardware:

```
Ok LIST
   10 CLS                   'Clear the screen
   20 ASK SCREEN environ% 'Get number of environments
   30 PRINT
   40 PRINT "THE NUMBER OF ENVIRONMENTS IS:" ; environ%
   50 PRINT
   60 IF environ% < 1 THEN PRINT "TEXT ONLY" : END
   70 FOR E% = 1 TO environ%
   80 SCREEN E%
   90 PRINT "ENVIRONMENT #" ; E%
   100 ASK COLOR col%       'Get number of colors
   110 PRINT "COLORS :"
   120 FOR C% = 0 TO col% -1
   130   PRINT C%          'Print the color number
   140   LINE (20,C%*8)-(80,(C%+1)*8-1),C%,BF  'Draw a box
   150 NEXT C%
   160 PRINT
   170 ASK VIEW X%, Y%
   180 PRINT
   190 PRINT "THERE ARE " ; X% ; " HORIZONTAL POINTS"
   200 PRINT "AND " ; Y% ; "VERTICAL POINTS"
   210 PRINT
   220 INPUT "PRESS ENTER TO CONTINUE: " ; ANY$
   230 PRINT
   240 NEXT E%
   250 END
Ok
```

Appendixes

CONTENTS

Appendix A. BASIC Input and Output

This appendix describes procedures and special considerations for using input and output. It contains lists of the commands and statements that are used with files, and explanations of how to use them. Several sample programs are included to help clarify the use of data files. If you are new to BASIC or if you are getting file I/O errors, read these procedures and program examples thoroughly to make sure that you are using all the file I/O statements correctly.

Note: You will need to convert data files created under BASICA to IBM RT PC. Use the operating system dosread utility to do this.

Specifying File Names

Filenames for files must conform to the operating system naming conventions for BASIC to be able to read them. See "Naming Files" in Chapter 5 of this manual to be sure you are specifying your files correctly.

Data Files – Sequential and Random Input and Output

Two types of data files can be created and accessed by a BASIC program: sequential files and random access files.

Sequential Files

Sequential files are easier to create than random files, but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored sequentially, one item after another, in the order that it is written. Each item is read back in the same way, from the first item in the file to the last item.

The statements and functions used with sequential files are as follows:

CLOSE	LOF
EOF	NAME
INPUT #	OPEN
INPUT$	PRINT #
KILL	PRINT # USING
LINE INPUT #	WRITE #
LOC	

Creating and Accessing a Sequential File

To create a sequential file and access the data in it, include the following steps in your program:

1. Open the file for output or append using the OPEN statement.
2. Write data to the file using the PRINT #, WRITE #, or PRINT # USING statements.
3. To access the data in the file, you must close the file (using CLOSE) and reopen it for input (using OPEN).
4. Use the INPUT # or LINE INPUT # statements to read data from the sequential file into the program.

These steps are shown in PROGRAM1:

```
Ok LIST
   10 REM PROGRAM1 - SEQUENTIAL FILES
   20 OPEN "data" FOR OUTPUT AS #1          'STEP 1
   30 FOR I=1 TO 100 WRITE #1,I             'STEP 2
   50 NEXT
   60 CLOSE #1                              'STEP 3
   70 OPEN "I",#1,"data"
   80 WHILE NOT EOF(1)
   90 INPUT #1,A
   100 PRINT A
   110 WEND
   120 CLOSE:END
Ok
```

Notice the two ways of writing the OPEN statement in line 20 and line 70.

In line 80 the EOF function tests for end of file. This usage prevents an **Input past end** error.

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
```

can be used to write numeric data to the file without explicit delimiters. The space at the end of the format string separates the items in the file.

The LOC function, when used with a sequential file, returns the number of characters that have been written to or read from the file since it was opened. The LOF function returns the number of characters allocated to the file.

Adding Data to a Sequential File

To add data to a sequential file, you cannot simply open the file for output and start writing data. When you open a sequential file for OUTPUT, you erase its current contents. Instead you should open the file for APPEND. See the OPEN statement in the *BASIC Reference* for details.

Random Files

Creating and accessing random files requires more program steps than creating and accessing sequential files, but there are advantages to using random files. For instance, numbers in random files are usually stored in binary formats, while numbers in sequential files are stored as ASCII characters. Therefore, in many cases random files require less space than sequential files.

The biggest advantage to random files is that data can be accessed randomly. It is not necessary to read through all the files up to the point where the information is stored as with sequential files. This is possible because the information is stored and accessed in distinct units called records. Each record is numbered and of the same size, as specified by the programmer.

Records can be any length up to 32767 bytes. The size of a record is not related to the size of blocks in the file system.

The statements and functions used with random files are as follows:

CLOSE	LOF
FIELD	NAME
GET	OPEN
KILL	PUT
LOC	

Statements and functions used to convert and move data into or out of random file buffers are as follows:

CVD	MKD$
CVI	MKI$
CVS	MKS$
LSET/RSET/MID$	

Creating a Random File

The following program steps are required to create a random file:

1. Open the file for random access. For example, this line

```
20 OPEN "datay" AS #1 LEN=32                    'STEP 1
```

   specifies a record length of 32 bytes. If the record length is omitted, BASIC assumes that it is 128 bytes.

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

```
30 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$ 'STEP 2
```

3. Use LSET, RSET, or MID$ to move the data into the random 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$ for a single-precision value; and MKD$ for a double-precision value.

```
90  LSET N$ = X$                                'STEP 3
100 LSET A$ = MKS$(AMT)                         'STEP 3
```

4. Write the data from the buffer to the file using the PUT statement.

```
120 PUT #1, CODE%                               'STEP 4
```

These steps are shown in PROGRAM2.

Note: Do not use a string variable that has been defined in a FIELD statement in an input statement or on the left side of an assignment (LET) statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Examine PROGRAM2. It writes to a random file the information entered at the keyboard. Each time the PUT statement in line 120 executes, it writes a record to the file. The two-digit code that is input in line 40 becomes the record number.

```
Ok LIST
   10 REM PROGRAM2 - CREATE A RANDOM FILE
   20 OPEN "datay" AS #1 LEN=32              'STEP 1
   30 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$ 'STEP 2
   40 INPUT "2-DIGIT CODE"; CODE%
   50 IF CODE%=99 THEN CLOSE: END
   60 INPUT "NAME"; X$
   70 INPUT "AMOUNT"; AMT
   80 INPUT "PHONE"; TEL$ : PRINT
   90  LSET N$ = X$                          'STEP 3
   100 LSET A$ = MKS$(AMT)                   'STEP 3
   110 LSET P$ = TEL$                        'STEP 3
   120 PUT #1, CODE%                         'STEP 4
   130 GOTO 40
Ok
```

Accessing a Random File

The following program steps are required to access a random file:

1. Open the file for random access.

   ```
   20 OPEN "datay" as #1 LEN=32              'STEP 1
   ```

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

   ```
   30 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$ 'STEP 2
   ```

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

3. Use the GET statement to move the desired record into the random buffer.

```
60 GET #1, CODE%                                    'STEP 3
```

4. The data in the buffer can now be accessed by the program. Numeric values must be converted back to numbers using the convert functions: CVI for integers; CVS for single-precision values; and CVD for double-precision values.

```
70 PRINT N$                                         'STEP 4
80 PRINT USING "$$###.##"; CVS(A$)                  'STEP 4
```

These steps are shown in PROGRAM3.

PROGRAM3 accesses the random file that was created in PROGRAM2. When the two-digit code is entered at the keyboard, the information associated with that code is read from the file and displayed.

```
Ok LIST
   10 REM PROGRAM3 - ACCESS A RANDOM FILE
   20 OPEN "datay" as #1 LEN=32                     'STEP 1
   30 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$          'STEP 2
   40 INPUT "2-DIGIT CODE"; CODE%
   50 IF CODE% =  99 THEN CLOSE : END
   60 GET #1, CODE%                                 'STEP 3
   70 PRINT N$                                      'STEP 4
   80 PRINT USING "$$###.##"; CVS(A$)               'STEP 4
   90 PRINT P$
   100 GOTO 40
Ok
```

With random files, the LOC function returns the current record number. The current record number is the latest one used in a GET or PUT statement. For example, the statement

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

stops the program if the current record number in file #1 is higher than 50.

Appendix B. How Variables are Stored

This appendix contains information on how variables are stored for PC Mode and Native Mode.

PC Mode

PC Mode integers range from −32768 to +32767.

Integers are stored in two bytes aligned on a halfword boundary as:

a	a + 1
lsb	msb

lsb = least significant byte
msb = most significant byte

Single-precision floating point numbers are stored in four bytes aligned on a word boundary as:

a	a + 1	a + 2	a + 3
lsb	...	s|msb	exp

Exponent bias = 0X80 (128 decimal)
lsb = least significant byte
msb = most significant byte

Single-precision mantissa size = 24 bits ~ 7 digits.

For example,

0x00, 0x00, 0x00, 0x80 = 0.5 (1/2)

Double-precision floating point numbers are stored in eight bytes aligned on a word boundary as:

a	a + 1	a + 2	a + 3	a + 4	a + 5	a + 6	a + 7
lsb	...	...	...	...	...	s|msb	exp

Exponent bias = 0x80 (128 decimal)
lsb = least significant byte
msb = most significant byte

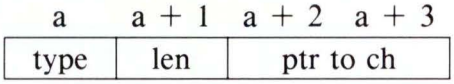
If exp <> 0, then there is a hidden most significant bit. This is conceptually overlaid by the sign bit.

Double-precision mantissa size = 56 bits ~ 17 digits.

For example,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x82 = 3

A string is described by a *String Descriptor Block*, or SDB. The layout of an SDB is:



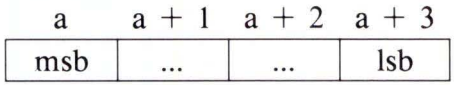
type occupies one byte and has values:

0 short string; characters are in a+2...
1 constant; ptr points into a constant area.
2 variable; ptr points into the heap.
3 temporary; ptr is an index into a table of pointers into the heap.
4 field; ptr points to a field descriptor block.

Native Mode

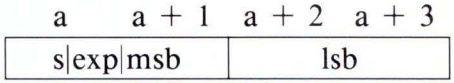
Native Integer range is −2,147,483,648 to +2,147,483,647.

Native integers are stored in four bytes aligned on a four-byte boundary as:



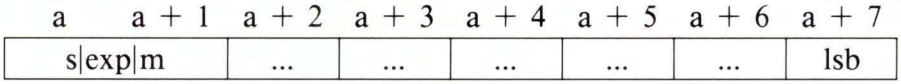
msb = most significant byte
lsb = least significant byte

IEEE FP (single-precision) numbers are stored in four bytes aligned on a four-byte boundary as:



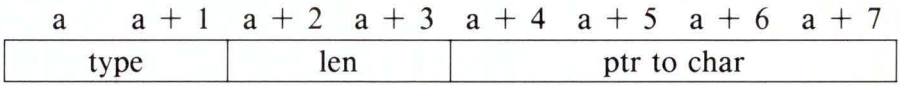
s = sign, 1 bit
exp = exponent, 8 bits
mantissa total size = 24 bits including hidden bit

IEEE DP (double-precision) numbers are stored in eight bytes aligned on a four-byte boundary as:



s = sign, 1 bit
exp = exponent, 11 bits
m = most significant part of mantissa, 4 bits
mantissa total size = 53 bitsincluding hidden bit

A string is described b a String Descriptor Block, or SDB. The layout of an SDB is:



type occupies two bytes and has values:

0 short string; characters are in a+4 ...
1 constant; ptr points into a constant area.
2 variable; ptr points into the heap.
3 temporary; ptr is an index into a table of pointers into the heap.
4 field; ptr points to a field descriptor block.

Appendix C. Communications

This appendix describes the BASIC statements required to support RS232 asynchronous communication with other computers and peripherals.

Opening a Communications File

OPEN "COM allocates a buffer for input and output in the same fashion as OPEN for other files.

Communications I/O

Since each communications adapter is opened as a file, all input/ouput statements that are valid for other files are also valid for communications.

Communications sequential input statements are the same as those for other files. They are

```
INPUT #
LINE INPUT #
INPUT$
```

Communications sequential output statements are also the same as those for other files, and are

```
PRINT #
PRINT # USING
WRITE #
```

See "INPUT" and "PRINT" for details of coding syntax and usage.

GET and PUT for Communications Files

GET and PUT are only slightly different for communications files than for other files. They are used for fixed length I/O from or to the communications file. Instead of specifying the record number to be read or written, you specify the number of bytes to be transferred into or out of the file buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM statement.

I/O Functions

The difficult aspect of asynchronous communication is processing characters as fast as they are received. At rates of 1200 bps or higher, it may be necessary to suspend character transmission from the other computer long enough to catch up. This can be done by sending XOFF (CHR$(19)) to the other computer and XON (CHR$(17)) when you are ready to resume. XOFF tells the other computer to stop sending, and XON tells it to start sending again.

Note: This is a commonly used convention, but it is not universal. Whether it is valid depends on the protocol implemented between you and the other computer or peripheral.

BASIC provides three functions that help determine when an overrun condition is likely to occur. They are:

LOC(f)
LOF(f)
EOF(f)

Note: A **Communications buffer overflow** error can occur if a read is attempted after the input buffer is full (that is, when LOF(f) returns 0).

INPUT$ Function

The INPUT$ function is preferred over the INPUT # and LINE INPUT # statements when reading communications files, since all ASCII characters may be significant in communications. INPUT # is least desirable because input stops when a comma or carriage return is seen. LINE INPUT # stops when a carriage return is seen.

INPUT$ allows all characters read to be assigned to a string. INPUT$(n,f) returns n characters from the #f file. The following statements can be used for reading a communications file:

```
10 WHILE NOT EOF( 1 )
20 A$ = INPUT$( LOC( 1 ), #1 )
    .
    .
    .
(process data returned as A$)
    .
    .
    .
100 WEND
```

When there are characters in the buffer, line 20 assigns them to A$, and they are processed. If there are more than 255 characters in the buffer, only 255 are returned at a time to prevent **String overflow**. Since EOF(1) is false, input continues until the input buffer is empty.

To process characters quickly, avoid examining every character as you receive it. If you are looking for special characters (such as control characters), you can use the INSTR function to find them in the input string.

A Sample Program

The following program allows the IBM RT PC computer to be used as a conventional dumb terminal in a full duplex mode. This program assumes a 300 bps line and an input buffer of 256 bytes.

```
Ok
    10 REM   dumb terminal example
    20 'set screen to text environment
    40 SCREEN 0
    50 'turn off key display; clear screen
    60 '   make sure all files are closed
    70 KEY OFF: CLS: CLOSE
    80 'define numeric variables as integer
    90 DEFINT A-Z
    100 'define true and false
    110 FALSE=0: TRUE=NOT FALSE
    120 'define the XON, XOFF characters
    130 XOFF$=CHR$(19): XON$=CHR$(17)
    140 'open communications to file number 1,
    150 ' 300 bps, EVEN parity, 7 data bits
    160 OPEN "COM1:300,E,7" AS #1
    170 'use screen as a file, just for fun
    180 OPEN "SCRN:" FOR OUTPUT AS #2
    190 'turn cursor on
    200 LOCATE ,,1
    400 PAUSE=FALSE: ON ERROR GOTO 9000
    490 '
    500 'send keyboard input to com line
    510 B$=INKEY$: IF B$<>"" THEN PRINT #1,B$;
    520 'if com buffer is empty, check key in
    530 IF EOF(1) THEN 510
    540 'if buffer more than 1/2 full, then
    550 'set PAUSE flag to say input suspended,
    560 'send XOFF to host to stop transmission
    570 IF LOC(1)>128 THEN PAUSE=TRUE
    575 PRINT #1,XOFF$
    580 'read contents of com buffer
```

```
590 A$=INPUT$(LOC(1),#1)
600 'remove linefeeds to avoid double spaces
610 'when input displayed on screen
620 LFP=0
625 'look for linefeed
630 LFP=INSTR(LFP+1,A$,CHR$(10))
640 IF LFP>0 THEN MID$(A$,LFP,1)=" "
645 GOTO 630
650 'display com input, and check for more
660 PRINT #2,A$;: IF LOC(1)>0 THEN 570
670 'if transmission suspended by XOFF,
680 'resume by sending XON
690 IF PAUSE THEN PAUSE=FALSE
695 PRINT #1,XON$;
700 'check for keyboard input again
710 GOTO 510
8999 'if error, print its number and retry
9000 PRINT "ERROR NO.";ERR: RESUME
Ok
```

Notes on the Program

- *Asynchronous* communication implies character I/O as opposed to line or block I/O. Therefore, all PRINTs (either to communications file or to screen) are ended with a semicolon. This stops the carriage return normally issued at the end of the list of values to be printed.
- Line 90, where all numeric variables are defined as integers, is coded because any program looking for speed optimization should use integer counters in loops where possible.
- Note that in line 510 that INKEY$ will return a null string if no character is pending.

Linking to Operating System Device Drivers

BASIC devices COM1: and COM2: are connected to operating system files **com1** and **com2**. When COM1: or COM2: is opened in a BASIC program, the current directory is searched for the appropriate system file and the connection is made. If the operating system file is not in the current directory, the file in the **/dev** directory is used. To link **com1** or **com2** to actual operating system device drivers, use the link command (see the *IBM RT PC AIX Operating System: Commands Reference*).

For example,

```
$ link /dev/tty1 /dev/com1
```

links COM1: to the operating system device driver **/dev/tty1** if the **com1** file in the **/dev** directory is used. If the **com1** file is in the current directory, the **link** command would be:

```
$ link /dev/tty1 com1
```

Two BASIC programs cannot link to the same system device driver at the same time. Two BASIC programs can, however, be using COM1: or COM2: concurrently if they are linked to different device drivers. This would be done by using **com1** or **com2** files that reside in different directories. For example, one program could link COM1: to **tty1** through a file in the current directory while another program could link COM1: to **tty3** through the **/dev** directory.

Appendix D. Control Sequences

This appendix lists the codes returned by the INKEY$ function when non-graphic keys are pressed on the keyboard. Graphic keys return the same character from INKEY$ as the character on the key which was pressed.

Key Name

The *Key Names* are shown as the name engraved on the key top or as a phrase describing the key function. **Ctrl** represents holding the **Ctrl** key while pressing the following key.

Code

The *code* values are given as the decimal integers returned by INKEY$. The values in the column labelled *Code* are the single values returned for the keys shown. Those in the column labelled *1-Code* are the second character of a two character sequence, the first character of which is always decimal value 1.

Use the LEN function to get the length of the string. Sequences longer than two characters are not listed here. Not all keyboards will return values for all possible key combinations, and some keyboards will return different values for some keys.

Key Name	Code	Key Name	Code
Ctrl 2	0	Ctrl J	10
Ctrl A	1	Ctrl K	11
Ctrl B	2	Ctrl L	12
Ctrl C	3	Ctrl M	13
Ctrl D	4	Ctrl N	14
Ctrl E	5	Ctrl O	15
Ctrl F	6	Ctrl P	16
Ctrl G	7	Ctrl Q	17

Key Name	Code	Key Name	Code
Ctrl H	8	Ctrl R	18
Ctrl I	9	Ctrl S	19
Ctrl T	20	Ctrl [	27
Ctrl U	21	Ctrl \	28
Ctrl V	22	Ctrl]	29
Ctrl W	23	Ctrl 6	30
Ctrl X	24	Ctrl -	31
Ctrl Y	25	Ctrl Backspace	127
Ctrl Z	26		

Key Name	1-Code	Key Name	1-Code
Break	1	Keypad Center	30
Cursor Down	2	Keypad Lower Left	31
Cursor Up	3	Keypad Lower Right	32
Cursor Left	4	Do	33
Cursor Right	5	Quit	34
Home	6	Command	35
Backspace	7	Previous Command	36
Delete Line	8	Next Pane	37
Insert Line	9	Previous Pane	38
Delete Character	10	Command Pane	39
Insert Mode On	11	End	40
Insert Mode Off	12	Help	41
Clear Screen	13	Select	42
Clear to End of Screen	14	Scroll Right	43
Clear to End of Line	15	Scroll Left	44
Scroll Forward	16	Tab	45
Scroll Backward	17	Back Tab	46
Next Page	18	New Line	47
Previous Page	19		
Set Tab Stop	20	Function Key 1	129
Clear Tab Stop	21	Function Key 2	130
Clear All Tabs	22	Function Key 3	131
Enter	23	Function Key 4	132
Soft Reset	24	Function Key 5	133
Hard Reset	25	Function Key 6	134
Print or Copy	26	Function Key 7	135
Last Line	27	Function Key 8	136
Keypad Upper Left	28	Function Key 9	137
Keypad Upper Right	29	Function Key 10	138

Appendix E. Converting Programs to IBM RT PC BASIC

Since IBM RT PC BASIC is very similar to many other microcomputer BASICS, many BASIC programs written for other microcomputers can be run using IBM RT PC BASIC. Some minor adjustments may be necessary before running them with IBM RT PC BASIC. Here are some specific things to look for when converting BASIC programs.

File I/O

In IBM RT PC BASIC, you read and write information to a file by opening the file to associate it with a particular file number; then using the I/O statements that specify the file number. File I/O is implemented differently in some other BASICs. Refer to the section in Chapter 5 called "Files," and to the OPEN statement in the *BASIC Reference* for more specific information.

Graphics

How you draw on the screen varies greatly between different BASICs. Refer to Chapter 6 for specific information about IBM RT PC BASIC graphics.

IF...THEN

The IF statement in IBM RT PC BASIC contains an optional ELSE clause, which is performed when the expression being tested is false. Some other BASICs do not have this capability. For example, in another BASIC you may have:

```
10 IF A=B THEN 30
20 PRINT "NOT EQUAL" : GOTO 40
30 PRINT "EQUAL"
40 REM CONTINUE
```

This sequence of code functions correctly in IBM RT PC BASIC, but it may also be conveniently recoded as follows:

```
10 IF A=B THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

IBM RT PC BASIC also allows multiple statements in both the THEN and ELSE clauses. This may cause a program written in another BASIC to perform differently. For example,

```
10 IF A=B GOTO 100 : PRINT "NOT EQUAL"
20 REM CONTINUE
```

In some other BASICs, if the test A=B is false, control branches to the next statement; that is, if A is not equal to B, "NOT EQUAL" is printed. In IBM RT PC BASIC, both GOTO 100 and PRINT "NOT EQUAL" are considered to be part of the THEN clause of the IF statement. If the test is false, control continues with the next program line; that is, to line 20 in this example. PRINT "NOT EQUAL" does not execute.

This example can be recoded in IBM RT PC BASIC as follows:

```
10 IF A=B THEN 100 ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

Line Feeds

In other BASICs, the line feed character is used to continue a statement on the next physical line. In IBM RT PC BASIC, if the underscore appears as the last character on a line, it is interpreted as a line continuation character.

Logical Operations

In IBM RT PC BASIC, logical operations (NOT, AND, OR, XOR, IMP, and EQV) are performed bit-by-bit on integer operands to produce an integer result. In some other BASICs, the operands are considered to be simple true (non-zero) or false (zero) values, and the result of the operation is either true or false. As an example of this difference, consider this small program:

```
10 A=9: B=2
20 IF A AND B THEN PRINT "BOTH A AND B ARE TRUE"
```

This example in another BASIC will perform as follows: A is non-zero, so it is true; B is also non-zero, so it is also true, so the program prints BOTH A AND B ARE TRUE.

However, IBM RT PC BASIC calculates it differently: A is 1001 in binary form, and B is 0010 in binary form, so A AND B (calculated bit-by-bit) are 0000, or zero; zero indicates false, so the message is not printed, and the program continues with the next line.

This can affect not only tests made in IF statements, but calculations as well. To get similar results, recode logical expressions like the following:

```
10 A=9: B=2
20 IF (A<>0) AND (B<>0)
      THEN PRINT "BOTH A AND B ARE TRUE"
```

MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to work properly.

Multiple Assignments

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. IBM RT PC BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With IBM RT PC BASIC, be sure all statements on a line are separated by a colon(:).

PEEKs and POKEs

Many PEEKs and POKEs are dependent on the particular computer you are using. You should examine the purpose of the PEEKs and POKEs in a program in another BASIC, and translate the statement so it performs the same function in IBM RT PC BASIC.

In particular, look at IBM RT PC BASIC's External Function's capability. An external function may be the best way to carry out the function you require.

Relational Expressions

In IBM RT PC BASIC, the value returned by a relational expression, such as A>B, is either –1, indicating the relation is true, or 0, indicating the relation is false. Some other BASICs return a positive 1 to indicate true. If you use the value of a relational expression in an arithmetic calculation, the results are likely to be different from what you want.

Remarks

Some BASICs allow you to add remarks to the end of a line using the exclamation point (!). Be sure to change this to a single quote (') when converting to IBM RT PC BASIC.

Rounding of Numbers

IBM RT PC BASIC rounds single- or double-precision numbers when it requires an integer value. Many other BASICs truncate instead. This can change the way your program runs because it affects not only assignment statements (for example, I%=2.5 results in I% equal to 3), but also affects function and statement evaluations (for example, TAB(4.5) goes to the fifth position, A(1.5) is the same as A(2), and X=11.5 MOD 4 will result in a value of 0 or X). Note in particular that rounding may cause IBM RT PC BASIC to select a different element from an array, possibly one that is out of range.

Scan Codes

Scan codes (keyboard codes) are not used in IBM RT PC BASIC. Instead, control sequences are passed to BASIC by the operating system. Refer to the KEY statement, ON KEY statement, and the INKEY$ variable for information on using control sequences. The control sequences passed to BASIC are listed in Appendix D of this manual.

Sound Statement

In IBM RT PC BASIC, clock ticks occur at a rate of 128 times per second. You may need to adjust the *duration* per second to get sounds of the same length.

Sounding the Bell

Some BASICs require PRINT CHR$(7) to send an ASCII bell character. In IBM RT PC BASIC, you may replace this statement with BEEP, although it is not required.

String Handling

String Length: Since strings in IBM RT PC BASIC are all variable lengths, you should delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the IBM RT PC BASIC statement DIM A$(J).

Concatenation: Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for IBM RT PC BASIC string concatenation.

Substrings: In IBM RT PC BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the character at position I in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	**IBM RT PC BASIC**
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J−I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	**IBM RT PC BASIC**
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J−I+1)=X$

Use of Blanks

Some BASICs allow statements with no separation of keywords:

```
20FORI=1TOX
```

With IBM RT PC BASIC be sure all keywords are separated by a space:

```
20 FOR I=1 TO X
```

Other

The BASIC language on another computer may be different from IBM RT PC BASIC in other ways than those listed here. You should become familiar with IBM RT PC BASIC as much as possible in order to be able to appropriately convert any function you may require.

Appendix F. BASIC Installation

The IBM RT PC BASIC Licensed Program Product provides both the BASIC Interpreter and the BASIC Compiler. This appendix explains how to install the IBM RT PC BASIC Licensed Program Product. Before you install BASIC on your system, you must have the AIX Operating System installed.

Installation Procedure

This section describes the installation procedure. If any error messages occur during the procedure, see the *IBM RT PC Messages Reference.*

Remove the BASIC diskettes from the plastic envelope in the back of the binder. Perform the following steps to install BASIC on your system.

1. Make sure that no one else is using the system and that no user programs are running. If the system is not in a quiet state, problems may occur as you install the various files for your licensed program product.
2. Log in as superuser or as a member of the system group. You must have superuser authority or be a member of the system group to install a licensed program product. See *IBM RT PC Using and Managing the AIX Operating System* for more information. After you log in, you will see the # prompt.
3. Type **installp** command. Then press **Enter**.

   ```
   # installp
   ```

4. The following message appears to remind you to make sure that the system is quiet:

```
000-123  Before you continue, you must make sure there is no other
         activity on the system. You should have just restarted the system,
         and no other users should be logged on. Refer to your messages
         reference book for more information.

         Do you want to continue with this command? (y or n)
```

   Type **y**, and press **Enter** to continue with the **installp** command.

5. Insert your program diskette volume 1 in response to the prompt. Then press **Enter**. You will be asked to insert subsequent volumes at the appropriate time.

```
Insert the program diskette into diskette drive
"/dev/rdf0" and then press Enter.
```

6. In response to the prompt, type **y** to indicate that you wish to continue with the installation. Then press **Enter**.

```
The program "BASIC Interpreter and Compiler"
will be installed.

Do you want to do this? (y or n)
```

7. If a version of this program has already been installed on your system, a message explains that the version of the program you are about to install is the same as or older than the version you already have installed on your system. Indicate whether you wish to go ahead with the installation.

```
You are about to install version "01.00.000" of this program.
This version is the same as or older than the version currently
on your system. Do you want to do this? (y/n)
If you type y and press Enter, the
installation process begins.
Please mount volume 1 on /dev/rfd0
```

   Your program diskette should already be in the diskette drive (/dev/rfd0). **Type return** is the same as **Press Enter**. As installation continues, various files are listed on the screen as they are copied to the fixed disk.

8. When installation is complete, remove the program diskette from the diskette drive and replaceit in its protective envelope in the binder.

```
The installation process has completed.
```

9. Log off as superuser or as a member of the system group.

You may now begin using your LPP.

Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

IBM RT PC
Programming Family

Reader's Comment Form

IBM RT PC BASIC
Language Handbook

SV21-8019

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding setup, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTA(
NECESSAI
IF MAILEI
IN THE
UNITED STA

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

tape

Cut or Fold Along Line

tape

Please Do Not Staple

IBM RT PC
Programming Family

Reader's Comment Form

IBM RT PC BASIC
Language Handbook

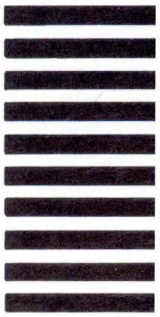
SV21-8019

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding setup, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTA
NECESSAI
IF MAILE
IN THE
UNITED STA



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

IBM RT PC BASIC Language Handbook SV21-8019

Book Evaluation Form

Your comments can help us produce better books. You may use this form to communicate your comments about this book, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Please take a few minutes to evaluate this book as soon as you become familiar with it. Circle Y (Yes) or N (No) for each question that applies and give us any information that may improve this book.

Y N Is the purpose of this book clear?

Y N Is the table of contents helpful?

Y N Is the index complete?

Y N Are the chapter titles and other headings meaningful?

Y N Is the information organized appropriately?

Y N Is the information accurate?

Y N Is the information complete?

Y N Is only necessary information included?

Y N Does the book refer you to the appropriate places for more information?

Y N Are terms defined clearly?

Y N Are terms used consistently?

Y N Are the abbreviations and acronyms understandable?

Y N Are the examples clear?

Y N Are examples provided where they are needed?

Y N Are the illustrations clear?

Y N Is the format of the book (shape, size, color) effective?

Other Comments

What could we do to make this book or the entire set of books for this system easier to use?

Optional Information

Your name _____

Company name _____

Street address _____

City, State, ZIP _____

No postage necessary if mailed in the U.S.A.

NO POSTAG
NECESSAR
IF MAILED
IN THE
UNITED STA

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

tape

Cut or Fold Along Line

International Business
Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Printed in the
United States of America

59X9289

About This Section

This section contains three new Usability Services commands that appear when you install the IBM RT PC BASIC Interpreter and Compiler Licensed Program Product. These commands are:

- **BASCOMPILE**
- **BASINTERPRET**.
- **BASNINTERPRET**.

You should read these pages before using your IBM RT PC BASIC Interpreter and Compiler Licensed Program Product with the Usability Services command interpreter. Insert this section in the chapter of your *Usability Services Reference* called "Licensed Program Product Commands."

BASCOMPILE—Using the BASIC Compiler

BASCOMPILE allows you to either compile BASIC source files or compile and link edit the BASIC source files and object files you specify. **BASCOMPILE** uses the AIX command **basicn -c**.

You can also use **COMPILE** to compile BASIC source files from the FILES window. See a description of **COMPILE** in the "Commands" chapter of *Usability Services Reference*. See the description of the **LINKEDIT** command in the "Commands" chapter of *Usability Services Reference* if you want to link edit BASIC object files separately from the **BASCOMPILE** command.

Steps for Compiling BASIC Source Files

1. Display a TOOLS window.
2. Select `PROGRAM DEVELOPMENT`. Then select `OPEN`
3. Select `BASCOMPILE`. Then select `RUN`.
4. Make the choices and press **Do** in each pop-up to run the command.
5. Follow the prompts on the display screen.

More Detailed Information

1. To display a TOOLS window:

   - If the window is already open, press the **Next Window** key until the window appears.

     OR

   - Open a window:

a. Select TOOLS from the Window Types pane of the WINDOWS window. The command bar changes to show you the commands that you can use with the selected window.

b. Select OPEN from the command bar. A TOOLS window appears on the screen.

2. From the TOOLS window:

   a. Select PROGRAM DEVELOPMENT. The command bar changes.

   b. Select OPEN from the command bar. The Program Development Tools Group appears.

3. From the Program Development Tools Group:

   a. Select BASCOMPILE. The command bar changes.

   b. Select RUN from the command bar. A pop-up appears that contains the choices for **BASCOMPILE**.

4. Make the choices and press **Do** in each pop-up. Depending on your choices, one or more pop-ups may appear. Pressing **Do** in the last remaining pop-up on the screen runs the command with your choices. Pressing **Quit** cancels a pop-up without saving your choices.

   The choices are:

   - Source File is the file you want to compile and link edit.

   - Use PC BASIC lets you decide whether or not to compile the program using features similar to PC Advanced BASIC (BASICA). Using this option may enable you to run a program that was written on a PC with fewer changes to the code than if you run without this option (in native mode). The default for this field is No.

   - Save Listing File allows you to save the listing file that the compiler produces. This listing file contains a formatted copy of your source program and shows any syntax errors in the program when and where they occur. The default value is Yes. If you select Yes, a pop-up appears that contains the name of the listing file

produced by the compiler. The default name is the source file name with the suffix `.lst` attached. You can change this name.

- `Linkedit After Compile` allows you to build a run file from the specified object files and any object files produced by the compile process. The default value is `Yes`. If you do not want to link edit the files now, you can use **LINKEDIT** to link them later. See the "Commands" chapter of *Usability Services Reference* for details of this command. If you select `Yes`, a pop-up appears. In this pop-up, you have the following choices:

  - `Other Object Files` are additional object files that you want to link edit along with any generated by the **COMPILE** command. You can use the pattern-matching characters *****, **?**, **[ ]**, and **!** to specify multiple object file names. See the discussion of pattern-matching characters in the "Using Commands" chapter of *Usability Services Reference* for more information.

  - `Library Directory` is the full path name of the directory that contains the libraries you want to use. Only one library directory can be specified.

  - `Library Names` are the names of the libraries that contain the files you want to use. Separate each name with a space. You can use the pattern-matching characters *****, **?**, **[ ]**, and **!** to specify multiple library names. See the discussion of pattern-matching characters in the "Using Commands" chapter of *Usability Services Reference* for more information.

  - `Run File` is the name for the single file created when the object files are link edited. The name you specify must contain the `.out` suffix.

- `Direct Output To` lets you decide to send the output of the command to the screen, the printer, or a file. By default, all output or messages go to the screen. If you select `File`, a pop-up asks you to type in the file name. If you select `Printer`, the output is sent to the printer.

5. Follow the prompts on the display screen. See "Running a Command in a FILES, TOOLS or APPLICATIONS Window" in *Usability Services Reference* for more details on the prompts.

BASINTERPRET—Running the BASIC Interpreter in PC or Native Mode

BASINTERPRET runs the BASIC interpreter, which allows you to enter one or more BASIC statements and run them immediately, or to run a program you have already written and saved in a file. You can run the interpreter in two different modes: either in PC Mode, which provides an environment similar to that of IBM PC Basic, or in Native Mode, which interprets your program using the full power available to you on your RT PC system. See "More Detailed Information" below for a further discussion of modes. **BASINTERPRET** is the Usability Services version of the AIX command **basic** or **basicn**.

You can run this command from the Program Development Tools Group of the TOOLS window (with either keyboard or file input, in either mode), or from the Applications Tools Group of the TOOLS window or from the APPLICATIONS window (with keyboard input, in PC Mode only). If you run this command from the Applications Tools Group or from the APPLICATIONS window, no pop-ups with choices appear.

You can also use **INTERPRET** to interpret BASIC source files from the FILES window. See a description of **INTERPRET** in the chapter called "Commands" in *Usability Services Reference*.

Steps for Using the BASIC Interpreter

1. Display a TOOLS window.
2. Select PROGRAM DEVELOPMENT. Then select OPEN
3. Select BASINTERPRET. Then select RUN.
4. Make the choices and press **Do** in each pop-up to run the command.
5. Follow the prompts on the display screen.

More Detailed Information

1. To display a TOOLS window:

   - If the window is already open, press the **Next Window** key until the window appears.

     OR

   - Open a window:

     a. Select TOOLS from the Window Types pane of the WINDOWS window. The command bar changes to show you the commands that you can use with the selected window.

     b. Select OPEN from the command bar. A TOOLS window appears on the screen.

2. From the TOOLS window:

   a. Select PROGRAM DEVELOPMENT. The command bar changes.

   b. Select OPEN from the command bar. The Program Development Tools Group appears.

3. From the Program Development Tools Group:

   a. Select BASINTERPRET. The command bar changes.

   b. Select RUN from the command bar. A pop-up appears that contains the choices for **BASINTERPRET**.

4. Make the choices and press **Do** in each pop-up. Depending on your choices, one or more pop-ups may appear. Pressing **Do** in the last remaining pop-up on the screen runs the command with your choices. Pressing **Quit** cancels a pop-up without saving your choices.

   The choices are:

- `Source File` is the name of the BASIC file that you want to interpret. You can type in only one name.

- `Use PC BASIC` lets you decide whether or not to interpret the program using features similar to PC Advanced BASIC (BASICA). BASICA enables you to run a program from a PC with few changes to the code. The default for this field is `No`.

- `Accept Input From` lets you decide if the program will be input from the keyboard or from a file. By default, the program is input from the keyboard. If you specify `File` input, a pop-up appears that lets you type in one file name.

- `Direct Output To` lets you decide whether to send your output to the screen, the printer, or a file. By default, all output is sent to the screen. If you select `Printer`, the output is sent to the printer. If you select `File`, a pop-up asks you to type in the file name. The file may be an existing file or a new file. If the file already exists, the output of this command will replaces and destroys the contents of the file.

5. Follow the prompts on the display screen. See "Running a Command in a FILES, TOOLS or APPLICATIONS Window" in *Usability Services Reference* for more details on the prompts.

BASNINTERPRET—Running the BASIC Interpreter in Native Mode

BASNINTERPRET runs the BASIC interpreter in Native Mode and allows you to enter one or more BASIC statements and run them immediately. Running in Native Mode enables you to interpret your program using the full power available to you on your RT PC system. **BASNINTERPRET** is the Usability Services version of the AIX command **basicn**.

You can run this command from the Applications Tools Group of the TOOLS window, or from the APPLICATIONS window. No pop-ups with choices appear when you run this command.

Use the **BASINTERPRET** command to interpret BASIC source files. See "BASINTERPRET—Running the BASIC Interpreter in PC or Native Mode" on page BC-6 for a description of this command. You can also use **INTERPRET** to interpret BASIC source files from the FILES window. See a description of **INTERPRET** in the chapter called "Commands" in *Usability Services Reference*.

The steps required for running **BASNINTERPRET** from the Applications Tools Group of the TOOLS window appear in the box below.

Steps for Using the BASIC Interpreter

1. Display a TOOLS window.
2. Select `APPLICATIONS`. Then select `OPEN`.
3. Select `BASNINTERPRET`. Then select `RUN`.
4. For more information on running this command, consult the reference books that came with your Licensed Program Product.

More Detailed Information

1. To display a TOOLS window:

   - If the window is already open, press the **Next Window** key until the window appears.

     OR

   - Open a window:

     a. Select TOOLS from the Window Types pane of the WINDOWS window. The command bar changes to show you the commands that you can use with the selected window.

     b. Select OPEN from the command bar. A TOOLS window appears on the screen.

2. From the TOOLS window:

   a. Select APPLICATIONS. The command bar changes.

   b. Select OPEN from the command bar. The APPLICATIONS Tools Group appears.

3. From the APPLICATIONS Tools Group:

   a. Select BASNINTERPRET. The command bar changes.

   b. Select RUN from the command bar.

4. For further information on running this command, refer to *BASIC Language Handbook* or *BASIC Language Reference*. These books came with your IBM RT PC BASIC Interpreter and Compiler Licensed Program Product.

Notes:

Notes: