Reference
Manual

AC/BASIC™

16-Bit BASIC Compiler for Apple II GS

absoft

High Performance Scientific/Engineering Software

AC/BASIC™

Compiler Reference Manual

absoft

High Performance Scientific/Engineering Software

NOTICE

All rights reserved. No part of this publication may be reproduced or used in any form by any means, without the prior written permission of Absoft Corporation.

The information contained in this publication is believed to be accurate and reliable. However, Absoft Corporation makes no representation of warranties with respect to the program material decscribed herein and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Absoft reserves the right to revise the program material and make changes therein from time to time without obligation to notify the purchaser of the revisions or changes. In no event shall Absoft be liable for any incidental, indirect, special or consequential damages arising out the purchaser's use of the program material.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Apple IIGS System Disk are copyrighted programs of Apple Computer, Inc. licensed to Absoft Corporation to distribute for use in any combination with AC/BASIC. Apple Software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of the execution of AC/BASIC. When AC/BASIC has completed execution Apple Software shall not be used by any other program.

AC/BASIC is a trademark of Absoft Corporation.
Apple, ImageWriter, LaserWriter, and ProDOS are registered trademarks of Apple Computer, Inc.
Macintosh and Apple IIGS are trademarks of Apple Computer, Inc.
Amiga is a trademark of Commodore-Amiga, Inc.
ORCA/M is a registered trademark of The Byte Works, Inc.
Microsoft is a registered trademark of Microsoft Corporation.

AC/BASIC™

AC/BASIC is a must for anyone writing BASIC programs.

AC/BASIC is the only BASIC compiler for the Apple II GS that is compatible with Microsoft BASIC programs written for the Macintosh.

High Performance
- Compatible with Microsoft BASICs
- True 16-bit compiler
- Unlimited program size
- Unlimited data and string space size
- IEEE math or Decimal math
- 32 and 64 bit floating point support
- Constant-folding optimizations
- Dynamic arrays and 32K strings

Structured Programming Features
- Independent subprograms
- Alphanumeric labels
- Block IF...THEN constructs
- FOR...NEXT constructs
- Global variable declaration for subprograms
- WHILE...WEND loops
- CHAIN to other programs

Graphics Features
- Full WINDOW and BUTTON support
- Create custom pull-down menus
- Both 320x200 and 640x200 Super Hi-Res
- CIRCLE, LINE and color PALETTE
- No Toolbox knowledge required
- Full color

Sound Features
- Powerful SOUND statement
- Create custom waveforms and instruments
- Music sequenced in background automatically

Options for Increased Flexibility
- "Include" files
- Batch compilation
- Programs do not require line numbers
- Supports alphanumeric labels
- Supports both sequential and random-access files
- Code is position independent and reentrant
- Support for assembly-language subroutines
- Develop stand-alone applications
- No license fee

Compile Time Options
- Compile for BCD math
- Generate error list
- Generate full list
- Process run-time events
- Link run-time library to object code
- Suppress default window and/or menu bar

System Requirements
- 512K RAM
- One 3.5 inch disk drive

2781 Bond Street
Auburn Hills, MI 48057
(313) 853-0050

absoft

High Performance Scientific/Engineering Software

AC/BASIC is a trademark of Absoft Corp. Apple IIGS and Macintosh are trademarks of Apple Computer. MS and Microsoft are trademarks of Microsoft Corporation.

TABLE OF CONTENTS

CHAPTER 1 ..... 1 - 1
Introduction ..... 1 - 1

CHAPTER 2 ..... 2 - 1
Getting Started ..... 2 - 1
Making a Backup ..... 2 - 1
Compiling a Demo Program ..... 2 - 2

CHAPTER 3 ..... 3 - 1
Using the AC/BASIC Compiler ..... 3 - 1
Preparing BASIC Programs ..... 3 - 1
An Interpreter File ..... 3 - 1
Operating the Compiler ..... 3 - 2
The File Menu ..... 3 - 3
Opening a File to Compile ..... 3 - 4
Creating a New File ..... 3 - 4
Editing an Existing File ..... 3 - 4
Choosing a Printer ..... 3 - 5
Printing a File ..... 3 - 5
Transfering to Another Program ..... 3 - 5
Quitting the Compiler ..... 3 - 5
The Control Window ..... 3 - 6
Compiling a Program ..... 3 - 7
BASIC Compiler Output ..... 3 - 8
Run Files ..... 3 - 8
Compilation Statistics ..... 3 - 9
Error Reporting ..... 3 - 10

CHAPTER 4 ..... 4 - 1
Using Compiler Options and Metacommands ..... 4 - 1
Compiler Options ..... 4 - 1
Selecting Compiler Options ..... 4 - 1
Options Affecting the Run File ..... 4 - 3
Actual Line Numbers ("A" Option) ..... 4 - 3
Process Run-time Events ("C" Option) ..... 4 - 4
Compile for Decimal Math ("D" Option) ..... 4 - 5
No Default Menu Bar ("M" Option) ..... 4 - 6
Run-time Line Numbers ("N" Option) ..... 4 - 6
Link Run-time Library ("R" Option) ..... 4 - 6
No Default Window ("U" Option) ..... 4 - 8

List File Options....4-8
Generate Errors List ("E" Option)....4-9
Generate Full List ("L" Option)....4-9
List Include Statements ("I" Option)....4-9
Compiler Metacommands....4-10
Changing the About box text....4-11
Ignoring Statements During Compilation....4-11
Including Source Files....4-12
Inserting Page Breaks in Compiler Listings....14

CHAPTER 5....5-1
Batch Compilation....5-1

CHAPTER 6....6-1
AC/BASIC Language Reference....6-1
Introduction....6-1
Conventions....6-2
The Character Set....6-3
The Program Line....6-4
Commenting a Program....6-5
Line Numbers and Labels....6-6
Line Numbers....6-6
Labels....6-8

CHAPTER 7....7-1
Constants and Variable Declarations....7-1
Constants....7-1
String Constants....7-1
Numeric Constants....7-2
Variables....7-3
Variable Names....7-4
Variable Types....7-4
Strings....7-4
Numeric Variables....7-5
Declaring Variable Types....7-6
Choosing Appropriate Variable Types....7-10
Array Variables....7-11
Assigning Values to Variables....7-12
LET....7-13
SWAP....7-14
Array Variable Statements and Functions....7-14
DIM....7-15
ERASE....7-16

OPTION BASE 7-17
LBOUND 7-18
UBOUND 7-19
User-Defined Functions 7-20

CHAPTER 8 8-1
Math Functions and Numeric Expressions 8-1
Expressions and Operators 8-2
Evaluation of Expressions 8-2
Hierarchy of Operations 8-3
Precision of Operands 8-4
Evaluation Errors 8-7
Logical (Boolean) Operators 8-8
NOT 8-8
AND 8-8
OR 8-8
XOR 8-8
EQV 8-9
IMP 8-9
Built-in Functions 8-9
ABS 8-9
ATN 8-10
CINT/CSNG/CDBL 8-11
COS 8-12
EXP 8-12
FIX 8-13
INT 8-14
LOG 8-15
RANDOMIZE 8-16
RND 8-16
SGN 8-18
SIN 8-18
SQR 8-19
TAN 8-20
VAL 8-20

CHAPTER 9 9-1
Strings 9-1
Relational String Operations 9-2
String Maximums 9-2
String Functions 9-3
ASC 9-3
CHR$ 9-4

DATE$ 9-5
HEX$ 9-6
INSTR 9-7
LEFT$ 9-8
LEN 9-8
MID$ 9-9
OCT$ 9-10
RIGHT$ 9-11
SPACE$ 9-12
STR$ 9-12
STRING$ 9-13
TIME$ 9-14
UCASE$ 9-15

CHAPTER 10 10-1
Control Statements 10-1
GOTO 10-2
GOSUB 10-3
RETURN 10-6
ON...GOTO / ON...GOSUB 10-7
IF Conditional Statements 10-11
IF...THEN...ELSE 10-11
IF...GOTO 10-13
Block IF Statements 10-13
Control Structures 10-16
FOR...NEXT 10-16
WHILE...WEND 10-19
System Control Statements 10-20
CHAIN 10-20
COMMON 10-21
RUN 10-23
END / STOP / SYSTEM 10-24

CHAPTER 11 11-1
I/O Statements and Functions 11-1
File and Path Names 11-2
System File Statements 11-2
CHDIR 11-3
FILES 11-4
FILES$ 11-5
KILL 11-6
NAME 11-7
Preparing File I/O 11-8

Sequential Files 11-8
Random-Access Files 11-9
File Buffers 11-9
OPEN 11-10
CLOSE 11-14
Sequential Input 11-15
INPUT 11-15
INPUT# 11-17
LINE INPUT 11-19
LINE INPUT# 11-20
INKEY$ 11-21
INPUT$ 11-22
Sequential Output 11-24
WIDTH 11-24
PRINT/PRINT#/LPRINT 11-26
SPC 11-29
TAB 11-30
PTAB 11-31
WRITE/WRITE# 11-32
PRINT USING 11-34
Auxiliary I/O Statements and Functions 11-39
LOCATE 11-39
CSRLIN 11-40
POS 11-41
LPOS 11-42
Random-Access I/O 11-42
FIELD 11-43
GET 11-45
PUT 11-46
LSET/RSET 11-48
MKI$/MKS$/MKD$ 11-49
CVI/CVS/CVD 11-50
MKSBCD$/MKDBCD$ 11-51
CVSBCD/CVDBCD 11-53
I/O Utility Functions 11-55
EOF 11-55
LOF 11-56
LOC 11-58
Reading Program Data 11-58
DATA 11-59
READ 11-60
RESTORE 11-61

CHAPTER 12 12-1
Graphics Statements and Functions 12-1
Screens and Windows 12-1
The Apple IIGS Colors 12-3
Pens 12-4
Screen and Window Statements 12-5
SCREEN 12-5
WINDOW 12-6
WINDOW OUTPUT 12-7
WINDOW CLOSE 12-8
WINDOW() 12-9
Drawing Statements 12-10
PRESET/PSET 12-10
LINE 12-11
CIRCLE 12-13
SCROLL 12-16
Changing the Apple IIGS Colors 12-17
PALETTE 12-17
COLOR 12-22
POINT 12-23
CLS 12-24
LCOPY 12-24
Array-Based Graphics 12-25
GET (graphics) 12-25
PUT (graphics) 12-27
SCREEN() 12-29
Picture-Based Graphics 12-29
PICTURE 12-29
Understanding the Colors 12-32
The 320x200 screen 12-32
The 640x200 screen 12-34

CHAPTER 13 13-1
Events and Event Trapping 13-1
Mouse Events 13-2
MOUSE() 13-3
Mouse Button Status 13-3
Mouse Pointer Coordinates 13-5
Mouse Event Trapping 13-7
Dialog Events 13-10
DIALOG() 13-10
Dialog Status 13-10
Dialog Event Trapping 13-13

BUTTON 13-14
EDIT FIELD / EDIT$ 13-16
Custom Menus 13-17
MENU 13-18
MENU() 13-19
Menu Event Trapping 13-21
Error Handling 13-23
Run-time Error Trapping 13-23
ERR and ERL 13-24
RESUME 13-26
ERROR 13-27
Timer Event Trapping 13-28
Break Event Trapping 13-29

CHAPTER 14 14-1
Sound Production 14-1
BEEP 14-2
Music Production Statements 14-3
SOUND 14-3
WAVE 14-7

CHAPTER 15 15-1
Subprograms 15-1
SUB 15-2
END SUB 15-6
EXIT SUB 15-7
SHARED 15-7
CALL 15-9

CHAPTER 16 16-1
Advanced Memory and Machine Statements 16-1
PEEK 16-1
POKE 16-2
VARPTR 16-3
CLEAR 16-4
FRE 16-5
RESET 16-5
Assembly-Language Interface 16-6
MACHINE 16-6
Array-based Input and Output 16-7
BLOAD 16-7
BSAVE 16-8
BSEEK 16-9

SWAP BYTES 16-10

CHAPTER 17 17-1
Editor 17-1
Starting The Editor 17-1
Help 17-2
Editor Modes 17-2
Entering Text 17-2
Moving Through A File 17-2
Small Movements 17-3
Medium Movement 17-3
Large Movements 17-4
Modifying Text 17-5
Deleting Text 17-6
Inserting Text 17-7
Advanced Features 17-7
Search / Search With Replace 17-7
Search 17-8
Search With Replacement 17-8
Cut, Copy, and Paste 17-9
Cut 17-9
Copy 17-10
Paste 17-10
Closing The File 17-10
Chapter Summary 17-11

CHAPTER 18 18-1
Editor Reference 18-1
Modes 18-1
Insert 18-2
Escape 18-2
Auto Indent 18-3
Select Text 18-3
Automatic Wrap 18-5
Macros 18-5
Command Descriptions 18-7
Setting Editor Defaults 18-21
Chapter Summary 18-23

APPENDIX A A-1
Internal Data Formats for Real Numbers A-1
Binary Versus Decimal Numbers A-1
Single and Double Precision Binary Numbers A-2

Single and Double Precision Decimal Numbers....A-4

APPENDIX B....B-1
ASCII Codes....B-1

APPENDIX C....C-1
BASIC Reserved Words....C-1

APPENDIX D....D-1
Machine-Language Example....D-1
Sieve of Eratosthenes....D-1
Using a Toolbox....D-5

APPENDIX E....E-1
Differences Between Versions of AC/BASIC....E-1
Macintosh-Specific Statements and Functions....E-1
Amiga-Specific Statements and Functions....E-2
Apple IIGS-Specific Statements and Functions....E-3
Macintosh Statement and Function Differences....E-3
Amiga Statement and Function Differences....E-5
Programming Techniques....E-10
Screen Resolution....E-10
Aspect Ratio....E-11
Colors....E-12
Sounds....E-12

APPENDIX F....F-1
Run-Time Error Messages....F-1
Loading Errors....F-1
Execution Errors....F-2

APPENDIX G....G-1
Compile-Time Error Messages....G-1
Alert Box Errors....G-1
Program Errors....G-2

APPENDIX H....H-1
AC/BASIC Installation Guide....H-1
Files on the AC/BASIC Compiler Disk....H-1
Copying Files....H-2
One or Two Disk Installation....H-2
Hard Disk Drive Installation....H-3
RAM Disk Installation....H-4

APPENDIX I ..... I - 1
Apple IIGS Toolbox Routines ..... I - 1
Text Routines ..... I - 1
Color Routines ..... I - 5
Graphics Pen Routines ..... I - 6
Line Drawing Routines ..... I - 10
Object Drawing Routines ..... I - 10
Mouse Cursor Routines ..... I - 14

APPENDIX J ..... J - 1
Bibliography ..... J - 1

INDEX

QUICK KEYWORD REFERENCE

CHAPTER 1

INTRODUCTION

The name BASIC is an acronym for "Beginner's All-purpose Symbolic Instruction Code". "Beginner's" indicates the language is easy to use and learn, "All-purpose" means that BASIC is well suited for a wide range of programming tasks, and "Symbolic Instruction Code" tells us that BASIC programs are made up of symbols and statements that look similar to the English language (one of the reasons it is easy to use).

For nearly three decades the BASIC language has evolved to meet the needs of programmers. AC/BASIC is a culmination of this evolution; it is one of the most powerful versions of BASIC to be found on any microcomputer. With it, you can quickly create programs that take full advantage of the sound and astounding graphics capabilities of the Apple IIGS computer.

AC/BASIC on the Apple IIGS is designed to be compatible with the Microsoft BASIC Interpreter and the Microsoft BASIC Compiler on the Macintosh

Once a program has been compiled, it becomes independent of the compiler and can be treated like any other program; it can be moved from disk to disk, machine to machine, and will run the same.

This manual is designed to expand fully on the topics discussed. Specifically, this manual provides:

- Instructions for compiling BASIC programs.
- A BASIC reference manual. All the BASIC statements and functions are grouped together by common topic and then explained. The syntax (rules for writing) of each statement or function is listed, along with an explanation and an example of its use.
- Nine functional appendices, including descriptions of internal data formats, an advanced installation

guide, tables of ASCII codes, reserved words, and run-time and compile-time error messages.

- Many fully-tested, working examples. Most of the examples are supplied on the disk, conveniently cataloged by chapter.

This manual assumes you understand the material presented in the two books supplied with the Apple IIGS computer:

- *Apple IIGS Owner's Guide* and
- *Apple IIGS System Disk User's Guide*

Furthermore, we will assume that you have worked, at least a little, with a BASIC interpreter like the Applesoft interpreter included with the Apple IIGS or a Microsoft BASIC interpreter on another computer. If you have used AC/BASIC on another computer you will find that AC/BASIC on the Apple IIGS is very similar

If you have never used a compiler before, you will find it to be quite different from an interpreter. Interpreters must examine a BASIC statement every time it is encountered. If the interpreter finds that the line is correct, then it is executed. Consider the following example:

```
FOR I = 1 TO 500
   J = I * 50 + (I - 10 )
   PRINT J
NEXT I
```

An interpreter must examine each of the above lines 500 times as the program executes. This can take a considerable amount of time.

In contrast, a compiler examines BASIC statements only while compiling the program. If a line is correct, then the compiler generates the appropriate machine language code to perform the statement. After fully examining the program, the compiler creates a file called a "run" file, and writes all the machine language code to it. The compiler never executes the BASIC statements; it merely translates them into their faster machine

language equivalents. The speed improvements seen when the run file is executed will vary widely depending on what the program does; programs that spend a lot of time doing file processing will be the least improved. This is because the speed of reading and writing files is limited by the physical speed of the disk. Programs that spend most of their time doing arithmetic and data manipulation will be the most improved over an interpreter.

Before using the compiler, spend some time reading through this manual, especially Chapters 2 through 5, which describe using the compiler, its options, and compiling multiple files. Although the compiler comes ready-to-use, you may easily relocate it to another floppy, a hard disk, or RAM memory to best fit your system's configuration. If the compiler and its overlays are to be moved, it is essential that this be done correctly; see Appendix H, "AC/BASIC Advanced Installation Guide", for the proper procedure.

CHAPTER 2

GETTING STARTED

This chapter presents a short tutorial which will guide you in compiling and running an example program supplied with the compiler.

Please take the time to make a copy of the AC/BASIC master disk. Then, as suggested, put the master disk in a safe place. Never use the master disk for anything but copies! It is always best to follow this procedure with any software you buy; a blank disk is much cheaper than buying the product again if the master disk is ever damaged.

For your convenience, this product is not copy-protected. Consequently, the AC/BASIC master disk can be safely copied as many times as you want. *Note that it is illegal to make copies for any purpose other than personal use.* Illegal distribution of copies (pirating) leads to higher software prices and reduced technical support. It may also delay introduction of improved versions of the software.

MAKING A BACKUP

Use the `FastCopy` program in the `SYS.UTILS` folder of the System Disk[1] to make a backup copy of the master disk. If you have a System Disk version 2.0 or earlier which does not include FastCopy, then use 'Duplicate a Disk' in the `SYSUTIL.SYSTEM` program. Usually, these are invoked from the Launcher by double-clicking on the folder and program names.

If you have never used FastCopy or System Utilities before consult the *Apple IIGS System Disk User's Guide*.

Now that you have a copy, put the master disk away!!

---

[1]The System Disk comes with the Apple IIGS. The *Apple IIGS System Disk User's Guide* explains how to use the System Utilities.

COMPILING A DEMO PROGRAM

The AC/BASIC disk is organized so that it can be used immediately — no special configuration or installation procedures are necessary. After a while, you may desire to rearrange the compiler files (perhaps for a hard disk) or integrate the compiler with a system disk. Appendix H is full of useful information concerning the compiler and the associated support files; read it if you need to reconfigure the disk setup.

The following is a brief example of using the compiler from the Program Launcher screen to compile the demo program "Hello". Although it is short, it gives the necessary steps for experimenting with the examples found throughout this manual and on the disk. You may follow this as a step-by-step tutorial:

1. Make sure the working copy of the master disk is *not* write-protected, then insert it in any drive and turn on the computer. The compiler disk has the required system files on it so you may remove your System Disk if necessary.

2. You will launch the compiler from the Launcher which should appear on the screen[2] . Double-click on the BASIC folder to reveal the AC.BASIC icon.

   Note: The system included on the compiler disk may be a different version than the system on your System Disk. If you have a one disk system, or if you choose to remove your System Disk, there may be a problem with the two systems being incompatible. To avoid having two different versions of the system, either boot from the compiler disk or install the compiler on your own disk as described in Appendix H.

[2] If you are using the Finder from your own System Disk, the Program Launcher will not appear. You can still click on the icons mentioned in this tutorial.

3. Double-click the `AC.BASIC` icon to start the compiler. The menu bar will be redrawn and the screen will be cleared.

4. A *file selector* will appear asking for you to choose a file name with the mouse. Double-click the file `Hello`. "Hello" is a BASIC program supplied with the compiler. A window will now appear displaying the compiler's *Control Window*; simply press the X key to compile and execute the program. The Control Window and its options will be discussed in Chapter 4.

5. You will see some statistics being displayed as AC/BASIC compiles the program to produce a run file named `Hello.Run`. After about half a minute you will see the message "Compilation Completed".

6. You will see the `Hello.Run` program running; when you want to end the program select `Quit` from the run-time `File` pull-down menu. You will be returned to the Program Launcher.

CHAPTER 3

USING THE AC/BASIC COMPILER

The topics in this chapter are arranged in sequential order. First, preparation of program files for input to the compiler is discussed. Then the details of starting up the compiler and using its menus are explained. Lastly, the compiler's screen output is reviewed. All the features on the compiler's Control Window are not explained until the next chapter (most of the options have nothing to do with operating the compiler; rather, they affect the run file produced). If you followed the tutorial at the end of the last chapter, much of the material presented here will be familiar.

PREPARING BASIC PROGRAMS

This section introduces the term "flat ASCII" to describe the required structure of an input file. "Flat" is used in the sense described in Webster's New World dictionary as "without gloss"; a flat ASCII file is one that contains only text and no other formatting or special information. The editor which comes with the AC/BASIC compiler works with and produces files that are flat ASCII.

An Interpreter File

If you are porting programs from an interpreter to run in AC/BASIC, note that most interpreters do not save all the *characters* of a program in a file. For increased speed and efficiency, they use a special compression technique to store the characters in memory as you type them. Such a file is not flat ASCII, and is not acceptable as input to the compiler. In order to support the loading of files created on other computers and word processors, an interpreter can usually open and save flat ASCII files. When loading a program, an interpreter automatically distinguishes between compressed and flat ASCII files. On the other hand (this is the important part), an

interpreter needs to be told to save a file in flat ASCII format; otherwise, it will default to the compressed mode.

In order to prepare a compressed file for use with the compiler, simply run the interpreter, load the program file, and then save it, by typing the following from command mode (press RETURN after each line). This works in the Microsoft Interpreter on the Macintosh and the Amiga Basic interpreter on the Amiga. As of this writing, there is no interpreter on the Apple IIGS which will save a program as an ASCII file.

```
load "filename"
save "filename",A
```

where `filename` is the name of the BASIC program file you wish to convert.

The above commands will cause an interpreter to save a program file in flat ASCII so that it is readable by the compiler and the editor (it will still be readable by the interpreter). If you have difficulty in saving files from the interpreter, refer to the description of the **SAVE** statement (command) in the interpreter's manual.

The file can be transfered to the Apple IIgs by using a terminal program on each computer.

OPERATING THE COMPILER

This section describes the operation of the compiler. This includes starting the compiler, using its controls, and understanding the information it presents.

You can start the compiler from either the Desktop, Finder, the Program Launcher, or APW[1]. To start the compiler from the Desktop, Finder, or Program Launcher, double-click on the AC.BASIC icon. To start the compiler from APW, type `AC.BASIC` in lower or upper case, and press RETURN.

[1] Apple IIGS Programmer's Workshop development environment.

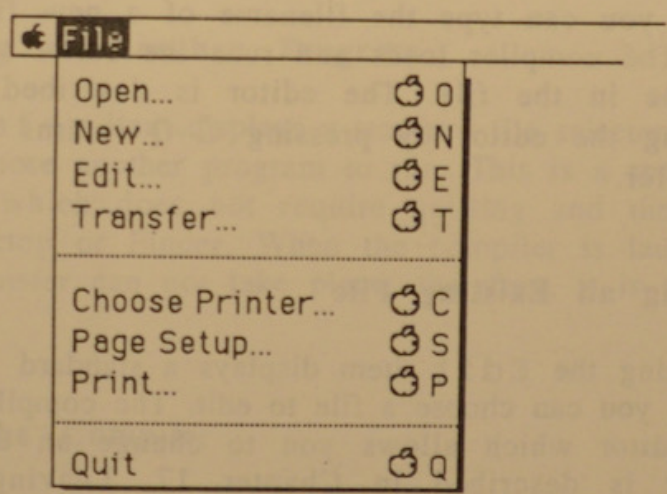
Once the compiler is started, a file selector appears. From this, you may choose a file to compile or click on the `CANCEL` button and then select an item from the `File` menu.

From the menu, the compiler can be operated by using the mouse to select items. For your convenience, all selections can also be made using the keyboard. In this manual, when the ⌘ key is referenced it means the open-Apple key next to the space bar key. The press of the ⌘ key will always be accompanied by typing some other key.

For example, ⌘-Q means you should press the ⌘ key and, while holding it down, press the "Q" key. This is a standard way of indicating multiple-key input on the keyboard.

The File Menu

The ⌘-key sequences for performing the menu activities are listed to the right of each item. From the `File` menu you can `Open` a file to compile (⌘-O), `Edit` a file (⌘-E), `Transfer` to another program (⌘-T), `Print` a file (⌘-P), or `Quit` (⌘-Q).



Each of the menu choices is described in the following sections.

Opening a File to Compile

The Open item in the File menu is used to specify the input file for the compiler. Normally, the input file will be a flat ASCII file containing your program. Selecting Open from the pull-down menu, or typing ⌘-O, causes a file selector to appear on the screen.

The file selector has two basic parts: the list of files in the current folder and some buttons. To select a file, double-click on its name. If you do not see the file name click on the up and down arrows to move the scroll bar through all the names in the folder. Double-clicking on a folder will open that folder and show the files within it. To close a folder, click on the Close button. To show the files on another disk, click the Disk button. If you do not want to select a file, click on Cancel. Note that pressing the RETURN key will have the same effect as clicking on the Open button.

Creating a New File

Choosing the New item displays a standard file selector from which you can type the filename of a new file to create and edit. The compiler loads and runs the editor which allows you to type in the file. The editor is described in Chapter 17. Leaving the editor by pressing ⌘-Q returns you back to the compiler.

Editing an Existing File

Choosing the Edit item displays a standard file selector from which you can choose a file to edit. The compiler loads and runs the editor which allows you to change an existing file. The editor is described in Chapter 17. Leaving the editor by pressing ⌘-Q returns you back to the compiler.

Choosing a Printer

You may need to select which kind of printer you want to be used by the `Print` item. To do this, use the `Choose Printer` item. You may use `Page Setup` to select a different page layout. Both of these items only need to be used once; the settings will remain in effect for each printing.

Printing a File

The `Print` item in the `File` menu is provided so you can print a file without leaving the compiler. For example, after a compilation using the list option you might want to have the list file printed. Selecting `Print` from the pull-down menu (or by pressing Ó-P), causes a standard file selector to be displayed.

Once a file name has been selected and the `Open` button clicked, the file will be printed on the attached printer. Remember to insure that your printer is ready (also called "on-line" or "selected") before you press RETURN or click the `Open` button.

Transfering to Another Program

The `Transfer` item displays a standard file selector from which you can choose another program to run. This is a speedy way of transfering which does not require quitting and then returning to the Desktop or Finder. When the compiler is launched from APW, a transfer can not take place, therefore, it is the same as `Quit`.

Quitting the Compiler

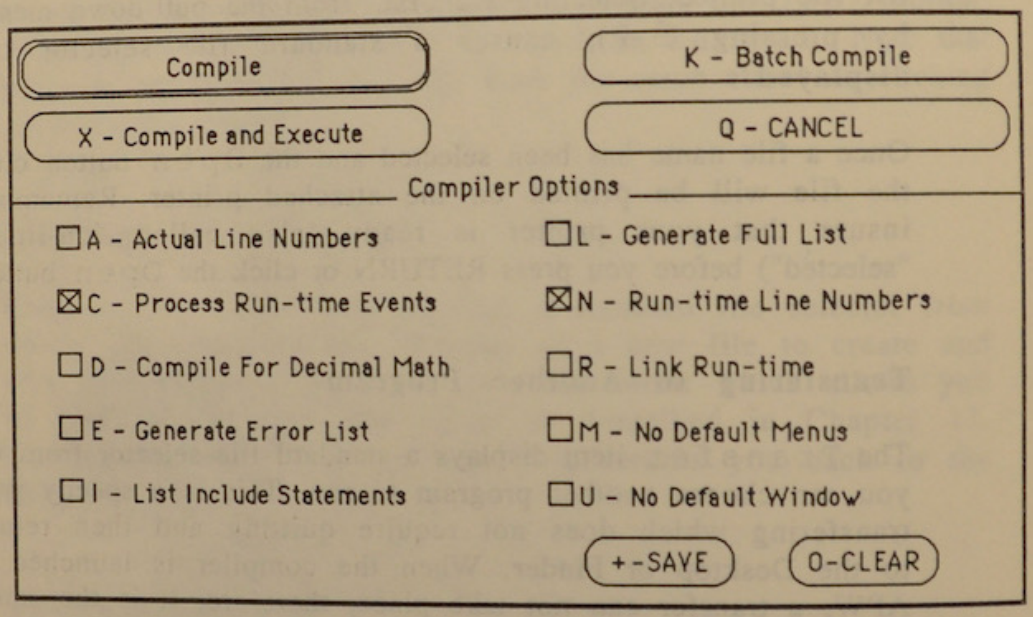
The `Quit` menu item is the exit from the AC/BASIC compiler. Selecting `Quit` from the pull-down menu, or typing Ó-Q, causes AC/BASIC to return to the system. Before exiting, the compiler will close any files it opened and free all the memory it used. If compiling more than one program, you do not need to exit and

restart the compiler for each file. Just compile one file, then the next, and so on; when you are all done, select `Quit` to exit.

The Control Window

As can be seen by the simple menu choices, the compiler is easy to use. There is one more detail that needs to be discussed before a general recipe for compiling programs can be given — you need to know how to use the Control Window. That will be explained in this section.

After selecting a file to compile, you will see a screen which looks like this, called the "Control Window":



The Control Window is divided into two functionally distinct halves. The lower half of the Control Window displays the settings of the available compiler options, all of which will be described in detail in Chapter 4. Compiler options are used to control how the compiler processes its input files (programs). Before shipping the compiler, we turned certain options on for you ("C", and "N"). Use them until you have time to read over the next chapter which discusses each option in detail.

The upper half of the Control Window is used to control the operation of the compiler itself: to start compilation, start a

batch compile (discussed in Chapter 5), and to cancel the current file selection.

Single-clicking the `Compile` box (or equivalently, pressing the RETURN key) will cause the compiler to begin compiling your program. Single-clicking the `Batch Compile` box will be discussed in detail in Chapter 5, "Batch Compilation." Single-clicking the `CANCEL` box will return you to the first AC/BASIC screen with the `File` pull-down menu.

COMPILING A PROGRAM

Except for learning which options are best for a particular program, you now know everything needed to use the compiler.

Each time you are ready to compile a program you will take the following steps:

1. Start the compiler.
2. Open a file to compile by selecting its name from the file selector.
3. Set the various compiler options you wish to use.
4. Click the `Compile` box (or press RETURN) to start compiling your program.
5. Read the compiler statistics and other significant information. (Explained in the next section.)
6. (Compile any other programs you want by repeating steps 2-5.)
7. Quit using the `File` pull-down and selecting `Quit` menu or by using Ó- Q.
8. Run the program(s) just like any other Apple IIGS software by clicking on the program's icon, or by

typing the name of the run file to launch it from APW.

BASIC COMPILER OUTPUT

The inputs and the controls of the compiler have been fully explained; all that is left to discuss are the various forms of compiler output. The next two sections describe the "run" file and the screen output you will see while the compiler is actually translating the input program into the machine-code run file.

Run Files

The primary output of the AC/BASIC compiler is a file of machine-language instructions (the "run" file). Machine-language instructions are the natural language of the Apple IIGS computer; they do not require any translation since the computer understands them directly. The run file is given the name of the input file, suffixed by `".Run"`. Thus the run file produced by compiling `"Prog"` will be `"Prog.Run"`. The run file is just like any other application produced by assemblers or other compilers: to run the compiled application from the Desktop, Finder, or Launcher, simply open (double-click) its icon, or from APW just enter, for example, `"Prog.Run"`.

Several errors may occur which will prevent AC/BASIC from compiling a program. They are:

`Failing load of system tools` — The AC/BASIC compiler and the compiled programs load many of the Apple IIGS Toolboxes from disk. This error occurs when the Toolboxes are not found on the disk or are outdated. The AC/BASIC compiler disk contains the the required tools which may be copied to your own disks. See Appendix H for a complete installation guide of the system Toolboxes.

`Failing read of run-time libraries` — The required run-time libraries called `BASIC.1L` and `BASIC.2L` were not found using the normal search path of folders. See the section

"Link Run-time Library" in Chapter 4 for a description of the search path and using the "R" option to link the run-time libraries to the compiled program.

`Not enough memory available` — The AC/BASIC compiler and compiled programs require at least 512K of memory. A large compiled program may require even more memory. Note that the RAM disk reduces the amount of available memory; delete some files from the RAM disk or use the Control Panel to reduce the RAM disk size.

The compiler will also produce various list files (of your program, its errors, or both) at your request. Listings are controlled by options in the Control Window which are discussed in Chapter 4, "List File Options."

Compilation Statistics

As the compiler processes the input file, it goes through several phases (called passes) in order to create the machine-language representation of the BASIC program. During each pass, the compiler creates and deletes a number of temporary files. The temporary file names are `Bprogr.1f`, `Bprogr.2f`, `Bprogr.3f`, and `Bprogr.4f`. These files can be safely deleted if the compiler ever leaves them around (this should only happen if a system error occurs during compilation). During compilation, a progress meter is shown along with the pass number below it.

Note that once the compiler is in this mode (actively compiling a program) the only menu selection available is `Quit`. If you want to stop the compiler at this point use Quit, ⌘-., or ⌘-Q.

After the compiler begins to compile a program, you will see the following message:

`Compiling` *`filename`*.

where *`filename`* is the name of the input file.

Error Reporting

The preceeding discussion referred to the compiler's screen output in cases where no errors were detected during compilation. The rest of this chapter discusses the window display that results from various types of errors in a program.

If there are syntactical errors (incorrect statement formats) or structural errors (e.g, **ELSE** not in an **IF**-block) in a program, they will be detected during the first compiler pass. The compiler will then display the line number, column number, and error message for each offending line. The letters L and C represent "line" and "column":

```
E110, L13, C5: ELSE or END IF without IF.
```

If you selected either the "Generate Errors List" or "Generate Full List" compiler options, then only this message will appear:

```
Pass 1: 1 error(s)
```

If control, context, or labelling errors (e.g., **GOTO** 10 where line number 10 is nowhere in your program) were encountered during the second pass, the compiler will display:

```
E18, L2: Undefined label or function (FN).
```

Again, the last line will be different if you set either the "Generate Errors List" or "Generate Full List" compiler options.

Certain errors are not detected on a line-by-line basis. For example, when the compiler detects a **FOR** loop statement, it makes a note of the fact and goes on about its business. At the end of the first pass (after the first pass the compiler has completely looked at each line of the program), the compiler determines if all the **FOR** statements had matching **NEXT** statements. If there is a missing **NEXT** statement, the compiler has no idea as to which line should contain the **NEXT** statement. Consequently, the error report will show the error to be on the last line of the file.

If there are several error messages, the text in the compiler window will scroll upwards; there will be at most a screen full. There are two ways to pause the printing:

1. Enter CTRL-S from the keyboard (the Control key and the "S" key simultaneously); to continue, enter CTRL-Q.

2. Press and hold the mouse button while the mouse cursor is over a menu title; when released, the scrolling will continue.

If there are a lot of errors, recompile your program with the "E" option (list errors) turned on. This will route the error list to a file instead of the window. (See the discussion of list options in the next chapter.)

A complete list of compile-time errors is in Appendix G.

CHAPTER 4

USING COMPILER OPTIONS AND METACOMMANDS

Options and metacommands are a means by which you can control various aspects of compilation from the Control Window or from within source files. The ten compiler options are described first, then the four metacommands are discussed.

Many of the discussions below relate to, and assume knowledge of, advanced features. As stated in Chapter 3, beginning BASIC programmers may use the "C" and "N" options to compile programs until they have learned how to use the more advanced compiler options. We strongly recommend that all programmers occasionally re-read this chapter as it contains considerable useful information.

COMPILER OPTIONS

In general, compiler options are a means of controlling how a BASIC program is compiled. You can think of the compiler options as being a set of on/off switches. The options are divided into two categories:

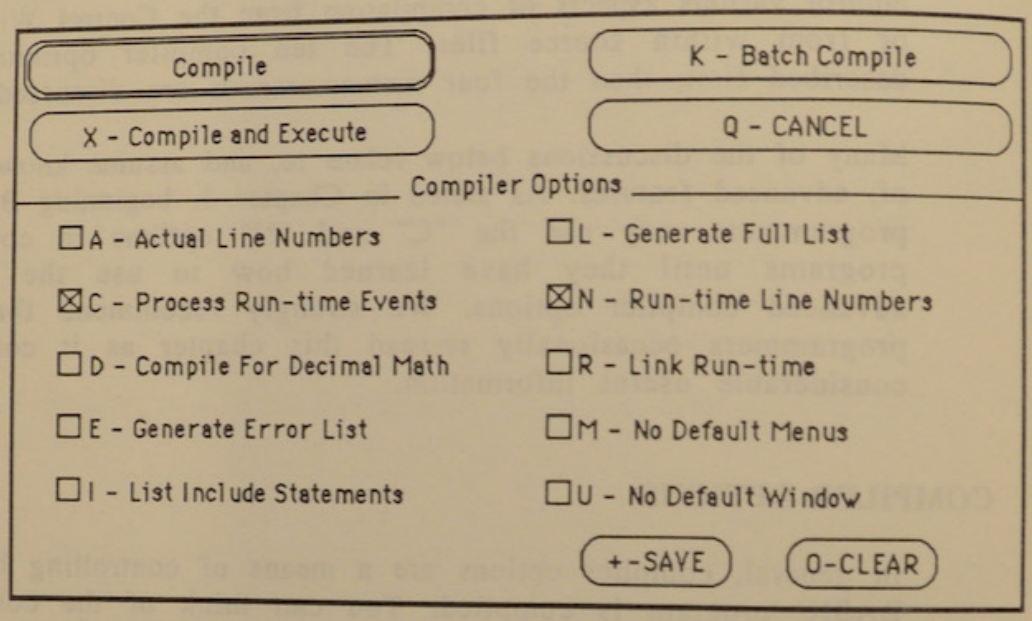
- Those which affect the run file.
- Those which control list files.

SELECTING COMPILER OPTIONS

All options can be set via the compiler's Control Window. In addition, a couple of options can also be controlled using compiler metacommands. Setting options via metacommands is discussed in the section "Using Compiler Metacommands."

With the Control Window, options are turned "on" and "off" by clicking on their respective boxes. You may also toggle options using the Apple IIGS keyboard; just type the letter corresponding to the box. In either case, an option is "on" when the box

has an X and "off" when it is blank. Any number of options can be turned on at the same time. The optimal combination of compiler options for any given program depends on its application and requirements (execution speed, error checking, etc.). For some applications, it may be appropriate to have all options turned off.



The large button marked SAVE is a convenience feature. As you become familiar with the compiler and the programming environment, you will probably find that a certain set of compiler options works well for nearly all programs (for example, turning "C" and "N" "on" and turning all the others "off"). You can set these options once from the Control Window and then click the SAVE button; thereafter, each time the compiler is started, the saved options will be automatically selected. The compiler saves the option settings as part of the file BASIC.OL. The plus sign "+" on the keyboard may be used in place of clicking on the SAVE button.

When compilation is started, the options are used as they appear in the Control Window, this makes it easy to override a saved set of options for a particular program. For example, if the "C" and "N" options have been previously saved and you also want "R" on for a certain program, then turn on "R" by clicking on it or by hitting the "R" key. This will override the saved options for that compilation.

Note: The compiler remembers the current option settings until you quit. So, in the above example, if you wanted to compile three different programs, the first with "C" and "N" and the other two with "C", "N", and "R", you would compile the first program without changing the options. Then turn on the "R" option and compile the remaining two programs.

The large button marked CLEAR turns off all options, making it easy to change a large number of options with just a few clicks of the mouse. The zero key "0" on the keyboard may be used in place of clicking on the CLEAR button.

OPTIONS AFFECTING THE RUN FILE

Run file options control features associated with the application produced by the compiler. They affect both the speed and size of the run file. As stated in the last chapter, the "C" (Process Run-time Events) and "N" (Run-time Line Numbers) produce programs with the greatest degree of compatibility with Microsoft BASIC Interpreters on other computers. However, they also cause the compiler to produce larger run files and may cause programs to run significantly slower than is necessary.

The following descriptions identify the function of each option. In addition, the effect of the option on speed and program size is discussed. For ease of reference, the options are listed in alphabetical order:

Actual Line Numbers ("A" Option)

When a BASIC program has explicit line numbers as in the following example, the line numbers are used by the compiler at run-time to specify error lines. With the "A" option turned off, this program will give a "Division by Zero" error at line 30.

```
10 PRINT "Small program"
20 x=0
30 y=1 / x
40 PRINT x,y
```

When a program is compiled with the "A" option, however, the line numbers are not used at run-time for error reporting. Instead, the actual line of the program is used. In this example, the error would be reported to be on line 3. The **ERL** fuction will also return a value of 3. The "A" option is especially useful for programs which normally do not have explicit line numbers. The editor, which shows physical line numbers, may be used to quickly find the line in which the error occurred.

Process Run-time Events ("C" Option)

To support a number of advanced event-driven features, AC/BASIC must do special processing between each statement in your program. This is a necessity, but it is time consuming. AC/BASIC gives you the ability to turn this special inter-statement processing on and off. Programs compiled with the "C" option off will be smaller and faster.

The primary purpose of the "C" option is to turn on run-time event checking, but it also enables the checking of the break key sequence Ó-. With the "C" option off, a program such as the following infinite loop can only be stopped by resetting or turning off the computer:

```
WHILE 1=1
   j=i+1
WEND
```

A special statement exists which will cause the events to be checked once. The **CONT** statement will always check for events even when the "C" option is turned off. For example, the following program segment will check for events such as Ó-. after executing a large series of mathematical statements.

```
WHILE pi <> ATN(1)*4
  GOSUB ComputePi
  CONT
WEND
```

Except for errors, all events can be detected by the program via two different techniques. One method, referred to as "polling", does not require inter-statement processing. The other, "trapping", requires processing between all statements and therefore requires the "C" option be turned on. Error events can only be detected with trapping. For more information on trapping and polling see Chapter 13, "Events and Event-Trapping".

When a program is compiled with the "C" option on, the run-time package will check to see if the event(s) being trapped have taken place. These checks take place between *every* statement that is executed. Remember that the polling method does not require the "C" option; in cases where a polling method of event detection will suffice, you can leave the "C" option off for additional speed. Alternatively, use the **CONT** statement carefully in those areas where you want all events detected without specific polling.

Compile for Decimal Math ("D" Option)

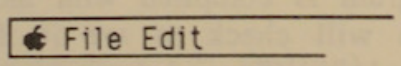
AC/BASIC supports two different methods of handling floating point values. The first method, called "binary" (the default), uses an IEEE format representation. This is the method used by most Microsoft interpreters and is faster than decimal; consequently, binary math will be appropriate for most programs. The "D" option is only turned on for the other type of math, called "decimal".

In general, programs compiled with "D" option turned on will have similar size run files as with the option turned off. Decimal programs tend to be slower than binary programs because all calculations in decimal math are performed in double precision (even if the operands are single precision). The biggest advantage of decimal math has to do with rounding errors that occur due to the internal representation of values in binary math. On the other hand, binary math representations

for both single precision and double precision support a larger number of significant digits than the decimal representations. In general, the benefits of one representation versus the other makes it appropriate to use binary for scientific programming and decimal for financial programming. Programs that are primarily sound or graphics oriented should stick to binary. For a complete technical discussion of the differences between these two representations, refer to Appendix A.

No Default Menu Bar ("M" Option)

With the "M" option off, an initial menu bar is created at run-time with the menus `File` and `Edit`:



With the "M" option on, however, no menu bar is created. This is useful when creating your own custom menus with the **MENU** statement and do not want the default menu bar.

Run-time Line Numbers ("N" Option)

With the "N" option on, the AC/BASIC compiler produces line and column number information for every statement. With the "N" option off, this information is not produced, thus making the compiled application smaller and slightly faster. The line and column information is required, however, to use error trapping and to have error reporting with line and column numbers at run-time. Therefore, it is appropriate to use the "N" option while debugging programs.

Turning the "N" option on automatically turns the "C" option on.

Link Run-time Library ("R" Option)

Every run file produced by AC/BASIC requires the support of additional files called the "run-time" libraries. Whenever a run file is launched it attempts to locate the run-time libraries. This

is done in one of two ways which is controlled by the "R" option. (The "R" option has no effect on the speed of a run file.) This section first explains the two different methods and then discuss the advantage of each.

Compiling with the "R" option turned on causes the compiler to automatically include a copy of the appropriate run-time libraries as part of the output run file. This means the run file will be considerably larger (by approximately 75 K-bytes). When the run file is launched, the run-time library is automatically loaded into memory, as it is part of the run file. The "R" option is labeled "Link Run-time" in the Control Window to indicate that the run-time library is to be linked to the run file.

Run files produced by compiling with the "R" option turned off do not have their own "personal" copy of a run-time libraries, and will be considerably smaller. When a run file produced this way is launched it will search the disks looking for the appropriate run-time libraries. (The search for the run-time libraries follows a specific path which may include more than one disk; see the order below for full details on the search path.) When the run-time libraries are located, they are loaded into memory by the run file. This method of associating the run file with the run-time libraries is called "dynamic linking", since the run-time libraries are actively (dynamically) loaded by the run file.

Search path for the run-time libraries `BASIC.1L` and `BASIC.2L`:

1. current folder
2. current folder's `BASIC.OVERLAYS` folder
3. root folder of boot disk
4. `System` folder of boot disk's root folder

As you can probably guess from the descriptions, the main disadvantage of dynamic linking has to do with the fact that the run file must be able to find the run-time libraries somewhere on a disk. This means that if the run file is moved to another disk, and consequently cannot find the run-time libraries an error will occur. By linking the run-time libraries you do not have to worry when moving the run file around because it contains a copy of the run-time libraries. The main

advantage of dynamic linking is the size of run files and their relation to the amount of free space on your disks: if you have a number of run files (programs) on one disk, each one will share the same run-time libraries. That means that less disk space will be needed and you will be able to support more files on your disk. On they other hand, if you link the run-time libraries to each run file you will use up the disk space much faster.

Note to Developers: *If you intend to distribute compiled applications (programs) you must first sign an agreement with Absoft Corporation which allows you to distribute the run-time libraries.* Distribution of the files `BASIC.1L` and `BASIC.2L`, or programs compiled with the "R" option enabled, is forbidden without first signing this agreement and returning it to Absoft. A licensing agreement is included with the compiler; if you have any questions about it, contact Absoft Corporation's Licensing Department at (313)-853-0050. There is no charge for this license.

No Default Window ("U" Option)

With the "U" option off, an initial full-screen window is created at run-time. With the "U" option on, however, no window is created. This is useful when creating your own window displays with the **WINDOW** statement and do not want the full-screen window.

LIST FILE OPTIONS

As previously discussed, the compiler has the ability to produce a source listing of the compiled program which is written to a file that you can examine later or send to a printer. The listing file will have the same name as the file being compiled, plus a `".lst"` extension; thus the file `"GAME"` when compiled will produce a file called `"GAME.lst"`. The following compiler options control the type of listing that is produced.

Generate Errors List ("E" Option)

Using the "E" option causes the compiler to write error messages about the source program to a list file with the extension ".lst" appended to the filename rather than to the screen. Each errant statement is listed, with error messages immediately underneath each statement. If no errors were found in the program, then no listing is produced.

Generate Full List ("L" Option)

Using the "L" option causes the compiler to generate a complete listing of the source program; each statement in the program is listed, and error messages are printed immediately below the statement that is in error. If no errors occurred then a symbol table and label reference is printed at the end of the listing.

List Include Statements ("I" Option)

Files containing BASIC statements included in a program with the **$INCLUDE** metacommand are normally not written into any of the aforementioned listings. If you want the statements in all the included files inserted in the listing, then turn on the "I" option. Note that turning on the "I" option automatically turns on the "L" option.

If you only want to see certain included files, but not all of them, then attach "/L" (quotes omitted) to the file name you used in the individual **$INCLUDE** metacommands; in this case, it is not necessary to use the "I" option. However, it is still necessary to use the full list option.

Consider the following main program:

```
DIM a(5)
REM $INCLUDE "INC1.BAS"/L
PRINT "Start"
REM $INCLUDE "INC2.BAS"
PRINT "End"
```

and the two include files:

```
REM This is file INC1.BAS
FOR z = 1 TO 5
   a(z) = z * 5
NEXT z

REM This is file INC2.BAS
b = a(1) + a(5) : PRINT b
```

The following BASIC statements would be printed in the listing:

```
DIM a(5)
REM $INCLUDE "INC1.BAS"/L
REM This is file INC1.BAS
FOR z = 1 TO 5
   a(z) = z * 5
NEXT z
PRINT "Start"
REM $INCLUDE "INC2.BAS"
PRINT "End"
```

COMPILER METACOMMANDS

Metacommands are special commands that you may use to control how a program is compiled; they are not part of the BASIC language and have no meaning outside the compiler. Metacommands instruct the compiler to perform various activities, such as including BASIC statements from other files or ignoring interpreter-specific statements like **LIST** and **DELETE**.

A metacommand is embedded within a BASIC **REM** statement, which must be the only statement appearing on the line. The **REM** statement under an interpreter is simply a way to enter remarks into a BASIC program; since metacommands are part of **REM** statements, they will be ignored by an interpreter. You can use them freely to control compiled programs without affecting the way the programs run under an interpreter.

The general form of a compiler metacommand is:

```
REM $metacommand [arguments]
' $metacommand [arguments]
```

where `metacommand` is one of the following:

Metacommand	Function
ABOUT	Sets the text of the "About" box
IGNORE	Ignores statements during compilation
INCLUDE	Inserts text from separate files
PAGE	Puts page breaks in compiler listings

Each is described in detail in the following sections.

Changing the About box text

General Form:

```
REM $ABOUT
'$ABOUT
```

During the execution of an AC/BASIC application, the Apple menu on the left of the menu bar contains the item "`About...`" which normally displays a box with the Absoft copyright notice along with the date the file was compiled. This notice may be changed with the **$ABOUT** metacommand which may specify up to 45 characters of text. For example,

```
REM $ABOUT "Program written by Adam Crooks."
```

Ignoring Statements During Compilation

General Form:

```
REM $IGNORE {ON | OFF}
'$IGNORE {ON | OFF}
```

This metacommand is especially useful for compiling programs that contain interpreter-specific statements. The **$IGNORE ON** metacommand causes the compiler to ignore all source lines until a **$IGNORE OFF** metacommand is found; you can place **$IGNORE ON** before those statements which are incompatible with the compiler and **$IGNORE OFF** following them. The program can then be compiled without those statements generating errors.

Another use of the **$IGNORE** metacommand is in conjunction with the **$INCLUDE** metacommand. You can use these two metacommands to create programs that run in both the interpreted and compiled environments and still take advantage of compiler-specific features. See discussion of the **$INCLUDE** metacommand below for examples of how to do this.

Including Source Files

General Form:

```
REM $INCLUDE "filename"[/L]
'$INCLUDE "filename"[/L]
```

The **$INCLUDE** metacommand causes the compiler to read BASIC statements from the specified file and process them as if they had actually been found in the program being compiled. When the compiler reaches the end of an **$INCLUDE** file it continues compiling the statements in the original file, beginning with the statement following the **$INCLUDE** command.

$INCLUDE files provide a convenient way of inserting code that you have already written and debugged into your program, or code that must be shared by several programs (variable declarations, **COMMON** statements, etc.). For example, if you are writing an application consisting of several programs that must **CHAIN** to each other, and have well-defined conventions for the usage of variable names, you may wish to **$INCLUDE** a declaration file in the following manner:

```
REM File: "DECLARES"
REM Declarations for ***** (application name)
DEFINT i-k,x-z : DEFSNG s,t  : DEFLNG m-p
DIM STATIC ProgData%(150,5),RunOptions$(5)
COMMON UserName$,CommandString$
COMMON ShapeArray%(1),ProgData%(),RunOptions$()
DIM ShapeArray%(256)
```

If the above statements were contained in file DECLARES, they may be included simply by placing an **$INCLUDE** command at the beginning of any files which require them:

```
REM Include application's variable types,
REM DIM statements, and COMMON declarations...
REM $INCLUDE "DECLARES"
.
.      (BASIC source code)
.
```

The code in the **$INCLUDE** file will be processed at compile time immediately after the **$INCLUDE** metacommand is encountered.

You can maintain groups of subroutines or subprograms that you use often and include them in programs; be sure to **$INCLUDE** subprogram definition files only at the *end* of your files. You may also use **$INCLUDE** in conjunction with the **$IGNORE** metacommand to produce BASIC programs that run under either an interpreter or the compiler:

```
REM Include file "compile.inc";
REM used for compilation only.
REM
REM Put compiler-specific statements here
```

Main Program:

```
REM If compiling, skip interpreter routines.
REM $IGNORE ON
REM
REM Put interpreter-specific statements here
REM $IGNORE OFF
REM Include the compiler-specific file
REM $INCLUDE "compile.inc"
.
.
```

When you compile the main program, the compiler will ignore the interpreter-specific portion and use the compiler-specific portion from the included file `compile.inc`.

$INCLUDE files must be "flat ASCII" text files just like any other source file you use with the compiler. You must take care that the **$INCLUDE** text does not contain variable declarations, subroutine names, or statement labels that conflict with the original file.

BASIC statements from **$INCLUDE** files do not normally appear in the compiler listing. If you wish the text from the **$INCLUDE** file to be in the listing, then append a "/L" (quotes omitted) after the file name (you can also use the "I" compiler option to force **$INCLUDE** files to be listed).

As with other metacommands, you must place the **$INCLUDE** metacommand alone on a single line in your source program. You must also take care that an included file does not contain another **$INCLUDE** metacommand. If an **$INCLUDE** metacommand is found within an **$INCLUDE** file the compiler will generate an error.

Inserting Page Breaks in Compiler Listings

General Form:

```
REM $PAGE
'$PAGE
```

The **$PAGE** metacommand causes the compiler to insert a page break in the compiler listing (See "List File Options"). When **$PAGE** is encountered, the compiler simply writes a form feed to the list file *in place of* the line containing the **$PAGE** metacommand; the **$PAGE** metacommand will not be included in the listing.

You may use **$PAGE** to control the presentation of important pieces of code, thus enhancing the readability of your listings.

CHAPTER 5

BATCH COMPILATION

Previous chapters have discussed the compilation of individual programs. This method works well if you only need to compile a few programs. If you have many files to compile at one time, separate compilation could be time-consuming. AC/BASIC has a facility that permits the compilation of multiple programs. Called "batch compilation", it allows you to give the compiler a list of files to be compiled. The compiler, in turn, compiles each file until the list is exhausted.

To use the batch compilation feature of AC/BASIC, first determine which files to compile, and which options are needed for each source file. Next prepare a "flat ASCII" text file, each line of which is of the following form:

```
sourcefilename {,compileroptions}
```

where `sourcefilename` is the file to be compiled, and `compileroptions` is a list of letters corresponding to the compiler options to be set for the compilation.

The filename must be separated from the list of options by a comma; the compiler option letters themselves may be separated by spaces, but spaces are not required. (See Chapter 4 for a list of valid compiler options and their letters). If no options are specified, then the default options will be used (compiler options that were set and then saved using the Control Window).

Examples:

```
AREA.BAS,CN
GRAPH2.BAS
C01E03.BAS,R C NL
```

All three examples above are correct. The first line will cause the compiler to use `AREA.BAS` as the source file and compile it with the "C" (Enable Run-time Tests) and "N" (Process Run-time

Events) options set. The second line will cause the compiler to use GRAPH2.BAS as the source file and compile it using whatever options were last saved on the Control Window. The third line is similar to the first only now C01E03.BAS is the source file and the compiler options to be set are "R", "C", "N", and "L".

Once you have constructed your batch-compile file, start the compiler as usual. Select the Open item from the File menu, or press ⌘-O, and select the batch-compile file name. When the Control Window appears, use the mouse and click the Compile List button to initiate compilation.

The compiler will now attempt to compile each named file in the batch-compile file you built. The normal statistics are displayed as each file is compiled. When all the files have been compiled, the File menu (or the ⌘-key shortcuts) can be used to do more compiling, printing, or quit.

If errors are encountered, either in the batch file or in one of the programs being compiled, the usual error messages will be written to the screen or a list file, dependent on the current option settings. When written to the screen, the compiler will wait for acknowledgement of the errors before compiling the next file.

If the same options are to be used for all files, an alternative to repeating the compiler option specifications for each file is to omit the options in the batch file and simply set them using the Control Window before initiating compilation.

CHAPTER 6

AC/BASIC LANGUAGE REFERENCE

This and the following chapters comprise a stand-alone reference manual which covers the full range of the AC/BASIC language. Rather than being listed indiscriminately in alphabetical order, the commands and their descriptions have been grouped into chapters by common topic, as follows:

Chapter 7	"Constants and Variable Declarations"
Chapter 8	"Math Functions and Numeric Expressions"
Chapter 9	"Strings"
Chapter 10	"Control Statements"
Chapter 11	"I/O Statements and Functions"
Chapter 12	"Graphics Statements and Functions"
Chapter 13	"Events and Event Trapping"
Chapter 14	"Sound Production"
Chapter 15	"Subprograms"
Chapter 16	"Advanced Memory and Machine Statements"

The manual is designed for quick and easy reference: examples are provided where appropriate and an extensive index is provided to help you locate any topic or keyword you may wish to reference.

The AC/BASIC compiler is almost completely compatible with the Microsoft BASIC Interpreter on the Macintosh except for sound statements and color graphics which are enhanced on the Apple IIGS.

In the following chapters, the term "AC/BASIC" by itself will refer to the language consisting of standard interpreted Microsoft BASIC on the Macintosh, not including those statements incompatible with the Absoft AC/BASIC compiler, *plus* the extensions provided by the AC/BASIC compiler.

INTRODUCTION

Before discussing the operations of the AC/BASIC statements and functions, a discussion of the structure of a program and the lines that make it up is in order. Programs are files of characters which are grouped into lines; a line may be blank, or may contain line numbers, line labels, variables, constants, and keywords which correspond to AC/BASIC statements or functions.

The function of the compiler is to translate a program into machine-language code that is understood directly by the Apple IIGS with no further modification. This is why a program linked with the run-time library can be moved to any disk or folder on the Apple IIGS and still be run. To effect this translation, each line in the program must adhere to the syntax that the compiler expects. Otherwise, errors will result at compile time.

This chapter describes the character set and the general format of AC/BASIC lines, line numbers, and labels. For information on specific statement syntaxes, refer to the statement's description in the appropriate chapter.

CONVENTIONS

The following is a list of conventions used in this manual for statement Formats and Definitions:

1. Square brackets ("[" and "]") in the "Format" sections of statement descriptions indicate optional items in the statement syntax.

2. Curly brackets ("{" and "}") in the "Format" sections of statement descriptions indicate a mandatory choice of *one* of the items within the brackets; each item will be separated by a vertical change bar ("|").

3. Words that are in the example font (i.e. `variable`) in a statement "Description" section usually refer to parameters listed in the statements "Format" definition.

4. Words like **GOTO**, **IF**, and **MOUSE** are AC/BASIC reserved words shown in bold-face capital letters.

5. Words that are in the example font (i.e. `variable`) in examples are not reserved words, but are variables, parameters, and other text.

6. An ellipsis ("...") is used to indicate repetition of a previous syntactic element in a "Format" statement syntax section.

THE CHARACTER SET

An AC/BASIC program is made up of printable ASCII characters. Certain printable characters are only allowed as part of a literal string or comment. These are:

Character	Definition
[	Opening bracket
]	Closing bracket
`	Grave accent
{	Opening brace
|	Vertical bar
}	Closing brace
@	"At" sign
~	Tilde
_	Underscore

In addition, each line in the program must end with the non-printable carriage-return character. A tab character is considered equivalent to a space (except in literal strings). Line feeds are allowed, but have no effect on your program.

THE PROGRAM LINE

AC/BASIC requires that each line in a program source file conform to one of the following formats:

```
[line-number] [statement-list] [comment]

[label:] [statement-list] [comment]

[line-number] [label:] [statement-list] [comment]
```

where `line-number` is an integer line number in the range 0 to 65529.

`label` is an alphanumeric line label, consisting of any combination up to 40 characters of letters, digits and periods (".") beginning with a letter, and containing no intermediate spaces.

`comment` is any non-executing comment statement you may use to make notes or to document the function of your program.

`statement-list` is a list of legal AC/BASIC statements, separated by colons (":").

Items in brackets are optional and may or may not be included; thus lines on which blanks, comments, statements, labels or line numbers exist alone are permitted.

Examples of legal AC/BASIC lines:

```
REM You could say "comment"
' Or you could say "comment"...
100  REM A line by itself
100  x=SQR(2) : PRINT "Square root of 2 is";x
10  StartLabel: PRINT "Welcome to AC/BASIC!" ' Say hi!
AskUser:  INPUT "Starting X coordinate"; Start.X
Main:
Wait.For.Click: WHILE MOUSE(0)=0 : WEND : RETURN
```

The maximum length of an AC/BASIC line is 255 characters.

COMMENTING A PROGRAM

AC/BASIC provides two formats for commenting programs: the **REM** statement, and the special comment character "'". If the **REM** statement is used, it must be separated from preceding statements by a colon, just like any other statement. The comment character may be used anywhere on a line to begin comment text, except within a quoted string literal, and it does not require the colon separation from other statements.

The following are examples of correct comment syntax, and are equivalent:

```
10 INPUT F$ : REM Get filename from user
```

and

```
10 INPUT F$ ' Get filename from user
```

All text following a comment on a line is considered to be part of the comment. The following contains only two statements, a **PRINT** statement and a comment — the word INPUT is not a statement, since it is part of the comment, and will never be executed:

```
PRINT "Which one"; : REM Where to go? : INPUT M$
```

Comments do not slow down a compiled program as they do an interpreted one. In fact, comments are simply ignored by the compiler at compile time; thus program designers may include as many comments as they wish without being concerned about decreasing execution speed.

AC/BASIC supports a special form of the comment, called a compiler metacommand, which instructs the compiler from within a program to alter the way the program is compiled or to perform various utilities at compile-time. Compiler metacommands are extremely useful for those programmers who wish to use some of the compiler's powerful extensions to an interpreted BASIC language, yet still maintain compatibility

with an interpreter. See Chapter 4 for a complete discussion of compiler metacommands and their uses.

LINE NUMBERS AND LABELS

Line numbers and statement labels are used to mark positions in the program to which program control can be directed or redirected by any of the AC/BASIC flow control statements (GOTO, GOSUB, etc.). There are no differences in the *use* of the two, with the exception of the ERL function, which returns the last line number encountered before an error. Otherwise, they may be used completely interchangeably, i.e. wherever a line number is required a label may be used, and vice-versa.

Line Numbers

The rules for line numbers are straightforward, as described above: any integer from 0 to 65529, inclusive, beginning in the first column of a line. Line numbers do *not* have to be in increasing order throughout a program.

Since the ERL function only reports error lines, and not labels, you may find it useful for error detection and handling to use line numbers to delimit functional bodies of code:

```
ON ERROR GOTO TrapErr
10 StartUp:  ' Perform initializations
.
.
500 REM *** User Input Section ***
.
.
1000 ProcessFiles:  ' Open files, read data, etc.
.
.
TrapErr:
    IF ERL=10 THEN                ' Startup error
        :
    ELSEIF ERL=500 THEN           ' User input error
        :
        etc.
```

Using these conventions, the line number returned by the **ERL** function after an error is a "signature" by which the type of error (user input, I/O, etc.) can be determined. For programs that do not use the error trapping facility, the "N" option may be used to pinpoint the physical line corresponding to an error (see Chapter 4, "Other Effects of the N Option").

Note: Although line number 0 is a perfectly valid line number, for some AC/BASIC (event-trapping) statements it has a symbolic, rather than literal, meaning:

```
ON ERROR GOTO 0
ON BREAK GOSUB 0
ON DIALOG GOSUB 0
ON MENU GOSUB 0
ON MOUSE GOSUB 0
ON TIMER GOSUB 0
```

These statements use line 0 to disable a program's ability to trap certain events (an **ERROR**, a **BREAK**, etc.). For example, if an "**ON ERROR GOTO** 0" statement is executed, the occurrence of an error will not be trapped and control will not pass to line 0; an error message will be displayed and execution will terminate.

See Chapter 13, "Events and Event-Trapping," for more on the **ERL** function and event-trapping environments.

Labels

The rules for constructing alphanumeric label names are the same as those for constructing legal variable names: any combination up to 40 characters of letters, digits, and periods (".") beginning with a letter. However, note that the characters used to declare variable types (#, !, %, and $) may *not* be used in label names. Labels are not case-sensitive (neither are variables), so that "Start" and "START" are equivalent labels.

Here are some valid labels:

```
ProcessFile
lab001
sort.subs
What.a.Label
```

The rules for declaring a label on a line are as follows:

- The label must begin when the first non-blank character on the line (after the line number, if one is present);
- The label name must be followed immediately by a colon (":"), with no intervening spaces; this informs the compiler or interpreter that the name is a label declaration, not a call to a subprogram of the same name.
- The label name must not be the same as any of AC/BASIC's reserved words; see Appendix C for a list of reserved words.

When referring to an alphanumeric label in a flow control statement, the trailing colon (":") is optional. For example,

GOTO CORRECT	is equivalent to	**GOTO** CORRECT:
GOSUB GetChar	is equivalent to	**GOSUB** GetChar:

Examples of *invalid* label usage:

GOTO 2nd.Label	(begins with a digit)
IF A%>5 **THEN GOSUB** .PRT	(begins with a period)
Start: 10 **PRINT** "HI!"	(Occurs before line no.)
10 Startup :**PRINT** "Program running"	(Space before colon)

CHAPTER 7

CONSTANTS AND VARIABLE DECLARATIONS

Constants and variables are used to represent values in a program. Constants explicitly specify a value; as such, a given constant always specifies the same value. As their name implies, variables are used to represent values which can change as your program executes. Both constants and variables are classified using two general categories. One category, the "strings", is used to represent character data (such as the letters and words used to make up this sentence). The other category is "numeric." Numeric values are data items used to represent numbers in an AC/BASIC program.

CONSTANTS

String Constants

A string constant (also called a string literal) is composed of text enclosed within double quotation marks, e.g.,

```
"This is a string"
" !#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHI..."
"AC/BASIC Compiler, Knock My Socks Off!"
```

In order to construct a string containing double-quote characters, or characters not accessible through the keyboard, the **CHR$** function may be used in conjunction with string concatenation (+):

```
REM Chapter7/Example1
DblQuote$=CHR$(34)
W$="John shouted "+DblQuote$+"It's here!"+DblQuote$
PRINT W$
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example would output the line

```
John shouted "It's here!"
```

A string literal may be used wherever a string *value* or expression is required by AC/BASIC statement syntax, but must not be used in place of arguments that will be assigned a value. In the following example, the use of a literal is permitted as an argument for the **LEFT$** function, which returns leftmost characters in a string, but not for the **MID$** statement, which replaces portions of text within a string variable:

```
SomeString$=LEFT$("1987 World Fair",4)   ' OK
MID$("1987 World Fair",6)="Tokyo"        ' WRONG
```

The maximum length of a string literal or constant is slightly under 255, limited solely by the maximum length of an AC/BASIC line (255).

Numeric Constants

Numeric constants may be specified in one of five formats:

1. Integer: a sequence of digits *without* a leading, trailing, or imbedded decimal point;

2. Floating-point: a sequence of digits *with* a leading, trailing, or embedded decimal point;

3. Exponential notation: specified in the form mmmmmE±ee, where mmmmm is an integer or floating-point mantissa, "E" indicates single precision ("D" for double), the optional "+" indicates a positive exponent ("-" for negative exponents), and ee is the integer exponent. This form represents the decimal number

   mmmmm * 10 ± ee

4. Hexadecimal: "&H" followed immediately by hexadecimal digits, e.g. "&H1AC5";

5. Octal: "&O" or "&" followed immediately by octal digits, e.g. "&137" or "&O771".

Using single-precision, the maximum specifiable value is up to, but not including, $3.4 * 10^{38}$, and the smallest is $1.18 * 10^{-38}$. A double-precision constant may have a value up to, but not including, $1.79 * 10^{308}$, and as small as $2.23 * 10^{-308}$.

Constants may be forced to conform to any one of the numeric types supported by AC/BASIC; this typing may affect the evaluation of numeric expressions. See the "Numeric Variables" section in this chapter for a discussion of integers, single-precision, and double-precision typing. See the "Evaluation of Expressions" section in the next chapter for a discussion of the effects of typing constants in expressions.

The AC/BASIC compiler performs *constant folding* at compile-time. This means that an expression of constants will be calculated at compile-time and only the final result will be placed in the run-time code. For example,

```
a=b+(639-19)/20
```

will produce run-time code as is the statement was,

```
a=b+31
```

Constant folding speeds execution and makes the run-time file smaller. A side-effect of constant folding is the generation of some possible errors at compile-time. For example, a "Division by zero" error will be generated at compile-time for:

```
a=b+10/(20-10*2)
```

VARIABLES

The purpose of a variable is to provide an area in memory to store a value that may or may not change. By storing it in memory, the value can be recalled when needed. Variables can be either string or numeric; when a variable is first referenced, its value is null for strings ("") and 0 for numerics.

Variable Names

All variables are subject to the same naming conventions used for labels. A variable name may be a sequence of up to 40 characters of letters, digits, and periods (."), beginning with a letter. Variable names may not have imbedded spaces, and may not be the same as any AC/BASIC reserved words, e. g. **PRINT**, **ATN**, **MOUSE**, etc. (see Appendix C for a list of reserved words). The case of letters within a variable name is not significant; thus the names "XCOORD" and "xcoord" are considered the same.

Variable names beginning with "FN" are reserved for user-defined functions. If you use such a variable, the compiler will attempt to reference the function of the same name. There *must* be a defined function available before this variable is encountered. If not, the error "Expecting a symbol" will be given when the program is compiled. See the "**DEF FN**" section in this chapter for more on how to define and use these functions.

Examples of function names:

```
FNcount
fnTax.Calculation
FNheader
```

Variable Types

As mentioned above, variables can be either string or numeric. There is one kind of string variable and three kinds of numeric variables: integers, single-precision floating point, and double-precision floating point.

Strings

A string variable is denoted by the presence of a "$" (dollar sign) immediately after the variable name. This symbol is required in all cases except when the given variable name has

been implicitly typed as a string with a **DEFSTR** statement. (See section "Declaring Variable Types".)

String variables may contain up to 32,767 characters of text, and occupy 6 bytes of storage plus one byte for each character in the string. AC/BASIC supports many different operations on strings, including truncating strings, replacing text in strings, and searching for text within strings. See Chapter 9, "Strings," for a complete description of all string-manipulation commands.

Examples of assigning string values to string variables:

```
A$="o boy"        ' Assignment of a literal
INPUT Video$      ' Request text input from user
Star$=STRING$(500,"*")          ' Construct a string
State$=RIGHT$("Macon, GA",2)  ' Extract part of a string
```

Numeric Variables

For reasons of efficiency in storage space and execution time, AC/BASIC provides several different numeric variable types for the storage of numeric values:

1. An <u>integer</u> may contain an integer value between -32768 and 32767, inclusive, and occupies two (2) bytes of storage.

   Examples:

   ```
   i% = -455
   i% = 3.27E4
   ```

2. A <u>single-precision</u> number may have up to seven significant digits, and has a range from $1.18 * 10^{-38}$ up to, but not including, $3.4 * 10^{38}$. Single-precision numbers occupy four (4) bytes of storage.

Examples:

```
i! = .1234567
i! = 1000001
i! = 9.805E-30
```

3. A double-precision number may have up to sixteen significant digits, and has a specifiable range from $2.23 * 10^{-308}$ up to, but not including, $1.79 * 10^{308}$. A "D" is used in place of an "E" when specifying double-precision numbers in exponential notation. Double-precision numbers occupy eight (8) bytes of storage.

   Examples

```
i# = 1234.567890123456
i# = 999958342066665.1
i# = 4.50967275D128        ' Note the "D"
```

Note: In the above list, the ranges in magnitude and significant digit limits of single and double precision variables are given for binary representation ("D" compiler option off). The limit values for decimal math can be found in Appendix A.

Declaring Variable Types

The above sections talked about the rules for forming variable names, and the rules that apply to a particular type of a variable; this section gives the rules for associating a variable name with a variable type. There are three ways to establish the connection between a variable and a type:

1. Implicitly: When the compiler encounters a variable name that has no associated type it automatically assigns a type. For programs compiled for binary math (the default), variable names are implicitly typed as single precision. For programs compiled for decimal math, variable names are implicitly typed as double precision. This is the simplest method to use, and works fine for most programs.

2. Type declaration characters: This method assigns the type to a variable based on a special character (a type declaration character) that is appended to every instance of the variable name. The available type declaration characters are:

Declaration Character	Type
%	Integer
!	Single precision
#	Double precision
$	String

A name ending with any of the above characters explicitly declares the type of the variable, e.g.:

```
i%, PageNum%, COUNT%     ' Integers
Pi!, ratio!, HW.SC!      ' Single-precision variables
Nat.Deficit#, Pi.dbl#    ' Double-precision variables
character$, CMD$, A.1$   ' Strings
```

The compiler considers variables of the same name and different types to be different variables; therefore VAR%, VAR!, VAR#, and VAR$ refer to separate and distinct variables.

One advantage of this method is that each variable name will clearly show its type. However, this means that every instance of a given variable must have a type declaration character. The next method of typing overcomes this disadvantage:

3. DEFtype statements (**DEFINT**, **DEFSNG**, **DEFDBL**, and **DEFSTR**): make the association of a variable to a type based on the first letter of the variable name. (The exact format of DEFtype statements is given below.) This method is similar to methods used in other high-level languages, such as FORTRAN.

Format of DEFtype statements:

```
DEFINT range-list
DEFSNG range-list
DEFDBL range-list
DEFSTR range-list
```

Description:

Invokes implicit variable typing for variables beginning with the letters in the specified `range-list`.

The `range-list` is a list of letters separated by commas; sequences of consecutive letters may be abbreviated. The list "A,B,C,F,X,Y,Z" is equivalent to "A-C,F,X-Z".

Variables and user-defined functions beginning with letters in the given `range-list` are typed according to the last three letters of the DEFtype statement used to declare them: INT (integer), SNG (single-precision), DBL (double-precision), and STR (string). All variables affected are reset to null or zero values. The case of the letters in the `range-list` is not significant and overlaps within a declaration will not generate an error:

```
DEFSTR a,c-z,v-z
```

is equivalent to

```
DEFSTR A,C-Z
```

Type declaration characters (%,!,#, or $) that are part of a variable name will always override any DEFtype declarations.

Any type declaration for a variable will remain in effect until the variable is re-typed with another DEFtype statement or until a **RESET** or **CLEAR** statement is encountered. During compilation, the

last DEFtype statement controls the implicit type of variables beginning with the specified letters.

Example:

```
REM Chapter7/Example2
DEFINT I-J          ' I-J are integers
I=52.67 : J=25.3 : K=12.34
PRINT I,J,K
DEFSNG J            ' Redefine J to single-precision
DEFINT K            ' K is int; previous value lost
J=ATN(1)*4          ' J=Pi
PRINT I,J,K
WHILE MOUSE(0) = 0 : WEND
```

The output from this example is

```
53          25          12.34
53          3.141593    0
```

As with user function definitions (discussed below in "User Defined Functions"), type declarations are processed as they occur in the source file and cannot be activated via **GOSUB**s or equivalent GOTOs:

```
a = 3
GOSUB InitDef
a = "abc"
.
.
InitDef:
DEFSTR a-z ' Make all implicit variables strings
RETURN
```

The compiler will report a "Type mismatch error" when the attempt is made to assign a string to "a". This is because the **DEFSTR** statement is not processed before it encounters the second assignment of "a"; the compiler only translates the **GOSUB** statement and does not actually perform the subroutine jump to the InitDef label.

Modification is required for programs designed for an interpreter which do explicit typing in conditional statements (**IF...THEN**); similar to the **GOSUB** example above, the compiler does not attempt to evaluate the conditional clause and will process *both* DEF declarations in the following code:

```
IF a = b THEN DEFINT a-z ELSE DEFSNG a-z
```

The compiler, because the letter ranges coincide exactly, will effectively ignore the **DEFINT** declaration, masked by the following **DEFSNG**.

DEFtype declarations are lost when the current program exits to another program; if you wish the variables in a chained program to have the same typing you must include the appropriate implicit declarations.

Choosing Appropriate Variable Types

Which variable type should a program use? The use of integer variables consumes the least space and execution time, while double-precision variables occupy four times as much space and slow execution considerably in computation-intensive programs. Thus, for programmers concerned with speed and efficiency, the optimal implementation will use variables typed according to the *least* precision necessary to accomplish a given task. **FOR** loop counter variables should always be specified as integers. The same applies to variables used to control **WHILE...WEND** loops. Consider the following program, which counts up to 30000 in a **WHILE** loop:

```
REM Chapter7/Example3
REM WHILE...WEND count to 30K
PRINT "Timing of loop from 1 to 30000"
StartTime!=TIMER     ' Remember starting time
i=0
WHILE i < 30000
   i=i+1
WEND
PRINT "Count took";TIMER-StartTime!"seconds"
WHILE MOUSE(0) = 0 : WEND
```

By changing the variable i to i%, i!, and i#, the following results were obtained:

Using i%	Using i!	Using i#
4 secs	28 secs	33 secs

Array Variables

Variables that have the same type can be grouped into a structure called an "array". AC/BASIC arrays have the following attributes: class, type, number of dimensions, and number of indices per dimension. The variables which compose the array are no longer referred to by individual names; instead, they are accessed by indices into the array. This is useful when the individual variables represent data that is related in some way (i.e. the yearly income of each employee of a corporation). In addition, many AC/BASIC statements and functions take entire arrays as parameters (graphics **GET** and **PUT**, **WAVE**, etc.), as may an AC/BASIC user-defined subprogram.

An array is an efficient means of storage because its data elements can be retrieved and processed easily. There may as many dimensions to an array as memory permits.

In a **DIM** statement, any character declaring the type of an array's elements is inserted between the last character of the array's name and the opening parenthesis of the list specifying the array's dimensions:

```
REM Chapter7/Example4
DIM digits$(9)
FOR i=0 TO 9
   digits$(i) = CHR$(i+48)
   PRINT digits$(i)
NEXT i
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The example saves the ten digits in string form in the array digits$.

If an array and a simple variable have the same name, they are considered separate entities. For example,

```
REM Chapter7/Example5
DIM i%(10)
FOR i%=0 TO 10
   i%(i%) = i% * 10
NEXT i%
FOR i% = 0 TO 10
   PRINT "Array element i%(";i%;") = ";i%(i%)
NEXT i%
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

Though confusing, this example is valid. The array and the simple variable do not conflict with one another.

ASSIGNING VALUES TO VARIABLES

A variable is not very useful unless its value can be changed (assigned). There have already been numerous examples that demonstrated the use of assignment statements. There are a number of other ways to associate a value with a variable, but most of them are specialized for a certain type of variable or have a special purpose. The statements below have no associated special processing; they merely set some variable(s) to some given value(s).

LET

Format:

```
[LET] variablename = expression
```

Description:

Assigns a value to a variable.

The keyword **LET** is optional and may be omitted without effect on the assignment; it has been included to provide compatibility with the syntax of older BASIC assignment statements.

`Expression` can be any expression from the simplest (a constant) to a complex formula.

The general type of the expression must match the type of the variable name. This means that only string expressions can be assigned to string variables and only numeric expressions can be assigned to numeric variables. However, it is perfectly legal to assign one type of a numeric expression to a different type of numeric variable when the destination variable has the range to store the value; an overflow error occurs otherwise.

Legal examples:

```
LET A% = 5
A% = 5
S$ = "HELLO WORLD"
A# = A%
```

Illegal examples:

```
LET B$=A%
F! = "GOOD BYE"
```

SWAP

Format:

```
SWAP variable1,variable2
```

Description:

Exchanges the values of two variables or array elements of the same type. If they are not of the same type, a "Type mismatch" error results.

Example:

```
REM Chapter7/Example6
REM Example showing how to use SWAP to reverse an array
DIM ArrayVal%(10)
FOR i%=0 TO 10 : ArrayVal%(i%) = i% : NEXT    ' Load array
FOR i%=0 TO 10 : PRINT ArrayVal%(i%); : NEXT ' Print array
PRINT
FOR i%=0 TO 4
  SWAP ArrayVal%(i%),ArrayVal%(10-i%) ' Exchange values
NEXT
FOR i%=0 TO 10                      ' Print reversed array values
  PRINT ArrayVal%(i%);
NEXT
PRINT
WHILE MOUSE(0) = 0 : WEND    '-- Wait for mouse click
```

The SWAP loop in the middle exchanges two values per pass for five passes until 2*5=10 elements have been swapped; the last element ArrayVal%(5) is not altered because it retains its value. The output is

```
0 1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1 0
```

ARRAY VARIABLE STATEMENTS AND FUNCTIONS

Previous sections of this chapter introduced constants, variables, and variable types; this section of the chapter is

devoted to statements and functions that define various attributes of arrays or allow programs to inquire about specific array attributes.

DIM

Format:

```
DIM subscripted-variable-list
```

Description:

Defines the type and sizes of arrays.

The `subscripted-variable-list` is a list of array variables, separated by commas and containing the maximum values of each dimension of the array in parentheses, e.g.

```
a(3),b%(3,4,5),c$(4,5,6,7)
```

Each index, or dimension, may have up to 32,767 elements and each array may have up to 7 indices, although the sizes of all the arrays in a program is dependent upon the limits of available memory. (Note that the number of elements per index is based on the minimum subscript value (either 0 or 1) as defined by the **OPTION BASE** statement.) The amount of space required by an array can be calculated by the product of the number of elements in each dimension times the storage required for the array type, as follows:

Array Type	Storage Required (Bytes)
Integer (%)	2
Single precision (!)	4
Double precision (#)	8
String ($)	6 plus 1 per character

Thus the three-dimensional single precision array declared by "**DIM** SingPrec!(15,5,5)" will use (15*5*5 elements)*(4 bytes/element)=1500 bytes of storage with **OPTION BASE** 1 in effect, and (16*6*6)*4=2304 bytes of storage with **OPTION BASE** 0 (the default).

If an array is referenced which has not been explicitly dimensioned with a **DIM** statement, it is implicitly dimensioned as an array with a maximum subscript of 10 for each dimension in the array reference. It is strongly recommended, however, that all arrays be explicitly declared with the appropriate form of the **DIM** statement. If a subscript greater than 10 is used in any dimension of any undeclared array, a "Subscript out of range" error will occur.

If an attempt is made to redimension an array before erasing it, an error will occur; otherwise, an array may be erased and redimensioned as many times as is necessary.

Advanced user may note that for multi-dimensional arrays, A(2,2) will directly follow A(1,2) in memory.

Example:

```
DIM PlanetShapes%(9,128),SunShape%(128)
```

ERASE

Format:

```
ERASE array-variable-list
```

Description:

ERASE deallocates storage for the given arrays. `Array-variable-list` is a list of array names separated by commas; array subscripts are not permitted.

This command is used to free up the memory associated with arrays; an error results if an attempt is made to redimension an array without erasing it first. An error is also generated if an array is erased which has not been declared with a **DIM** statement. **ERASE** may not be used in a subprogram.

Example:

```
REM Chapter7/Example7
DIM NewMemberNames$(125)
   PRINT UBOUND ( NewMemberNames$ )
ERASE NewMemberNames$
DIM NewMemberNames$(150)
   PRINT UBOUND ( NewMemberNames$ )
PRINT "done"
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The array is redimensioned from 125 to 150 elements.

OPTION BASE

Format:

```
OPTION BASE {0|1}
```

Description:

Sets the base (minimum) array subscript value to 0 or 1 If any other expression is passed to **OPTION BASE** besides the literals "0" or "1" (quotes removed), a "Syntax error" will be generated when the program is compiled.

The default minimum subscript value is 0. If no **OPTION BASE** specification were present in a program, the statement "**DIM** a(4)" would allocate an array with five elements. If the **OPTION BASE** statement were used to set the starting array subscript to 1, then the same **DIM** statement would allocate a four-element array, a(1) through a(4), and an attempt to access a(0) would result in a "Subscript out of range" error.

If present, the **OPTION BASE** statement should occur in the program before any arrays are dimensioned or used, but it may be used more than once in a compiled application.

OPTION BASE statements may be executed conditionally in a compiled program:

```
REM Chapter7/Example8
OPTION BASE 1
DIM array(5)
PRINT LBOUND(array)
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
END
```

The program will print a "0" when run as a compiled program.

LBOUND

Format:

```
LBOUND (array-name[,dimension])
```

Description:

Returns the lowest subscript of the specified `dimension` of the given array; if the `dimension` specification is omitted, the lowest subscript of the *first* dimension of the array is returned.

LBOUND returns either 0 or 1, depending upon the previous use of the **OPTION BASE** statement. If no **OPTION BASE** statement has been executed, **LBOUND** will always return 0, the default lower bound. If **OPTION BASE** has been used to set the minimum array dimension subscript to be 1, subsequent **LBOUND** function calls will always return 1.

Example:

```
REM Chapter7/Example9
OPTION BASE 0 ' Default base subscript for arrays
DIM a(10,20)
PRINT LBOUND(a),LBOUND(a,2)
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The example prints:

```
0    0
```

See also **UBOUND**, below.

```
REM Chapter7/Example10
OPTION BASE 1
DIM a(10,20,30)
PRINT LBOUND(a),LBOUND(a,2)
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example prints:

```
1    1
```

UBOUND

Format:

```
UBOUND(array-name[,dimension])
```

Description:

Returns the maximum index of the specified `dimension` in the given array. If the `dimension` specification is omitted, the maximum index of the first dimension is returned.

UBOUND is not affected by the **OPTION BASE** setting in any way. This function is useful when an array whose size is not known is passed to a subprogram.

Example:

```
REM Illustrates use of UBOUND and LBOUND to allow
REM processing of different size arrays.
DIM DataArray1%(10,200),DataArray2%(3,50)
Index1%=6
CALL ExamineIndex(Index1%,DataArray1%(),"DataArray1")
Index2%=1
CALL ExamineIndex(Index2%,DataArray2%(),"DataArray2")

SUB ExamineIndex(ArrayIndex%,Array%(),ArrName$)
  REM Check array boundary conditions before
  REM attempting to examine any elements ...
  IF ArrayIndex% > UBOUND(Array%) THEN
    PRINT "First index for array ";ArrName$;" too great"
  ELSEIF ArrayIndex% < LBOUND(Array%) THEN
    PRINT "First index for array ";ArrName$;" too small"
  ELSE              ' Print partial contents of array
    PRINT "Dumping array ";ArrName$;" contents ";
    PRINT "using first index";ArrayIndex%
    FOR i%=LBOUND(Array%) TO UBOUND(Array%)
      PRINT "Element (";ArrayIndex%;",";i%;") equals";
      PRINT Array%(ArrayIndex%,i%)
    NEXT
    PRINT "End of dump."
  END IF
END SUB
```

USER-DEFINED FUNCTIONS

This section discusses the method by which you can make up new functions beyond those supplied "built-in". It is discussed in this chapter due to the close relation functions have with variables; both specify a value and both have types and names.

To declare a new function in AC/BASIC you must define its name and formula using the following format:

```
DEF FNname[(parameter-list)]=function-definition
```

Description:

Associates a numeric or string expression with a function name.

The `name` field may be any legal variable name and must immediately follow the "FN" without any intervening spaces. If the function name does not have an explicit type declaration character (#,!,$,%) it will be affected by any DEFtype statements which apply to variables beginning with the same first letter. (See **DEF**, in "Declaring Variable Types".)

`Parameter-list` is optional and consists of a list of variables (separated by commas) that will be passed as arguments to the function.

`Function-definition` is an expression that is resolved using the arguments passed in the function parameter list and/or global variables, and returned as the function result, and may be any legal BASIC string or numeric expression. This is the "formula" for the new function.

When the function is referenced, the values of the parameters are assigned to the corresponding function arguments in the `function-definition;` the parameter variables (the arguments) are local to the function and are distinct from program variables of the same names. If a variable used in the `function-definition` does not occur in the `parameter-list`, the current global value of that variable will be used.

User function definitions are lost when the current program exits to another program. As with other statements (**COMMON**, DEFtype, etc.), definitions or declarations desired in chained programs must be appropriately included.

Example:

```
REM Example of a string-valued function
DEF FNFile$(FileName$)=FileName$+".DAT"
INPUT "Enter data file name"; A$
IF A$<>"" THEN OPEN FNFile$(A$) FOR INPUT AS FILE 1
.
.
```

This example prompts the user for a data file name and automatically appends a ".DAT" extension before opening it.

Any user function definitions must precede their use in a program's source code. Consider the following example.

```
    REM Chapter7/Example11
    GOSUB InitRoutine
    PRINT FNDouble(5)
    END
  InitRoutine:
    DEF FNDouble(x) = x*2
    RETURN
```

When compiled, a "Function not defined previously" error would be reported for the "**PRINT** FNDouble(5)" statement, because the function definition occurs after the function's usage.

The following would compile and execute successfully, producing correct output:

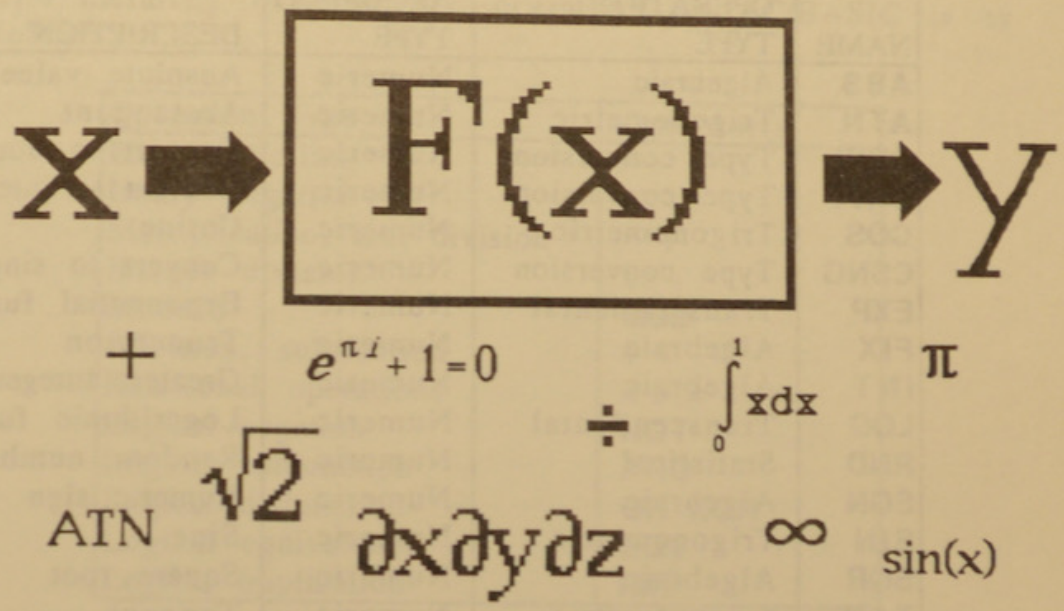
```
REM Chapter7/Example12
DEF FNDouble(x) = x*2
PRINT FNDouble(5)
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
END
```

In the case of multiple function definitions (i.e., with the same name), each definition has effect for the body of code between it and the next definition (lexical scope):

```
    REM Chapter7/Example13
    pass=0
    DEF FNz%(num%)=num%^2 ' FNz returns num% squared
DoAgain:
    PRINT FNz%(5);
    DEF FNz%(num%)=num%^3 ' FNz is now cube function
    PRINT FNz%(2);
    IF pass=0 THEN pass=1 : GOTO DoAgain
    PRINT "done"
    WHILE MOUSE(0) = 0 : WEND
    END
```

Run as a compiled application, it will print " 25 8 25 8 " because the first FNz% definition has effect for all the statements up to the second definition. See also conventions for **DEF**type declarations, in this chapter.

CHAPTER 8



MATH FUNCTIONS AND NUMERIC EXPRESSIONS

AC/BASIC provides a number of built-in functions which allow you to do trigonometric operations, type conversions, logarithmic operations, and other operations which return numeric results. This chapter explains in detail all the built-in functions, including differences between the binary and decimal implementations. It also includes information about how AC/BASIC evaluates mathematical and relational expressions, and about how numerical types are converted during mixed-type arithmetic operations. Built-in functions relating to other features of AC/BASIC such as I/O and graphics are discussed in the appropriate chapters.

The following table is a summary of the functions described in this chapter, listing the operation type, argument type, and a brief description of each. Not listed, but included in this chapter because of its close association with the **RND** function, is the **RANDOMIZE** statement.

Function Summary

NAME	OPERATION TYPE	ARGUMENT TYPE	DESCRIPTION
ABS	Algebraic	Numeric	Absolute value
ATN	Trigonometric	Numeric	Arctangent
CDBL	Type conversion	Numeric	Convert to double-precision
CINT	Type conversion	Numeric	Convert to integer
COS	Trigonometric	Numeric	Cosine
CSNG	Type conversion	Numeric	Convert to single-precision
EXP	Transcendental	Numeric	Exponential function
FIX	Algebraic	Numeric	Truncation
INT	Algebraic	Numeric	Greatest integer less than
LOG	Transcendental	Numeric	Logarithmic function
RND	Statistical	Numeric	Random number
SGN	Algebraic	Numeric	Numeric sign
SIN	Trigonometric	Numeric	Sine
SQR	Algebraic	Numeric	Square root
TAN	Trigonometric	Numeric	Tangent
VAL	String utility	String	Numeric value in string

EXPRESSIONS AND OPERATORS

The following sections describe the formation rules for mathematical expressions in AC/BASIC. A number of important topics are discussed; the rules for type conversion are especially important. Many of the rules are stated twice: once for binary math and once for decimal math. If you are undecided as to which math type to use, these sections can give you additional insight into the differences.

Evaluation of Expressions

The evaluation of any numeric expression depends on the precedence of the operators and the precision of the operands involved.

Hierarchy of Operations

The hierarchy of arithmetic operations in AC/BASIC is as follows:

Operation Type	Operator(s)
Exponentiation	^
Unary negation	-
Multiplication, real division	* /
Integer division	\
Modulo	MOD
Addition, subtraction	+ -
Relational operations	< > = <= >=
Logical negation	NOT
Logical conjunction	AND
Logical disjunction	OR XOR
Logical equivalence	EQV
Logical implication	IMP

Operations with the same precedence are evaluated in order of occurrence in the expression; as in standard mathematics, any operation has explicit priority when enclosed in parentheses, e.g.

```
AVG% = (X1% + X2%)/2
```

The discussion of logical operators is presented in a later section; all the other operators are self-explanatory with the exception of the modulo and "\" (integer division) operators; these two are explained in the following paragraphs.

MOD is often referred to as the "remainder function" as it gives the remainder of the integer division of its two arguments. More specifically, the the value of

```
x MOD y
```

is the remainder of x divided by y. Thus the result of "13 **MOD** 10" is 3, the result of "25 **MOD** 7" is 4, and so on.

The statement

```
TEMP=X MOD Y
```

is exactly equivalent to

```
TEMP=X-Y*(X\Y)
```

and

```
TEMP=X
WHILE TEMP >= Y
 TEMP=TEMP-Y
WEND
```

MOD is often useful for performing periodic activities based on a counter variable's value. The following example counts up to 10000, printing out the counter variable every 50 units:

```
FOR count%=1 TO 10000
 IF count% MOD 50 = 0 THEN
  LOCATE 1,1
  PRINT count%;
END IF
NEXT
```

Integer division, indicated by the "\" backslash operator, involves three conversions to integer values: the numerator and the denominator are both rounded before division, and the result is truncated after division.

Examples:

Expression	Yields
2.3\1	2
2.51\1.3	3
999\1000	0

Precision of Operands

If a mixed-type expression is encountered, i.e., one that has several different numeric types as operands, each variable or

subexpression result is converted to the type of the highest-precision variable in its subexpression. A subexpression is any expression within another expression which must be evaluated before the larger expression can be evaluated. An expression has an overall numeric type equivalent to the type of the highest-precision variable found within it. This will be illustrated shortly with an example. The correspondence between type and precision is as follows, in increasing order of precision:

Integer	(lowest)
Single-precision	
Double-precision	(highest)

Users concerned with the accuracy of arithmetic results should be careful when using mixed-type expressions with the binary math package, especially those that involve floating-point division and a conversion to higher-precision. The binary math package is the default on the compiler and in most cases produces the same arithmetical results as most interpreters. For example:

```
1 REM Example A
10 A% = 1
20 B! = 3
30 C# = 10
40 D# = A% / B! * C#
50 PRINT D#
```

In the expression "A% / B! * C#", the integer A% is converted to single-precision to match the (higher) precision of B! in the subexpression "A%/B!"; no rounding or conversion errors occur here. When the subexpression is evaluated it produces the single-precision result .3333333!. In order to evaluate the final subexpression ".3333333! *C#", the single precision term ".3333333!" must be converted to double-precision; this results in a conversion error with the compiler (if the program is compiled without the decimal math library), due to the internal storage format for binary numbers. A conversion error is not a run-time error: it means that inaccuracy is introduced into a numeric expression. Thus what is printed is

```
3.33333343267441
```

The same conversion errors can be introduced with constant expressions, e.g.

```
A# = 1/3
```

Since the default type for a floating-point division is single-precision, the same type of error as in the previous example occurs when converting the single-precision result to double precision for assignment to the variable A#. This error can be avoided by explicitly typing the expression:

```
A# = 1#/3
```

or

```
A# = 1/3#
```

Both divisions are performed in double precision so that conversion before assignment to A# is unnecessary.

Note that downwards precision-conversion presents no problems; for instance, this would be the case if the variable D# at line 40 in the example above were single-precision instead of double-precision.

When it is necessary to have conversion to a higher precision, use the **CDBL** and **CSNG** functions described in this chapter. If line 40 in Example A above were replaced by the line

```
40 D# = A% / CDBL(B!) * C#
```

the division operation would be performed in double precision, and no conversion would be necessary to match the precision of C#. The result of the division operation would then cause the example program to print

```
3.333333333333333
```

Another way to avoid rounding and/or conversion errors is to compile programs with the "D" option set ("Compile for Decimal Math"), which causes AC/BASIC to use the more accurate

decimal math formats for numbers instead of the default binary format. With the decimal math package, all mathematical functions and all arithmetical subexpressions are calculated in double precision, regardless of the type of the operands, before assignment to a destination variable. In Example A in the previous section, compilation with the "D" option set would cause the program to print the correct result, since the subexpression "A%/B!" would automatically be converted to the internal equivalent of "**CDBL**(A%)/**CDBL**(B!)".

The conversion functions **CVSBCD** and **CVDBCD**, **MKSBCD$** and **MKDBCD$** have been provided for conversion of single-precision or double-precision decimal format numbers to equivalent binary format numbers. (**MKSBCD$** and **MKDBCD$** are described in the next chapter.) No corresponding functions exist for the conversion of integers, since they have the same internal representation regardless of the math library used.

For more information on binary versus decimal math see Appendix A, "Internal Data Formats for Real Numbers".

Evaluation Errors

If during the course of execution of a program a zero occurs in the denominator of a division operation or if zero is raised via exponentiation to a negative power, AC/BASIC produces the run-time error code for "Divide by zero". If error-trapping is in effect, control will pass to the subroutine indicated with the **ON ERROR GOTO** statement; otherwise, the error number is displayed (and the physical file line number, if compiled with the "N" option set), and execution terminates.

Numeric overflow errors occur when a number is generated that exceeds the range of the current expression's numeric type, or when a number exceeds the numeric type of a destination variable in an assignment statement. For example,

```
I%=30000
J%=30000
M!=I%+J%
```

Logical (Boolean) Operators

The following sections describe the operators used to perform bitwise operations on integers. If a non-integer operand is given, it will be rounded to the nearest integer value before the operation is performed.

To determine the result of a logical operation, each bit of the operands (or single operand, in the case of the **NOT** operator) is examined and the operator's truth table is used to determine whether the corresponding bit of the result will be set, where a "1" indicates the bit is set and an "0" indicates the bit is off .

For AC/BASIC operations which return a logical result, like **EOF** and all the comparison operations (">", "<", "=", etc.), TRUE is indicated by a value of -1 (all bits set), and FALSE is indicated by 0 (no bits set). The tables below show -1 as 1 for simplicity.

NOT

A	NOT A
1	0
0	1

AND

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

OR

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

XOR

A	B	A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

EQV

A	B	A EQV B
1	1	1
1	0	0
0	1	0
0	0	1

IMP

A	B	A IMP B
1	1	1
1	0	0
0	1	1
0	0	1

BUILT-IN FUNCTIONS

With the rules for expressions out of the way, the various built-in functions will now be discussed. Since built-in functions return a value, they may be used as operands in more complicated expressions. If AC/BASIC does not provide a function required by a program, one can be defined by the **DEF FN** statement discussed in Chapter 7, "Constants and Variable Declarations".

ABS

Format:

ABS (x)

Description:

Returns the absolute value of the numeric expression x.

Example:

```
REM Chapter8/Example1
LET x% = -5
x% = ABS(x%)
PRINT x%
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The output from this program is:

```
5
```

ATN

Format:

ATN (x)

Description:

Returns the arctangent of x. The result is expressed in radians and will always be in the range -pi/2 to +pi/2.

If you are using the binary math run-time library, the calculation is performed in single precision unless x is a double precision expression. If you use the decimal math library, the calculation is performed in double precision.

Example:

PRINT ATN (3.6021)

will produce the following results:

Binary Math	Decimal Math
1.3	1.299999824834

CINT/CSNG/CDBL

Format:

```
CINT(x)
CSNG(x)
CDBL(x)
```

Description:

CINT returns the integer equivalent of the numeric expression x, **CSNG** returns the single-precision equivalent of x, and **CDBL** returns the double-precision equivalent of x.

The permissible ranges for each function argument are listed in the following table:

Function	Min x	Max x
CINT	-32768	32767
CDBL	±2.23E-308	±1.79E+308
CSNG	±1.18E-38	±3.4E+38

The values listed for **CSNG** and **CDBL** are for the default binary math package — (see Appendix A for the decimal ranges). Arguments outside of the specified ranges will produce an "Overflow" error at run-time.

As discussed in the section "Evaluation of Expressions" at the beginning of this chapter, a division operation such as

```
i%=10000
j%=30000
x#=i%/j%
```

would produce a slightly incorrect result (a single-precision .3333333 plus "garbage" in the double-precision places), because of a conversion error. Using a **CDBL** as follows would produce an accurate double-precision representation of the fraction 1/3, forcing the expression to be calculated in double precision:

```
x#=CDBL(i%)/j%
```

COS

Format:

COS (x)

Description:

Returns the cosine of x, where x is expressed in radians.

If you use the binary math run-time library, the calculation is performed in single precision, unless x is a double-precision expression. If you are using the decimal math run-time library, the calculation is always in double precision.

Example:

```
PRINT COS(.5)
```

will produce the following results:

Binary Math	Decimal Math
.8775826	.8775826189037

EXP

Format:

EXP (x)

Description:

Returns the value of e^x.

The constant e is the base of natural logarithms. It is approximately equal to 2.718282.

If x is too large, an "Overflow" error is generated. The size of x is dependent on the precision of the x and the type of floating point math being used (binary or decimal).

If you use the binary math run-time library, x in single precision can be up to 88.02996. x in double precision can be up to 709.5.

If you are using the decimal math run-time library, x can be no larger than 145.062.

Example:

```
PRINT EXP(1);EXP(2);EXP(3)
```

The example produces the following output:

```
Binary Math
2.718282  7.389056  20.08554

Decimal Math
2.718281828459  7.3890560989306  20.085536923188
```

FIX

Format:

```
FIX(x)
```

Description:

Returns the whole number portion of x.

Note that **FIX** simply truncates the fractional part of the expression x. If you want to get the whole number portion with rounding, use the **INT** function (described in the next section).

Example:

```
REM Chapter8/Example2
sng1! = 5.67 : sng2! = -5.6
PRINT FIX(sng1!);FIX(sng2!)
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The output of the example is:

```
5 -5
```

INT

Format:

```
INT(x)
```

Description:

Returns the greatest integer less than or equal to x.

If x is positive or zero, its value is simply truncated (as with **FIX**), and returned. For negative values, the integer returned is *less* than or equal to x.

Example:

```
PRINT INT(-5.6),INT(5.6)
```

This example prints

```
-6 5
```

LOG

Format:

LOG (x)

Description:

Returns the natural logarithm (base 2.718282) of x.

As with any logarithmic evaluation, the argument x must be non-zero and positive; otherwise, an "Illegal function call" error will occur. The upper limit of x is the largest number that is representable in the current math format (binary or decimal). If this value is exceeded, an "Overflow" error will occur.

When you use the binary math run-time library, the calculation is performed in single precision unless the expression x is in double precision. If you using the decimal math run-time library, the calculation is in double precision.

Example:

```
PRINT LOG (2)
```

The example prints

Binary Math	Decimal Math
.6931472	.69314718055993

RANDOMIZE

As stated earlier, **RANDOMIZE** is *not* a built-in function, but is presented here due to its direct relation to the built-in function **RND** (discussed in the next section).

Format:

RANDOMIZE [expression|**TIMER**]

Description:

Reseeds the random number generator.

Expression, when rounded, must be an integer between -32768 and 32767, or it may be **TIMER**. If you omit expression, your program will stop and prompt you for a random number seed when it encounters the **RANDOMIZE** statement:

```
Random Number Seed (-32768 to 32767)?
```

If you do not reseed the random number generator, it will return the same sequence of values every time the program is executed. The simplest and most random way to reseed the random number generator each time is by using **RANDOMIZE TIMER**. This causes the seed to be the number of seconds past midnight.

RND

Format:

RND [(x)]

Description:

Returns a random value between 0 and 1.

Using **RND**, a random number in any numeric range can be obtained. For example, a program seeking values from 15 to 19 would use the following line:

```
value%=15+INT(RND*5)
```

x is a numeric expression that effects the returned value. If x is greater than zero, the next number in the sequence is returned. If x is equal to zero, the last number generated is repeated.

RND(x), where x is negative, will cause the random number sequence to repeat starting with the first number generated after the first **RND**(x), where x is negative. See the example below for an illustration of its use.

When a program begins execution, the seed for the random number generator is defined. By default, it is always the same. This causes the sequence of random numbers to be the same every time a program is run. There are 2 ways of avoiding this. By either passing a negative value in x as explained above, or by reseeding the random number generator through the use of the **RANDOMIZE** statement.

Example:

```
REMDraw 1000 random dots in 4 colors
RANDOMIZE TIMER
FOR COL%=0 TO 51 STEP 17
   A=RND(-1) ' Restart sequence
   FOR Points%=1 TO 1000
      PSET (RND*WINDOW(2),RND*WINDOW(3)),COL%
   NEXT
NEXT
WHILE MOUSE(0)=0 :WEND ' Wait for mouse click
```

This example plots 1000 random points, then uses **RND**(-1) to plot the same points in two other colors.

SGN

Format:

SGN (x)

Description:

Returns information about the sign of expression x.

Returns a -1 if the numeric expression x is negative, 1 if positive, and 0 if the expression's value is zero.

Example:

```
ON SGN(x)+2 GOTO 100,200,300
```

This example branches to line 100 if x is negative, line 200 if x is zero, and line 300 if x is positive.

SIN

Format:

SIN (x)

Description:

Calculates the trigonometric sine function of angle x, where x is expressed in radians.

If you want to convert degrees to radians, multiply the degrees by pi/180.

If you are using the binary math run-time library, the calculation is performed in single precision unless expression x is double precision. If you use the decimal math run-time library, the calculation is in double precision.

Example:

```
REM Chapter8/Example3
Pi=ATN(1)*4 ' Close approximation of pi
DEF FNRadians(DegreeVal)=DegreeVal*Pi/180
INPUT "Calculate sine of which angle (degrees)";Angle
AngleInRads=FNRadians(Angle)
PRINT SIN(AngleInRads)
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example requests input of an angle in degrees, converts it to radians, and prints its sine.

SQR

Format:

SQR (x)

Description:

Returns the square root of the value of the variable or expression x; if x is negative, an "Illegal function call" error will occur.

If you are using the binary math run-time library, the calculation is performed in single precision unless expression x is double precision. If you use the decimal math run-time library, the calculation is in double precision.

Example:

```
PRINT SQR(9);SQR((4^3)*15-7)
```

will produce the following results:

Binary Math	Decimal Math
3 30.8707	3 30.870698080867

TAN

Format:

TAN (x)

Description:

Returns the tangent of the angle x, where x is in radians.

If you are using the binary math run-time library, the calculation is performed in single precision unless expression x is double precision. If you use the decimal math run-time library, the calculation is always in double precision.

Example:

PRINT TAN (.5)

will produce the following results:

Binary Math	Decimal Math
.5463025	.54630248984379

VAL

Format:

VAL (x$)

Description:

Returns a numeric value from the string expression x$.

The **VAL** function begins its scan with the first character in the string, ignoring leading blanks, tabs, and line feeds. The scan will end with the first character that cannot be part of a number. Thus a string may be passed to **VAL** that is not composed entirely of numeric characters. If the first character in the string is not a number or a decimal point (."), **VAL** returns 0.

The number returned is in the lowest precision that can accurately represent the value. For example,

Expression	Data type returned
VAL("1")	integer
VAL("101.1")	single-precision
VAL("1234.56789")	double-precision

If the number to be returned is larger than can be represented in double precision, an "Overflow" error results.

Note: Do not use **VAL** for the conversion of numbers stored in random-access files with **MKI$**, **MKS$**, or **MKD$** — use the **CVI**, **CVS**, **CVD** conversion functions described in Chapter 9.

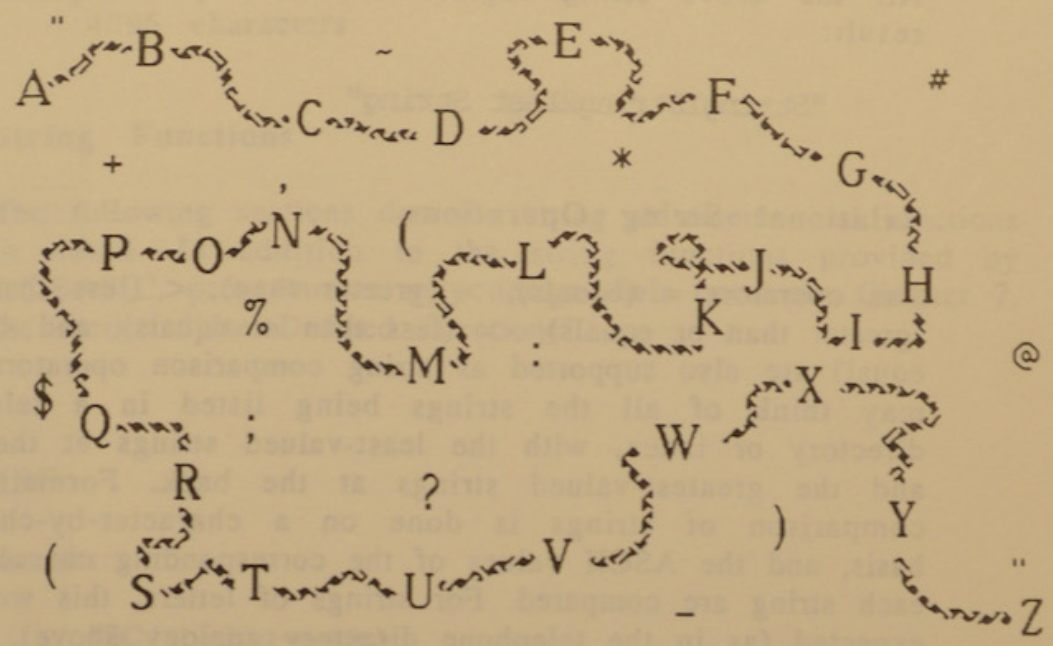
Examples:

```
a$="3"
b$="D3"
PRINT VAL(a$+b$)
PRINT VAL("43.2HOWDY")
PRINT VAL("50 E2")
PRINT VAL("4 5 5 . 2 E 2 0")
```

The output from these statements would be:

```
3000
43.2
5000
4.552D+22
```

CHAPTER 9



STRINGS

AC/BASIC has a large number of string statements and functions; however, there is only one string construction operator. Consequently, there is no need to discuss things like precedence in relation to string expressions; all string expressions are evaluated purely from left to right. String functions which may become operands in a string expression must be evaluated before the operation can take place.

The string construction operator provided by AC/BASIC is "+" and is called "concatenation". The concatenation operation merely joins two strings to form one string. Because string expressions are always evaluated left-to-right, there is no need for parentheses is string expressions, but parentheses may be used for clarity.

For example:

```
"String1"+"String2"+"Last String"
("String1"+"String2")+"Last String"
"String1"+("String2"+"Last String")
```

All the above string expressions above produce the same result:

```
"String1String2Last String"
```

Relational String Operations

The operators = (equals), > (greater than), < (less than), >= (greater than or equals), <= (less than or equals), and <> (not equal) are also supported as string comparison operators. One may think of all the strings being listed in a telephone directory or index, with the least-valued strings at the front and the greatest-valued strings at the back. Formally, the comparison of strings is done on a character-by-character basis, and the ASCII values of the corresponding characters in each string are compared. For strings of letters, this works as expected (as in the telephone directory analogy above), except that lower case letters are always valued higher than any upper case letters. When in doubt, consult the table of ASCII values in Appendix B.

Some examples of TRUE string comparisons (ones that return a logical result of -1):

```
"abc" > "ABC"
"B" < "B "
("howdy"+CHR$(12)) > ("howdy"+CHR$(11))
"ZZZ@" > "ZZZ."
```

String Maximums

Maximum length:
32767 characters.

Total string space:
limited only by available memory

Maximum length of a **BUTTON**, **WINDOW** title, or **MENU** string:
128 characters

Total string space for **BUTTON**, **WINDOW** title, and **MENU** strings:

4096 characters

String Functions

The following sections describe string statements and functions in detail. In addition to the string functions provided by AC/BASIC, programmers may design their own; see Chapter 7, the section "User-Defined Functions".

ASC

Format:

```
ASC(string-expr)
```

Description:

Returns the ASCII code of the first character in `string-expr`, which will always be a value between 0 and 255, inclusive.

See Appendix B for a table of the ASCII codes and their associated characters.

Example:

```
REM Chapter9/Example1
LET x$ = "String"
First% = ASC(x$)
PRINT First%
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The variable `First%` now contains the ASCII code for "S", which is 83.

The functions **ASC** and **CHR$** are inverses of each other: that is, the statements

```
AsciiVal%=ASC(CHR$(AsciiVal%))
```

and

```
SomeChar$=CHR$(ASC(SomeChar$))
```

where SomeChar$ is a string containing one character, do nothing. Neither AsciiVal% nor SomeChar$ have their values changed by the assignment statements.

CHR$

Format:

```
CHR$(x)
```

Description:

Returns the character with ASCII code x.

The most common use of the **CHR$** function is to output special (non-printable) characters. See Appendix B for a table of ASCII codes and their associated characters. X must be a value between 0 and 255 inclusive.

Example:

```
A$=CHR$(34)+"A quoted string"+CHR$(34)
PRINT A$
```

This illustrates the only way to include the double-quote character in a string from within a program (i.e., without user input). The output would be

```
"A quoted string"
```

The **CHR$** and **ASC** functions are inverses of each other; see the **ASC** function description above.

DATE$

Format:

```
DATE$                                  (function syntax)
DATE$=string-expression                (statement syntax)
```

Description:

Returns the system date in the form of a string or sets the date to `string-expression`. The **DATE$** statement is included in this section because of its close tie to the **DATE$** function.

The string generated by and set with **DATE$** is ten characters long and is of the form `mm-dd-yyyy`, where

`mm` is the month (01 to 12)
`dd` is the day of the month (01 to 31)
`yyyy` is the year (1900 to 2155)

The **DATE$** statement can be used to set the system date. The following two statements will each set the date to July 4, 1976,

```
DATE$="07-04-76"
DATE$="07-04-1976"
```

Example:

```
REM Chapter9/Example2
OPTION BASE 1
DIM month.name$(12)
FOR I%=1 TO 12
  READ mm$
  month.name$(I%)=mm$
NEXT I%
month% = VAL(LEFT$(DATE$,2))
day$ = MID$(DATE$,4,2)
year$ = RIGHT$(DATE$,4)
PRINT "Today is ";month.name$(month%);day$;", ";year$
WHILE MOUSE(0) = 0 : WEND '-- Wait for mouse click
DATA "January ","February ","March ","April "
DATA "May ","June ","July ","August ","September "
DATA "October ","November ","December "
```

HEX$

Format:

```
HEX$(x)
```

Description:

Returns the value of x as a hexadecimal string.

If x is non-integer, its value is rounded to the nearest integer before hexadecimal conversion. If x is negative, then the two's complement form is used. X must be between -65535 and 65535

Example:

```
PRINT HEX$(35)
```

This example prints "2 3", the hexadecimal equivalent of the decimal value 35.

See also **STR$** and **OCT$** in this chapter.

INSTR

Format:

```
INSTR([n,]x$,y$)
```

Description:

Searches for the first occurrence of the string y$ in x$ starting at the nth character in x$. If n is omitted, the search begins with the first character in x$.

If y$ is not present in x$, **INSTR** returns a 0; otherwise, the value returned is the starting position of y$ in x$. If n is greater than the length of x$, **INSTR** will always return a 0.

If n is a non-integer value it will be rounded; its value must be in the range 0 to 32767, inclusive.

Example:

```
REM Chapter9/Example3
PRINT INSTR ("string","ing")
PRINT INSTR(3,"Neat feat","ea")
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example prints:

```
4
7
```

LEFT$

Format:

```
LEFT$(x$,i)
```

Description:

Returns the first i characters of the string expression x$.

If i is non-integer, its value will be rounded; the value LEFT$ uses can be any valid positive integer (0 to 32767). If the length of x$ is less than or equal to i, then x$ is returned as the result.

Example:

```
REM Chapter9/Example4
REM Quotes shows exactly what strings are produced
Quote$=CHR$(34) ' Double-quote character
PRINT Quote$;LEFT$("string",3);Quote$
PRINT Quote$;LEFT$("Short string", 15000);Quote$
Cmd$="PRINT FILES"
PRINT Quote$;LEFT$(Cmd$,INSTR(Cmd$," ")-1);Quote$
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example prints:

```
"str"
"Short string"
"PRINT"
```

LEN

Format:

```
LEN(x$)
```

Description:

Returns the length of the string expression x$

Example:

```
REM Chapter9/Example5
LET x$ = "2 blank string"
PRINT LEN (x$)
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The value printed is 14.

MID$

Formats:

```
MID$(string-expr,n[,len])                (function syntax)
MID$(string-var,n[,len])=string-expr     (statement syntax)
```

where

0 < n =< 32767
0 =< length =< 32767.

Description:

The **MID$** function returns a string of length len characters starting with the nth character in string-expr. If the length specification is omitted, all characters from position n to the end of the string-expr are returned.

The statement form of **MID$** replaces a maximum of len characters in string-var, starting at position n, with the text of string-expr. If the length of string-expr is less than length, or if the len specification is omitted, **MID$** will attempt to replace only as many characters as are in the string-expr. In all cases the length of string-var is unchanged: if the replacement result is longer than string-var, the additional characters will be truncated on the right.

If n or len are not integer values they will be truncated before evaluation proceeds; an "Illegal function call" error will be generated for values outside of the specified ranges.

MID$ may be used to assign values to fielded strings which are defined by the FIELD statement described in Chapter 11.

Example:

```
REM Chapter9/Example6
a$="BASIC compiler"
PRINT MID$(a$,7,7)    ' 7 chars starting at position 7
PRINT MID$(a$,7)      ' All chars starting at position 7
PRINT MID$(a$,5,30)   ' 30 chars or to end from pos 5
MID$(a$,2)="a" : PRINT a$
MID$(a$,4,50)="i" : PRINT a$
MID$(a$,7)="COMPILER FOR THE IIGS"
PRINT a$
WHILE MOUSE(0) = 0 : WEND '-- Wait for mouse click
```

The output from this example would be:

```
compile
compiler
C compiler
BaSIC compiler
BaSiC compiler
BaSiC COMPILER
```

OCT$

Format:

OCT$(n)

Description:

Returns the value n as an octal string.

The value of n is rounded to the nearest integer before the conversion to octal. If n is negative, then the two's complement form is used.

Example:

```
PRINT OCT$(8)
```

This example prints 10, which is the octal value of decimal 8.

See also **HEX$** and **STR$** in this chapter.

RIGHT$

Format:

```
RIGHT$(string-expr,n)
```

Description:

Returns the last n characters of the specified string expression.

If n is a non-integer expression, it is rounded before the **RIGHT$** result is evaluated; the (rounded) value must be a valid positive integer (0 to 32767) or an "Illegal function call error" will be generated. If n is greater than or equal to the length of x$, **RIGHT$** will return the text of string-expr.

If the string-expr is null, the resulting string is always null.

Example:

```
REM Chapter9/Example7
LET x$ = "string"
PRINT RIGHT$(x$,3)
PRINT RIGHT$(x$+" expr",15)
WHILE MOUSE(0)=0 : WEND
```

The output from this example will be:

```
ing
string expr
```

SPACE$

Format:

```
SPACE$(n)
```

Description:

Returns a string of n spaces.

N can be any value from 0 to 32767, and is rounded before the string of spaces is built. If n is negative, an "Illegal function call" is generated. If n is greater than 32767, an "Overflow" error results.

Example:

```
PRINT " 12345"
PRINT "[" SPACE$(5) "]"
```

Prints:

```
 12345
[     ]
```

STR$

Format:

```
STR$(x)
```

Description:

Returns the string representation of the numeric expression x. STR$ returns a string with a leading space and no trailing space if x is positive and a leading minus sign if x is negative.

Example:

```
REM Chapter9/Example8
INPUT int1%
digits% = LEN(STR$(int1%)) - 1 ' Subtract 1 for leading space
PRINT digits%
WHILE MOUSE(0) = 0 : WEND '-- Wait for mouse click
```

This example stores the number of digits in `int%` in `digits%`. See also **VAL**, **HEX$**, and **OCT$** in this chapter.

STRING$

Format:

```
STRING$(n,j)
STRING$(n,x$)
```

Description:

The first format returns a string of `n` characters, each having ASCII value `j`.

The second form of **STRING$** returns a string of `n` characters, where each is the first character found in the string expression x$.

N can be from 0 to 32767, and `j` must be in the range 0 to 255; both will be rounded if non-integer.

Example:

```
REM Chapter9/Example9
LET dashes$ = STRING$(5,45)
PRINT dashes$;"HEADER";dashes$
PRINT STRING$(16,"*")
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The output is:

```
-----HEADER-----
****************
```

TIME$

Format:

TIME$	(function syntax)
TIME$="string-expression"	(statement syntax)

Description:

Returns the system time in the form of a string or sets the time to `string-expression`. The **TIME$** statement is included in this section because of its close tie to the **TIME$** function.

The string generated by or set with **TIME$** is eight characters long and is of the form `hh:mm:ss`, where

`hh`	is the hour (00 to 23)
`mm`	is the minutes (00 to 59)
`ss`	is the seconds (00 to 59)

The time may be set with the **TIME$** statement using the same string format. For example,

```
TIME$="12:34:56"
```

The system time may also be set with the Apple IIGS Control Panel.

UCASE$

Format:

```
UCASE$(x$)
```

Description:

Returns the equivalent of the string expression x$ with all alphabetic characters in upper case.

UCASE$** is very convenient for "genericizing" string expressions for comparison. For example, if a program requires a user to input a command string, then **UCASE$ can be used for easy identification of a keyword:

```
INPUT Cmd$
Cmd$=UCASE$(Cmd$)
IF Cmd$="PRINT" THEN
ELSEIF Cmd$="QUIT" THEN
```

Only alphabetic characters are changed; others characters remain the same..

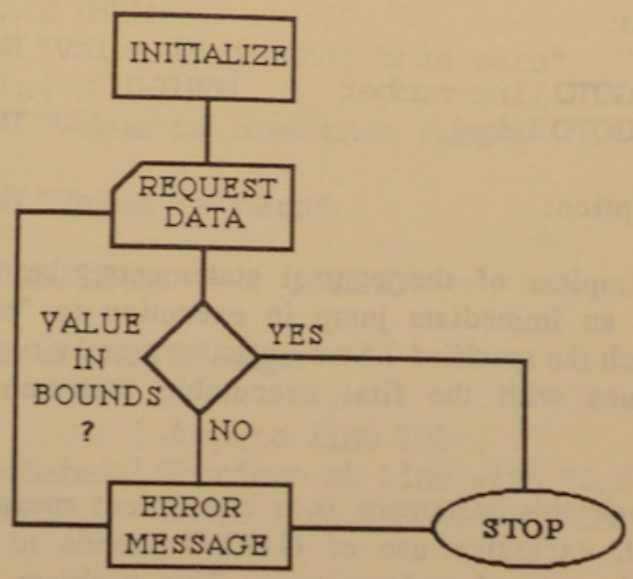
Examples:

```
PRINT UCASE$("Hello")
PRINT UCASE$("AaBbCcDd")
PRINT UCASE$("z&&y((x!!")
```

The output from the above would be:

```
HELLO
AABBCCDD
Z&&Y((X!!
```

CHAPTER 10



CONTROL STATEMENTS

Ordinarily, BASIC programs are executed sequentially by stepping through and processing each statement, one at a time. Certain statements can, however, cause the flow of execution to be interrupted and control to pass to another statement not immediately following the one currently executing. This chapter is devoted to the explanation of the function and use of these statements.

GOTO

Format:

```
GOTO line-number
GOTO label
```

Description:

The simplest of the control statements, the **GOTO** statement causes an immediate jump in execution (a "branch") to the line on which the specified `line-number` or `label` resides. Execution continues with the first executable statement on or after that line.

Although this statement is a convenient means of program flow control, excessive use of **GOTO**s tends to make the program difficult to read and maintain. This problem can be avoided by using AC/BASIC language statements such as **IF...THEN...ELSE** and **WHILE...WEND**. These are structures which perform jumps to sections of code, but do so in a much more orderly and uncluttered fashion. As an example, consider the following code:

```
     REM Chapter10/Example1
     INPUT "Enter X%";x%
     IF x% > 0 THEN GOTO 260
     IF x% = 0 THEN GOTO 280
     PRINT "Value is less than zero"
     GOTO 300
 260 PRINT "Value is greater than zero"
     GOTO 300
 280 PRINT "Value is zero"
 300 WHILE MOUSE(0) = 0 : WEND
```

Although this use of **GOTO** is fairly efficient, the code is opaque and difficult to read. The same function can be coded with an **IF...THEN...ELSE** construct, with a significant increase in clarity:

```
REM Chapter10/Example2
INPUT "Enter X%";x%
IF x% > 0 THEN
   PRINT "Value is greater than zero"
ELSEIF x% < 0 THEN
   PRINT "Value is less than zero"
ELSE
   PRINT "Value is zero"
END IF
WHILE MOUSE(0) = 0 : WEND
```

Other examples of **GOTO** usage:

```
GOTO 100          ' Jump to line 100
GOTO InfernoLabel ' Continue at line with "InfernoLabel:"
GOTO PRINT.DATA   ' Go to line labelled "PRINT.DATA:"
```

GOSUB

Format:

```
GOSUB line-number
GOSUB label
```

Description:

Like **GOTO**, **GOSUB** causes an immediate jump or branch in execution to the statements at and following the specified line (a subroutine), but when a **RETURN** statement is encountered, execution will resume at the next statement following the **GOSUB**.

A subroutine is simply a section of code designed to be invoked by a **GOSUB** call and exited via one of the forms of the **RETURN** statement. A subroutine may contain more than one **RETURN** statement, may appear anywhere in the main program or within subprograms, and may be called any number of times during the execution of a program. Also, any subroutine may be called from within another subroutine; the limits of subroutine "nesting" are dependent only upon the amount of memory available at run time.

Subroutines may be used to eliminate repetitions of sections of code or to perform often-used services or functions. For example,

```
IF ERR=20 THEN BEEP : GOSUB AskContinue


PRINT "At a stopping point ..."
GOSUB AskContinue

AskContinue:
   INPUT "Continue";ACAnswer$
   IF UCASE$(LEFT$(ACAnswer$,1)) = "Y" THEN
      RETURN ' Answer was yes, continue
   ELSEIF UCASE$(LEFT$(ACAnswer$,1)) = "N" THEN
      PRINT "Closing all files"
      CLOSE
      STOP
   ELSE
      PRINT "Please respond with YES or NO"
      GOTO AskContinue
   END IF
```

This subroutine will perform correctly only if accessed via a GOSUB statement, otherwise a "RETURN without GOSUB" error will occur if the user enters a "Y" response. The same is true for all subroutines or sections of statements with **RETURN**s; it is good practice to ensure that there are **STOP**, **SYSTEM**, **END**, or other control statements before the bodies of all subroutines to avoid accidental execution.

Example:

```
        REM Chapter10/Example3
        GOTO 100
SUB1:   REM *** Print an integer
        PRINT "The integer value is:",int1%
        RETURN
100     INPUT "Please enter an integer value";int1%
        GOSUB SUB1
        WHILE MOUSE(0) = 0 : WEND
        END
```

The possibility of accidental execution can be avoided with a much slicker version of the subroutine, the "subprogram"; subprograms cannot be accidentally executed and are much more modular and portable than subroutines. See Chapter 15 for all the details about subprograms.

Subroutine labels are not to be confused with subprogram names, e.g.

```
REM Chapter10/Example4
GOSUB SUB1
END

SUB SUB1
  PRINT "Inside a subprogram"
END SUB
```

The example will generate an "Undefined label" error during compilation.

RETURN

Format:

```
RETURN
RETURN line-number
RETURN label
```

Description:

All three forms of **RETURN** are used to exit from subroutines called by **GOSUB**.

The **RETURN** statement without a `line-number` or `label` causes execution to resume with the statement immediately following the calling **GOSUB**, i.e., the most recent **GOSUB** executed.

If `line-number` or `label` is specified, the subroutine is exited and control passes immediately to the line with the given line number or label. If using **RETURN** within a subprogram, you may not use this form of the **RETURN** statement to return to lines in the main program or in other subprograms; likewise, you cannot use **RETURN** to transfer control from the main program into a subprogram.

RETURNing to a line within a looping structure without having executed the loop control statement (**FOR** or **WHILE**) is a dubious practice. In a compiled application, the results will be unpredictable. For example,

```
    REM Chapter10/Example5
 10 GOSUB 60
 20 FOR I=1 TO 5
 30   array(I) = I
 40 NEXT I
 45 WHILE MOUSE(0) = 0 : WEND '-- Wait for click
 50 END
 60 I = 3
 70 RETURN 30
```

You may use as many **RETURN** statements as your program requires, wherever it may require them; however, if a **RETURN**

is executed in any section of code that has not been called via a **GOSUB**, a "RETURN without GOSUB" error will be generated.

Example of **RETURN** usage:

```
    GOSUB CheckSub
    PRINT "Value OK - proceeding"
    .
    .
ErrorTrap:
    PRINT "Value out of range"
    .
    .
CheckSub:
    REM Examine TestVal & take appropriate action
    IF TestVal% > 255% THEN RETURN ErrorTrap
    RETURN
```

For more information on **RETURN** and subroutines, see the reference to **GOSUB** in this chapter.

ON...GOTO / ON...GOSUB

Format:

```
ON expression GOTO line-list
ON expression GOSUB line-list
```

where `line-list` is of the form

```
{line-number|label} [,[line-number|label] ...]
```

Description:

Executes a **GOTO** or **GOSUB** branch to a `line-number` or `label` in `line-list` depending upon the rounded result of the given expression: a result of one will cause a branch to the first line or label in `line-list`, a result of two to the second line or label, etc. If the **ON...GOSUB** form is used, each line and label in the `line-list` should be the beginning of a subroutine.

If the rounded result of `expression` is zero, or greater than the number of elements in the `line-list`, execution will continue with the statement immediately following the **ON...GOTO** or **ON...GOSUB** statement.

The line numbers and labels in `line-list` must be separated with commas. If the `line-list` contains two consecutive commas and the expression evaluates to the value which would select the line or label at that position, execution continues with the next statement.

The **ON...GOTO** and **ON...GOSUB** statements are convenient for eliminating lengthy **IF...THEN...ELSE** constructions: the following code,

```
   REM Chapter10/Example6
GetChoice:          ' Let user select a feature
   INPUT "Input your choice (1 to 4)";Choice%
   IF Choice%=1 THEN
      GOSUB Accounts
   ELSEIF Choice%=2 THEN
      GOSUB Records
   ELSEIF Choice%=3 THEN
      GOSUB Employees
   ELSEIF Choice%=4 THEN
      GOSUB Products
   ELSE
      PRINT "Please enter a number from 1 to 4"
      GOTO GetChoice
   END IF
   WHILE MOUSE(0) = 0 : WEND : END

Accounts:
   PRINT "Accounts Processing"
   RETURN
Records:
   PRINT "Records Processing"
   RETURN
Employees:
   PRINT "Employee Processing"
   RETURN
Products:
   PRINT "Products Processing"
   RETURN
```

may be replaced by a cleaner version that uses the **ON...GOSUB** statement:

```
   REM Chapter10/Example7
GetChoice: ' Let user select a feature
   INPUT "Input your choice (1 to 4)";Choice%
   IF (Choice% >= 1) AND (Choice% <= 4) THEN
      ON Choice% GOSUB Accounts,Records,Employees,Products
   ELSE
      PRINT "Please enter a number from 1 to 4"
      GOTO GetChoice
   END IF
   PRINT "Done"
   WHILE MOUSE(0) = 0 : WEND : END

Accounts:
   PRINT "Accounts Processing"
   RETURN
Records:
   PRINT "Records Processing"
   RETURN
Employees:
   PRINT "Employee Processing"
   RETURN
Products:
   PRINT "Product Processing"
   RETURN
```

Other examples of **ON...GOTO** and **ON...GOSUB** usage (note that these are only program fragments and are not complete programs):

```
' On values 0,3, or > 6 this does no GOTOs
ON Command% GOTO 100,200,,300,anylab,exitlab

' Branches to sections of code to perform actions
' based on user-entered command character.
' Specifying "P" will drop through the ON...GOTO: note ",,".
ON INSTR("CPLQ",CMD.CHAR$)+1 GOTO Illegal,Copy.,,List.,QUIT
PRINT CMD.CHAR$;" - command not implemented"
```

IF CONDITIONAL STATEMENTS

IF statements allow conditional execution of a statement or block of statements based on the evaluation of a given expression.

IF...THEN...ELSE

Formats:

```
IF expression THEN then-clause
IF expression THEN then-clause ELSE else-clause
```

Description:

If `expression` is found to be true, i.e., evaluates to a non-zero value, `then-clause` is executed, which consists of all statements to the end of the line if no **ELSE** is present, or all statements on the line up to the corresponding **ELSE**. If an **ELSE** clause follows, and the **IF** expression is false, i.e., resolves to zero, `else-clause` is executed. If no **ELSE** clause is specified and the **IF** expression is false, program execution continues with the following line.

With either form of the **IF...THEN** one-line statement, a line number or label may be specified in place of the **THEN** or **ELSE** clause statements, and will be implicitly understood to be a **GOTO** statement. The following two lines are exactly equivalent:

```
IF Command$ = "Continue" THEN 200 ELSE FinishUp
IF Command$ = "Continue" THEN 200 ELSE GOTO FinishUp
```

Care must be taken when attempting to call a parameterless subprogram immediately after then **THEN** or **ELSE** keywords: the statement **IF** `c$="CURVE"` **THEN** `PlotCurve` will try to **GOTO** the label `PlotCurve` and will never invoke the subprogram `PlotCurve`. If you wish to invoke a parameterless subprogram immediately after the **THEN** or **ELSE** keywords, you must use the explicit **CALL** syntax, e.g. **IF** `c$="CURVE"` **THEN CALL** `PlotCurve`.

Any clause on the line may contain multiple statements, separated by colons (":"), which will all be executed if and only if the result of the expression selects that clause. Examples of multiple-statement conditionals are

```
IF V%<0 THEN PRINT "Bad Value!" : GOTO GetValue
IF Num>=0 THEN N = SQR (Num) : PRINT N ELSE N=0 : BEEP
```

In the first line, the **PRINT** and **GOTO** statements will be executed if the value of V% is less than zero; in the second, either the two statements following **THEN** or the two statements following **ELSE** will be executed.

Both the `then-clause` and `else-clause` may be omitted if it is desired that no action be performed, for example,

```
IF LEN (a$) <100 THEN ELSE PRINT "Too long" : STOP
```

Nesting **IF...THEN...ELSE** statements is permissible. If the statement does not contain the same number of **ELSE** statements as **THEN** statements, each **ELSE** is matched with the closest unmatched **THEN**. For example, in the line

```
IF (I < 5) THEN IF (I < 0) THEN I = 0 ELSE I = 5
```

the "**ELSE** I = 5" clause is the alternative to the second **THEN** and will be executed only if the value of I is greater than or equal to zero and less than five.

Example:

```
REM Chapter10/Example8
mypoints% = 100
yourpoints% = 500
win% = mypoints% > yourpoints%
IF win% THEN PRINT "I win"
IF NOT win% THEN PRINT "You win"
WHILE MOUSE (0) = 0 : WEND '--Wait for click
```

IF...GOTO

Format:

```
IF expression GOTO {line|label} [ELSE else-clause]
```

Description:

Causes a branch to the given `line` or `label` if the expression is true.

IF...GOTO `line` is functionally equivalent to **IF...THEN GOTO** `line` and **IF...THEN** `line`.

The **ELSE** case to this statement works exactly as described in the reference to **IF...THEN...ELSE**. If the `else-clause` is a line number or label, program control is transferred to that line or label.

Example:

```
IF A =B GOTO 100
```

Execution continues at line 100 if A and B are equal.

Block IF Statements

Format:

```
IF expression THEN
.
.
[ELSE]/[ELSEIF]
.
.
END IF
```

The block-**IF** statements represent extended forms of the one-line **IF...THEN...ELSE** statement, allowing conditional execution of whole sections (or "blocks") of code.

In its simplest form, the block-IF statement may be used to execute a block of statements if the IF expression is true (resolves to a non-zero value):

```
    REM Chapter10/Example9
    PRINT "Waiting for mouse click..."
waiting:
    IF MOUSE(0) <> 0 THEN
        PRINT "The mouse button was clicked"
        PRINT "X coordinate:" MOUSE(1)
        PRINT "Y Coordinate:" MOUSE(2)
        GOTO the.end
    ELSE
        GOTO waiting
    END IF
the.end:
    WHILE INKEY$ = "" : WEND '-- Wait for key
```

If an ELSE block is included, the statements between the ELSE and the END IF will be executed if the IF expression is false (resolves to a value of zero):

```
    REM Chapter10/Example10
    INPUT a!,b!,c!
    SqrExpr! = b!^2 - 4*a!*c!
    IF SqrExpr! >= 0 THEN
        PRINT "Equation has real roots"
        Root1! = (-b!+SQR(SqrExpr))/(2*a!)
        Root2! = (-b!-SQR(SqrExpr))/(2*a!)
        PRINT Root1!,Root2!
    ELSE
        PRINT "Equation has imaginary roots"
        PRINT "(";-b!;"+";SQR(-SqrExpr!);"i )/";2*a!
        PRINT "(";-b!;"-";SQR(-SqrExpr!);"i )/";2*a!
    END IF
    WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The ELSEIF statement is a special case of ELSE. ELSEIF conditionally executes a block of statements when the IF expression is false. This is different from a nested IF statement, since the ELSE statement that might follow is on the same level as the IF and ELSEIF statements. The ELSE block

may be thought of as the alternative to all the previous **IF** and **ELSEIF** blocks:

```
REM Chapter10/Example11
a = 1
IF a < 1 THEN
   PRINT "a is less than one"
ELSEIF a = 1 THEN
   PRINT "a is equal to one"
ELSE
   PRINT "a is greater than one"
END IF
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The **END IF** statement must be used to close a block-**IF** structure, even if the last block has only one statement, as above. The associated clause will become all the lines up to the **END IF** statement.

If program execution is transferred to a labeled **ELSE** or **ELSEIF** statement (with a **GOTO**, for instance), execution continues at the first line following the corresponding **END IF** statement.

CONTROL STRUCTURES

FOR...NEXT

Format:

```
FOR counter=initial TO final [STEP increment]
.
.
NEXT [counter[,counter...]]
```

where

`Counter` is a simple variable (not an array element) that will maintain the loop count.

`Initial` is the beginning value that is assigned to `counter`.

`Final` is compared with `counter` to determine when the loop is to be terminated.

`Increment` is the value that is added to `counter` each time a loop is completed. If the **STEP** keyword is omitted, the increment defaults to one.

Description:

Causes a block of statements to be executed a number of times in a loop; the **FOR** block is defined as all statements from the **FOR** statement to the corresponding **NEXT** statement.

When a loop begins, the `counter` is always assigned the specified `initial` value. After each iteration of the loop, the `increment`, which may be positive or negative, is added to the `counter`; the `counter` is then compared with the specified `final` value. If `increment` is positive and `counter` is greater than `final`, then the loop is exited. If `increment` is negative and `counter` is less than `final`, then the loop is exited.

If the `increment` is positive and the `initial` value is not greater than the `final` value, the **FOR** block will be executed at least once. The same is true if the `increment` is negative and the `initial` value is not less than the `final` value.

If the `increment` is positive and the `initial` value is greater than the `final` value, or if the `increment` is negative and the `initial` value is less than the `final` value, the **FOR** loop will not be executed at all and control will pass to the statement immediately after the closing **NEXT** statement. The following loops will not be executed at all:

```
FOR i%=1 TO 0
  .
  .
NEXT i%
PRINT i% ' Prints 1


FOR counter!=7.5 TO 10 STEP -.5
  .
  .
NEXT counter!
PRINT counter! ' Prints 7.5
```

When a **FOR** loop exists entirely inside another **FOR** block, the inner **FOR** loop is said to be "nested" within the outer loop. Nesting is used when a **FOR** loop construct must be executed more than once. An example is a bubble sort loop:

```
temp1.size = array.size - 1
FOR i=0 TO temp1.size
   temp2.size = array.size - i
   FOR j=2 TO temp2.size
      IF (array(j-1) > array(j)) THEN
         SWAP array(j),array(j-1)
      END IF
   NEXT j
NEXT i
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The limit of loop nesting is 16 levels. This limit includes all nested **WHILE...WEND** loops as well.

The **NEXT** statement defines the end of a **FOR** block. **NEXT** can take on one of three forms:

1. **NEXT** — (without a counter variable specified), the **NEXT** statement closes the block of the last **FOR** statement that is not currently matched with another **NEXT** statement.

```
   REM Chapter10/Example12
10 OPTION BASE 1 ' Set array base subscript to 1
20 DIM array%(10,20)
30 FOR I=1 TO 10
40    FOR J=1 TO 20
50       array%(I,J) = J + ((I - 1) * 20)
55       PRINT array%(I,J),;
60    NEXT  ' Terminates the FOR J statement at 40.
65    PRINT
70 NEXT     ' Terminates the FOR I statement at 30.
75 WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

2. **NEXT** `counter` — closes the block of a **FOR** statement with the same counter variable.

```
   REM Chapter10/Example13
10 OPTION BASE 1 ' Set array base subscript to 1
20 DIM array%(10,20)
30 FOR I=1 TO 10
40    FOR J=1 TO 20
50       array%(I,J) = J + ((I - 1) * 20)
55       PRINT array%(I,J),;
60    NEXT J    ' Terminates the FOR statement at 40.
65    PRINT
70 NEXT I       ' Terminates the FOR statement at 30.
75 WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

3. **NEXT** `counter1, counter2, ...` — Terminates more than one **FOR** loop. The counter variables must be ordered correctly:

```
    REM Chapter10/Example14
 10 OPTION BASE 1 ' Set array base subscript to 1
 20 DIM array%(10,20)
 30 FOR I=1 TO 10
 40    FOR J=1 TO 20
 50       array%(I,J) = J + ((I - 1) * 20)
 55       PRINT array%(I,J)
 60 NEXT J,I                 ' Terminates both loops
 65 WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

Note that if line 60 were "60 **NEXT** i,j" it would attempt to close the "FOR i" loop before the "FOR j" loop and would result in a "FOR without NEXT" error.

WHILE...WEND

Format:

```
WHILE expression
.
.
WEND
```

Description:

Loops through the block of statements delimited by the **WHILE** and **WEND** statements while `expression` is true (evaluates to a non-zero value).

When the **WHILE** statement is entered, the given `expression` is evaluated. If the result is false (zero), execution continues at the statement following the loop's associated **WEND** statement. If true (non-zero), the ensuing block of statements is executed up to the matching **WEND** and the process is repeated.

WHILE...WEND statements may be nested, and follow the same nesting rules as **FOR...NEXT** statements. The maximum depth of nesting (loops occurring entirely within other loops) is 16; this figure includes all nested **FOR** and **WHILE** loops. A **WHILE** statement is matched to the nearest **WEND** that is not

part of a nested internal loop, i.e., already matched with another **WHILE**.

Example:

```
REM Chapter10/Example15
LET I% = 1
WHILE (I% < 5)
   I% = I% + 1
   PRINT I%
WEND
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This **WHILE** loop iterates 4 times.

SYSTEM CONTROL STATEMENTS

CHAIN

Format:

```
CHAIN filespec
```

Description:

Exits the current program and begins execution of another program.

The `filespec` string expression must be present and must be the name of an AC/BASIC application; remember to include the ".RUN" extension if you have not renamed the compiled run file. An application compiled for binary math may **CHAIN** (or **RUN**) applications compiled for decimal math ("D" option), and vice-versa, but there may be complications — see the **$OPTION** metacommand description in Chapter 4.

If the specified filename does not exist, a "File not found" error results.

Note that the **CHAIN MERGE**, **CHAIN ... DELETE**, etc., variants are *not* supported by the compiler.

CHAIN is very similar to the **RUN** statement. The differences are:

1. **RUN** closes all files and devices (except when the R parameter is used), and **CHAIN** leaves them open.

2. **RUN** clears the output window and closes all screens and windows; **CHAIN** leaves all windows and screens as they are.

3. **RUN** terminates any current sounds being played; **CHAIN** does not.

4. **RUN** does not allow the passage of variables; with **CHAIN**, variables may be passed using **COMMON** statements.

Note that for both **CHAIN** and **RUN**, event trapping is disabled, the default array base subscript is reset to zero, all **DEF**type declarations are cleared, and all user defined functions (**DEF FN**) are discarded. The **PALETTE** color settings are retained in both cases.

In order for a variable to be passed from the chaining program to the chained program, it must be contained in an identical **COMMON** statement in both programs. See discussion of the **COMMON** statement in the next section.

COMMON

Format:

COMMON `variable-list`

Description:

`Variable-list` is a list of variable names separated by commas.

COMMON statements are used in conjunction with the **CHAIN** statement to pass variables to a chained program. The

variables to be passed must be contained in the `variable-list` of **COMMON** statements in both the chaining and chained programs. Any number of **COMMON** statements may be present in a program, but a variable can be contained in the `variable-list` of at most one **COMMON** statement.

The variables must also be declared common in the same lexical order in both programs; the number of **COMMON** statements used to do this is irrelevant. Thus the following columns of declarations are equivalent:

```
COMMON a%,b$,c!     COMMON a%,b$     COMMON a%
                    COMMON c!        COMMON b$
                                     COMMON c!
```

If they are not in the proper order, the mapping of values to program variables will not be correct and the results of program execution will be unpredictable.

A good method to use when chaining between programs with **COMMON** statements is to place the **COMMON** statements in a separate file and use the **$INCLUDE** directive to include the declarations in both programs (see Chapter 4, "Using Compiler Options and Metacommands").

Arrays appearing in **COMMON** statements must include the number of dimensions within parenthesis and must be dimensioned *after* the **COMMON** statement. For example,

```
REM Sequence used to declare arrays common
COMMON array%(3),str.array$(1)
DIM array%(3,4,5),str.array$(25)
```

RUN

Format:

RUN filespec[,R]

Description:

Transfers control to another AC/BASIC program.

Filespec is a string expression representing the name of the program to run. The R option following the file specification causes **RUN** to leave all data files open when exiting the current program. If R is not specified, the **RUN** statement will automatically close all open files.

Unlike **CHAIN**, the **RUN** statement cannot pass variables with the **COMMON** statement. Also, the **RUN** statement will close all screens and windows, and terminate any sounds being played; **CHAIN** will not.

As with **CHAIN** under the compiler, filespec must be the name of an AC/BASIC compiled application; remember to include the extension ".RUN". Applications compiled for binary math may **RUN** (or **CHAIN**) applications compiled for decimal math ("D" option), and vice-versa, but complications may arise — see the metacommand section of Chapter 4.

RUN line is not supported by AC/BASIC.

Example:

```
RUN "PROG2.RUN"
```

END / STOP / SYSTEM

Format:

```
END
STOP
SYSTEM
```

Description:

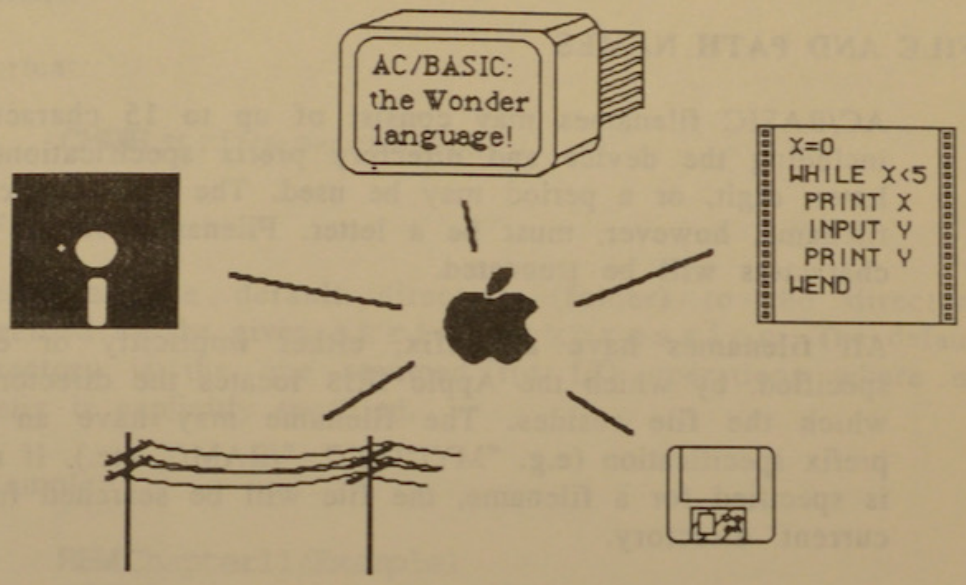
Each of these statements terminates execution of a program and closes all opened files. They statements are not required in a program, but are useful in stopping execution of a program at any point. An **END**, **STOP**, or **SYSTEM** statement need not follow the main program since execution will terminate without it. All three statements are included to maintain compatibility with various versions of BASIC.

Example:

```
REM Chapter10/Example16
RANDOMIZE 100
WHILE 1=1
   I% = RND * 50     ' Random value from 0 to 49
   PRINT I%
   IF (I% > 40) THEN END
WEND
```

This example will halt program execution if the value of I% becomes greater than 40.

CHAPTER 11



I/O STATEMENTS AND FUNCTIONS

The abbreviation "I/O" stands for "input/output", and in this chapter, refers to those statements which transfer data to and from files and devices.

In AC/BASIC a file is an entity where data can be stored and retrieved. In general, this refers to physical files on the disk(s), but can also refer to the Apple's screen and keyboard. The screen, the keyboard, the printer, and the serial port are treated as special files, and therefore are given the special designation "devices."

This chapter begins by discussing statements that refer to files on the Apple's disk(s) in their entirety. They do not directly relate to program I/O. Then, the two types of files (sequential and random-access) are discussed with the **OPEN** and **CLOSE** statements, which are performed before and after (respectively) all I/O operations on files. The next three sections explain sequential and random-access input and output statements, including related functions. The **CLOSE** section discusses a number of functions that may be used to obtain information about currently open files. The last section

presents statements which relate to data stored in the program itself.

FILE AND PATH NAMES

AC/BASIC filenames may consist of up to 15 characters, not including the device and directory prefix specification(s). Any letter, digit, or a period may be used. The first character of a filename, however, must be a letter. Filenames longer than 15 characters will be truncated.

All filenames have a prefix, either implicitly or explicitly specified, by which the Apple IIGS locates the directory within which the file resides. The filename may have an optional prefix specification (e.g. "MYDISK/", "/RAM/", etc.). If no prefix is specified for a filename, the file will be searched for in the current directory.

A prefix may be an asterisk (*) which represents the root directory of the disk from which the machine was booted.

Examples (the filenames are underlined in each):

LIB TXT	In current directory
*/LIB.DAT	In root directory of boot disk
*/DATA/text.file	In directory DATA of boot root
ACCOUNTS/PERSONAL	In the directory ACCOUNTS in the current directory

SYSTEM FILE STATEMENTS

These statements, with the exception of **CHDIR**, affect files in their entirety, and may be considered a level above direct program I/O. All of them relate to or change files in system directories, and most have Desktop equivalents.

CHDIR

Format:

```
CHDIR string-expression
```

Description:

Redefines the default directory (folder) to the directory specified by the given `string-expression`. The default directory is the one assumed for I/O operations where no prefix is explicitly specified.

Example:

```
REM Chapter11/Example1
OPEN "new.file" FOR OUTPUT AS #1
CLOSE 1
CHDIR "/RAM5"
OPEN "new.file" FOR INPUT AS #1
CLOSE 1
```

If the file `new.file` did not previously exist on the volume `/RAM5`, running this example will result in error 53 "File not found" at the second **OPEN** statement. The full name of the second file opened is `/RAM5/new.file`. A prefix name may still be explicitly specified to supercede the **CHDIR** prefix. For example, `/RAM5/new.file` will *always* refer to `new.file` on the `/RAM5` volume.

As with **OPEN**, any AC/BASIC statement that takes a file name as an argument will also be affected by any change to the default directory. These are, in full:

CHAIN	**FILES**	**KILL**
NAME	**OPEN**	**RUN**

The new directory is not only the default during program execution, but is also the default directory when the program exits to the operating system.

FILES

Format:

```
FILES [filespec]
```

Description:

Prints the names of the files on the specified volume, in the specified directory (folder), or checks for a file's existence.

If `filespec` is a filename, the file is searched for. If found, the name is displayed on the screen, otherwise, error 53 "File not found" is generated.

If `filespec` is a directory (folder), all files in the specified directory are displayed. If `filespec` is omitted, all files in the current (default) directory are displayed.

FILES may be used to detect if a file is present on the disk. To do this, error trapping must be set-up to detect error 53 "File not found" when the file does not exist. See Chapter 13 for an explanation of error trapping.

Examples:

```
FILES          ' List all files in current directory
FILES "/RAM5"  ' List all files in RAM5 directory
FILES "lightning.run"     ' Search for this file
```

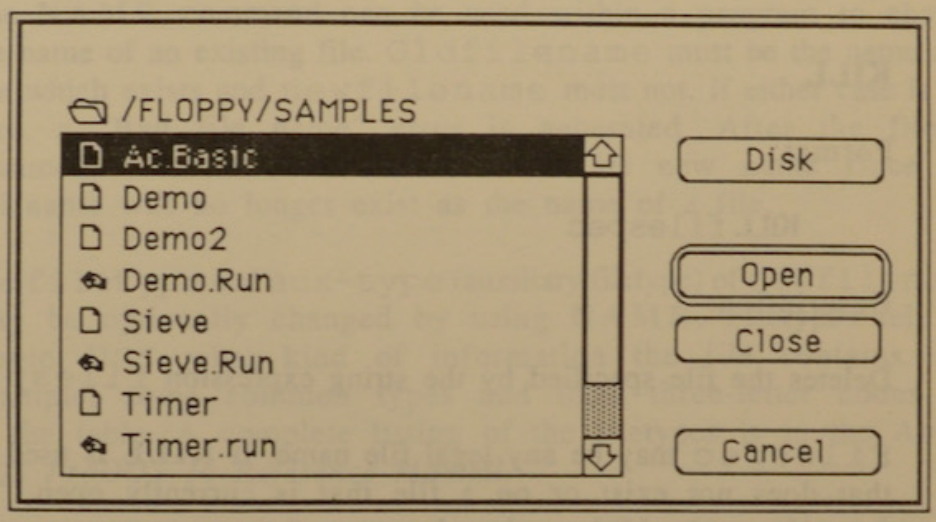
FILES$

Format:

```
FILES$ (function-number[,filetype-string])
```

Description:

Displays a standard file selector, similar to the one shown, allowing the user to select an existing filename or type a new filename.



When `function-number` is a `0`, the user is requested to type a new filename in the field provided. The `filetype-string` is ignored when `function-number` is `0`.

When `function-number` is a `1`, only an existing filename may be selected. The optional `filetype-string` may be used to display only the files of the specified types. To display all text and source files, for example, the `filetype-string` "TXTSRC" may be used. A partial table of filetypes appears in the description of the **NAME** statement a couple pages ahead. When `filetype-string` is omitted, files of all types are displayed in the file selector.

Neither form of the **FILES$** function opens, closes, runs, deletes, or renames a file. To do so, use the appropriate statements found in this chapter.

If the CANCEL button on the file selector is clicked, a null string is returned.

Examples:

```
old$=FILES$(1)              ' Get an existing name
new$=FILES$(0)              ' Ask for a new filename
appl$=FILES$(1,"S16EXE")    ' Get an application filename
text$=FILES$(1,"TXT")       ' Get a text-file name...
KILL text$                  ' ...and delete it
```

KILL

Format:

```
KILL filespec
```

Description:

Deletes the file specified by the string expression filespec.

Filespec may be any legal file name. If **KILL** is used on a file that does not exist or on a file that is currently open, "File not found" and "File already open" errors are produced, respectively.

Examples:

```
KILL "oldfile"
KILL DeleteFileName$
KILL filename$+".BAS"
```

NAME

Format:

NAME oldfilename **AS** newfilename [,filetype[,aux-type]]

Description:

Renames the file specified by the string expression oldfilename to newfilename.

The **NAME** command can be used within a program to change the name of an existing file. Oldfilename must be the name of a file which exists and newfilename must not. If either case is not true, a "Bad file name" error is generated. After the file is renamed, it must be referenced with its new name since the old name will no longer exist as the name of a file.

The filetype and aux-type (auxiliary filetype) of newfilename may be optionally changed by using **NAME**. Filetypes tell the Apple IIGS what kind of information the file contains. For example, some common types and their three-letter codes are in the table. A complete listing of the filetypes is in the *Apple IIGS ProDOS 16 Reference Manual.*

Description	Filetype
ProDOS 16 application program	S16
ProDOS 16 shell application	EXE
ASCII text file	TXT
APW source file	SRC
Directory file	DIR
Applesoft program file	BAS
Font file	FNT
Bad block file	BAD

Auxiliary types may be used by other applications, such as word processors or editors, to hold specific information about the file. The auxiliary type is specified as an integer number from 0 to 65535. The AC/BASIC compiler does not use the auxiliary type.

`Filetype` may also be specified as an integer number from 0 to 255. If an illegal filetype number is specified, the filetype "BAD" is assigned to the file.

Using **NAME** on an open file will cause an error.

Examples:

```
NAME "file1" AS "file2"
NAME A$+".BAS" AS A$+".BAK","TXT",0
NAME "old" AS "new",&HB5,&H0100
```

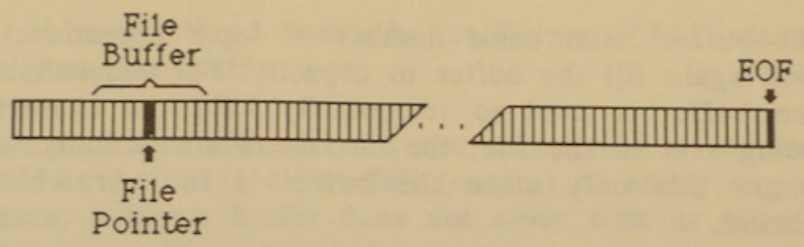
PREPARING FILE I/O

Before any I/O operations can be performed on a file or device, it must first be "opened" with the **OPEN** statement in one of the three available modes: input, output, and random-access. After the file has been opened, data may be read or written in one of several ways, depending upon the mode in which it was opened. After a file is no longer needed for I/O operations in a program, it may be "closed" with the **CLOSE** statement.

Sequential Files

Sequential files are composed of a sequence of ASCII characters that must be read or written one after another; data read from a sequential file cannot be re-read without first closing and re-opening the file. I/O operations on this type of file are very much like input from and output to the screen: characters may be read one-by-one or in groups (via **INPUT$** and **INPUT#**) or by line (**LINE INPUT#**), and may be output in the same way (by the **PRINT#** or **WRITE#** statements).

The fact that data may only be read or written to these files in sequential order gives rise to a construct called the *file pointer*, which is maintained internally by AC/BASIC to point to the next character in the file to be read or written. For some programs, it may be important to know where the file pointer will be after certain input operations and for determining certain file conditions, like end-of-file (**EOF**).



Sequential files are appropriate for use as data files when the data to be stored is variable in length or format, like the lines of a program, or text in a letter. Otherwise, for larger, more diverse collections of data, the random-access file is a much more efficient means of storage.

Random-Access Files

Random-access files differ from sequential files in that they are ordered by record rather than by character or line. These records are each numbered and may be read and re-read in any order without re-opening the file; hence the name "random-access." The length of each record is usually indicated by the **LEN** specification on the **OPEN** statement, and the structure of the records by one or more **FIELD** statements. The **FIELD** statements split each record into partitions as it is read in (via **GET**) and cause an automatic assignment of the data in the partitions to string variables. The data can then be manipulated from the program through these variables (using **LSET** and **RSET**), and any changes can be stored in a record by writing the contents of the record back to the file (via **PUT**).

Sequential I/O statements may be used for random access files. In such cases, the buffer is treated as a file of the length determined by the buffer size. When using **PRINT #**, a subsequent **PUT** will actually write the data to the file are the proper record.

File Buffers

A file buffer is an area in memory which AC/BASIC uses to store the partial contents of a file. When a sequential file is opened for input, AC/BASIC will read in as much of the file as will fit into the buffer; when the file pointer reaches the end of

the buffer (after some number of input operations), AC/BASIC will again fill the buffer to capacity. For sequential output files, the buffer is used to temporarily hold the characters that are being sent to the file; the characters are actually written to the output file only when the buffer is full, or when the file is closed.

For random-access files, the buffer is used to hold the contents of each record read or to be written. The **GET** statement fills the buffer with a new record, and the **PUT** statement writes the contents of the buffer to the file.

In general, larger buffers speed up programs that perform a large amount of I/O, since the number of disk read and write operations is reduced. File buffers may be up to 32,767 bytes in length, limited only by the amount of free memory on the Apple IIGS.

OPEN

Format:

```
OPEN filespec [FOR mode] AS [#]filenumber [LEN=buffer-size]
OPEN mode,[#]filenumber,filespec[,buffer-size]
```

Description:

Prepares a file or device for input or output (I/O) operations using the specified mode and buffer sizes, and associates it with the specified file number.

When a file is successfully opened, the integer expression `filenumber` is used to relate further I/O operations to the file with the name defined by the string expression `filespec`. As long as the file is opened, the number will refer to the file. The `filenumber` can be any number from 1 to 255, inclusive.

The `buffer-size` defines the size of the temporary holding area (the file buffer) for data being transferred to and from the file. For random-access files, this represents the record length. The default `buffer-size` is 128 bytes and may be as large as

32767, although there must be sufficient RAM memory to accommodate the buffer.

For sequential file I/O, data is not transferred to or from the file until the buffer is either full (output) or empty (input). For this reason, a larger buffer does not move data as often and is therefore more efficient and faster.

The mode string parameter specifies the type of I/O that will take place on the file being opened. In the first format, mode is specified by keywords as follows:

OUTPUT	sequential output mode
INPUT	sequential input mode
APPEND	sequential append

If no mode specification keywords are present, the mode defaults to random-access.

In the second, abbreviated form of the **OPEN** statement, the first character of the mode string expression is used to declare the mode:

I	sequential input mode
O	sequential output mode
A	sequential append mode
R	random-access mode

If a file is opened for input, the file pointer is positioned to the first character in the file; no output operations may be performed on a file opened for input.

If a file is opened in output mode, <u>its contents are first deleted</u>. No input operations may be performed on a file opened for output.

Append mode is a variant of output mode, the difference being that the contents are <u>not</u> deleted when the file is opened; instead, the file pointer is positioned after the last line or character in the file. This mode is convenient for adding to the contents of existing files.

If a file is opened for random-access, both input and output operations are permitted; these are usually done by the **GET** and **PUT** statements. See the "Random-Access I/O" section below.

There is a ProDOS limit of allowing no more than 8 files to be open at the same time.

The **OPEN** file specification can be a device name. The valid device names for the Apple IIGS are:

Device name	Addresses the
SCRN:	Screen
KYBD:	Keyboard
LPT1:	Printer
COM1:	Serial I/O port
CLIP:	Clipboard

The COM1: and LPT1: devices use the modem and printer ports, respectively, on the back of the Apple IIGS. Some additional settings, such as hardware hand-shaking, for these ports may be changed using the Control Panel which is accessed with the key combination Ó-CTRL-ESC. For more information about the Control Panel, refer to Appendix A of the *Apple IIGS Owner's Guide*.

COM1: may have up to four parameters specified on the open statement. The format is as follows:

```
COM1:[baud[,parity[,data-bits[,stop-bits]]]]
```

where:

baud is the baud rate, one of the following: 110, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, or 19200.

parity defaults to "E" (even), but may also be "O" (odd), or "N" (none).

data-bits specifies how many bits in each byte transmitted that represent actual data, and not parity bits or stop bits. May be 5, 6, 7, or 8.

stop-bits specifies how many bits will be used to indicate the end of each transmitted byte.

Example:

```
OPEN "COM1:4800,E,6,2" AS 9
```

LPT1: may have one of two possible parameters — DIRECT and PROMPT. The device specification LPT1:DIRECT is the same as having just LPT1: which causes printing to begin without any user intervention. LPT1:PROMPT, on the other hand, will prompt the user before printing with two printer dialog boxes from which various print styles, qualities, and formats may be chosen. Note that the *type* of printer to be used (i.e. LaserWriter or ImageWriter) must first be selected from the Choose Printer menu item. To use the LaserWriter, don't forget to select AppleTalk in the Control Panel and run the installing program CHOOSER.II on the System Disk when you first turn on the Apple IIGS.

The devices SCRN:, LPT1:, CLIP:, and COM1: can be opened for output. KYBD: and COM1: can be opened for input. Only COM1: can be opened for random-access I/O.

Example:

```
REM Chapter11/Example2
OPEN "NEWFILE" FOR OUTPUT AS 2 ' First syntax
OPEN "I",#1,"OLDFILE"          ' Alternate syntax
WHILE NOT EOF(1)               ' Loop until EOF.
   LINE INPUT #1,a$            ' Copy line from old
   PRINT #2,a$                 ' to new.
WEND
CLOSE                          ' Close all files
```

The example copies the contents of OLDFILE to NEWFILE.

CLOSE

Format:

```
CLOSE [[#]filenumber[,[#]filenumber]] ...
```

Description:

Closes all files with the specified file numbers. If no file numbers are specified, the **CLOSE** statement will close *all* open files.

The `filenumber` parameters are the numbers that were associated with files opened for input or output (I/O) operations with the **OPEN** statement. If a filenumber is specified which has no associated open file, no action is taken.

As in the **OPEN** statement syntax, the pound-sign ("#") is optional before the filenumber specification and has no effect on the statement's action.

After a file is closed, its number may be associated with another file using the **OPEN** statement. There can be no further I/O performed on the file after its channel number has been closed, unless it is opened with another **OPEN** statement.

Although all open files are automatically closed upon program exit, executing a **CLOSE** statement as soon as you are done with a file assures that all output to the file has been saved on disk. When performing output to files, the system will fill the file buffer (in memory) to capacity before actually writing data to your file; when a **CLOSE** statement is executed, the contents of the buffer are automatically written to the file. For this reason, if the system were to go down before the program exited, the contents of any output files would be potentially incomplete.

The **END**, **CLEAR**, **RESET**, **STOP**, **SYSTEM**, and **RUN** (without the "R" option) statements always close all files automatically.

Example:

```
REM Chapter11/Example3
OPEN "new.file1" FOR OUTPUT AS 1
OPEN "new.file2" FOR OUTPUT AS 2
PRINT #2,"This is a new file"
CLOSE #2                    ' Close "new.file2", only.
OPEN "i",#2,"new.file2"     ' Open again, for input
LINE INPUT #2,a$
PRINT #1,a$; ", too"
CLOSE                       ' Close all open files.
```

SEQUENTIAL INPUT

The sequential input statements provide a means of obtaining any number of characters or lines of data from the keyboard and from disk files, in contrast to random-access input (**GET**), which returns a structured block of fixed size.

The most common means of doing this is discussed first; then more exotic and specialized input statements are discussed, namely **INKEY$**, which gets a single character from the keyboard, and **INPUT$**, which retrieves a specified number of characters from the keyboard or a file.

INPUT

Format:

```
INPUT[;][prompt-string{;|,}]variable-list
```

Description:

Requests data to be input from the keyboard and assigns it to the variables specified in `variable-list`.

The variable list is a list of string and/or numeric variables, separated by commas, which are to be assigned values input by the user.

The **INPUT** statement suspends execution and if `prompt-string` is included, it will be printed before the program waits for input. `Prompt-string` must be a literal string enclosed in double-quotes; if you wish to use the contents of a string variable as a prompt, you must first use **PRINT** to display the prompt and then use the appropriate form of **INPUT**.

If a semicolon is used immediately before the variable list, or if only a variable list is given, the program will display a question mark on the screen to indicate that it is waiting for input from the keyboard; if a comma is used instead of a semicolon, no question mark is printed.

Examples of legal **INPUT** statements:

```
INPUT A$
INPUT FileName$, Record%, Code%
INPUT "Your name"; YourName$
INPUT ;"Which option?",opt%
```

With the compiler, the text pen position can be forced to remain at the end of the input data by including a semicolon immediately following the keyword **INPUT**; the carriage return will not be echoed to the screen.

If there is more than one variable in the **INPUT** variable list, the user must separate the data items with commas when entering at the keyboard. These data items will be assigned to the variables in the order in which they appear in the variable list. Strings need not be enclosed in quotes unless they contain commas or significant leading or trailing blanks; the quotes will be removed before assignment to the corresponding string variable. If the first character in the inputted string is not a quote, the string will consist of all the characters up to the next comma, regardless of any quotes that may be present.

The number of data items entered must match the number of variables in the variable list, and the types of the values entered must match the types of the corresponding variables. If any discrepancy is detected, the message "`?Redo from start`" will be displayed and the user will be prompted again.

Example:

```
INPUT "Test Values"; AnyString$,BVal%
```

Sample Input	AnyString	BVal%
somedata,5	somedata	5
"Oh, what a day",20	Oh, what a day	20
He said "Drink.",-30	He said "Drink."	-30
He said "Why, yes!",1	(?Redo from start)	
HOP,4,SKIP	(?Redo from start)	

The maximum number of characters which may be typed on an **INPUT** line at run-time is 256.

The second-from-last input line fails because of the comma inside the string; the last input line fails because three items are input when only two are requested.

For information on how to input an entire line of data from the keyboard, see **LINE INPUT**. A fancier form-like method of entry is available with the **EDIT FIELD** statement described in Chapter 13.

INPUT#

Format:

```
INPUT#filenumber,variable-list
```

Description:

Reads the variables specified in `variable-list` from the file associated with the given `filenumber`.

The `filenumber` is the number used when the file was opened for input. The `variable-list` contains variables, separated by commas, which will be assigned values read from the file.

The input format of the data follows the same conventions as that of user-entered data (see **INPUT** above), with a few

additions: when reading numeric data, the first character read which is not a space is assumed to be the start of a number. The number terminates on a comma, a space, or at end-of-line or end-of-file. With an **INPUT#** operation, the following numbers, `1.3 500 -35` in a file opened as filenumber 1 could be read by the statement

```
INPUT #1, A!,B%,C%
```

When an unquoted string is read, the string starts on the first non-blank character that is not a space. The string ends when a comma, line terminator, or the end-of-file is encountered. Quoted strings terminate at the next quotation mark or end-of-file; unlike the format for user-entered data with the **INPUT** statement, no comma is necessary between quoted strings and other data, only a space:

```
"Martin" "Ken" 25631 "Plumber"
```

The above data in a file could be read by

```
INPUT #1, LName$,FName$,SSN%,Occup$
```

Any string input from a file can be up to 32767 characters in length. Trailing blanks are removed from the strings.

Although not required for appropriately delimited data read by the **INPUT#** file input statement, the use of commas reduces the possibility of error and increases data file readability. The **WRITE#** statement is designed for this purpose, and is especially convenient for sending data to a file that will later be read by **INPUT#** statements: it performs the same action as **PRINT#**, but automatically quotes all strings and places commas between the outputted data items. See the reference to **WRITE#** in this chapter.

For a discussion on how to input an entire line of data from a file, see **LINE INPUT#**, in this chapter.

Example:

```
REM Chapter11/Example4
INPUT "Name"; AName$
INPUT "Rank"; Rank$
INPUT "Serial number"; SNo#
OPEN "DATAFILE" FOR OUTPUT AS 1
WRITE #1,AName$,Rank$,SNo#
CLOSE 1
OPEN "DATAFILE" FOR INPUT AS 1
INPUT #1,InName$,InRank$,InSer#
PRINT InName$,InRank$,InSer#
CLOSE 1
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example uses the **WRITE#** statement to write data to a file that can be directly re-read by the **INPUT#** statement.

LINE INPUT

Format:

```
LINE INPUT[;]["prompt-string";]string-variable
```

Description:

Reads an entire line of data from the keyboard into a string.

If `prompt-string` is supplied (in quotes), it is printed before the program waits for input. When the user types characters followed by a return, all characters up to (but not including) the carriage return and line feed characters will be assigned to the specified `string-variable`. If the user enters a string that contains commas or quotes, the entire string is assigned, commas and all. If an **INPUT** statement were used and unquoted commas were embedded in the input, the `"?Redo from start"` message would be displayed. Unlike the **INPUT** statement, **LINE INPUT** does *not* print a question mark.

With the compiler, the pen position can be forced to remain at the end of the inputted line by including a semicolon immediately following the keyword **INPUT**.

The maximum number of characters which may be typed on an **LINE INPUT** line at run-time is 256.

Example:

```
LINE INPUT "Enter name.",user$
LINE INPUT ;WholeLine$
```

LINE INPUT#

Format:

```
LINE INPUT #filenumber,string-variable
```

Description:

Reads data from a file, starting at the file pointer position in the specified file, up to a line terminator (carriage return) character or the end-of-file.

`Filenumber` is the number that was associated with the file using the **OPEN** statement. Characters will be read from the file until the line terminator or end-of-file is encountered, then assigned to the specified `string-variable` (not including the line termination characters).

The standard line terminator is a carriage return. If the sequence of carriage return and line feed appear in a file, both are used to terminate the line. After an **LINE INPUT#** statement, the file position pointer will be positioned after the line terminator and the next input will occur at the character following the carriage return character.

LINE INPUT# is especially useful for reading text files, which may contain commas and quotes that would make use of the **INPUT#** statement unacceptable.

This example program requests the user to input a file, opens it, and prints out all the lines in the file:

```
REM Chapter11/Example5
PRINT "Choose a file to display..."
File$=FILES$(1)
IF File$="" THEN END          ' Check for CANCEL
OPEN File$ FOR INPUT AS #1
WHILE NOT EOF(1)
   LINE INPUT #1, Text$
   PRINT Text$
WEND
CLOSE 1
PRINT : PRINT "Listing of "File$" complete."
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

INKEY$

Format:

INKEY$

Description:

Returns a character from the keyboard if one has been typed; otherwise, if no characters have been typed, **INKEY$** returns a null string ("").

INKEY$** does not wait for input, and characters are not echoed to the screen when retrieved by **INKEY$. The only keystroke sequence that is not passed to the program is Ó-., which causes a break in program execution.

If there is no window that is active for input, pressing a key causes the screen to flash and a beep to be generated. The key that was pressed is ignored.

One cautionary note: **INKEY$** does not retain its value from one call to the next. If you wish to use the value retrieved by a particular **INKEY$** call more than once, you must store the retrieved value in a string variable, e.g.

```
IF (INKEY$ <> "Y") AND (INKEY$ <> "N") THEN
   PRINT "Please respond with a Y or N"
   .
   .
END IF
```

This example will not work as expected — the input character must first be stored in a variable by a statement such as "ResponseChar$=**INKEY$**" before being examined in the **IF** statement.

Example:

```
10 a$=INKEY$ : IF a$="" GOTO 10
20 PRINT a$
```

This example pauses until any key is pressed. Line 10 is functionally equivalent to the statement a$=**INPUT$**(1) described in the next section.

INPUT$

Format:

INPUT$(x[,[#]filenumber])

Description:

Returns x characters in a string from a file or device.

The expression x indicates the number of characters to read. If filenumber is specified, the **INPUT$** operation reads characters from the file or device which was opened with that file number; otherwise, characters are read from the keyboard. Characters read from the keyboard with **INPUT$** are not echoed to the screen.

For file input, the maximum allowable size of x depends on the access mode: for sequential files, x can specify up to 32,767 characters; for random access files, x must be less than or equal

to the buffer size specified by the **LEN** option of the **OPEN** statement (128 bytes is the default).

When reading from a file, if more characters are specified than exist from the current read position to the end of file, the error "Input past end" will occur at run-time.

Example:

```
REM Chapter11/Example6
REM Copying files using INPUT$ and LOF functions
PRINT "Select source file name:"
source.file$=FILES$(1)
OPEN source.file$ FOR INPUT AS #1
file.length! = LOF(1) ' # chars in the source file
IF file.length! > 32767 THEN
   PRINT "Too large to copy with this method."
ELSE
   PRINT "Enter destination file name:"
   dest.file$=FILES$(0)
   OPEN dest.file$ FOR OUTPUT AS #2
   PRINT #2,INPUT$(file.length!,1) ' Input from file #1
END IF
CLOSE ' All open files
PRINT "done"
WHILE MOUSE(0) = 0 : WEND '-- Wait for mouse click
```

This example copies a source file to a destination file. The method used copies the contents of the source file into AC/BASIC's string space and immediately outputs to `dest.file$`. By using the string space, the copy can be done with one AC/BASIC statement, but the maximum length of the source file is restricted to the maximum size of a string (32767 characters).

SEQUENTIAL OUTPUT

This group of statements and functions relates to the outputting of text to the screen, disk files, and a printer. First, the **WIDTH** statement is discussed, since it may be used to control the line width of any device to which text output may

be sent. The simple **PRINT** statement, its variants, and associated functions (**SPC**, **TAB**, and **PTAB**) are explained next, followed by fancier means of outputting formatted text, the **WRITE** and **PRINT USING** statements. Lastly, those statements and functions are discussed which concern the position or location of text on output devices.

WIDTH

Format:

```
WIDTH output-device, [line-width] [,zone-width]
WIDTH #filenumber, [line-width] [,zone-width]
WIDTH LPRINT [line-width] [,zone-width]

WIDTH (string-expression)                    (function syntax)
```

Description:

Sets the `line-width` and `zone-width` (tab stop) for a file or device. The **WIDTH()** function returns the length, in pixels, of the `string-expression` with the current font characteristics; the function does not return the width of a device.

`Line-width` is the maximum number of characters that may be contained on a line of the specified file or device, and must be a variable or expression having value between 1 and 255, inclusive. If it were set to 10, a **PRINT** statement would force a carriage return at the tenth character position.

`Zone-width` is most like tab stops on a typewriter. If a comma is used in a **PRINT** statement, it forces a skip to the beginning of the next zone on the output device line.

Each file and device (including the screen) has default line and zone widths. The default line width for the screen, printer, and all files is 255, which actually indicates that the width is infinite; no **PRINT** or **WRITE** statements will ever cause an automatic carriage-return at any character position on the output line. The default zone width is always 14. If either of these arguments are omitted in a **WIDTH** statement, their values will not be altered.

The `output-device` can be any one of the predefined I/O devices ("SCRN:", "CLIP:", "LPT1:", or "COM1:"); if no output device is specified, "SCRN:" is assumed.

`Filenumber` associates a specific file with the **WIDTH** operation, and is the number that was associated with the file using an **OPEN** statement.

Using the **LPRINT** keyword alters the line and zone widths of the printer. It is equivalent to opening the device "LPT1:" and executing **WIDTH** "LPT1:" with the same parameters.

The **WIDTH**() function returns the length of the `string-expression` in pixels. It is useful for centering text horizontally. For example, the text "Hello World!" is centered horizontally within the current window:

```
text$="Hello World!"
window.width=WINDOW(2)
text.width=WIDTH(text$)
PRINT PTAB((window.width-text.width)/2);text$
```

Examples:

```
WIDTH 100      ' Set screen line width only
WIDTH 40,10    ' Set screen line & zone widths
WIDTH Dev$,10  ' Set device line width to 10
WIDTH "COM1:", UserLineW%, UserZoneW%
```

PRINT/PRINT#/LPRINT

Format:

```
PRINT [expression-list]
PRINT #filenumber, expression-list
LPRINT [expression-list]
```

Description:

PRINT outputs the data items in `expression-list` to the screen; the optional `expression-list` is a list of numeric and/or string expressions to be printed, separated by spaces, commas, or semicolons.

PRINT `#filenumber` outputs data in the same manner as **PRINT**, but sends it to the file that was associated with the given number by an **OPEN** statement. The expression-list is *not* optional with this syntax.

LPRINT also outputs data according to the following rules, but directs it to an attached printer. To specify printing quality of styles, refer to **OPEN** statement section about device "`LPT1:`".

Numeric items are printed with a leading space or minus sign, depending on the sign of the number, and a trailing space. If a single precision number has more than 7 digits (6 in BCD) or a double has more than 16 digits (14 in BCD), the number is formatted using scientific notation. For example, 0.00000001 in single precision would be printed as " `1E-08` ", quotes removed. The scientific notation format for double-precision numbers is the same as for single-precision numbers, except a "`D`" is used instead of an "`E`".

String items are printed as is, with no leading or trailing spaces added. If you wish to output a literal string, you must enclose it in quotes, e.g.

```
PRINT "Final Results: "; x.total!, y.total!
```

The positioning of the output items depends upon the separators used after each item: a semicolon or space instructs

PRINT to output the next expression result immediately following the previous expression result.

```
REM Chapter11/Example7
x%=50 : y!=.000012345# : Q$="YIELDS"
PRINT x%;Q$;y!      ' Same as PRINT X% Q$ Y!
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example prints

```
50 YIELDS 1.2345E-5
```

A comma causes **PRINT** to move to the next print zone on the line before printing the next expression; the default print zone width is 14 and may be set using the **WIDTH** statement. Commas may be used to align output items in a tabular format:

```
REM Chapter11/Example8
x%=5 : y%=8 : Delta!=y%/x%
PRINT ,," X TO Y"
PRINT " X"," Y"," CHANGE"
PRINT x%,y%,Delta!
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The spaces in the X and Y column headers are included to compensate for the leading spaces tacked on to the output numbers; this example would print

```
                X TO Y
X       Y       CHANGE
5       8       1.6
```

Ordinarily, the **PRINT** statement outputs an end-of-line sequence after printing all the expression list items. If a semicolon or comma follows the expression list, the cursor will remain on the same line, after the last character printed (if a semicolon is used), or tabbed over to the next print zone (if a comma is used).

See the **WIDTH** statement description earlier in this chapter for more information about print zones.

When **PRINT#** is used on a random-access file, the buffer is *not* padded with blanks or a newline character. Printing to a full buffer will give an overflow.

A question mark ("?") may be used in place of the **PRINT** keyword, with no effect upon the conventions described above — it is simply a time-saving convenience.

Example:

```
REM Chapter11/Example9
REM -- PRINT using the default zone width (14) --
PRINT 1,2,3
REM -- Make the zone width smaller --
WIDTH ,10
PRINT 1,2,3
REM -- Make the line width smaller --
WIDTH 15
PRINT 1,2,3
REM -- PRINT some strings --
WIDTH 255
PRINT "12345";"0123456789"
PRINT "1234567890123456789"
WIDTH 15
PRINT "12345";"0123456789"
PRINT "1234567890123456789"
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The output of the example is:

```
1             2             3
1       2             3
1       2
3
123450123456789
1234567890123456789
12345
0123456789
123456789012345
6789
```

SPC

Format:

```
SPC(x)
```

Description:

The **SPC** function is used in **PRINT** and **LPRINT** statements to output the number of blanks specified by the numeric expression x.

The value of x must be greater than or equal to 0 and less than 32768. If not, "Illegal function call" or "Overflow" errors will be generated at run-time.

The **SPC** function can only appear in **PRINT** and **LPRINT** statements, and is always output as if there were a semicolon following; that is, if the output expression list ends in the **SPC** function, the next output statement will begin writing where **SPC** terminated, not at the left margin of the next line.

When used in a **PRINT** statement to the screen, the spaces generated by **SPC** are non-destructive, unlike the spaces produced by **STRING$**, **SPACE$**, or literal equivalents.

Example:

```
REM Chapter11/Example10
PRINT "12345"
PRINT SPC(5);
PRINT "string"
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

Prints:

```
12345
     string
```

TAB

Format:

```
TAB(x)
```

Description:

In **PRINT** and **LPRINT** statements, **TAB** causes a "tab" out to position x on the current line. **TAB** may only be used in these two statements; if the print position on the current line is greater than x, **TAB** will cause a jump to position x on the next output line.

The argument x must be in the range 1 to 255, or an "Illegal function call" error will be generated.

Example:

```
PRINT "Table of squares from 1 to 5"
PRINT " X"; TAB(20); "X SQUARED"
FOR S%=1 TO 5
  PRINT S%; TAB(20); S%^2
NEXT
```

This example prints:

```
Table of squares from 1 to 5:
 X                   X SQUARED
 1                   1
 2                   4
 3                   9
 4                   16
 5                   25
```

PTAB

Format:

PTAB (x)

Description:

PTAB positions the text pen to pixel x on the current line.

The **PTAB** function does not return a value and can only appear in **PRINT** and **LPRINT** statements. It is always implicitly followed by a semicolon; if the output statement ends in the **PTAB** function, the next output statement will begin writing where **PTAB** terminated, not at the left margin of the next line.

The x argument indicates the horizontal pixel position to which the text pen position will be moved. If the current text pen position is past that point, the pen is reset to the given pixel and the vertical positioning remains the same. If the pixel is beyond the current line width, the pen is placed at the beginning of the next line. The value of x can be between 0 and 32767.

PTAB does not affect the current position of the graphics pen.

Example:

```
REM Chapter11/Example11
SLIDE " THE", 10
SLIDE " AC/BASIC", 11
SLIDE " COMPILER", 12
SLIDE " IS", 13
SLIDE " HERE!", 14
WHILE MOUSE(0)=0 : WEND

SUB SLIDE(x$,ROW%) STATIC
LOCATE ROW%
FOR PIXEL%=0 TO WINDOW(2)/2-LEN(x$)*4 STEP 2
   PRINT PTAB(PIXEL%); x$;
NEXT
END SUB
```

This example uses **PTAB** to "slide" text from the left margin to a centered position on the screen.

WRITE/WRITE#

Format:

```
WRITE [expression-list]
WRITE #filenumber[,expression-list]
```

Description:

Similar to **PRINT** and **PRINT#**, **WRITE** and **WRITE#** send data to the screen or an output file or device. The difference is that with the **WRITE** statements, every output item is automatically separated by commas. All strings are quoted and numeric values have no leading or trailing spaces.

`Expression-list` syntax is almost identical to that of the **PRINT** statement (any list of string or numeric expressions), with the exception that items in the list *must* be separated by commas.

Filenumber is the number associated with the file or device using the **OPEN** statement.

As with **PRINT** statements, if no expression list is present, a carriage-return is output. The double quote characters are counted in the line width set by the **WIDTH** statement.

The main use of the **WRITE#** statement is to output values which will later be read with the **INPUT#** statement, which, in general, requires that the items to be read be separated by commas.

When **WRITE#** is used on a random-access file, the buffer is padded with blanks and one newline character. A subsequent **WRITE#**, without a **PUT**, will give an overflow. A **GET** before a **WRITE#** also gives an overflow error.

Warning: if a string containing double-quote characters is output using a **WRITE#** statement, it will not be in the proper format for an **INPUT#** operation:

```
A$="SOMECHARS"+CHR$(34)+"OTHERCHARS" ' Insert a quote
B%=5
WRITE #1, A$,B%
```

This will cause the line

```
"SOMECHARS"OTHERCHARS",5
```

to be written to the file opened on I/O device 1, which is not readable by the **INPUT#** statement. (See the **INPUT#** description in this chapter.)

Example:

```
REM Chapter11/Example12
I%=55
J$="65816"
p!=3.14159
WRITE I%,J$,p!
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The output from this example is

```
55,"65816",3.14159
```

PRINT USING

Format:

```
PRINT USING string-expr; expression-list
PRINT #filenumber, USING string-expr; expression-list
LPRINT USING string-expr; expression-list
```

Description:

Prints numbers and/or strings that are formatted according to the given string expression.

PRINT USING directs formatted output to the screen.

PRINT `#filenumber`, **USING** directs output to the file that was associated with the given file number by an **OPEN** statement.

LPRINT USING sends output to an attached printer.

`String-expr` is used to format the data to be printed. The method of formatting uses predefined format characters and character sequences. When the **USING** string expression "requires" a value to output, one is supplied by the expression-list.

`Expression-list` is made up of a number of string and/or numeric expressions that are separated by commas or semicolons. If the **USING** string expression runs out of formatting characters before the expression-list is exhausted, formatting continues from the beginning of the string expression.

Literal characters, that is, not one of the special formatting characters, may be embedded in the **USING** string. When **PRINT USING** encounters such characters, it will print them at the current cursor position.

The formatting characters are divided into two types, string and numeric. There are four methods of formatting string output with **PRINT USING**:

1. !

The exclamation mark takes the first character from the expression-list's current argument and prints it.

Example: **PRINT USING** "!!"; "Albert","Einstein"
Output: AE

2. \n-spaces\

Backslashes allow a portion of a string argument to be printed. Note that the two backslashes themselves count as two character positions. Thus the number of characters that will be printed with backslash formatting is 2+n. If the string argument does not fill the field specified, it is padded on the right with blanks.

Example: **PRINT USING** "\ \"; "MARCH"
Output: MAR

3. &

The ampersand simply prints an entire string argument without truncation or padding.

Example: **PRINT USING** "& &"; "Day","One"
Output: Day One

4. _

To print a literal character that is one of the special formatting characters, precede it with an underscore (_).

Example: **PRINT USING** "Variable: &_!"; "Single"
Output: Variable: Single!

If the current expression in the expression list is not of type string when a string-formatting character is encountered (!, double-backslash, or &), a "Type mismatch" error will occur at run-time.

Examples of "Type mismatch" errors:

```
PRINT USING "\ \ ##";20
PRINT USING "INITIALS: !!!"; A$,B$,C%
```

There are nine ways to format numeric data with **PRINT USING**:

1. #

Each number sign indicates a digit position to be printed. Sequential number signs indicate how many digits of the current numeric argument to print. If a number does not fill each digit position, it is padded with spaces on the left (except in certain formats).

Example: `PRINT USING "Number ##"; 25`
Output: Number 25

2. .

The period forces a decimal point to be inserted in a printed number at the specified position. If the period is not adjacent to any number signs, it is output literally.

Example: `PRINT USING "###.#"; 5.3`
Output: 5.3

3. +

The plus sign, placed on the right or left of a numeric field, causes the sign of the number to be output in the corresponding position. If the plus sign is encountered by itself, it is output literally.

Example: **PRINT USING** "+### #.#+"; 15.3,-1.1
Output: +15 1.1-

4. -

The minus sign is a formatting character only when following a numeric field. If the number to be printed is negative, a minus sign is output following it; otherwise, a blank is printed.

Example: **PRINT USING** "###.##-"; 5,-100.9
Output: 5.00 100.90-

5. ,

The comma, when present to the left of the decimal point, inserts a comma every third digit starting from the decimal point. If no decimal point is present, a comma at the end of the number sign sequence has the same effect. Otherwise, the comma is treated like a standard literal character.

Example: **PRINT USING** "######, ######,.###"; 1250, 50123.09
Output: 1,250 50,123.090

6. **

The double-asterisk format sequence forces any blanks in a number to be printed as asterisks. The two asterisks also count as two number sign formatting characters.

Example: **PRINT USING** "**##"; 500
Output: *500

7. $$

Double-dollar formatting produces a "floating" dollar sign that will always be in the position immediately to the left of the formatted number. This also adds two digit positions, but one is reserved for the dollar sign.

Example: **PRINT USING** "$$ $$## $$##.##"; 5;5;5
Output: $5 $5 $5.00

8. **$

This formatting sequence is a combination of the two previous formats. The double-asterisk with a dollar sign prints the number with a "floating" dollar sign and asterisk padding rather than blanks. The three format characters add three more digit positions, but as above, one is used by the dollar sign.

Example: **PRINT USING** `"**$##.## "; -4.50; 50.23`
Output: `**-$4.50 **$50.23`

9. ^^^^

Scientific notation formatting is performed by entering 4 carets or up-arrows immediately after the digit specifiers. This will cause the number to be converted into the format mE±ee, where m is the mantissa, ee is the two-digit exponent, and "E" indicates the number formatted was single-precision. If the number is double-precision, the "E" is replaced by a "D". The mantissa will hold as many digits as there were number signs in the USING string. Any implied or explicit decimal places are lined up and the exponent is adjusted accordingly. A digit position is always taken for the sign unless a position has already been reserved (see "+" and "-" formatting).

Example: **PRINT USING** `".#^^^^ ##.###^^^^"; 500, 32767`
Output: `.5E+03 3.2767E+04`

It is possible for a formatted number to overflow the specified number of digits. If this occurs, a percent sign (%) is output immediately before the number.

If an attempt is made to reserve more than 24 digit places, an error will result.

The keyword **USING** must appear immediately after the words **PRINT** or **LPRINT** or after the optional `#filenumber` specification.

Each of the following is a valid **PRINT USING** statement followed by its output:

```
PRINT USING " DATE: ## \ \ ##"; 6,"JANUARY", 87
DATE:  6 JAN 87

PRINT USING "Your final bill is **$###.##"; 430.95
Your final bill is **$430.95

PRINT USING " ##.## "; 1.1,22.22,50
  1.10 22.22 50.00
```

AUXILIARY I/O STATEMENTS AND FUNCTIONS

LOCATE

Format:

LOCATE [row][,column]

Description:

Moves the pen or cursor to the specified row and column position in the current output window.

The row and column parameters must be greater than zero and less than 32768. If either is zero or less, an "Illegal function call" error will result. If greater than 32767, an "Overflow" error will be generated.

The height of a row is determined by the size of the character zero ("0") in the current text font. Row 1 is at the top of the window. A column is defined as the current character width. Column 1 is at the far left of the window.

If either parameter is omitted from the **LOCATE** statement, the current column or row value is used.

Moving a window has no effect on the position of the pen position within the window.

See also **POS** and **CSRLIN** in this chapter.

CSRLIN

Format:

```
CSRLIN
```

Description:

Returns the vertical line position of the cursor in the current output window.

This function returns the current line that the cursor is on, where row 1 is at the top of the window. A height of a row is based on the current font and text styles.

The current cursor column position can be obtained with the **POS** function, also described in this chapter.

Example:

```
.
.
curline% = CSRLIN ' Mark current line
PRINT "Counter at:";
FOR i%=1 TO 200
   LOCATE curline%,12
   PRINT i%;
NEXT
```

This example will display the value of the loop counter variable i% as the loop is executed, on the current cursor line on the screen. The **LOCATE** positions the cursor to the same line in column 12 each time.

POS

Format:

```
POS(x)
```

Description:

Returns the column position where the next character will be output in the currently active window.

x is ignored by **POS** and can be any numeric expression.

Note that the width of a column is defined as the width of a "0" in the current font. If the font is proportionally spaced (each individual character has its own width), **POS** may return a value that does not represent the current number of characters on the current line. For instance, if a "." is half the width of a "0", printing 20 periods on a line will cause **POS** to return the value 10.

See also **CSRLIN**, **LOCATE** and **LPOS**.

Example:

```
REM Chapter11/Example13
LOCATE 10,10
PRINT "Please enter your name";
y% = CSRLIN
x% = POS(0)
INPUT ;nam$
CLS
LOCATE y%,x%
PRINT nam$
BEEP
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example prompts for a name, then clears the screen and positions the cursor where "Please" was originally printed.

LPOS

Format:

LPOS (x)

Description:

Returns the logical column of the print head on the printer. The physical column position may be different because **LPOS** counts **TAB** characters as moving the print head only one position.

As with the **POS** function, the argument x is ignored by **LPOS** and can be any numeric expression.

Example:

```
IF LPOS (x) > 70 THEN LPRINT
```

This will begin a new line on the printer if the current line is more than 70 characters long.

RANDOM-ACCESS I/O

As outlined in the "Random-Access Files" section at the beginning of this chapter, random-access files and their associated statements are a very efficient means of data storage. The chief advantage over sequential files is that the data in random-access files may be read and written in any order.

A alternate group of Unix-style random-access file I/O statements which work directly with arrays is described in Chapter 16.

FIELD

Format:

```
FIELD [#]filenumber,fieldwidth AS string-variable[, ... ]
```

Description:

Associates string variables with portions of a random file buffer.

`Filenumber` is the number associated with the random file when it was opened. Random-access files must be opened without the **INPUT**, **OUTPUT**, or **APPEND** keywords, or with file mode "R", using the abbreviated **OPEN** syntax, e.g.

```
OPEN "datafile" AS 1 LEN=100 'Defaults to random-access
OPEN "R",1,"datafile",100    'Specifies random-access
```

If a **FIELD** statement is used for a file opened in input, output, or append mode, a "Bad file mode" error will result at run-time.

`Fieldwidth` defines the number of bytes or characters in the buffer to be assigned to the corresponding `string-variable`. **FIELD** always begins with the first character in the buffer and traverses the buffer sequentially: for example, the statement

```
FIELD #8, 10 AS A$, 10 AS B$, 10 AS C$
```

assigns the first 10 characters in the buffer to `A$`, the second 10 to `B$`, and the third 10 to `C$`. The file whose buffer is being fielded must have been opened with a buffer size equal to or greater than the sum of all the field widths in a **FIELD** statement; otherwise, a "FIELD overflow" error will result. Thus the above AC/BASIC line requires a buffer size of 30 characters (bytes) or greater. A file's buffer size is set by the **LEN** option on the **OPEN** statement.

Numeric variables must first be converted to strings before being stored in a random-access file buffer, and must be retrieved as strings and then converted to numerics when being read from a buffer. While it is possible to use the **STR$** and **VAL** functions for these purposes, it is much more space-

efficient to use the **MKI$**, **MKS$**, and **MKD$** functions and their inverses, **CVI**, **CVS**, and **CVD** for conversion of numerics to strings and vice-versa.

A random file buffer can be partitioned over and over without affecting the variable assignments set up by other, currently active **FIELD** statements. If a random file buffer is large or must be highly partitioned, the **FIELD** statements may be broken up:

```
OPEN "ShowInfo" AS FILE 5 LEN=125
FIELD #5, 10 AS ShowDate$, 35 AS PerformerName$, 55 AS Loc$
FIELD #5, 100 AS dummy$, 15 AS ShowType$, 10 AS TicketPrice$
```

10	35	55	15	10

The use of `dummy$` in the third line cause **FIELD** to skip the hundred bytes assigned in the previous **FIELD** statement, so that the variable `ShowType$` references bytes 101 through 115 in the buffer, etc.

Fielded strings must be assigned new values in your program only by **LSET**, **RSET**, or the **MID$** statement; if a fielded string is assigned as a normal string variable via **LET**, **SWAP**, or **INPUT**, it no longer references an area in the random file buffer.

In a compiled program, when random files are closed (by **CLOSE**, **CLEAR**, or **RESET**), the fielded variables are reset to null.

Example:

```
OPEN "infile" AS #2 LEN=72
FIELD #2,20 AS name$,20 AS street$,30 AS city$,2 AS state$
```

This example allocates the record in the file buffer as having a 20-character name field, a 30-character street field, a 30-character city field and a 2-character state field.

See also **GET**, **PUT**, **LSET**, **RSET**, and **OPEN** in this chapter.

GET

Format (files):

```
GET [#]filenumber[,record-number]
```

Description:

Loads a random-access file buffer defined by **FIELD** with the next record in the file, or the record specified by `recordnumber`.

`Filenumber` parameter is the number that was assigned to a file with the **OPEN** statement. The `record-number` indicates the random file record within the file that is to be retrieved, and can range from 1 to 16,777,215. If no record number is specified, the record following the last one read (**GET**) or written (**PUT**) will be retrieved.

The data in the random file buffer can be manipulated through the use of fielded strings; see the discussion of the **FIELD** statement in this chapter.

If a **GET** attempts to retrieve a record past the end of the file, no error is reported and the last record in the file is retrieved instead. The end-of-file status can be determined with the **EOF** function, which will return true if the current record contains the logical end of file.

This example reads a file created by the example program for **PUT** in the next section. It uses **GET** to search for a specified check number and, if found, prints out to whom the check was paid, the date it was written, and the amount of the check.

```
  REM Chapter11/Example14
  OPEN "CHECKS.DAT" AS 1 LEN=35
  FIELD #1, 2 AS CHKNUM$,20 AS ToWhom$,9 AS WRDATE$,4 AS CHKAMT$

READLOOP:
  INPUT "CHECK NUMBER"; CN%
  IF CN%=0 THEN ENDCHECKS
  REC%=1
SEARCHLOOP:
  GET #1,REC%
  IF CVI(CHKNUM$)=CN% THEN
     PRINT "PAYEE: "; ToWhom$
     PRINT "DATE WRITTEN: "; WRDATE$
     PRINT USING "AMOUNT: $#####,.##"; CVS(CHKAMT$)
     PRINT
  ELSEIF NOT EOF(1) THEN
     REC%=REC%+1
     GOTO SEARCHLOOP
  ELSE
     PRINT "CHECK NUMBER NOT FOUND."
     PRINT
  END IF
  GOTO READLOOP
ENDCHECKS:
  CLOSE
  PRINT "CHECK DATA FILE CLOSED."
  WHILE MOUSE(0)=0 : WEND
```

PUT

Format (files):

```
PUT [#]filenumber[,record-number]
```

Description:

Writes the current contents of the random file buffer to the specified record.

`Filenumber` associates a random file with the **PUT** statement. The specific record number in the file being written to is

indicated by the record-number, and can range from 1 to 16,777,215. If no record number is specified, the buffer is written to the last record read (**GET**) or to the record after the last written (**PUT**), whichever operation was performed most recently. If a record number greater than the number of records in the file is used, **PUT** will automatically extend the file and then put the record.

The contents of a random file buffer are accessed through the use of fielded variables. See the discussion of the **FIELD** statement in this chapter.

This example opens a random-access file, inputs checkbook accounting data from the user, and stores it in the the file using **PUT**. After running this file, you may use the complementary example for **GET** to read the data.

```
  REM Chapter11/Example15
  OPEN "CHECKS.DAT" AS 1 LEN=35
  FIELD #1, 2 AS CHKNUM$, 20 AS ToWhom$, 9 AS WRDATE$, 4 AS CHKAMT$

READLOOP:
  REC%=REC%+1
  INPUT "CHECK NUMBER"; CN%
  IF CN%=0 THEN ENDCHECKS
  INPUT "PAYEE "; PAYEE$
  INPUT "DATE WRITTEN"; DT$
  INPUT "AMOUNT "; AMT!
  LSET CHKNUM$=MKI$(CN%)
  LSET ToWhom$=PAYEE$
  LSET WRDATE$=DT$
  LSET CHKAMT$=MKS$(AMT!)
  PUT #1,REC%
  PRINT "RECORD" LOC(1) "WRITTEN"
  GOTO READLOOP

ENDCHECKS:
  CLOSE
  PRINT "CHECK DATA FILE CLOSED."
  WHILE MOUSE(0)=0 : WEND
```

LSET/RSET

Format:

```
LSET stringvar=string-expression
RSET stringvar=string-expression
```

Description:

Used to load data into random file buffer strings that have been declared with the **FIELD** statement; also used for the right- and left-justification of text in strings.

LSET left justifies the `string-expression` in the `stringvar`, and **RSET** right justifies the `string-expression` in `stringvar`. If `stringvar` is fielded, the data is saved in a random file buffer. If `string-expression` is shorter than `stringvar`, then **LSET** pads with spaces on the right; **RSET** pads on the left. If `string-expression` is longer than `stringvar`, characters are always truncated on the right.

The following example illustrates the use of **LSET** and **RSET** in conjunction with **FIELD** to store data in the first record of a random data file.

```
REM Chapter11/Example16
OPEN "addresses" AS #2 LEN = 77
FIELD #2, 20 AS namestr$, 20 AS street$, 30 AS city$
FIELD #2, 70 AS dummy$, 2 AS state$, 5 AS zip$
LSET namestr$ = "Jake Smith"
LSET street$ = "123 Main Street"
LSET city$ = "Tree City"
LSET state$ = "MI"
RSET zip$ = "48000"
PUT #2
CLOSE #2
PRINT "Data sent to file: addresses"
WHILE MOUSE(0)=0 : WEND
```

In order to justify a non-fielded string variable, its length must be defined before the use of **LSET** or **RSET**:

```
A$ = SPACE$(35)
RSET A$="Total Bills"
```

The resulting string is of length 35 and contains the given literal, right-justified.

See also **FIELD**, and **MKI$**, **MKS$**, and **MKD$** in the next section.

MKI$/MKS$/MKD$

Format:

```
MKI$(integer-expression)
MKS$(single-precision-expression)
MKD$(double-precision-expression)
```

Description:

Packs a numeric expression into a string.

These functions are used to store numbers efficiently, especially in random file buffers, as numbers converted to strings using the **STR$** function generally take up more room. For example, as a string, the number 32767 ("32767") occupies five bytes, but using the **MKI$** ("Make an Integer into a string") function its representation takes only two bytes.

MKI$**, **MKS$, and **MKD$** convert `integer`, `single-precision`, and `double-precision` variables or expressions into 2-, 4-, and 8-byte strings, respectively.

The functions **CVI**, **CVS**, and **CVD** are inverses of **MKI$**, **MKS$**, and **MKD$**, respectively; see the next section for a complete description.

Example:

```
REM Chapter11/Example17
OPEN "outfile" AS #1 LEN=14
LET int1% = 10
LET sng! = 10.54321
LET dbl# = 10.12345678#
FIELD #1, 2 AS inte$, 4 AS sngl$, 8 AS dble$
LSET inte$ = MKI$(int1%)
LSET sngl$ = MKS$(sng!)
LSET dble$ = MKD$(dbl#)
PUT #1
CLOSE #1
PRINT "Data PUT to file: outfile"
WHILE MOUSE(0)=0 : WEND
```

This example puts the values of `int1%`, `sng!`, and `dbl#` into an 14-byte random file buffer; storing these values in unconverted string form would require a minimum buffer of 24 bytes.

CVI/CVS/CVD

Format:

```
CVI(2-byte-string-expr)
CVS(4-byte-string-expr)
CVD(8-byte-string-expr)
```

Description:

CVI converts a string expression into an integer, CVS converts a string expression into a single-precision number, and CVD converts a string expression into a double-precision number.

If a string or string expression is supplied that is longer than necessary for a CV-function, the extra characters are ignored; if shorter than required, an "`Illegal function call`" error (number 5) is generated at run-time.

In contrast to the **VAL** function, which returns any literal numeric value beginning a string, the CV-functions cause AC/BASIC to view the bit patterns in a string as an internal representation of a number.

For instance, **VAL**("11") produces the value 11, whereas **CVI**("11") produces the value 12593. This is because the ASCII value of a "1" is 49, which has a bitwise representation of 00110001, making the internal representation of "11" the following binary number,

0011000100110001

which is equivalent to the decimal value 12593.

The CV-functions may be easily used with their inverses, the **MKI$**, **MKS$**, and **MKD$** functions, to convert any AC/BASIC number to or from the format required for random-access file storage. In addition, these functions may be used for efficient storage and retrieval of numbers in files via **PRINT** and **INPUT$**. For instance, the single-precision number represented by the string "2.123456E-30", requires 12 bytes of storage; if printed to or stored in a file using the **MKS$** function, it only requires 4 bytes.

See also the **FIELD** statement and the **MKI$/MKS$/MKD$** functions in this chapter.

MKSBCD$/MKDBCD$

Format:

MKSBCD$(single-precision-expression)
MKDBCD$(double-precision-expression)

Description:

Provides a means of exchanging floating point data between programs compiled for binary math with programs compiled for decimal math.

MKSBCD$** and **MKDBCD$ serve the same purpose as **MKS$** and **MKD$**, described above, but they convert the binary values to decimal before returning the string result. The strings can then be saved in a random file buffer.

Integer and string variables are directly compatible with both versions and require no special translations.

Note that these functions can be used in programs compiled for decimal math. In this case, they are exactly the same as **MKS$** and **MKD$**.

Further discussion of numeric formats can be found in the reference to **CVSBCD** and **CVDBCD**, and also in Appendix A.

The following program writes decimal floating point data to a random file.

```
REM Chapter11/Example18
' This program should be compiled for binary math.
OPEN "random.file" AS #1 LEN=16
FIELD #1,4 AS single1$, 4 AS single2$, 8 AS Double$
LSET single1$=MKSBCD$(123.45)
LSET single2$=MKSBCD$(56.789)
LSET Double$=MKDBCD$(123456.789#)
PUT #1,1
CLOSE 1
PRINT "Data PUT to file: outfile"
WHILE MOUSE(0)=0 : WEND
```

This program reads the decimal data that was saved by the previous program.

```
REM Chapter11/Example19
' This program should be compiled for decimal math.
OPEN "random.file" AS #1 LEN=16
FIELD #1,4 AS single1$, 4 AS single2$, 8 AS Double$
GET #1,1
PRINT CVS(single1$),CVS(single2$),CVD(Double$)
CLOSE 1
PRINT "Click mouse to exit."
WHILE MOUSE(0)=0 : WEND
```

The output of this example will be:

```
123.45    56.789    123456.789
```

CVSBCD/CVDBCD

Format:

```
CVSBCD(4-byte-string-expression)
CVDBCD(8-byte-string-expression)
```

Description:

AC/BASIC supports two math libraries, binary (the default) and decimal; single-precision or double-precision numbers stored in (random-access) files via **MKS$** and **MKD$** under one library ordinarily will not be compatible with a program compiled with the other. These functions provide a means of converting strings representing numbers stored in decimal format to their correct binary values.

CVSBCD converts a string expression representing a decimal single-precision value to a binary format single-precision number, and **CVDBCD** converts a string expression representing a decimal double-precision number to a binary format double-precision number.

If the string supplied is too long, the functions will only use as many bytes of the string as is required; strings that are too short will produce an "`Illegal function call`" at run-time.

Examples:

```
REM Chapter11/Example20
REM Program "A", to be compiled with "D" option on
single!=4.5E+30        ' A single-precision #
Double#=1.75D+30       ' A double-precision #
REM Now output numbers to a file in decimal format:
OPEN "BCD.TEMP" FOR OUTPUT AS 1
   PRINT #1, MKS$(single!);
   PRINT #1, MKD$(Double#);
CLOSE 1
PRINT "Datafile BCD.TEMP ready for Program B."
WHILE MOUSE(0)=0 : WEND
```

```
REM Chapter11/Example21
REM Program "B", to be compiled without "D" option
OPEN "BCD.TEMP" FOR INPUT AS 1
single$=INPUT$(4,1)        ' Get stored single-prec #
Double$=INPUT$(8,1)        ' Get stored double-prec #
PRINT "Using CVS and CVD:"
PRINT "Single:" CVS(single$)
PRINT "Double:" CVD(Double$)
PRINT "Using CVSBCD and CVDBCD:"
PRINT "Single:" CVSBCD(single$)
PRINT "Double:" CVDBCD(Double$)
WHILE MOUSE(0)=0 : WEND ' Wait for mouse click
```

This example illustrates the use of **CVSBCD** and **CVDBCD** to correctly convert strings representing decimal numbers to their binary equivalents. Program "A", which should be compiled using the "D" option, outputs two strings representing single-precision and double-precision numbers in decimal format to the file "BCD.TEMP". Program "B", compiled without the "D" option (and therefore defaulting to binary math), shows the erroneous values produced by using the **CVS** and **CVD** functions on decimal strings. The values "4.50E30" and "1.75D30" were stored in the data file, and the following results from running program "B":

```
Using CVS and CVD:
Single: 1.419534E+19
Double: 1.192358102290304D+150
Using CVSBCD and CVDBCD:
Single: 4.5E+30
Double: 1.75D+30
```

I/O UTILITY FUNCTIONS

These functions provide information about files that are open, and may be used effectively for flexible file processing.

EOF

Format:

```
EOF(file-number)
```

Description:

Returns a logical result (-1 for true, 0 for false) indicating, for sequential files, whether all data has been read from the file or, for random-access files, whether the last record read by a GET is the last record in the file.

Example:

```
REM Chapter11/Example22
INPUT "Count lines in which file"; FIL$
OPEN FIL$ FOR INPUT AS #1
COUNT% = 0
WHILE NOT EOF(1)
   LINE INPUT #1,a$
   COUNT% = COUNT% + 1
WEND
CLOSE 1
PRINT COUNT% "lines in file " FIL$
WHILE MOUSE(0)=0 : WEND
END
```

This example counts the number of lines in the sequential file named in the string `FIL$`.

LOF

Format:

LOF(`file-number`)

Description:

Returns the current size, in bytes, of the file associated with the given `file-number`.

Sequential files: if performing input, **LOF** returns the size of the file as it exists on the disk. Otherwise, the value returned is the current file size, taking into account the amount of data in the output file buffer that has not been flushed to the disk.

Random-access files: **LOF** always returns a number of bytes that is a multiple of the record size of the file. If an accurate file length is desired, you may first open the file for input, execute a **LOF**, close the file, then re-open it for random-access.

If the device associated with `file-number` is `COM1:` then a 0 is returned if no incoming data is available and a 1 is returned if data is available.

Note that the **LOF** function always returns 0 for file numbers associated with the devices `SCRN:`, `KYBD:`, and `LPT1:`.

Example:

```
REM Chapter11/Example23
OPEN "old.file" FOR OUTPUT AS #1
PRINT #1,"1234567890";
CLOSE
OPEN "old.file" FOR INPUT AS #1
PRINT LOF(1)
CLOSE
OPEN "new.file" FOR OUTPUT AS #1
PRINT #1,"1234567890";
PRINT LOF(1);
PRINT #1,"1234567890";
PRINT LOF(1)
CLOSE
OPEN "random.file" AS #1 LEN=6
FIELD #1,2 AS integer$,4 AS single$
FOR currec=1 TO 6
   LSET integer$=MKI$(currec)
   LSET single$=MKS$(currec)
   PUT #1,currec
   PRINT LOF(1);
NEXT currec
CLOSE
PRINT"done"
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
END
```

The output of the example is:

```
10
10 20
6 12 18 24 30 36
```

LOC

Format:

```
LOC(file-number)
```

Description:

Returns the number of records or buffers read from the file opened with the specified `file-number`.

The size of a record in a file is defined by the **LEN** specification on the **OPEN** statement; the default is 128 bytes. For sequential files, **LOC** indicates how many buffers of this size have been read or written; for random-access files, **LOC** returns the current record number.

For a file opened as `"KYBD:"`, **LOC** returns 1 if a character is waiting, 0 otherwise. For a file opened as `"COM1:"` (serial I/O), **LOC** returns the number of characters that are waiting to be input.

Example:

```
IF LOC(1) > 100 THEN STOP
```

This example stops execution when the 100th record in the file is passed.

READING PROGRAM DATA

If information needed by a program can be easily listed, an alternative to disk storage is to place the data elements in **DATA** statements within the program. This avoids slow disk accesses and eliminates the need for a tag-along data file.

The **DATA** statement is also an effective weapon against repetitious array initialization; the initial values can often be read from the **DATA** statements in a simple loop.

DATA

Format:

```
DATA constant[,constant...]
```

Description:

The **DATA** statement is used to set up lists of numeric and/or string constants that can be assigned to variables using **READ** statements.

All `constant` entries must be separated by commas and all strings must be quoted when they contain commas, colons, or significant leading or trailing blanks. Expressions, either numeric or string, are not allowed.

Note: Only the **REM** statement may be used to include comments on the same line as a **DATA** statement; comments beginning with the special comment character (') will be considered part of the current **DATA** item.

If two consecutive commas appear in a data statement, it is considered to be a null item. If the variable being assigned with a **READ** statement is numeric, a zero value is used. If the variable is a string, a null value ("") is assigned.

There may be as many constant fields after a data statement as will fit on the current line, and **DATA** statements may appear anywhere in a program without affecting execution. **DATA** statements are accessed with the **READ** statement lexically (in order of physical location in the file).

A **RESTORE** statement may be used to begin re-reading the fields in **DATA** statements starting with the first **DATA** statement in the program or with or the data statement at the specified line or label. See discussion of the **READ** and **RESTORE** statements in this chapter for information on how to read and re-read **DATA** list items.

Example:

```
REM Chapter11/Example24
DATA 45,36
READ int1%
READ int2%
READ sing!
READ txt$
WRITE int1%,int2%,sing!,txt$
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
DATA 3.5,"text"
```

The output of this example would be:

```
45,36,3.5,"text"
```

READ

Format:

```
READ variable-list
```

Description:

Retrieves values from **DATA** statements in the program and assigns the values to the variables in `variable-list`.

The variables in the variable list are separated by commas, and may be string or numeric variables (or array elements). **DATA** element values are read sequentially and assigned to the variables in the list, one by one. The number and placement of **DATA** statements in a program is irrelevant; when all the elements of a **DATA** statement have been read, **READ** will skip to the first item of the next **DATA** statement in the program. If no data remains for any variable in a **READ** list, an "Out of data" error results at run-time.

You can cause **READ** to begin reading or rereading **DATA** values at a specific line or label in the program with the

RESTORE statement. See discussion of this statement in the next section.

Example:

```
REM Chapter11/Example25
DATA 1,ONE,2
DIM Num%(5), NumName$(5)
FOR I% = 1 TO 5
   READ Num%(I%), NumName$(I%)
   PRINT Num%(I%) "= " NumName$(I%)
NEXT
DATA TWO,3,THREE,4,FOUR,5,FIVE
PRINT "Click mouse to exit."
WHILE MOUSE(0)=0 : WEND
```

Reads the values 1 through 5 and their names into the arrays Num% and NumName$, respectively.

RESTORE

Format:

```
RESTORE [line|label]
```

Description:

RESTORE without a `line` or `label` specification causes subsequent **READ** statements to retrieve information from the first **DATA** statement in the program.

If a `line` or `label` is specified, subsequent **READ** statements will begin retrieving values from the first **DATA** statement on or after the specified line or label which may be in a different subprogram.

Example:

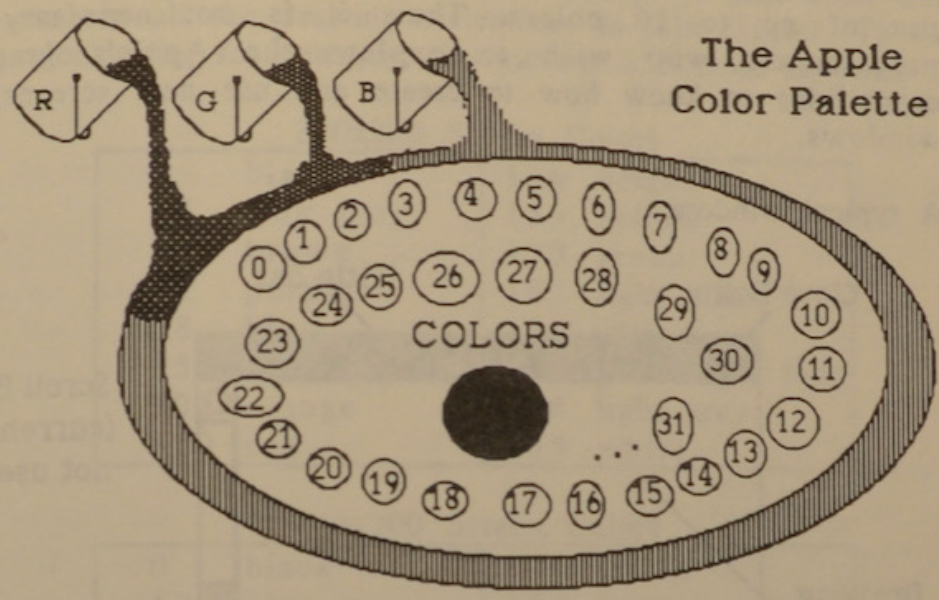
```
    REM Chapter11/Example26
  Start:
    DATA LONDON BRIDGE IS
    D1:
    DATA FALLING DOWN, FALLING DOWN
    READ a$, b$, c$      ' Read from first DATA statement
    PRINT a$ : PRINT b$;", ";c$;", ";
    RESTORE D1           ' Back to data at D1:
    READ D$              ' Read one item
    PRINT D$
    RESTORE LastData     ' Next read starts at LastData:
    READ E$
    PRINT E$
    WHILE MOUSE(0) = 0 : WEND '-- Wait for click
    END
  LastData:
    DATA (MY FAIR LADY)
```

The output of this example is:

```
LONDON BRIDGE IS
FALLING DOWN, FALLING DOWN, FALLING DOWN
(MY FAIR LADY)
```

CHAPTER 12

GRAPHICS STATEMENTS AND FUNCTIONS



SCREENS AND WINDOWS

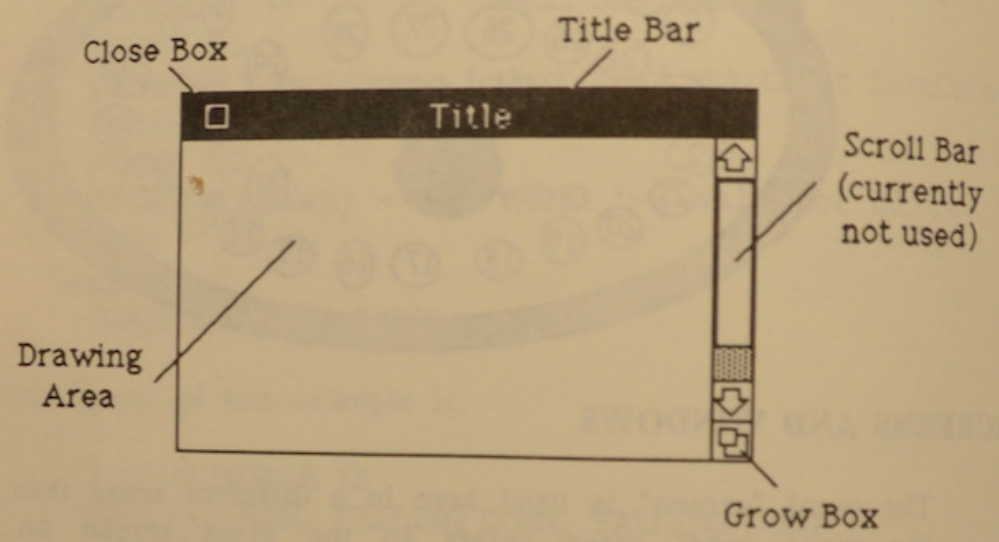
The word "screen" is used here in a different sense than usual; the word most often refers to the glass screen on video monitors. On the Apple IIGS, the screen is the medium through which all display is done, and two types of screens may be defined. With each screen is associated certain basic parameters of display, such as the maximum number of colors and the vertical and horizontal resolution.

One screen has a resolution of 640 pixels horizontally and 200 pixels vertically. It is referred to as the "640x200 screen". The other screen type has the same number of pixels vertically, but half the number of pixels horizontally. It is called the "320x200" screen. Although the 640x200 screen has twice the resolution, its colors are not quite as clear as the 320x200 screen"

On a screen, you can create and manipulate your own windows from the three different types available. Using the mouse, you can stack, move, or resize windows that are defined with those features.

When a program is started, AC/BASIC automatically provides a full-size window on the screen (unless the "U" compile option was selected), which has 640x200 resolution and permits the use of up to 16 colors. Thus, it is not necessary for programmers who wish to explore the Apple's graphics capabilities to know how to create and use new screens and windows.

A typical window:



There are two types of window states: a window can be said to be the *currently active* window, and/or the *current output* window. The currently active window has a highlighted title bar, and all other windows' title bars are ghosted (dimmed). All keyboard and mouse input is done through the currently active window. Any window can be made the currently active window by use of **WINDOW** commands within the program (see the **WINDOW** statement description below).

The current output window is the one to which all output will be directed, via **PRINT**, **PSET**, etc., and may be selected using the **WINDOW OUTPUT** statement described below.

THE APPLE IIGS COLORS

Each of the two screens has its own defined set of colors as shown in the following tables. These are the color numbers used in all the drawing statements in this chapter. To use the color white, for example, the number 255 must be used.

320x200 Screen Colors

0	black	136	beige
17	dark gray	153	yellow
34	brown	170	green
51	purple	187	light blue
68	blue	204	lilac
85	dark green	221	periwinkle blue
102	orange	238	light gray
119	red	255	white

640x200 Screen Colors

0	black	136	green
17	blue	153	aqua
34	olive	170	chartreuse
51	gray	187	pastel green
68	red	204	light gray
85	purple	221	light blue
102	orange	238	light yellow
119	peach	255	white

All the numbers are multiples of 17. The reason behind this is explained in the section "Understanding the Colors" but does not have to be understood unless you want to create new color combinations. The color numbers between 0 and 255 which are not in the tables represent *striped* colors made of two separate colors.

This example randomly displays colored boxes in the current window. (hit the mouse button to stop):

```
REM Chapter12/Example1
w=WINDOW(2)
h=WINDOW(3)
WHILE MOUSE(0)=0
   col=INT(RND*16)*17        ' Select a random color
   LINE (RND*w,RND*h)-(RND*w,RND*h),col,BF
WEND
```

To see all the colors, including the striped ones, change the fifth line to be:

```
col=INT(RND*255)          ' Select a random color
```

PENS

When AC/BASIC puts your program output on the Apple IIGS screen, it uses colored pens to draw the image. AC/BASIC maintains two separate pens which are used to draw text and graphics. They each specify the current position on the screen, size, and foreground and background color.

The pen used to draw text is called the *text pen* which is used and changed by statements such as **PRINT**, **INPUT**, **COLOR**, and **LOCATE**.

The *graphics pen* is used by statements such as **LINE** and **CIRCLE**. Its color can also be changed with **COLOR**.

Most of the statements in this chapter which use or change a pen, use or change the graphics pen

SCREEN AND WINDOW STATEMENTS

SCREEN

Format:

SCREEN screen-type (statement syntax)

Description:

SCREEN changes the resolution of the Apple IIGS screen.

The screen-type is either the number 1 or 2 to indicate 320x200 or 640x200 resolution, respectively.

What are the advantages and disadvantages between the two screen resolutions? The 640x200 screen has only four true colors but twice the number of pixels on the 320x200 screen which has 16 true colors. For a complete explanation, see the section "Understanding the Colors" in this chapter.

screen type	resolution	aspect ratio	number of true colors
1	320x200	1.2 : 1	16
2	640x200	2.4 : 1	4

When the **SCREEN** statement is executed, all window, buttons, and edit fields are closed. In addition, all the applicable Apple IIGS Toolboxes are closed and re-opened for the new screen resolution. This causes the tools to be reloaded from disk which will take some time — about 10 to 30 seconds. Screen types other than 1 or 2 will not cause an error but will change the screen resolution to 640x200.

WINDOW

Format:

```
WINDOW window-id[,[title$][,[(x1,y1)-(x2,y2)][,[type]]]]
WINDOW window-id
```

Description:

WINDOW creates and/or displays a window with the given attributes. `Window-id` must be a number from 1 to 6, inclusive, or an "Illegal function call" error is generated.

If the window specified does not exist, or was previously closed with a **WINDOW CLOSE** statement, it becomes the currently active window (title bar highlighted) and the current output window and it appears in front of others windows on the screen. The same is true for a **WINDOW** statement which changes *any* parameters of the window with the specified `window-id`; in this case, if the window is currently open, it is also cleared and its pen positions reset (see **CLS**). Since the abbreviated form of the **WINDOW** statement is simply the longer form with all specifications omitted, the discussions of default parameter values below apply.

If `title$` is specified, it is displayed in the title bar of the window, truncated on the right if longer than the title bar width. If omitted, `title$` will default to the previous title used for a window with the same `window-id`. The default window (displayed if the "U" option is off) has a default title which is the application file name.

The dimensions and placement of the window on the Apple IIGS screen are defined by the rectangle specification, which is of the form `(x1,y1)-(x2,y2)`, indicating the upper left-hand and lower right-hand interior corners of the window being created. This rectangle, plus the window's border dimensions, should fit within the limits of the screen type with which the window is defined; the **WINDOW**(2) and **WINDOW**(3) functions return the width and height for the current output window, and may aid with window positioning. If the rectangle specification is omitted, the window's previous size is used — this is handy for resizeable and/or moveable windows which

you want to reappear as last altered by a user. The initial default for each window is the whole screen.

The `type` parameter specifies the window type from the four listed below:

1. Document window with a title bar and a grow box.
2. Window with a double-line border.
3. Window with a single-line border.
4. Same as type 3. (Included for compatibility with Microsoft BASIC on the Macintosh)

The default window type is a document window (type 1). If a type of 1 is specified and no title is given, the window will have the title of the program. If the type specified is less than 1 or greater than 4, an "Illegal function call" error is generated.

WINDOW OUTPUT

Format:

WINDOW OUTPUT `window-id`

Description:

Selects the specified window as the current output window. The window with the given `window-id` must be defined and must not have been closed with **WINDOW CLOSE**, or an "Illegal function call" results.

All subsequent text and graphics output is routed to the new current output window. However, if a program requests user input which requires printing a prompt or echoing typed characters, the currently active window (through which all input is done) is made the output window automatically. The statements to which this applies are **INPUT** and **LINE INPUT**; input statements which do *not* echo to the currently active window are **INKEY$** and **INPUT$**.

The switching speed of **WINDOW OUTPUT** between windows is very fast, making it possible to update information in many different windows with little delay.

WINDOW CLOSE

Format:

```
WINDOW CLOSE window-id
```

Description:

WINDOW CLOSE removes the specified window from the Apple IIGS screen.

`Window-id` must be a number from 1 to 6, inclusive; if the window does not exist, no action is taken and no errors are reported.

All the attributes of a window are saved in memory after a **WINDOW CLOSE** statement. Thus, if the following statements,

```
WINDOW 2,"My Window",(30,30)-(150,100),2
WINDOW CLOSE 2
WINDOW 2
```

are executed, window 2 would disappear and then reappear with the same title, size, location, and window type. All text and graphics in the window are lost when the window is closed.

WINDOW()

Format:

WINDOW (n) (function syntax)

Description:

Returns information about the current output window based on the value of n.

The permissible values for n and the information retrieved are listed below.

0 Currently active window's ID (see **WINDOW** statement). If there is no currently active window, zero is returned.

1 Current output window's ID (see **WINDOW** statement).

2 Current output window's interior width in pixels.

3 Current output window's interior height in pixels.

4 X coordinate of the current output window's graphics pen position.

5 Y coordinate of the current output window's graphics pen position.

The useable widths and heights of windows will never be the full screen size (640x200, etc.), because the menu bar and window borders use space.

Example:

```
REM Chapter12/Example2
LOCATE 8,25
x1%=WINDOW(4)-1 : y1%=WINDOW(5)-9
PRINT "WINDOW-ID";WINDOW(1);"IS CURRENT OUTPUT";
x2%=WINDOW(4) : y2%=WINDOW(5) + 1
LINE (x1%,y1%)-(x2%,y2%),68,b
WHILE MOUSE(0)=0 : WEND     ' Wait for click
```

This example uses the values returned by **WINDOW**(4) and **WINDOW**(5) to outline a box of text; the offsets -1 and -9 are used for the first coordinate pair because the text position values returned by the **WINDOW** functions represent the *base line* of the text to be printed.

DRAWING STATEMENTS

PRESET/PSET

Format:

```
PSET [STEP] (x,y)[,color]
PRESET [STEP] (x,y)[,color]
```

Description:

Sets the color of the specified point in the current output window.

If the **STEP** keyword is used, the values of x and y are relative to the current graphics pen position.

The values of x and y must be in the integer range, -32768 to 32767, or an "Overflow" error will be generated.

Color refers to one of the Apple IIGS colors which are given in a table on page 3 of this chapter. If an attempt is made to use a color that is outside the range of 0 to 255, an "Illegal function call" error is generated.

The only difference between the two statements is that the default color for **PSET** (pixel set) is the current foreground color, and the default for **PRESET** (pixel reset) is the current background color.

Example:

```
REM Chapter12/Example3
FOR I% = 0 TO 100
   PSET (I%,I%)
NEXT
FOR I%=100 TO 0 STEP -1
   PRESET (I%,I%)
NEXT
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example draws a diagonal line on the screen from the upper left-hand corner to (100,100), then erases it in the reverse order that it was plotted.

LINE

Format:

```
LINE [[STEP](x1,y1)]-[STEP](x2,y2)[,[color][[,fcolor][,B[F]]]]
```

Description:

Draws a line, a box, or a filled box in the current output window.

Without the `B` or `BF` options, **LINE** draws a line between the points specified by `(x1,y1)` and `(x2,y2)`. The `B` and `BF` options cause **LINE** to outline or fill, respectively, the rectangle whose opposite corners are at the specified coordinates.

`Color` and `fcolor` refer to the Apple IIGS colors which are given in a table on page 3 of this chapter. If an attempt is made to use a color that is outside the range of 0 to 255, an "Illegal function call" error is generated.

x1, y1, x2, and y2 must all be values within the range -32768 to 32767 (integers), or an "Overflow" error is generated. If the points (x1, y1) or (x2, y2) lie outside the boundaries of the current window, the line or box is drawn and clipped at the window's edge.

The first coordinate pair may be left off, for example,

```
LINE -(X%,Y%)
```

and **LINE** will use the current graphics pen position as the starting point for the line or box to be drawn.

The **STEP** keyword specifies that the following coordinates are to be taken as relative to the current graphics pen position; if used before the second coordinate pair, x2% and y2% will be taken as offsets from the first coordinate pair. These two **LINE** statements draw the same line:

```
LINE (100,100)-(130,60)
LINE (100,100)-STEP(30,-40)
```

The default drawing color is the current foreground color (set by the **COLOR** statement), initially color number 0. A line or box may be drawn in a different color if a number is passed as the second argument in a **LINE** statement. If an attempt is made to use a color that is beyond the limit of 0 to 255, an "Illegal function call" message is generated.

When a filled box is drawn by specifying the BF after the color, the optional fcolor may be used to specify the color for which the rectangle will be filled. For example,

```
blue=17 : black=0
LINE (100,10)-(200,110),blue,black,BF
```

will draw a black rectangle with a blue border.

The graphics pen position will remain at the point specified by the second coordinate. This is true even if one or more of the coordinates is off the screen.

Example:

```
REM Chapter12/Example4
LINE (0,0)-(100,100)
   WHILE MOUSE(0)=0 : WEND
LINE (0,0)-(100,100),68,b
   WHILE MOUSE(0)<>0 : WEND
   WHILE MOUSE(0)=0 : WEND
LINE (0,0)-(100,100),,17,BF
   WHILE MOUSE(0)<>0 : WEND
   WHILE MOUSE(0)=0 : WEND
```

The first **LINE** statement draws a line from the upper left-hand corner to the point (100,100); the second draws an empty rectangle in the Apple's color 68 whose diagonal is the previous line; the third **LINE** statement draws the same rectangle, but fills it with the color 17. The example program waits for the user to click the mouse between each drawing.

CIRCLE

Format:

```
CIRCLE [STEP] (x,y),radius[,[color][,[start],[end][,aspect[,fcolor]]]]
```

Description:

Draws a circle, an arc, or an ellipse.

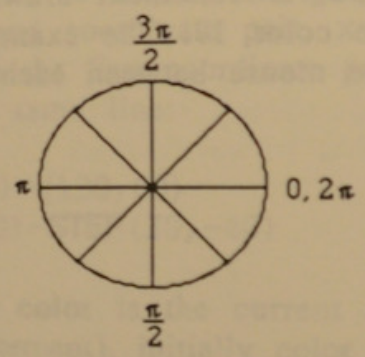
The `x` and `y` parameters are the center of the circle to be drawn. The **STEP** option indicates that the coordinates will be relative to the last point referenced (the current graphics pen position).

The `radius` of the figure to be drawn is in vertical pixels; if `radius` is zero, the circle is drawn as a dot. If negative, an "Illegal function call" error results. Discussion of the `aspect` parameter below describes how the radius applies to ellipses.

`Color` refers to one of the Apple IIGS colors which are given in a table on page 3 of this chapter. If an attempt is made to use a

color that is outside the range of 0 to 255, an "Illegal function call" error is generated.

`Start` and `end` specify the extent of an arc in radians. If the start angle is omitted, it is assumed to be 0. If the end angle is left off, a full circle is drawn from the point specified by the start angle. If an angle is negative, it is made positive, and a line is drawn from that point on the circle to the center. The radial line will be drawn only if both start and end angles are explicitly specified; this feature is useful for drawing pie charts and for other applications that require the subdivision of a circle. Finally, if the start angle is greater than the end angle, an arc is drawn from the start angle, past angle 0 (the horizontal), to the end angle. The angles are best shown with a picture:



`Aspect` describes the relationship of the y radius to the x radius, or the number of y-pixels to use per x-pixel; this is called the aspect ratio. If the aspect ratio is less than one, the radius will represent the width of the figure in x-pixels; if the aspect is greater than one, the radius will indicate the height of the figure in y-pixels. If `aspect` is omitted, the aspect ratio of the current screen will be used and a circle will be drawn.

Example:

```
REM Chapter12/Example5
CIRCLE (70,70),30,,,,.2
CIRCLE (70,70),30,,,,2!
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The first circle command will draw a short and wide ellipse, centered at (70,70) with a horizontal radius of 30 x-pixels. The

second will draw a tall and narrow ellipse with a vertical radius of 30 y-pixels.

After the circle, arc, or ellipse is drawn, the current graphics pen position is left at the designated center coordinates.

`Fcolor` is an optional number representing the color with which the circle will be filled. The `color` argument specifies the border color whereas the `fcolor` argument specifies the interior color. If an attempt is made to use a color that is beyond the limit of 0 to 255, an "Illegal function call" message is generated.

This example draws six circles centered at (100,100):

```
REM Chapter12/Example6
FOR radius% = 5 TO 30 STEP 5
   CIRCLE (100,100),radius%
NEXT radius%
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This example specifies negative angles on the **CIRCLE** statement to partition a circle into `NumParts!` slices:

```
REM Chapter12/Example7
NumParts!=12
Pi!=3.14159
FOR Angle!=0 TO 2*Pi! STEP 2*Pi!/NumParts!
   CIRCLE (300,90),50,0,-Angle!,Angle!-.01
NEXT
WHILE MOUSE(0)=0 : WEND ' Wait for click
```

The `Angle!-.01` is used to go "full circle" after plotting at `Angle!`; if `Angle!` were used in its place, this example program would draw a star with `NumParts!` radial lines.

SCROLL

Format:

```
SCROLL (x1,y1)-(x2,y2),delta-x,delta-y
```

Description:

Scrolls the contents of the specified rectangular area in the current output window.

`(x1,y1)-(x2,y2)` indicates the upper left and lower right corners of the portion of the screen to be scrolled, respectively.

`Delta-x` and `delta-y` are the values that determine the direction and extent (in pixels) of the scroll. Negative `delta x` and `delta y` scroll left and up, respectively. Positive `delta x` and `delta y` scroll right and down.

If the rectangle specified has a background contained within it, the background will be scrolled as well. After scrolling, the graphics pen position remains the same.

Example:

```
REM Chapter12/Example8
LOCATE 10,10
PRINT "TEXT"
SCROLL (70,80)-(115,100),8,8
LOCATE 10,10
PRINT "TEXT";
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The example displays the word `TEXT` in the window. The word is then scrolled down by one row and right by one column. To indicate the distance moved, the word `TEXT` is printed again in its original position.

CHANGING THE APPLE IIGS COLORS

One of the very attractive features of the Apple IIGS is its ability to display a wide range of colors. Applications can redefine the default colors to achieve a tailored appearance, or create and use their own colors to increase the visual clarity of screen presentations. This section discusses the statements which work with the Apple IIGS color palettes.

All of the screen colors the Apple IIGS uses come from a color palette which has 16 entries — one entry for each color on the 320x200 screen. The **PALETTE** statement can be used to change the entire palette or just a single entry on the palette. Although the Apple only uses one palette at a time, there are 16 palettes from which the *current palette* may be chosen. Any of the 16 palettes may be changed with the **PALETTE** statement.

Changes in the current palette affect the screen color *immediately* since the Apple graphics hardware is constantly looking-up the screen colors in the current palette. To select one of the 16 palettes as the current palette, use the **COLOR** staetment.

PALETTE

Format:

```
PALETTE [pal-number][,pal-entry,red,green,blue]
PALETTE pal-number USING array-name
PALETTE pal-number GET array-name
```

Description:

Alters the color intensities on one of the 16 Apple IIGS palettes specified by `pal-number`. A single palette entry is specified by `pal-entry`.

Both the palettes and the palette entries are numbered 0-15.

Each of the 16 color pens may be any one of 4,096 colors. The red, green, and blue parameters are decimal numbers from 0 to 1.0 which indicate the percentage of full intensity of each primary color to be used in the composition of the palette entry specified by pal-entry.

There are 16 intensities for each primary color (16x16x16 = 4096 colors total); decimal values will be "rounded" to the nearest 1/16. Thus, .09 will be "rounded" to .0625, or 1/16, etc. If a value outside of the range from 0 to 1 is specified, an "Illegal function call" error will be generated.

The Apple IIGS color palettes are reset to their previous status upon exit from the AC/BASIC application. A palette may also be reset by omitting the optional pal-entry, red, green, and blue parameters. For example, this statements resets palette number 0,

```
PALETTE 0
```

The current palette is referenced by omitting the pal-number parameter. To change the current palette to another palette number besides the initial palette of 0, use the COLOR statement. Note that this also changes the current palette for the menu bar.

The following is the master table of all sixteen colors on the 320x200 screen as defined through combinations of the primaries. It includes the color number which is used by the drawing statements, the palette entry which is used by PALETTE and the ROM routines in Appendix I, and the default RGB intensities:

Master 320x200 Color Table

color	color number	pal-entry	red	green	blue
black	0	0	0	0	0
dark gray	17	1	.4375	.4375	.4375
brown	34	2	.5	.25	.0625
purple	51	3	.4375	.125	.8125
blue	68	4	0	0	1
dark green	85	5	0	.5	0
orange	102	6	1	.4375	0
red	119	7	.875	0	0
beige	136	8	1	.6875	.625
yellow	153	9	1	1	0
green	170	10	0	.9375	0
light blue	187	11	.25	.875	1
lilac	204	12	.875	.6875	1
periwinkle blue	221	13	.4375	.5	1
light gray	238	14	.8125	.8125	.8125
white	255	15	1	1	1

A four point gray-scale could be defined by:

color	red	green	blue
black	0.00	0.00	0.00
dark gray	0.33	0.33	0.33
light gray	0.67	0.67	0.67
white	1.00	1.00	1.00

Any color definition in effect for the specified color number will be overridden by the **PALETTE** statement.

Note to advanced users: The **PALETTE** statements may also be used to set the values for the entire palette by including the **USING** keyword and specifying an array. By using the keyword **GET**, the array is filled with the entries of the specified palette. In this more advanced form of **PALETTE**, the array does not contain intensities 0 to 1, but rather the palette as the machine uses it. Each palette entry requires one word with each color intensity as a 4-bit value as shown, therefore, the array `array-name` should be dimensioned for 16 integers:

0	R	G	B

Example:

```
REM Chapter12/Example9
SCREEN 1                                  ' Use 320x200 screen
LINE (50,50)-(250,140),34,BF              ' Draw a brown box
LINE (50,80)-(250,110),102,BF             ' Draw an orange box
FOR RedMix!=0 TO 1 STEP 1/16
  FOR BlueMix!=0 TO 1 STEP 1/16
    PALETTE ,2,RedMix!,2/16,BlueMix!
    FOR delay=1 TO 100 : NEXT ' Delay
  NEXT
NEXT
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This program draws a brown box using color 34 which uses palette entry #2, blocks out the middle portion with another box of a different color, then illustrates the use of the **PALETTE** statement to change the color of the part of the screen drawn with palette entry #2, without redrawing.

PALETTE may also be used to create animation simply by changing palette entries or by changing the entire palette. This type of animation is really created by changing the color palettes of existing fixed objects in much the same way Las Vegas lights show movement by changing the electrical supply.

Since a palette for the 640x200 screen is composed of 4 mini-palettes, when a palette entry is changed, care must be taken as to changing other entries too. Since using **PALETTE** on the 640x200 screen is a tricky business, only one useful example is given which sets the 640x200 palette to have four true colors referenced as 0, 85, 170, and 255 without dithering:

```
REM Chapter12/Example10
FOR count=0 TO 12 STEP 4
   PALETTE ,count+0,0,0,0      ' Black
   PALETTE ,count+1,1,0,0      ' Red
   PALETTE ,count+2,0,0,1      ' Blue
   PALETTE ,count+3,1,1,1      ' White
NEXT count
LINE (99,9)-(501,141),0,B      ' Black border
col=85
FOR y=10 TO 130 STEP 10
   LINE (100,y)-(500,y+10),col,BF
   col=340-col
NEXT y
LINE (100,10)-(250,79),170,BF
WHILE MOUSE(0) = 0 : WEND  ' Wait for click
```

The menu bar initially uses palettes 1-6 for the six Apple menu logo colors. On the 320x200 screen, the text uses palette entry #0, the menu bar background uses #15, but the color of each Apple logo scan line is entry #1. The only difference on the 640x200 screen is that the mini-palette entries are used, thus, #3 is the background color. This small program fragment will "fade out" the window border, controls, and the menu bar on the 320x200 screen to give a full-screen effect. Note that color 0 becomes white and even the mouse cursor disappears.

```
PALETTE 0,0,1,1,1  ' Black to white on main palette
FOR x=1 TO 6
     PALETTE x,0,1,1,1  ' Menu text to white
     PALETTE x,1,1,1,1  ' Logo colors to white
     PALETTE x,15,1,1,1 ' Background to white
NEXT x
```

COLOR

Format:

COLOR [text-fore] [,text-back] [,pen-fore] [,pen-back] [,pal]

Description:

Changes the current foreground and/or background text pen and graphics pen colors. It also changes the current palette.

The text-fore, text-back, pen-fore, and pen-back values refer to the Apple IIGS palette entries as given in the following tables. Using the **PALETTE** statement, the default colors may be redefined. The numbers in the 320x200 table are simply the color numbers on page 3 each divided by 17.

320x200 screen			
0	black	8	beige
1	dark gray	9	yellow
2	brown	10	green
3	purple	11	light blue
4	blue	12	lilac
5	dark green	13	periwinkle blue
6	orange	14	light gray
7	red	15	white

640x200 screen	
0	black
1	purple
2	green
3	white

The foreground text pen and graphic pen colors are initially black (0) in both screen modes. The background text pen and graphic pen colors are initially white (15 for 320x200 mode and 3 for 640x200 mode).

If an attempt is made to use a color that is beyond the range for the current screen mode, an "Illegal function call" message is generated.

There must be at least one argument passed to COLOR or a "Missing argument" error will occur at compile time. However, any of the arguments may be omitted.

Pal is an integer 0 to 15 which specifies the current palette to use for all the screen colors. See the introduction to this section for a description of the Apple IIGS palettes.

Example:

```
REM Chapter12/Example11
PRINT "This is default text color"
COLOR 2
PRINT "The foreground text color was changed"
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The example sets the text pen foreground color to palette entry #2.

POINT

Format:

```
POINT(x,y)
```

Description:

Returns the palette entry at the coordinates (x,y). For the 320x200 screen, this is a value from 0 to 15. For the 640x200 screen, the value returned is from 0 to 3 which is a mini-palette entry as described in the section "Understanding the Colors".

With the 320x200 screen, simply multiply the value returned by POINT() with the number 17 to calculate the color number.

If the point specified is outside the limits of the current window, a -1 is returned by the function.

Example:

```
REM Chapter12/Example12
THE.COLOR%=POINT(10,10)
PRINT "Color of pixel at (10,10) is";THE.COLOR%
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

CLS

Format:

```
CLS
```

Description:

This command clears the current output window. No other windows are affected.

The text pen is reset to the upper left corner of the window and the graphics pen is moved to the window's center.

The window is cleared by filling it with the current background graphics pen color which is set by the **COLOR** statement or the **BACKCOLOR** Toolbox call. For a complete description of this call, see Appendix I. For example,

```
SCREEN 1          ' Use 320x200 screen
BACKCOLOR 7       ' Set background color to red
CLS               ' Clear window with red
BACKCOLOR 15      ' Reset background color to white
```

LCOPY

Format:

```
LCOPY [color-flag]
```

Description:

Prints the entire screen on the attached printer.

LCOPY send a "screen dump" to the current printer selected from the `Choose Printer` item in the `File` menu. If the optional `color-flag` value is 0 then the screen dump will be made for a printer with a single-color ribbon. If color-flag is non-zero, a color screen dump will be produced on printers which support color like the ImageWriter II.

ARRAY-BASED GRAPHICS

Rectangle areas of the screen may be stored as a graphic image into an array and saved or placed elsewhere on the screen. In addition to the normal file I/O methods, an array containing an image may be written and read from a disk file by using the array-based input and output statements described near the end of Chapter 16.

GET (graphics)

Format:

```
GET (x1,y1)-(x2,y2),array-name[(index1[,index2...])]
```

Description:

Saves the specified rectangular portion of the screen in an array, which can then be redisplayed on the screen with the graphics **PUT** statement.

The screen segment being saved is defined by `(x1,y1)` and `(x2,y2)`, where the first coordinate pair is the upper left corner and the second is the lower right corner of a rectangle.

The array being used must be a numeric array and must be large enough to hold the screen data being saved. The minimum array size, in bytes, can be determined by using the **SCREEN()** function:

```
req-size = SCREEN(x1,y1,x2,y2)
```

where x1, y1, x2, and y2 are the elements of the coordinate pairs that define the rectangle on the.

The first 4 bytes of a graphic array contain information about the saved image; the first two bytes hold the image width and the second two hold the image height. If an integer array is used, these values can be accessed as the first and second elements.

The precision of the array used is irrelevant, as long as the array is big enough. Double-precision (#) array elements are 8 bytes long, single-precision (!) 4 bytes, and integers (%) are 2 bytes; thus, to store a rectangle defined by the coordinates (0,0)-(100,100) on the 640x200 screen, 3240 bytes are required, and any of the following declarations would declare an array of sufficient size:

```
DIM Array%(1620)      ' OPTION BASE 0 assumed
DIM Array!(810)
DIM Array#(405)
```

The Index specifications define where in the array the image is to be stored. If omitted, storage of the screen image begins with the first element in the array.

Note that by using a multi-dimensional array, several images can be stored in the same array. This is useful for quickly displaying several images using the **GET** and **PUT** statements. In this case, you should specify a size for the first dimension of the array that is adequate to hold the largest of the images being retrieved, and use the second dimension (or combination of the other dimensions) to identify or number each image. For example, to display 8 images which require 500 bytes storage each, declare an array in this format:

```
DIM ImageArray%(250,8)
```

Then the statement

```
GET (200,100)-(240,150),ImageArray%(0,5)
```

would retrieve an image from the screen and store it as image number 5 in the array.

Example:

```
REM Chapter12/Example13
PRINT "Hello World!"
DIM a%(SCREEN(0,0,96,8))
GET (0,0)-(96,8), a%        ' GET it
PUT (0,0),a%,XOR            ' Erase it
WHILE MOUSE(0) = 0
   x%=RND*500
   y%=RND*180
   PUT (x%,y%),a%                ' Display text
   FOR x=1 TO 300 : NEXT         ' Delay a little
   PUT (x%,y%),a%,XOR            ' Erase text
WEND
```

Since in the default text font characters are about 8 pixels by 8 pixels, the "Hello World!" text is 8 pixels by 96; the example program stores the text in an array, uses the XOR mode to erase it, and then PUTs it randomly to the screen until the mouse is clicked.

PUT (graphics)

Format:

```
PUT [STEP] (x1,y1)[-(x2,y2)],array[(ind[...,ind])][,mode]
```

Description:

Copies an image saved in an array by a graphics GET statement to the screen at the specified coordinates.

The coordinate pairs (x1,y1) and (x2,y2) represent the upper left- and lower right-hand corners of the image's destination rectangle. The image will be scaled to fit the rectangle on the screen. If only (x1,y1) is specified, no scaling is done.

The graphic image data is retrieved from `array-name;ind` are array index specifications are used to define the point inside the array where the graphic data begins. The same protocol as for **GET** applies when addressing one of several objects stored in a multi-dimensional array.

The manner in which the data in the array is transferred to the screen depends upon `mode`. `Mode` specifies the logical operation to be used to combine the data from the array with the screen data. The five modes are **PSET**, **PRESET**, **AND**, **OR**, and **XOR** (the default).

PSET causes the array image to be copied directly to the destination rectangle, erasing its previous contents.

PRESET displays the image with inverted colors: each color in the image is displayed as the maximum (255) minus its color number.

AND displays only those parts of the image which have the same color as the background pixels of the destination rectangle.

OR combines the shape with the background, displaying colors with numbers that result from a logical **OR** of the image pixel color numbers and the background pixel color numbers.

XOR performs the same type of function as **OR** and **AND**, but the logical **XOR** operation is used to determine the destination colors. If an image is **PUT**, then **PUT** again to the same location with **XOR** mode, all the pixel **XOR** operations will result in the original color before the first **PUT**, which is equivalent to erasing the image.

See the example for the **GET** statement above.

A **GET** image from the 640x200 screen can not be **PUT** on the 320x200 screen, and vise-versa, since the number of bits representing a color pixel is different for each screen type. Error number 155 will be generated in such a case.

SCREEN()

Format:

```
SCREEN(x1,y1,x2,y2)
```

Description:

The **SCREEN()** function returns the number of bytes needed to store the graphic image in the rectangle defined by `x1,y1,x2,` and `y2`. The screen resolution is automatically taken into effect since an image on the 320x200 screen requires about twice as much memory as the same size image on the 640x200 screen.

It is useful for dynamically declaring the size of arrays needed to store an image using the **GET** statement. See the example of the **GET** statement earlier in this section which uses the **SCREEN()** function.

PICTURE-BASED GRAPHICS

PICTURE

Format:

```
PICTURE ON
PICTURE OFF
PICTURE [(x1,y1)[-(x2,y2)]][,picture-string]
PICTURE$
```

Description:

The **PICTURE** statements may be used to route all drawing commands to a string for later use instead of the screen.

PICTURE ON starts the recording of the screen display. During the recording, all drawing commands like **LINE**, **CIRCLE**, and **PRINT** which normally draw on the screen, instead, encode their operation into the predefined string **PICTURE$**. No drawing will appear on the screen while the picture is being recorded.

PICTURE OFF stops the picture recording. All drawing commands after **PICTURE OFF** is executed appear on the screen as usual.

The **PICTURE** statement will draw the recorded picture in `picture-string` at the location specified by `(x1,y1)-(x2,y2)`. The picture will be scaled appropriately to fit the destination rectangle. If only the upper left-hand corner `(x1,y1)` is specified, the picture is drawn unscaled. The most recently recorded picture in **PICTURE$** will be drawn if the `picture-string` argument is omitted.

Picture graphics differ from array-based graphics, such as **GET** and **PUT**, in that the drawing is encoded internally as Apple IIGS Toolbox commands unlike array-based graphics which store graphics as a bit image. What does this mean? The array-based graphics are slightly faster for complex drawings since the image is not calculated and "drawn" for each **PUT**; instead, the bit-image is quickly placed on the screen changing the entire rectangle. Picture graphics, on the other hand, are calculated and "drawn" for each **PICTURE** statement, but the storage space for the picture can be significantly smaller than what is required for array-based graphics.

In this example, four pictures are recorded and stored into an array of strings. Each one is drawn, in sequence, at the mouse position. Click the mouse button to stop the program:

```
REM Chapter12/Example14
DIM colors(4),frame$(4)
colors(0)=17  : colors(1)=68   ' blue, red
colors(2)=136 : colors(3)=102  ' green, orange
FOR count=0 TO 3              ' Record 4 pictures
   PICTURE ON                 ' Start recording
   LINE (0,0)-(80,40),,colors(count),BF
   LINE (10,5)-(70,35),,colors((count+1) MOD 4),BF
   LINE (20,10)-(60,30),,colors((count+2) MOD 4),BF
   LINE (30,15)-(50,25),,colors((count+3) MOD 4),BF
   PICTURE OFF                ' Stop recording
   frame$(count)=PICTURE$
NEXT count

count=0
WHILE MOUSE(0)=0              ' Stop when clicked
   PICTURE (MOUSE(1),MOUSE(2)),frame$(count)
   count=(count+1) MOD 4
WEND
```

A picture recorded from the 640x200 screen can not be displayed on the 320x200 screen and vise-versa. Error number 155 will be generated in such a case.

A picture may be saved to a disk file and loaded back in by the same program or even another program. The following two code fragments show a simple way to do the saving and loading.

```
REM Write the string PICTURE$ to a file
f$=FILES$(0)    ' Get a filename
OPEN f$ FOR OUTPUT AS #1
PRINT #1,PICTURE$
CLOSE #1
      .
      .
REM Load a picture from a file into string pic$
f$=FILES$(1)
OPEN f$ FOR INPUT AS #1
pic$=INPUT$(LOF(1),1)
CLOSE #1
PICTURE pic$   ' Draw the picture
      .
      .
```

UNDERSTANDING THE COLORS

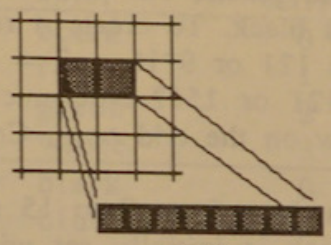
This section gives a complete explanation of the color characteristics and is *not* needed in order for you to use all the graphics statements and all the colors. It describes color palettes and shows how colors other than those in the previous tables can be displayed.

There is a trade-off between the two screen modes. The 320x200 screen has lower resolution and 16 colors. The 640x200 screen has twice the resolution, but its 16 colors are produced by dithering and are not as clear.

The 320x200 screen

Each pixel on the 320x200 screen requires 4 *bits* of memory to define the color. A bit is either on (1) or off (0). Four bits can uniquely represent 16 numbers from 0 to 15. For the screen pixels, these 16 numbers represent the 16 possible colors. Although the bit is the smallest piece of information the computer can process, on the screen the Apple works with bytes which each contain 8 bits. Four bits represent 16 numbers, but 8 bits represent 256 numbers. The following

picture shows a byte divided into its 8 bits which define two pixels:



8 bits
2 pixels

A horizontal scan-line on the 320x200 screen requires 160 bytes. The upper (or left) 4 bits of a each byte give the color for an even pixel; the lower 4 bits are for an odd pixel. The following table shows the 16 possible colors in their 4-bit forms

320x200 Screen Palette Entries

0	black	8	beige
1	dark gray	9	yellow
2	brown	10	green
3	purple	11	light blue
4	blue	12	lilac
5	dark green	13	periwinkle blue
6	orange	14	light gray
7	red	15	white

When the computer draws on an even pixel, it uses the color in the upper 4 bits; the lower 4 bits for an odd pixel. If the colors in the upper and lower halves are not the same, a striped color appears on the screen.

To create a byte-value color which the computer will accept, use the following formula where `even` is the 4-bit color number, from the previous table, of the even pixel and `odd` is the 4-bit color of the odd pixel.

```
color = even * 16 + odd
```

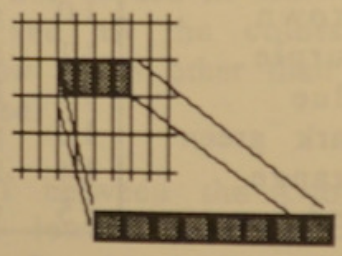
For example, to have the color yellow on all drawn pixels, the color should be specified as 9 (yellow) in each 4-bit half. Thus,

the color is 9*16 + 9 = 153 which is the color to be used in all graphics statements. If just the number 9 is used, yellow will appear on all odd pixels, but the even pixels will have the 4-bit color 0 which is black. To create a red and yellow striped color, use 7*16 + 9 = 121 or 9*16 + 7 = 151. What is the difference between using 121 or 151? Not much — 121 has red on even pixels and yellow on the odd ones. Color 151 is the opposite.

The 16 color numbers from 0 to 15 are actually entries into the color palette which is initially set with the colors shown in the table "320x200 Screen Palette Entries" on the previous page. The colors may be changed with the **PALETTE** statement.

The 640x200 screen

Each pixel on the 640x200 screen requires only 2 bits of memory which can represent 4 color numbers from 0 to 3. Dithering is used to produce the effect of having 16 colors available, however. Each byte on the screen holds the colors for 4 pixels — 2 bits each.



8 bits
4 pixels

The 2-bit color numbers 0 to 3 do not directly represent a single color. Instead, the color of each pixel comes from the palette which is divided into four mini-palettes with mappings as shown.

640x200 Screen Palette Entries

palette entry	color	mini-palette entry	pixel mapping
0	black	0	third
1	red	1	third
2	green	2	third
3	white	3	third
4	black	0	fourth
5	blue	1	fourth
6	yellow	2	fourth
7	white	3	fourth
8	black	0	first
9	red	1	first
10	green	2	first
11	white	3	first
12	black	0	second
13	blue	1	second
14	yellow	2	second
15	white	3	second

Each screen pixel is not mapped sequentially into the byte. Instead, when the computer draws on the *first* pixel of each four-pixel group on the screen, it gets the color from the *third* 2-bit pair. The other pixel mappings are given in the table. This technique of mixing different colors to produce what looks like a solid color is called *dithering*.

To create a byte-value color which the computer will accept, use the following formula where `first`, `second`, `third`, and `fourth` are the 2-bit color numbers (mini-palette entries) of the screen pixels from the table.

```
color = first * 64 + second * 16 + third * 4 + fourth
```

For example, to make a purple color, use entry #1 for each pixel. This will make every pixel blue or red depending on its screen position; the net result is purple. When the `first/second` and `third/fourth` entries are the same, the color produced is fairly solid much like having two 4-bit colors on the 320x200 screen. For example, if `first` and `second` are 1 and 2, respectively, and `third` and `fourth` are also 1 and 2, then the resulting color is 102, solid orange. The 16 solid color

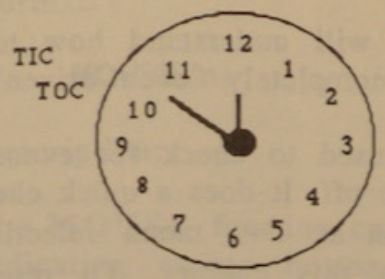
numbers are given in the table "640x200 Screen Colors" a few pages back. When the `first/second` and `third/fourth` entries are not the same, the color is striped.

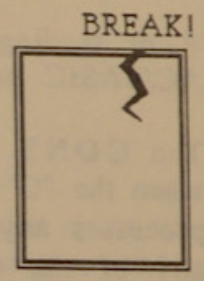
Two pixels with the same color must appear next to each other horizontally for the 640x200 screen colors to appear properly. This is because the 640x200 screen can physically show only four different colors; your eye, however, will put pixel pairs together so you can see up to 16 different colors. For example, a red line which is only one pixel wide will appear orange unless another red line is drawn immediately next to it:

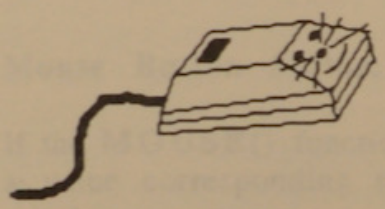
```
LINE (10,50)-(10,100),68 ' It looks orange-ish
a$=INPUT$(1)
LINE (11,50)-(11,100),68 ' Now, it looks red
```

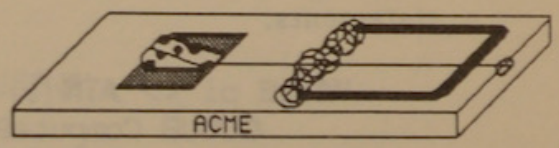
To change the palette colors, use the **PALETTE** statement described in this chapter.

CHAPTER 13









EVENTS AND EVENT TRAPPING

Events and event detection are fundamental concepts. Much of the application software currently available is "event driven". This means that the user controls the flow of the program by selecting menus, clicking the mouse, etc. This chapter explains how to detect and process events in AC/BASIC.

There are many different types of events that your programs can detect and process. These events include menu selection, mouse, keyboard activity, run-time errors, break events (Ó-.), and timer events.

There are two ways to check for events of interest: polling and trapping. Polling, like the word implies, is like going from door to door and asking about something of interest. In BASIC, this means that the program will constantly be checking to see if an event occurred via functions provided for this purpose. Trapping, again like the name implies, is like setting a mouse trap; you set the trap, if a mouse event occurs (a mouse in your house), the trap catches it, and you know the event occurred. Polling for the same mouse event in the example would entail

(pun intended) standing around with a hammer watching for a mouse.

After reading this chapter, you will understand how to write AC/BASIC programs which are completely "event-driven".

The **CONT** statement may be used to check for events even when the "C" and "N" options are off. It does a quick check and processes any current events such as Ó-., menu selections, or any of the events described in this chapter. To repeat the example from Chapter 4, the following program segment will check for events after executing a large series of mathematical statements:

```
WHILE pi <> ATN(1)*4
    GOSUB ComputePi
    CONT
WEND
```

MOUSE EVENTS

Mouse events, like menu events, are at the heart of the Apple IIGS user interface capabilities. With the mouse information available to the AC/BASIC programmer, the user can select graphics images, buttons, and edit fields, or have dynamic access to the **LINE**, **CIRCLE**, and **PSET** graphics drawing statements. The mouse event-trapping commands (**ON MOUSE GOSUB** and **MOUSE {ON|OFF|STOP}**) allow design of a program whose flow is completely mouse-driven.

MOUSE()

Format:

MOUSE (n)

Description:

The **MOUSE** function can return seven different types of values indicating mouse button activity, button status, and mouse pointer positions.

Mouse Button Status: MOUSE(0)

If the **MOUSE**() function is passed an argument of 0, it returns a value corresponding to the mouse button status in the active window.

MOUSE(0) will return an integer from -3 to 3; the following list gives the button status corresponding to each possible value:

0 - The button is currently up and has not been clicked since the last **MOUSE**(0) function call.

1 - The button was clicked and released.

-1 - The button was clicked, and is currently down.

2 - The button was double-clicked and released.

-2 - The button was double-clicked, and is currently down.

3 - The button was triple-clicked and released.

-3 - The button was triple-clicked, and is currently down.

When **MOUSE**(0) returns a negative number, it usually indicates that the user is holding down the button while moving the mouse; the values retrieved by **MOUSE**(3) through **MOUSE**(6) can be used to determine the starting and ending

coordinates of the movement. The mouse button status is updated each time the mouse button is clicked.

The mouse button can be either single, double, or triple-clicked. Note that if the delay between clicks is too long, the action will be interpreted as two or more separate actions. For example, if the user intends to perform a double-click, and the delay between the first and second clicks is too long, the Apple IIGS will interpret this as two single-clicks. You may use the Apple IIGS Control Panel to adjust the amount of time after which the computer considers mouse clicks to be separate.

Conversely, due to the increase in execution speed provided by the compiler, **MOUSE**(0) followed by subsequent calls to **MOUSE**(0) may "dissect" double or triple clicks into multiple single clicks. The following example uses polling to detect mouse events and return mouse button status, with a delay long enough to allow the user to triple-click the mouse.

```
    REM Chapter13/Example1
mouseproc:
    Delay!=TIMER
    WHILE TIMER < Delay!+1          ' Wait 1 second
    WEND
    mouse.event = MOUSE(0)          ' Get button status
    IF ABS(mouse.event) <= 3 AND mouse.event <> 0 THEN
      IF mouse.event < 0 THEN
        PRINT "The button is down after it was ";
      ELSE
        PRINT "The button has been ";
      END IF
      mouse.event=ABS(mouse.event)
      IF mouse.event=1 THEN PRINT "single-clicked."
      IF mouse.event=2 THEN PRINT "double-clicked."
      IF mouse.event=3 THEN PRINT "triple-clicked."
    ELSE
      PRINT "No mouse button event."
    END IF
GOTO mouseproc
```

Mouse Pointer Coordinates

Passing parameter values of 1 through 6 cause the **MOUSE()** function to return mouse pointer coordinates in the current active window. These values are updated *only* after a call to **MOUSE**(0) has been executed.

Current X Coordinate: **MOUSE**(1) returns the current horizontal pixel position of pointer.

Current Y Coordinate: **MOUSE**(2) returns the current vertical pixel position of the pointer.

Starting X Coordinate: **MOUSE**(3) returns the horizontal pixel position of the pointer when the button was last clicked.

Starting Y Coordinate: **MOUSE**(4) returns the vertical pixel position of the pointer when the button was last clicked.

Ending X Coordinate: **MOUSE**(5) returns two different values, depending on the state of the mouse button during the last **MOUSE**(0) call. If the button was up, the pointer's horizontal pixel position at the button's last release is returned. If down, the value returned is the horizontal pixel position of the pointer when **MOUSE**(0) was called, and will be the same as **MOUSE**(1).

Ending Y Coordinate: **MOUSE**(6) returns two different values, depending on the state of the mouse button during the last **MOUSE**(0) call. If the button was up, the pointer's vertical pixel position at the the button's last release is returned. If down, the value returned is the pointer's vertical pixel position when **MOUSE**(0) was called, and will be the same as **MOUSE**(2).

All pixel position values are relative to the origin of the current active window, with minimum values of zero; **MOUSE**(1) through **MOUSE**(6) never return negative values.

If the mouse button is clicked and held down while the mouse is moved, the **MOUSE**(3) through **MOUSE**(6) functions support actions based on the starting and ending coordinates. For example, the statement

```
LINE (MOUSE(3),MOUSE(4))-(MOUSE(5),MOUSE(6)),,B
```

will draw a rectangle in the current output window enclosing the starting and ending coordinates of the mouse movement.

If the parameter of **MOUSE()** is negative or greater than 6, an "Illegal function call" error is generated.

The first example is an enhanced version of the example for **MOUSE(0)** above, displaying all the current **MOUSE()** function values dynamically:

```
REM Chapter13/Example2
FOR MouseFun%=0 TO 6
   LOCATE 1,MouseFun%*10+1 ' Tab to proper column
   PRINT USING "MOUSE(#)";MouseFun%; ' Print header
NEXT MouseFun%
WHILE INKEY$=""          ' Until a key is pressed
   PAUSE!=TIMER
   WHILE TIMER < PAUSE!+.5    ' Wait 1/2 a sec
   WEND
   FOR MouseFun%=0 TO 6
      LOCATE 2,MouseFun%*10+1
      PRINT MOUSE(MouseFun%);  ' Print MOUSE value
   NEXT MouseFun%
WEND
```

The next example displays the current pointer position in the upper left corner of the window. If the mouse button is pressed, a dot is drawn. If the button is held down as the mouse is moved, a series of dots will be drawn.

```
REM Chapter13/Example3
LOCATE 19,10
PRINT"Press ESCape to quit"
WHILE INKEY$ <> CHR$(27)      ' Check for ESC key
  clicks%=MOUSE(0)        ' Update pointer coordinates
  x%=MOUSE(1) : y%=MOUSE(2)        ' Get coordinates
  IF ABS(clicks%) > 0 THEN
    PSET (x%,y%)   ' Plot a point if clicked
  END IF
  LOCATE 1,1
  PRINT USING "(###&###)";x%,",",y%
WEND
```

Mouse Event Trapping

Format:

```
ON MOUSE GOSUB {line-number|label}
MOUSE {ON|OFF|STOP}
```

NOTE: any AC/BASIC program containing these statements requires compilation with the "N" option set.

Description:

After an **ON MOUSE GOSUB** statement has been executed, a **MOUSE ON** statement activates mouse event trapping, meaning that any subsequent clicks of the mouse button will update the mouse button status *and* automatically perform a **GOSUB** to the specified line number or label. The `line-number` or `label` specified *must* be located in the main program.

Mouse events are processed only if the mouse pointer is inside the currently active window; see Chapter 12 for a discussion of window states. Within the subroutine, the **MOUSE**(0) function may be used to determine how many times the mouse button was clicked, whether it is currently up or down, and to update the **MOUSE**() function pointer coordinate values.

MOUSE OFF turns off mouse event trapping. After a **MOUSE OFF** is executed, no mouse events will be trapped and handled

by the subroutine indicated by the **ON MOUSE GOSUB** statement. The statement **ON MOUSE GOSUB** 0 will also disable the mouse event trapping; however, after this statement, a **MOUSE ON** will have no effect until another **ON MOUSE GOSUB** is executed specifying a program subroutine line or label.

MOUSE STOP suspends mouse event processing; if mouse events occur while trapping is suspended, AC/BASIC will remember only the most recent one, which will be processed as soon as a **MOUSE ON** is executed.

All forms of the **MOUSE {ON|OFF|STOP}** statement do not alter the current mouse pointer coordinates, leaving them as last updated by a **MOUSE**(0) call.

Example:

```
    REM Chapter13/Example4
    ON MOUSE GOSUB mouseproc
    MOUSE ON
10
    MOUSE STOP
    FOR I=1 TO 1000 : NEXT I
    MOUSE ON
    GOTO 10
    END

mouseproc:
    mouse.event = MOUSE(0)
    m1=MOUSE(1) : m2=MOUSE(2)
    PRINT "Mouse is currently at (";m1;",";m2;")"
    IF ABS(mouse.event) <= 3 AND mouse.event <> 0 THEN
      IF mouse.event < 0 THEN
        PRINT "The button is down after it was ";
      ELSE
        PRINT "The button has been ";
      END IF
      mouse.event=ABS(mouse.event)
      IF mouse.event=1 THEN PRINT "single-clicked."
      IF mouse.event=2 THEN PRINT "double-clicked."
      IF mouse.event=3 THEN PRINT "triple-clicked."
    END IF
    MOUSE ON
    RETURN
```

The example displays information about the current state of the mouse button and the position of the mouse in the window.

DIALOG EVENTS

Dialog events include mouse clicks on screen buttons, window alterations, and edit field interactions. Using dialogs, the user can create and control various kinds of on-screen buttons with AC/BASIC commands. Edit fields can be used to create customized interactive data entry forms.

DIALOG()

Format:

DIALOG (n)

Description:

The **DIALOG** function can return information about six different types of dialog events. The value of n may range from 0 to 5.

Event trapping is not required to use the **DIALOG()** function.

Dialog Status: DIALOG(0)

When the **DIALOG()** function is passed as argument of 0, it returns information as to which type of dialog event has occurred.

DIALOG(0) will return an integer value from 0 to 7; the following list of values explains the corresponding dialog event:

0 There have been no new dialog events since last checked with **DIALOG**(0).

1 A button has been selected in the current output window. **DIALOG**(1) can be used to get the button number.

2 The current edit field has been changed or the mouse has been clicked in an edit field by the user.

DIALOG(2) can be used to get the new edit field number.

3 An inactive window has been selected by the user. **DIALOG**(3) can be used to get the window number. The currently active window is not changed automatically. To change the currently active window, use the **WINDOW** statement.

4 A window's Close Box has been selected by the user. **DIALOG**(4) can be used to get the window number. To close the window, use the **WINDOW CLOSE** statement.

5 An output window has been partially or completely erased and needs to be redrawn. This occurs when window are moved or resized. **DIALOG**(5) can be used to get the window number. You should redraw the window content.

6 The RETURN key was used when an edit field or button in the currently active window will not accept the RETURN key.

7 The TAB key was used in an edit field.

DIALOG(1) returns the number of the most recently selected button.

DIALOG(2) returns the number of the current edit field. It returns zero if there is no edit field.

DIALOG(3) returns the window number of the selected window. It returns zero if no window selection change has been made.

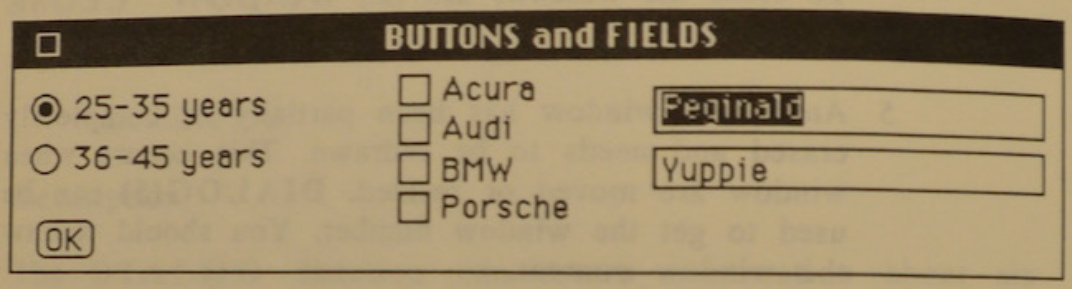
DIALOG(4) returns the window number of the window whose Close Box was selected. It returns zero if no Close Box has been selected.

DIALOG(5) returns the window number of the most recently partially erased window.

The button, edit field, and window numbers are assigned in the first argument of the **BUTTON**, **EDIT FIELD**, and **WINDOW** statements, respectively.

If the parameter of **DIALOG()** is negative or greater than 5, an "Illegal function call" error is generated.

The following example creates two windows: one has some buttons and edit fields as shown in the picture while the other is used as an output window that displays the dialog actions. Try clicking buttons and typing in the edit fields.




```
REM Chapter13/Example5
REM Compile with "N" option
WINDOW 2,"Output Window",(5,140)-(400,195),1
WINDOW 1,"BUTTONS and FIELDS",(5,30)-(630,110),1
BUTTON 1,2,"25-35 years",(5,10)-(120,22),3
BUTTON 2,1,"36-45 years",(5,30)-(120,42),3
BUTTON 3,1,"Acura",(150,5)-(220,17),2
BUTTON 4,1,"Audi",(150,20)-(220,32),2
BUTTON 5,1,"BMW",(150,35)-(220,47),2
BUTTON 6,1,"Porsche",(150,50)-(220,62),2
EDIT FIELD 2,"Yuppie",(400,27)-(600,40),1,1
EDIT FIELD 1,"Reginald",(400,10)-(600,23),1,1
BUTTON 7,1,"OK",(10,60)-(40,75),1
ON DIALOG GOSUB ShowPerson : DIALOG ON
WHILE ok=0 : WEND                    ' Loop until ok is 1
DIALOG OFF : BUTTON CLOSE 0 : EDIT FIELD CLOSE 0
END
```

```
ShowPerson:
  d0=DIALOG(0) : d1=DIALOG(1)
  IF d0=4 OR d0=6 THEN ok=1 : RETURN   ' Close box
  IF d0=1 AND d1=7 THEN ok=1 : RETURN  ' OK button
  DIALOG OFF
  IF d0=1 THEN
     BUTTON d1,3-BUTTON(d1)  ' Change state on screen
     IF d1=1 THEN BUTTON 2,3-BUTTON(2)  'Can be only
     IF d1=2 THEN BUTTON 1,3-BUTTON(1)  'one age
  END IF

  p$=EDIT$(1)+" is "
  IF BUTTON(1)=2 THEN
     p$=p$+"a young "+EDIT$(2)+" with: "
  ELSE
     p$=p$+"an old "+EDIT$(2)+" with: "
  END IF
  IF BUTTON(3)=2 THEN p$=p$+"Acura  "
  IF BUTTON(4)=2 THEN p$=p$+"Audi  "
  IF BUTTON(5)=2 THEN p$=p$+"BMW  "
  IF BUTTON(6)=2 THEN p$=p$+"Porsche  "
  WINDOW OUTPUT 2
  PRINT p$
  WINDOW OUTPUT 1
  DIALOG ON
  RETURN
```

Dialog Event Trapping

Format:

```
ON DIALOG GOSUB {line-number|label}
DIALOG {ON|OFF|STOP}
```

Description:

The **ON DIALOG GOSUB** statement sets up dialog event trapping so that dialog events will automatically cause the computer to perform a **GOSUB** statement to the specified `line-number` or `label`. **DIALOG ON** is required to turn on the trapping.

DIALOG OFF turns off dialog event trapping. After **DIALOG OFF** is executed, no dialog events will be trapped and processed by the subroutine specified in the **ON DIALOG GOSUB** statement. **DIALOG ON** may be used to turn the trapping back on. **ON DIALOG GOSUB** 0 may also be used to turn off dialog event trapping, however, a subsequent **ON DIALOG GOSUB** statement is required to start it again.

DIALOG STOP suspends dialog event processing. If dialog events occur while trapping is suspended, AC/BASIC will remember only the most recent one which will be processed after a **DIALOG ON** is executed.

All forms of the **DIALOG {ON|OFF|STOP}** statement do not alter the current dialog information returned from the **DIALOG()** function. The information is left unchanged as it was from the last **DIALOG()** call.

BUTTON

Format:

```
BUTTON button-id,state[,title,(x1,y1)-(x2,y2)[,type]]
BUTTON CLOSE button-id
BUTTON (button-id)                    (function syntax)
```

Description:

Draws a button in the current output window.

The **BUTTON** statement displays a button on the screen. It will remain on the screen until it is removed by using **BUTTON CLOSE** or closing the window containing the button.

There are two ways of detecting button click by the user: The **BUTTON()** function may be used to "poll" the buttons for changes. Or, dialog event trapping may be used to automatically detect changes in the button states.

`Button-id` is the number from 1 to 253 by which the button may be uniquely referenced. It can be any value greater or equal to 1. It is customary to number the buttons 1, 2, 3, ... in a window.

`State` is the initial button state:

0 Inactive button, dimmed.

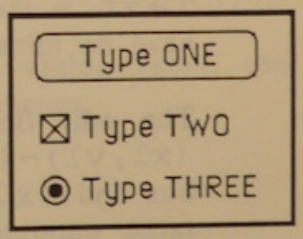
1 Active button, not selected.

2 Active button, selected

`Title` is a string which will appear beside or in the button depending upon the button type.

The button is displayed in the rectangle defined by `(x1,y1)-(x2,y2)` in the current output window. `(x1,y1)` is the upper-left corner; `(x2,y2)` is the lower-right corner. Note that a button is selectable anywhere within its rectangle, not just in the "button". The rectangle region is cleared to the background color of the window before the button is drawn.

`Type` specifies one of three different type of buttons specified as 1, 2, or 3 which are shown in their active state. Type 1 is the default as is called a push button. Type 2 is a check button. Type 3 is called a radio button usually meaning only one button of type 3 is on at a time, like the pre-set station buttons on a radio



BUTTON CLOSE removes the specific button with `number-id` from the window. To remove all the buttons, use **BUTTON CLOSE** 0.

The **BUTTON()** function will return the following information about the button with the `button-id`:

0 The button is inactive and is dimmed.

1 The button is active but is not selected.

2 The button is active and is selected.

EDIT FIELD / EDIT$

Format:

```
EDIT FIELD field-id[,text,(x1,y1)-(x2,y2)[,[type][,justify]]]
EDIT FIELD CLOSE field-id
EDIT$ (field-id)
```

Description:

EDIT FIELD creates a box which contains editable text on the current output window. The **EDIT$** function will return the current text in the box.

`Field-id` is the number from 1 to 255 by which the edit field may be referenced. Each edit field must have a unique `field-id`. It is customary to number the edit fields of a window 1, 2, 3... and so on..

`Text` is the initial text which is displayed in the edit field. To have no initial text, use the null string "".

The edit field is a rectangle box on the screen defined by `(x1,y1)-(x2,y2)` where `(x1,y1)` is the upper-left corner of the box and `(x2,y2)` is the lower-right corner. The box must be large enough to accommodate some text (11 pixels high and fairly wide)

`Type` is one of four available edit field types specified as a number 1 to 4:

1 Box around the edit field. RETURN key is not allowed.

2 Box around the edit field. RETURN key is allowed.

3 No box around the edit field. RETURN key is not allowed.

4 No box around the edit field. RETURN key is allowed.

`Justify` specifies one of three different kinds of text justification within the edit field:

1 Left justified text. Left

2 Centered text. Center

3 Right justified text Right

Unlike the **INPUT** statement, **EDIT FIELD** does not directly assign the text to a variable. Instead it stores the text internally until requested by the **EDIT$()** function which returns the string of the specified edit field. Edit fields will appear in the window and remain active until closed explicitly with **EDIT FIELD CLOSE** or by closing the window.

Edit fields are used in conjunction with the **DIALOG()** function and usually with dialog event trapping.

EDIT FIELD CLOSE is used to remove the edit field number `edit-id` from the current output window. To close all the edit fields use **EDIT FIELD CLOSE** 0.

To activate an edit field without creating a dialog event, use **EDIT FIELD** `field-id` with no further parameters.

CUSTOM MENUS

The ability to modify or add to existing title-bar menus is one of AC/BASIC's user interface strong points. Custom menus permit any program to take on a professional, tailored appearance, and allow clear presentation of choices available to the user. As with mouse events, menu events may be conveniently trapped or polled, depending on the preferences of the programmer or the importance of execution speed. The following sections discuss the creation and modification of menus and menu items, determination of selected menu items, and menu event trapping.

MENU

Format:

```
MENU
MENU menu-id,item-id,state[,title-string]
MENU RESET
```

Description:

You can use the **MENU** command to rewrite existing menu(s), or add additional menus; AC/BASIC always sets up the menu bar when your program starts. Other commands and functions allow you to determine whether a menu item was selected, or to branch to subroutines when an item is selected.

Although **MENU** may be used to change existing menus, it can not change the default menus. The default `File` and `Edit` menus may be replaced or removed by using:

```
MENU 1,0,0,"" ' Delete File menu
MENU 2,0,0,"" ' Delete Edit menu
```

The default `File` and `Edit` menus may also be removed automatically by using the "M" compiler option described in chapter 4.

MENU with no parameters changes an inversed menu title back to normal.

A menu can be assigned a `menu-id` from 1 to 10, and each menu may have up to 15 items underneath, which are referenced by item-id (1 to 15). Item 0 for each menu is the menu's title.

Each menu item can assume one of three different states: a value of 0 disables the item, of 1 enables the item, and of 2 enables the item and flags it with a check mark. Menu item 0 (the title) can not have a check mark.

To remove a menu, set its title string to null ("") and its state to 0. To reset the menus to their original form with just the File and Edit menus, use the statement **MENU RESET**.

Example:

```
MENU 3,0,1,"Display Mode"
MENU 3,1,2,"AutoDisplay (ON/OFF)"
A$=INPUT$(1)          'Wait for a key press
```

If 0 is specified for the item-id, then the state (0 or 1) affects the entire menu. This allows disabling of an entire menu without having to disable each item separately. It is not possible to flag the menu title with a check mark; thus if the item-id is 0, a state of 2 has no effect.

If the title string is omitted for any item, its current title is not altered.

To change the text in the box which is displayed by selecting the "About..." item in the Apple menu, use the **$ABOUT** metacommand described in Chapter 4.

Note: You *must* execute a **MENU** statement with item-id set to 0 (to initialize the menu title) before working on a menu. If you do not, an "Illegal Function Call" error will occur.

MENU()

Format:

```
MENU(0)
MENU(1)
```

Description:

MENU(0) returns the menu-id and **MENU**(1) returns the item-id of the most recently selected menu item.

The **MENU()** function does not require that a program be compiled with the "N" option; it is only when you use the **ON**

MENU GOSUB statement that a menu event-driven program requires this option.

MENU(0) is reset to 0 after every reference; you can poll the menu bar in the same manner you poll the keyboard with **INKEY$**. If the menu-id of a selection must be accessed more than once, the initial value of **MENU**(0) must be stored in a temporary variable first. **MENU**(1) is not reset with every reference.

These functions do not report on the selection of default menus or their items; only user-defined menus are recognized by this function. The default menus can be overridden by user menus with the **MENU** command described in the previous section. These new menus will then interact with the **MENU**() function.

Example:

```
REM Chapter13/Example6
FOR Menu.Id%=2 TO 9          ' Construct menus 2 - 9
   MENU Menu.Id%,0,1,"MENU"+STR$(Menu.Id%)+" "
   FOR Menu.Item%=1 TO 15      ' Set up menu items 1 - 15
      MENU Menu.Id%,Menu.Item%,1,"Item"+STR$(Menu.Item%)
   NEXT
NEXT
WHILE INKEY$=""              ' Continue until a key is typed
   Menu.Poll%=MENU(0)        ' Poll menu status
   IF Menu.Poll% THEN        ' An item was selected
      LOCATE 1,1
      PRINT "Item" MENU(1) "was selected ";
      PRINT "from menu" Menu.Poll%" "
      MENU                   ' Invert menu title to normal
   END IF
WEND
```

The example polls the menu bar with **MENU**(0) until a selection is made. **MENU**(1) is used to report the item number selected from the menu. Selecting "Quit" will terminate execution.

Menu Event Trapping;

Format:

```
MENU {ON|OFF|STOP}
ON MENU GOSUB {line-number|label}
```

NOTE: any AC/BASIC program containing these statements requires compilation with the "N" option set.

Description:

The **ON MENU GOSUB** statement may be used in conjunction with the **MENU ON/OFF/STOP** commands to set up automatic menu event-trapping, which causes program control to be transferred to a subroutine when a menu item is selected.

ON MENU GOSUB specifies the subroutine `line-number` or `label` to jump to when a menu event has taken place. Any number of **ON MENU** statements may be used in a program; the last one used will determine the subroutine to which control will pass. If the statement **ON MENU GOSUB** 0 is executed, menu event-trapping is disabled. A menu event occurs whenever the user selects an item from one of the menus defined with the **MENU** statement. Within the subroutine, the **MENU**(0) and **MENU**(1) functions may be used to determine which menu and item was selected.

MENU ON enables the trapping of menu events, and must be executed before the **ON MENU GOSUB** statement will have effect. **MENU OFF** disables the trapping of any events. The **MENU STOP** statement causes menu event trapping to cease temporarily; the most recent event will be "remembered", but no action will be taken until a **MENU ON** statement is executed.

The selection of items from the default AC/BASIC menus *does not* cause a menu event.

Example:

```
REM Chapter13/Example7
REM Compile with "N" option
DIM menuitems$(5)
DATA "HELP"
DATA "OPEN"
DATA "CLOSE"
DATA "ABOUT"
DATA "QUIT"
ON MENU GOSUB menuproc
MENU ON
MENU 1,0,1,"MY PROGRAM"  '-- Set up main title
FOR Z = 1 TO 5
   READ menuitems$( Z )  '-- Read in menu items
   MENU 1,Z,1," "+menuitems$( Z ) '-- Build menu
NEXT Z
PRINT "Select an item from menu #1..."
PRINT "Select QUIT to end program"

eventloop:
   GOTO eventloop        '-- This program is event driven
menuproc:                '-- Menu Selection Handler
   menunumber = MENU(0) : itemnumber = MENU(1)
   IF itemnumber = 5 THEN END '-- Menu item was "QUIT"
   PRINT "Selected item number";itemnumber
   IF formeritem <> 0 THEN
      MENU 1,formeritem,1," "+menuitems$(formeritem )
   END IF
   MENU
   MENU 1,itemnumber,2," "+menuitems$(itemnumber )
   formeritem = itemnumber
   RETURN
```

The example sets up menu #1. Each time you select a menu item, it is checked, and all others are set back to normal. The item selected is displayed on the screen. If you select Quit from the menu, then the program will end.

ERROR HANDLING

Run-time Error Trapping

Format:

ON ERROR GOTO `{line-number|label}`

NOTE: any AC/BASIC program containing this statement requires compilation with the "N" option set.

Description:

After the execution of an **ON ERROR** statement, any run-time error causes a branch to the specified `line-number` or `label`, which is usually the beginning of an error-handling routine. The specified `line-number` or `label` *must* be located in the main program. Once control has been passed to the error-handling routine, further error trapping is temporarily disabled; if another program error occurs while inside the error handling routine, AC/BASIC will report the error as it does an untrapped error, with the standard message as shown in Chapter 3.

For execution to continue with error-trapping in effect, the error-handling routine must execute a **RESUME** statement; **RESUME** is to an error-handling routine what **RETURN** is to a subroutine. See the discussion of the various forms of the **RESUME** command later in this chapter.

Error-trapping can be disabled by the statement **ON ERROR GOTO** 0. Note that this statement does not cause the error-handler to pass control to line 0 upon subsequent errors. Errors occurring after trapping has been disabled will cause the error message above to be printed and execution to be terminated.

Example:

```
REM Chapter13/Example8
ON ERROR GOTO errorroutine
zero%=0
infinity%=1/zero%
END

errorroutine:
PRINT "Divide by zero error"
PRINT "Click mouse to end"
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The example will print "Divide by zero error".

ERR and ERL

Format:

```
ERR
ERL
```

Description:

Returns the current error code (**ERR**) and the line number (**ERL**) of the line which caused an error.

This information is useful for writing "intelligent" error handlers that can process different types of errors that occur in different places within a program.

The **ERR** and **ERL** functions are used in conjunction with the **ON ERROR GOTO** statement to trap and handle program errors without system intervention. **ERR** returns the error code associated with the generated error; see Appendix F for a complete list of run-time errors and their codes. If the line at which the error occurred is not labeled, the nearest previous line number encountered is returned. If there are no line numbers at or before the error line, **ERL** returns a 0.

When the "A" compiler option is used to have actual line numbers, the line number returned by **ERL** is the physical line of the source file rather than any line numbers. For more information and an example of the "A" option, refer to Chapter 4.

On program startup, the **ERR** and **ERL** variables are always zero; **ERR** is reset upon each execution of a **RESUME** statement.

Example:

```
   REM Chapter13/Example9
   REM Compile without "A" option and with "N"
10 ON ERROR GOTO error.proc
20 PRINT #1,"string" ' "Bad file number" error
25 WHILE MOUSE(0) = 0 : WEND '-- Wait for click
30 END
40 error.proc:
50 IF ERR=52 THEN PRINT "Bad file number at";ERL
60 RESUME NEXT
```

The example will print:

```
Bad file number at 20
```

RESUME

Format:

```
RESUME
RESUME 0
RESUME NEXT
RESUME {line|label}
```

NOTE: any AC/BASIC program containing these statements requires compilation with the "N" option set.

Description:

The various forms of **RESUME** are used to return from an error-handling routine after the processing of an error is complete.

In a program that uses error-trapping (see the **ON ERROR GOTO** statement above), a **RESUME** must be executed before the error-trapping facility is re-enabled; error-trapping is automatically disabled when an error is generated.

RESUME can continue the program in 3 different ways. **RESUME** or **RESUME** 0 causes execution to resume with the statement that caused the error; **RESUME NEXT** causes execution to continue with the statement immediately after the statement that caused the error; and **RESUME** `line` or **RESUME** `label` causes execution to continue at the specified line or label in the main program. **RESUME**s, as with **GOTO**s and **GOSUB**s, cannot transfer control to line numbers or labels between different subprograms or between the main program and any subprograms.

If a **RESUME** statement is executed and no error has been trapped, a "`RESUME without ERROR`" error results.

Example:

```
    REM Chapter13/Example10
    REM *** Must compile with "C" and "N" option on
 5  DIM a(45)
10  ON ERROR GOTO caught
15  subscript = 47
20  a( subscript ) = 6
25  PRINT "OK"
30  WHILE MOUSE(0) = 0 : WEND
32  END
35  caught:
40  PRINT "Bad Subscript!"
45  subscript = 5
50  RESUME
```

The example will branch to `caught` when line 20 is executed the first time. The error handler will assign 5 to `subscript`, and then resume back to line 20. The line will be executed properly, and then program will print "OK".

ERROR

Format:

```
ERROR integer-expression
```

Description:

Forces the generation of an error with error code of the given `integer-expression`.

The integer expression must be between 0 and 32767. When the **ERROR** statement is executed, the run-time error processor continues as if it had actually encountered the specified error value.

If the value of the `integer-expression` matches one of the AC/BASIC run-time error codes, the error will be processed as if an AC/BASIC error had occurred. The values specified in

Appendix F are either run-time errors or reserved for future implementation.

By defining its own error codes, a program can print error messages specific to an application.

Example:

```
    REM Chapter13/Example11
    ON ERROR GOTO errorroutine
    PRINT "Number > 1000 gives a user-defined error."
  inputline:
    INPUT "Enter a number: ",I
    IF I >= 1000 THEN ERROR 200 ELSE inputline
  errorroutine:
    IF ERR=200 THEN
      PRINT "Value entered is out of range"
    END IF
    RESUME inputline
```

TIMER EVENT TRAPPING

Format:

```
ON TIMER(n) GOSUB {line|label}
TIMER {ON|OFF|STOP}
```

NOTE: any AC/BASIC program containing these statements requires compilation with the "N" option set.

Description:

The **ON TIMER** statement causes the subroutine at the given line or label to be executed every n seconds; n may be any single-precision value between 0 and 86400 (the number of seconds in 24 hours). The specified subroutine *must* be located in the main program. The **TIMER** subroutine interrupts will not begin until a **TIMER ON** statement is executed; **TIMER OFF**, or, equivalently, **ON TIMER GOSUB** 0, halts the generation of **TIMER** interrupts.

TIMER STOP causes timer event trapping to be disabled until a subsequent **TIMER ON** is encountered. The first event after a **TIMER STOP** that would have occurred otherwise will be "remembered" and will be processed immediately after a **TIMER ON** is executed.

```
    REM Chapter13/Example12
    PRINT "CLICK MOUSE TO EXIT"
    TIMER ON
    ON TIMER(1) GOSUB displaytime
    WHILE MOUSE(0) = 0 : WEND : END
  displaytime:
    LOCATE 1
    pixelloc%=WINDOW(2)-8*8 ' 8 chars x 8 pixels/char
    PRINT PTAB(pixelloc%); TIME$;
    RETURN
```

The example prints the Apple's internal time once every second in the upper right part of the screen.

BREAK EVENT TRAPPING

Format:

```
ON BREAK GOSUB {line|label}
BREAK {ON|OFF|STOP}
```

<u>NOTE</u>: any AC/BASIC program containing these statements requires compilation with the "N" option set.

Description:

The **ON BREAK** and **BREAK** statements are used to handle user interrupts, which are generated by the Ó-period (".") key sequence in a program's current active window.

ON BREAK GOSUB causes the subroutine specified by the given line or label to be executed whenever a break event occurs. The **BREAK** subroutine *must* be located in the main program. This statement has no effect until a **BREAK ON** is

executed, which starts break event processing; **BREAK OFF** terminates break event trapping.

BREAK STOP suspends break event trapping until the next **BREAK ON** statement is encountered. If a break event occurs after a **BREAK STOP** has been executed, it will be saved and processed after the next **BREAK ON** is encountered. Only one break event will be saved; the rest are discarded.

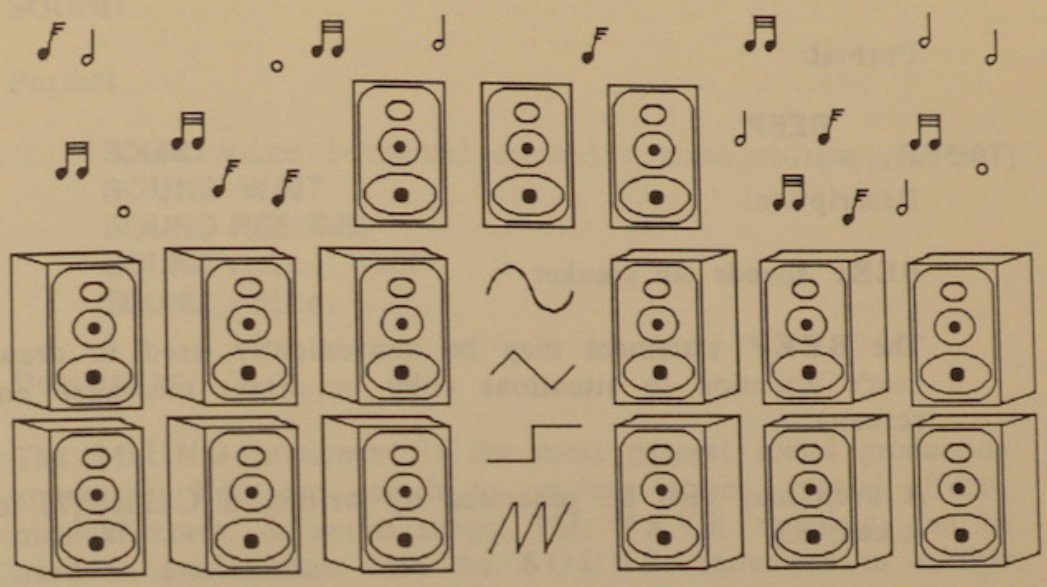
Example:

```
    REM Chapter13/Example13
    REM Compile with "N" option
    ON BREAK GOSUB breaksub
    PRINT "Press or Apple-. to break ..."
    BREAK ON
    breakcount% = 0
Loop:                ' Loop forever.
    GOTO Loop
breaksub:
    LOCATE 2,1
    breakcount% = breakcount% + 1
    PRINT "User break" breakcount% "detected"
    IF breakcount% = 10 THEN STOP
    RETURN
```

The example prints the message when the ⌘ key and the period are pressed at the same time. After the tenth break event, the program exits.

If a break event occurs, and break trapping is not active, the program will terminate.

CHAPTER 14



SOUND PRODUCTION

This chapter discusses the AC/BASIC statements that may be used to produce music and sounds. The **BEEP** statement can be used to give a simple audible warning. The general sound production statements, **SOUND** and **WAVE**, generate sounds of a wide range of frequencies and waveforms, allowing you to play arrangements of musical notes or construct special audio effects.

The Apple IIGS can play up to 15 notes at the same time thanks to the powerful Ensoniq® Digital Oscillating Synthesizer (DOC) chip. AC/BASIC allows up to four different voices with each having their own waveform and instrument definition; each voice may be thought of as one musician in a four-piece band. All 15 notes may be played with the same voice or with a combination of all four voices.

When a note is played with the **SOUND** statement, AC/BASIC requests for a available channel, assigns it the waveform and instrument for the voice, then plays the note.

BEEP

Format:

```
BEEP
```

Description:

BEEP sounds the speaker.

The **BEEP** statement may be conveniently used to draw the user's attention to situations (like important messages on the screen).

A beep may also be generated by writing a **CHR$**(7) to the screen, e.g.

```
PRINT "Beeping ...";CHR$(7);CHR$(7)
```

In this example, if there are no open windows the text will not be printed but the beep sound will still occur.

Example:

```
    REM Chapter14/Example1
  the.loop:
    INPUT "Enter item cost (less than 1.00)"; cost!
    IF cost! >= 1! THEN
      BEEP
      PRINT "Error: Incorrect item cost...try again"
      GOTO the.loop
    END IF
    PRINT "Cost value OK - click mouse to exit."
    WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

The above example shows the use of the **BEEP** statement to draw the user's attention to a data entry error.

MUSIC PRODUCTION STATEMENTS

SOUND

Format:

```
SOUND voice,interval,duration,tone,volume[,REPEAT]
SOUND WAIT
SOUND RESUME
SOUND voice,STOP
SOUND(voice)
```

Description:

The **SOUND** statement is the most general sound production statement. You can use it to produce noises, special effects, musical notes, or entire songs. The **WAVE** statement can be used in conjunction with the **SOUND** statement to further control the sounds.

The **SOUND()** function returns the number of unplayed notes in the sound buffer. It returns 0 if no notes are waiting to be played. If there is a repeat sequence, described below, it returns minus the number of notes in the sequence.

If you have not executed a **SOUND WAIT** statement before a **SOUND** statement, then the sound is played immediately. The **SOUND WAIT** statement will cause sounds to be stored in a list. When a **SOUND RESUME** statement is executed, the stored sounds will be played in order. You must be careful about using the **SOUND WAIT** statement. If you attempt to store too many sounds, you will receive error 159, "SOUND error".

When the **REPEAT** keyword is appended to the **SOUND** statement, the sequence of notes on the line is repeated automatically while other statements are executed until a **SOUND STOP** is executed. More than one note may be specified for a repeat sequence. This small example will make two short beeps once every second on voice 0:

```
SOUND 0,10,5,72,100,50,5,72,100,REPEAT
WHILE INKEY$=""
   PRINT "Like the beeps?"
WEND
SOUND 0,STOP
```

Additional notes can not be added to an existing **REPEAT** sequence; error 159, "SOUND error" is generated in such a case. First use **SOUND STOP** before creating a new **REPEAT** sequence.

A specific sound is generated through the use of five parameters:

`Voice` is a number from 0 to 3 which defines the waveform/instrument combination to use when playing the note. The waveform and instrument are set with the **WAVE** statement. If you are using only one waveform/instrument for all the notes, then the `voice` number is insignificant and may be any of the four possible.

`Interval` is the time in sixtieths of a second before the next note on the same voice should begin. `Duration` is the time, also in sixtieths of a second, the note should be played. To overlap notes, the `interval` value should be less than the `duration` value. To pause between notes, the `interval` should be greater than the `duration`. When `interval` and `duration` are equal, the next note will begin immediately after the first is finished.

For example, the first note will play for one and a half seconds, but after one second the next note begins also:

```
SOUND 0,60,90,69,255
SOUND 0,60,60,72,255
```

The first note plays for one second, then there is a half-second period of silence before the next note begins:

```
SOUND 0,90,60,69,255
SOUND 0,60,60,72,255
```

The first note plays for one second then the next two notes begin immediately thereafter:

```
SOUND 0,60,60,69,255
SOUND 0,0,60,72,255
SOUND 0,0,60,65,255
```

`Tone` is an integer value for the note as shown in the following table. Only integer values may be used. Semi-tones may be produced by using the instrument array. See the **WAVE** statement for more information.

`Volume` may be an integer value from 0 to 255 where 0 means no sound and 255 is the maximum possible volume.

Tone	Note name
0	C
1	C#
2	D
3	D#
4	E
5	F
6	F#
7	G
8	G#
9	A
10	A#
11	B
12	C
13	C#
14	D
15	D#
16	E
17	F
18	F#
19	G
20	G#
21	A
22	A#
23	B
24	C
25	C#
26	D
27	D#
28	E
29	F
30	F#
31	G

Tone	Note name
32	G#
33	A
34	A#
35	B
36	C
37	C#
38	D
39	D#
40	E
41	F
42	F#
43	G
44	G#
45	A
46	A#
47	B
48	C
49	C#
50	D
51	D#
52	E
53	F
54	F#
55	G
56	G#
57	A
58	A#
59	B
60	C (Middle C)
61	C#
62	D
63	D#

Tone	Note name
64	E
65	F
66	F#
67	G
68	G#
69	A
70	A#
71	B
72	C
73	C#
74	D
75	D#
76	E
77	F
78	F#
79	G
80	G#
81	A
82	A#
83	B
84	C
85	C#
86	D
87	D#
88	E
89	F
90	F#
91	G
92	G#
93	A
94	A#
95	B

Tone	Note name
96	C
97	C#
98	D
99	D#
100	E
101	F
102	F#
103	G
104	G#
105	A
106	A#
107	B
108	C
109	C#
110	D
111	D#
112	E
113	F
114	F#
115	G
116	G#
117	A
118	A#
119	B
120	C
121	C#
122	D
123	D#
124	E
125	F
126	F#
127	G

The **SOUND** statement is said to be "asynchronous". That is, after the sound is started, execution of your program continues while the sound is being played. However, sound is stopped when a program ends execution.

The following example randomly constructs inversions of A-major chords, plays them using **SOUND**. Each note plays for 10/60 of a second and sets the interval to 10/60 of a second so notes don't overlap.

```
REM Chapter14/Example2
DEFSNG a
DIM a(10)
a(0)=45          ' A below middle C
a(1)=49          ' C# above low C
a(2)=52          ' E above low C
FOR I=3 TO 10        ' Duplicate
  a(I)=a(I-3)+12     ' in higher
NEXT                 ' octaves
RANDOMIZE TIMER
WHILE MOUSE(0)=0
FOR I%=1 TO 40
  whichchord%=INT(RND*9)+1    ' Pick a note
  SOUND 0,10,10,a(whichchord%),150     ' Construct
  SOUND 1,10,10,a(whichchord%-1),150   ' harmonic
  SOUND 2,10,10,a(whichchord%+1),150   ' major chord
NEXT
dummy=RND(-1)              ' Repeat previous sequence
WEND
```

WAVE

Format:

```
WAVE voice,wave-array[,[size][,[address][,instr-array]]]
WAVE voice GET,instrument-array
```

Description:

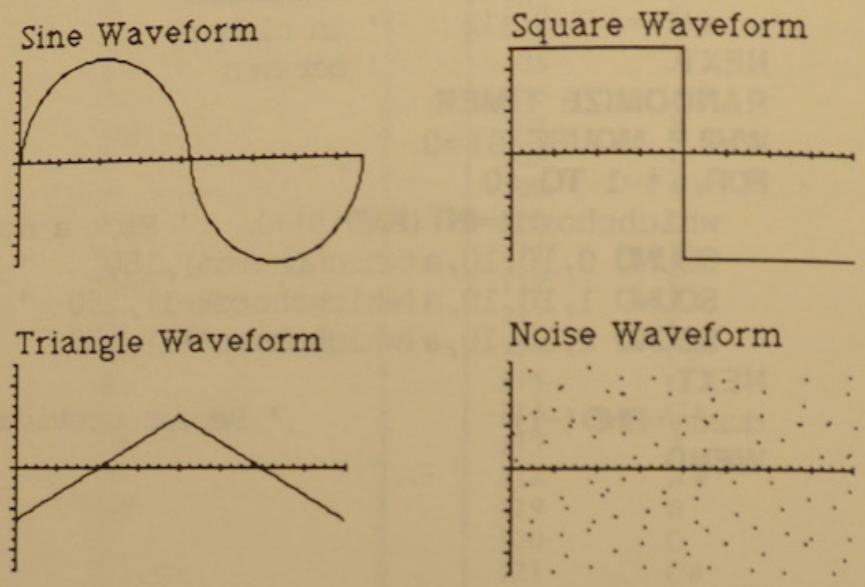
The **WAVE** statement can be used to "tailor" the type of sound you want to produce. Specifically, the **WAVE** statement modifies the shape of the sound wave that is used for a given channel. It can also specify a new instrument to use for the voice. Instruments are described in detail in their own section at the end of this chapter.

`Voice` can be a value 0 to 3. See the explanation for `voice` in the description of **SOUND**.

`Wave-array` is an integer array. It must have at least 256 elements, where each element has a value between 1 and 255,

inclusive. The array must be defined and initialized before executing the **WAVE** statement. To save space, **WAVE** arrays can be erased after the **WAVE** statement has been executed.

In addition, `wave-array` can be specified as **SIN**, **SQR**, or **TRI** to produce sine, square, and triangle waves:



`Size` is the size of `wave-array` in pages where each page is 256 bytes. For example, the `size` for a 512 byte waveform is 2. The default size, if omitted, is 1 for a 256 byte array.

For advanced users of waveforms, the `address` in the DOC memory may be specified for the waveform to be stored. `Address` may be a value from 1 to 255 which specifies a page address in DOC memory. Normally, this argument is omitted and DOC memory allocation is automatic starting at page 1.

`Instr-array` is an array which contains the characteristics of an instrument for the voice. For example, vibrato characteristics and attack and decay times are instrument features. Initially, the instrument for each of the four voices is a 44 byte array defined in the table. It contains far too much information to absorb immediately, therefore, explainations for most of the entries follow.

Default Instrument Array

Offset	Default value	Description
0	127	Envelope 0 breakpoint
1	0	Envelope 0 increment (LSB)
2	127	Envelope 0 increment (MSB)
3	127	Envelope 1 breakpoint
4	0	Envelope 1 increment (LSB)
5	0	Envelope 1 increment (MSB)
6	0	Envelope 2 breakpoint
7	0	Envelope 2 increment (LSB)
8	127	Envelope 2 increment (MSB)
9	0	Envelope 3 breakpoint
10	0	Envelope 3 increment (LSB)
11	0	Envelope 3 increment (MSB)
12	0	Envelope 4 breakpoint
13	0	Envelope 4 increment (LSB)
14	0	Envelope 4 increment (MSB)
15	0	Envelope 5 breakpoint
16	0	Envelope 5 increment (LSB)
17	0	Envelope 5 increment (MSB)
18	0	Envelope 6 breakpoint
19	0	Envelope 6 increment (LSB)
20	0	Envelope 6 increment (MSB)
21	0	Envelope 7 breakpoint
22	0	Envelope 7 increment (LSB)
23	0	Envelope 7 increment (MSB)
24	2	Release envelope number
25	3	Priority decrement
26	1	Pitch bend
27	0	Vibrato depth
28	32	Vibrato speed
29	0	spare (not used)
30	1	A-wave count
31	1	B-wave count

Offset	Default value	Description
32	127	Maximum pitch for instrument
33	automatic	Waveform address (in pages)
34	automatic	Waveform size (in pages)
35	0	DOC mode
36	0	Relative pitch (LSB)
37	0	Relative pitch (MSB)
38	127	Maximum pitch for instrument
39	automatic	Waveform address (in pages)
40	automatic	Waveform size (in pages)
41	0	DOC mode
42	4	Relative pitch (LSB) 4 makes it fuller
43	0	Relative pitch (MSB)

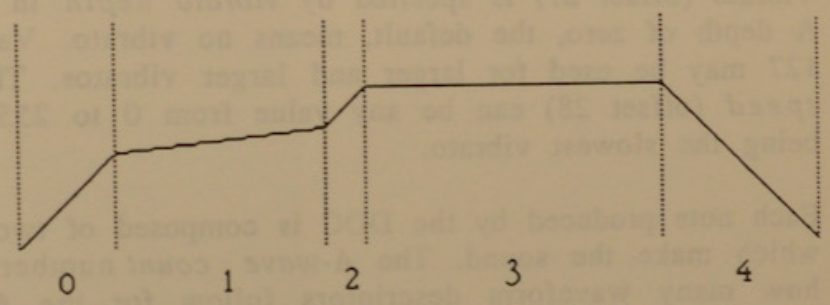
The **GET** form of the **WAVE** statement may be used to put the instrument definition of the specified voice into the array `instrument-array` which must be at least 64 bytes (32 integer elements). `Instrument-array` may be specified as an array element; in this case, the array is filled sequentially starting at the specified element. The `UNPACK` subprogram can be used to convert the byte-oriented instrument array into an easy to use integer form:

```
SUB UNPACK(in%(1), out%(1))
FOR count=0 TO 21
   out%(2*count)=in%(count) MOD 256
   out%(2*count+1)=ABS(INT(in%(count) / 256))
NEXT count
END SUB
```

The `PACK` subprogram does just the opposite; it converts an integer array of 44 elements into a byte-oriented array which can then be used to set an instrument with the **WAVE** statement:

```
SUB PACK(in%(1), out%(1))
FOR count=0 TO 21
   lsb=in%(2*count)
   msb=in%(2*count+1)
   out%(count)=lsb+msb*256
NEXT count
END SUB
```

First, a quick tutorial about envelopes. Every sound the Apple IIGS plays has up to six parts called *envelopes*. Each envelope describes a volume change, or *increment*, for its sixth of the note. The volume increases during an envelope until it reaches the value of the *breakpoint* at which time, the next envelope takes over; this is similar to the funtion of a **FOR...NEXT** loop which steps to an ending value then continues in the program. The breakpoint is the ending volume level (0 to 127) for the envelope. The picture shows how three envelopes can be used in the attack phase, one for sustain, and one for the decay:



If the increment is zero, the volume stays the same or is *sustained*. As described so far, the note can increase in volume and can also stay at the same volume. One of the envelopes must be defined as the *decay* portion of the note which lowers the volume down to the specified breakpoint when the note is turned off.

Let's suppose a note has a fast attack (the rising volume), a sustain portion, and a long decay. When the note is played, you quickly hear the volume rise to the level specified in the **SOUND** statement, then, when the note is finished, the long decay begins. For example, with the statement

```
SOUND 0,60,60,84,150
```

the decay envelope begins after one second. But, the default decay is very short. To make it longer, the instrument must be modified. The following example makes a longer decay:

```
REM Chapter14/Example3
DIM instrument%(31)
WAVE 0 GET,instrument%      ' Get the initial instrument
instrument%(8/2)=1          ' Decay LSB=1, increment=256
WAVE 0,,,,instrument%       ' Change instr. for voice 0
SOUND 1,60,60,84,150        ' Play original instrument
a$=INPUT$(1)
SOUND 0,60,60,84,150        ' Play with longer decay
a$=INPUT$(1)
```

At offset 24 in the instrument array is the *release envelope number* which specifies one of the six envelopes to be used for the decay portion of a note. It can have a value 0 to 5.

Vibrato (offset 27) is specifed by *vibrato depth* in semi-tones. A depth of zero, the default, means no vibrato. Values up to 127 may be used for larger and larger vibratos. The *vibrato speed* (offset 28) can be any value from 0 to 255, with 255 being the slowest vibrato.

Each note produced by the DOC is composed of two oscillators which make the sound. The *A-wave count* number specifies how many waveform descriptors follow for the A-oscillator. Likewise, the *B-wave count* number specifies how many waveform descriptors follow for the B-oscillator. AC/BASIC assumes one waveform for each oscillator, thus, both of these should stay 1.

Offsets 32 to 37 are the descriptor for the A-oscillator. 38 to 43 are for the B-oscillator. Both descriptors are described together.

The *maximum pitch* entry should be 127 for all possible tones to be played by the waveform. If it were 80, for example, all tones 81 and higher would not be played by this oscillator.

The *waveform address* and *waveform size* are specified automatically by the **WAVE** statement. If `size` and `address` were specified in the **WAVE** statement, those values were placed in these two entries. Advanced users can change these

values to address custom waveforms or combinations of waveforms which are already in DOC memory, but be careful since AC/BASIC does not manage the DOC memory. A `size` of 0 specifies 256 bytes. For complete details, refer to the *Apple IIGS Hardware Reference Manual.*

DOC mode can control, for advanced users, the individual oscillators. See the Note Synthesizer section of the *Apple IIGS Toolbox Reference Manual* for details.

The *relative pitch* entries can be used to change the actual pitch relative to the pitch specified in a **SOUND** statement. The LSB (Least Significant Byte) of the relative pitch raises the pitch by 1/256 of a tone. The LSB of the relative pitch of the B-oscillator descriptor is initially 4 (or 4/256) which gives a slightly fuller sound to all notes since the pitches of the two oscillators are slightly off. The MSB (Most Significant Byte) of the relative pitch raises the pitch by whole tones.

CHAPTER 15

SUBPROGRAMS

Subprograms are one of the most versatile additions to the BASIC language, essentially allowing the programmer to create new commands beyond those already available. For example, consider the AC/BASIC statement

```
CIRCLE (100,100),35,0
```

which draws a circle of radius 35 centered at (100,100) in color number 0. If you needed to draw a half circle frequently, you could create a subprogram called HCIRCLE and execute it with a statement which looks as though it were part of the AC/BASIC language:

```
HCIRCLE 100,100,35,0
WHILE MOUSE(0)=0 : WEND          ' Wait for click

SUB HCIRCLE(X%,Y%,Radius%,Col%) STATIC
CIRCLE (X%,Y%),Radius%,Col%,0,3.14159
END SUB
```

This example would draw the same circle but from only angles 0 to 3.14159.

Not only do subprograms have the advantage over subroutines of being able to be passed parameters, but subprograms are also much more transportable than subroutines, since the variables within a subprogram are distinct from any main-program variables of the same names; one can create libraries of useful subprograms and append them to programs, where they can be used with little or no modification. When using subroutines in programs for which they were not originally designed, there is always the concern that temporary or work variables used by the subroutine will alter or corrupt important variables used by the programs.

Lastly, subprograms permit what is known in software design as *functional modularity*: the programmer has the ability to

design sections of code which perform certain specific functions, and which are insulated in a controlled way from other sections of code. A large program can be broken up into smaller tasks, each of which can be performed by a subprogram. For example, here is a sample fragment from a large database management program:

```
Start:
   DIM Personnel$(1000), SSN#(1000), Dept$(15)
   SetFileDefaults DBFile$,TempExt$
   InitOptions Opt$,NumOptions%
   LoadDataBase Personnel$(),SSN#,Depts$() ' Load data
PromptLoop:
   PrintHeader              ' Print version, time, date
   GetCommand UserCmd$      ' Get a command line
   ParseLine UserCmd$,CmdNo%,ErrFlag%       ' Parse it
   .
   .
```

At the same that it makes the construction of large, complex programs more manageable, this design increases readability and facilitates the task of debugging. Any single program block such as the main program or a subprogram can not be larger than 64K, thus, subprograms are useful for breaking extremely large programs into smaller workable modules.

The following sections describe in detail how to define and use subprograms from within an AC/BASIC program.

SUB

Format:

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

Description:

The **SUB** statement marks the beginning of a subprogram definition. Each **SUB** must be matched by an **END SUB** statement, or a "SUB without END SUB" error will occur at compile time. Subprogram definitions may *not* be nested.

With the compiler, all subprograms *must* be at the very end of the main program. If there is any statement other than another subprogram definition after a **SUB / END SUB** block, the error "Statement not in a subprogram block" will be reported at compile-time. The program SORTSUBS on the master AC/BASIC disk has been provided to move all subprogram definitions to the end of a program. SORTSUBS is useful for porting programs from other versions of BASIC which permit subprograms to appear anywhere.

Subprogram names follow the same conventions as label and variable names: any combination of up to 40 characters of letters, digits, or periods ("."), beginning with a letter. No two subprograms may have the same name.

Parameter-list is a list of variable and/or array names, separated by commas, that will receive the values of the corresponding arguments passed to the subprogram when invoked at run-time. The number of parameters that may be declared in or passed to a subprogram is limited only by the maximum length of an AC/BASIC line (255 characters). Arrays in the parameter list must be followed by parentheses, or, optionally, by the number of dimensions (*not* the size of the dimensions) in parentheses:

```
DIM Octopus$(8,3),Sushi#(4,20,5,1)
CALL Pauline(Octopus$(),Sushi#())
.
.
SUB Pauline(Octopus$(),Sushi#(4))
.
.
END SUB
```

Arrays passed as parameters may not be redimensioned or erased within a subprogram.

At run-time, the variables in the parameter list are "aliased" with the argument variables passed from the main program or calling subprogram, unless passed by value (see the **CALL** statement below). This means that any changes to a parameter variable within a subprogram are reflected in the value of the

corresponding argument variable in the subprogram invocation parameter list. An example makes this clear:

```
Var1%=100
Var2%=50
PRINT "Before subprogram call:"; Var1%;Var2%
CHANGEVARS Var1%,(Var2%)          ' Invocation
PRINT "After subprogram call:"; Var1%;Var2%
WHILE MOUSE(0)=0 : WEND      ' Wait for click

SUB CHANGEVARS(X1%,X2%)
X1%=X1%+11
X2%=X2%-6
END SUB
```

When the subprogram CHANGEVARS is invoked, all changes to the variable X1% also alter the value of the main program variable Var1%; since Var2% is parenthesized in the invocation statement, indicating passage by value, it is *not* affected by changes to X2% within the subprogram. Thus, the example would output

```
Before subprogram call: 100 50
After subprogram call: 111 50
```

All variables used *within* a subprogram are local to the subprogram, except those explicitly linked to the main program with the keyword **SHARED**, **SHARED** statements (see **SHARED** in this chapter), or arrays that have been dimensioned in the main program with **DIM SHARED** statements. Thus, in the following example, the input variable I% is totally unrelated to the subprogram variable of the same name used as a loop counter, and the same value will be printed out that is typed in:

```
INPUT I%
CALL CountUp
PRINT "Main program I% is still" I%
WHILE MOUSE(0)=0 : WEND ' Wait for mouse click

SUB CountUp
  FOR I%=1 TO 10
    PRINT "Subprogram I% is" I%
  NEXT I%
END SUB
```

A subprogram may be defined with the keyword **SHARED** which causes *all* variables and arrays in the main program to be accessed and changed from within the subprogram. This is useful for reducing the main program size by placing large sections of code into new subprograms which share all the variables. A subprogram may be declared **SHARED** *and* **STATIC** by placing a space between the two keywords. Any individual **SHARED** statements in a subprogram which is declared with the **SHARED** keyword will generate an error.

If a subprogram is defined as static, by using the **STATIC** keyword, all local variables within the subprogram will retain their values from call to call, whereas non-static subprogram variables are reset to zero or null at each invocation. While any subprogram may invoke another, different subprogram, only non-static subprograms can call themselves. A standard example of a non-static subprogram that calls itself recursively is one that computes the factorial function:

```
Loop:
 INPUT "Compute factorial of"; x%
 IF x% < 0 THEN END
 result%=1
 FACTORIAL x%,result%
 PRINT x% "factorial is" result%
 GOTO Loop

SUB FACTORIAL (number%,result%)
 IF number% > 0 THEN
 result%=result%*number%
 FACTORIAL number%-1,result%
 END IF
END SUB
```

Any AC/BASIC statement is legal within a subprogram, except user function definitions (**DEF FN**), **COMMON**, **ERASE** and **CLEAR** statements, and, since subprogram definitions cannot be nested, the **SUB** and **END SUB** statements.

END SUB

Format:

```
END SUB
```

Description:

The **END SUB** statement ends a subprogram definition begun with a **SUB** statement. The statements that make up the body of the subprogram are placed between the **SUB** and **END SUB** statements. There may be only one **END SUB** for each **SUB**.

When the **END SUB** statement is encountered during execution of a subprogram, control returns, as when a **RETURN** is executed in a subroutine called with **GOSUB**, to the statement immediately following the calling statement. The **EXIT SUB** statement performs this same function, but, unlike the **END SUB** statement, may be used more than once within a subprogram.

EXIT SUB

Format:

EXIT SUB

Description:

Exits the current subprogram and returns control to the statement immediately following the subprogram invocation.

EXIT SUB may be used as often as desired in a subprogram. However, since the statement is essentially a **GOTO** to the end of the subroutine, it is good programming practice to use as few **EXIT SUB**s as possible.

SHARED

Format:

SHARED `variable-list`

Description:

Causes the main-program variables included in `variable-list` to be available to the subprogram containing the **SHARED** statement.

SHARED statements may only appear in subprograms. Subprograms may contain more than one **SHARED** statement. A variable in the main program can be **SHARED** among more than one subprogram, and will not generate an error if declared as **SHARED** more than once within the same subprogram. However, all **SHARED** statements in a subprogram *must* occur before the first executable statement in the subprogram.

The variable list is a list of variable and/or array names, separated by commas. Array names must be followed by parentheses to distinguish them from simple variables, e.g.

```
DIM Cube%(5,5,10)
Filename$="DF1:CubeData"
CALL LoadCube
.
.
SUB LoadCube
 SHARED Filename$,Cube%()
 .
 .
END SUB
```

Normally, variables used in the main program are distinct from those in a subprogram, regardless of whether they have the same names. Through the **SHARED** statement, main-program variables can be accessed and modified directly by a subprogram without being passed as arguments.

Example:

```
REM Chapter15/Example1
main.var=5
CALL mysub
PRINT "main.var =";main.var
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
END

SUB mysub
SHARED main.var ' References to this var are global
main.var=10 ' Alters the value here & in main program
END SUB
```

The output of this program is:

```
main.var = 10
```

If the **SHARED** statement were omitted, `main.var` would retain its original value of 5 after execution of the subroutine call.

CALL

Format:

```
CALL subprogram-name [(argument-list)]
subprogram-name [argument-list]
```

Description:

The **CALL** statement is used to execute AC/BASIC subprograms.

The arguments, if any, passed to the subprogram are enclosed in parentheses, separated by commas, and must correspond in both number and type with the arguments in the subprogram declaration. For example,

```
CALL ScreenMsg("Printing in progress",F$,1)
.
.
SUB ScreenMsg(Msg$,Filename$,BeepFlag%) STATIC
  .
  .   (body of subprogram)
  .
END SUB
```

The **CALL** keyword is not required for subprogram invocation and may be omitted: using this syntax, the parentheses around `argument-list` *must* be omitted. The above **CALL** would then look like this:

```
ScreenMsg "Printing in progress",F$,1
```

Ordinarily, if the values of variables passed to a subprogram are altered within the subprogram, they will retain their changed values. This is called "two-way" passing: the values are passed in when the subprogram is invoked, and are passed back when the subprogram is exited. If you do not wish changes in passed argument variables to be reflected in the calling program, you may surround the variable(s) with parentheses; this is "one-way" passing, or passing "by value." Using the following subprogram **CALL** syntax, the variable F$

will have the same value before and after execution of the subprogram, regardless of any actions performed in the subprogram:

```
ScreenMsg "Printing in progress",(F$),1
```

In certain situations, the **CALL**-omitted invocation syntax may be confused with declaration of, or jump to, a label. Consider the following:

```
CALL MYSUB: newvalue# = oldvalue#
```

This is a line containing two statements; a **CALL** statement and a variable assignment. Using the alternate syntax for a **CALL** statement, the line would be:

```
MYSUB: newvalue# = oldvalue#
```

This presents the problem of deciding whether `MYSUB` is a call statement with no arguments or an alphanumeric label. When this problem occurs, `MYSUB` is considered to be a label. Inserting a blank between `MYSUB` and the colon (":") breaks from the label declaration syntax and resolves the conflict:

```
MYSUB : newvalue# = oldvalue#
```

Similar confusion may occur between labels and subprogram calls in the **THEN** `label` and **ELSE** `label` forms of the **IF** statement: see "**IF...THEN...ELSE**", in Chapter 10 "Control Statements" for the related discussion.

Though similar in function, there are some important differences between the **CALL** and **GOSUB** statements. For one, the **CALL** statement allows explicitly passed parameters. A second difference is that subprograms may be **CALL**ed from within other subprograms, while **GOSUB**s may only be used to execute subroutines local to the main program or to the current subprogram (see "**GOSUB**" in Chapter 10).

Another difference has to do with avoiding accidental execution of subroutines. A subprogram can *only* be executed when explicitly called. For example,

Example:

```
REM Chapter15/Example2
message$ = "Assigned in MAIN"
CALL mysub (message$)
PRINT message$
WHILE MOUSE(0) = 0 : WEND '-- Wait for click

SUB mysub (a$)
a$ = "Assigned in mysub"
END SUB
```

The example program changes the main program variable `message$` to reflect its assignment within the subprogram. Note that, unlike a subroutine of the same function, the subprogram body will *not* be executed following the mouse click.

AC/BASIC Reference Manual

CHAPTER 16

ADVANCED MEMORY AND MACHINE STATEMENTS

This chapter explains the statements and functions which access the Apple IIGS memory directly and the statements which allow the use of machine-language subroutines. A simple variable assignment statement alters the contents of memory, but with the statements in this chapter, any memory location in the Apple IIGS may be referenced.

The commands in this chapter are usually used only by advanced BASIC programmers and should be used with care, especially the **POKE** statement, which is used to overwrite the contents of memory.

PEEK

Format:

PEEK (address)

Description:

Returns the value at memory location address.

The address must be in the range of values representable by 24 bits of data (0 to 2^{24}-1). **PEEK** will return a byte value between -128 and 127.

See the **POKE** statement description for information on how to write bytes to memory addresses.

Example:

```
REM Chapter16/Example1
a%=5            ' Define a%
b%=0            ' Define b%
a.ptr=VARPTR(a%)                ' Get the pointer to a%
b.ptr=VARPTR(b%)                ' and b%
POKE b.ptr,PEEK(a.ptr)          ' Copy contents of a% into b%.
POKE b.ptr+1,PEEK(a.ptr+1)
PRINT a%;b%
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

This is a roundabout method for assigning the contents of a% to b%. Two **PEEK**s are used because the values being moved are integers and thus occupy one word (two bytes) of storage. The **PRINT** statement will display:

```
5 5
```

See also the example for **POKE** below.

POKE

Format:

```
POKE address,value
```

Description:

Writes data directly into the memory location specified by address.

The address is a numeric expression that must be in the range addressable by 24 bits (0 to 2^{24}-1). The allowable range of value is 0 to 255.

As **POKE** is used to write directly to memory, **PEEK** reads data from memory. See the **PEEK** statement description above.

Warning: Writing data directly to memory is potentially dangerous, as it is possible to overwrite system

memory. If this happens, you will need to reset the Apple IIGS. Use the **VARPTR** statement for numeric variable addresses to ensure that the address to which you are **POKE**ing is valid.

Example:

```
A$="Your grade for the course is: "
grade%=ASC("C")
GradeAddr=VARPTR(grade%)
NewCharVal%=ASC("A")      ' ASCII value of "A"
POKE GradeAddr,NewCharVal%
PRINT A$;CHR$(grade)
WHILE MOUSE(0)=0 : WEND ' Wait for mouse click
```

This example uses **VARPTR** to get the location of `Grade` in memory and **POKE**s in a new value in its place. The final result is,

```
Your grade for the course is: A
```

VARPTR

Format:

```
VARPTR(variable)
```

Description:

Returns the address of the specified variable's data.

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to a machine language routine.

VARPTR returns an address that requires four bytes of storage. The only data types that cannot be used to hold an address are short integer (only two bytes) and string (not for numeric data).

The address returned for string arguments references the string descriptor, and consequently, is of little use to BASIC programmers.

CLEAR

Format:

CLEAR [,heap-size][,stack-size]

Description:

The following is a list of the **CLEAR** statement's effect:

1. All numeric variables are set to 0.
2. String space is compacted and all strings are set to null ("").
3. All variables are reset. That is, all **DEFINT**, **DEFSNG**, **DEFDBL**, and **DEFSTR** declarations are cleared.
4. All I/O buffers are flushed and all files are closed.
5. All memory in use by the program is returned to the operating system.
6. The next **DATA** element read will be the first in the program.

The `heap-size` parameter sets the size of the heap space which AC/BASIC uses at run-time to store buffers, strings, and other information. If a value less than 4096 is used, an Out of Memory error will occur.

The `stack-size` parameter is ignored and has been included to maintain compatibility with other versions of BASIC.

Windows, buttons, and edit fields are not affected by **CLEAR**.

CLEAR may not appear in a subprogram.

FRE

Format:

FRE (x)

Description:

FRE() returns information about memory usage. The table shows what information is returned in bytes for a given value of x.

x	Returned value
-1	Size of Heap space
-2	Size of Stack space
-3	Total of free memory
-4	Size of largest free block
-5	Total memory in machine

FRE() with a value other than those in the table returns **FRE**(-1) + **FRE**(-2).

RESET

Format:

RESET

Description:

Writes the contents of all file buffers to the corresponding open files, then closes all open files.

RESET is exactly equivalent to the **CLOSE** statement with no file numbers specified, and has been included for compatibility with older versions of BASIC.

ASSEMBLY-LANGUAGE INTERFACE

The statement **MACHINE** is used to access machine-language subroutines which are stored in binary form in an array. Three additional Unix-style statements are specifically designed to ease the use of loading a machine-language routine from a disk file into an array. An example machine-language program which computes prime numbers using the Sieve of Eratosthenes is given in Appendix D.

MACHINE

Format:

MACHINE `array1[,array2[,array3[,array4[,array5]]]]`

Description:

The position independent machine-language program in `array1` is executed. Information may be passed to and from the machine-language program with the optional arrays `array2` through `array5`.

The addresses of `array2` through `array5` are stored in the 65816 direct-page starting at hexadecimal offset $24.

parameter	offset
array2	$24
array3	$28
array4	$2C
array5	$30

The area of the direct-page between the D register and the stack pointer is available for use. This is typically about 14 256-byte pages.

How should the machine-language program be placed into `array1`? The statement **BLOAD** is designed to load a machine language program into an array; see the **BLOAD** description

below. **READ** statements may also be used to read the program from **DATA** statements.

ARRAY-BASED INPUT AND OUTPUT

Although this section could have been contained in Chapter 11 "Input and Output Statements", these Unix-style I/O statements are designed for interfacing AC/BASIC with machine-language programs. They are also extremely useful for storing and retrieving array-based graphics, waveforms, and instrument definitions.

BLOAD

Format:

```
BLOAD [#]filenumber,array-name,byte-count
BLOAD                                     (function syntax)
```

Description:

The **BLOAD** statement loads the `byte-count` number of bytes into the array `array-name`. A random-access file referenced by `filenumber` must have already been opened before using **BLOAD**. The buffer size for the random-access file can be any value — **BLOAD** ignores it.

The **BLOAD** function returns the number of bytes actually loaded by the previous **BLOAD** statement. The value 0 is returned if the end of file was reached and no data was read from the previous **BLOAD**; reaching end of file does not produce an error.

If `byte-count` is greater than the number of bytes in the array, an error will be generated.

BLOAD can be used to load the header of an Apple object file and retrieve starting address and length information before using **BLOAD** again to load the actual machine-language program. The sieve example in Appendix D shows this procedure.

Array-name may be an array element. In such a case, the array is read sequentially starting with that element.

BLOAD is also useful for reading array-based graphics from disk which were written using **BSAVE**. For example, the following two programs store and retrieve an image:

```
REM Chapter16/Example2
size=SCREEN(0,0,100,10)
DIM Image%(size)
LOCATE 1,1                    ' Draw a picture
PRINT "Some text!"
LINE (0,0)-(100,10),0,B
GET (0,0)-(100,10),Image%
OPEN "Image.File" AS #1
BSAVE #1,Image%,size
CLOSE
```

```
REM Chapter16/Example3
size=SCREEN(0,0,100,10)
DIM Image%(size)
OPEN "Image.File" AS #1
BLOAD #1,Image%,size
CLOSE
PUT (200,50),Image%
WHILE MOUSE(0) = 0 : WEND '-- Wait for click
```

BSAVE

Format:

```
BSAVE [#]filenumber,array-name,byte-count
```

Description:

The **BSAVE** statement writes byte-count number of bytes from the array array-name to the random-access file referenced by filenumber. A random-access file referenced by filenumber must have already been opened before using

BSAVE. The buffer size for the random-access file can be any value — **BSAVE** ignores it.

If `byte-count` is greater than the number of bytes in the array, an error will be generated.

Any array-based information such as array-based graphic images, waveforms, and instrument definitions may be written using **BSAVE** for later retrieval with **BLOAD**. `Array-name` may be an array element. In such a case, the array is written sequentially starting with that element.

This program copies a file by using 32K array, thus resulting in a fast copy operation for large files.

```
REM Chapter16/Example4
size=32767              ' Largest array size
DIM store%(size/2)      ' Integers take two bytes
source$=FILES$(1)           ' Get source file name
destination$=FILES$(0)      ' Get destination file name
OPEN source$ AS #1
OPEN destination$ AS #2
l=1
WHILE l<>0
   BLOAD #1,store%,size
   l=BLOAD
   BSAVE #2,store%,l
WEND
CLOSE
```

BSEEK

Format:

```
BSEEK([#]filenumber,offset[,option])
```

Description:

Positions to `offset` number of bytes from the position defined by `option` within the file referenced by `filenumber`. The

possible values of `option` are in the table. If omitted, `option` defaults to 0.

option	meaning
0	Position from start of file
1	Position from current file position
2	Position from end of file

The value returned from **BSEEK()** is the file position in bytes from the file start after the seek.

BSEEK() is useful for positioning within an Apple object file to the beginning of a code or data segment. It may also be used to easily position within a file of many array-based graphic images.

SWAP BYTES

Format:

SWAP BYTES, `array-name[,n]`

Description:

Byte swaps `n` words of the array `array-name`. This may be useful since the Apple IIGS microprocessor, the 65816, is byte-swapped.

`Array-name` may also be an array element from which the swapping will begin. If `n` is omitted, the bytes will be swapped up to the end of the array.

The primary purpose of **SWAP BYTES** is to make instrument and ROM call array data easier to read in **DATA** statements. For example, the following program fragment reads a pattern which is two box outlines:

```
DEFINT a-z
DIM pattern(x)
FOR x=0 TO 15
   READ a
   pattern(x)=a : pattern(x+8)=a
NEXT x

DATA &H0000, &H0000
DATA &HFF0F, &HF0FF
DATA &HFF0F, &HF0FF
DATA &H0000, &H0000
```

The shape of the pattern is easier to see when the data is stored in a byte-swapped format:

```
DEFINT a-z
DIM pattern(x)
FOR x=0 TO 15
   READ a
   pattern(x)=a : pattern(x+8)=a
NEXT x
SWAP BYTES,pattern

DATA &H0000, &H0000
DATA &H0FFF, &HFFF0
DATA &H0FFF, &HFFF0
DATA &H0000, &H0000
```

CHAPTER 17

EDITOR

This chapter covers most of the editor features. There are small examples for you to try functions as they are introduced. Topics covered in this chapter are:

- Editor modes.
- Entering text.
- Moving through a file.
- Modifying text.
- Search / search and replacement.
- Cut, copy, paste.

The editor has over fifty commands. Many of the features will not be covered now. When you complete this chapter, you may want to look through Chapter 18, which covers *all* of the editor features.

Starting The Editor

In order to work through all of the commands covered in this chapter, a file with a fair amount of text will be needed so that you can see how the commands work. For this purpose, we will use the example "Hello" program which you compiled and executed in Chapter 2. When it comes time to exit the editor, don't save the changes! The last section of this chapter will guide you through closing the file.

Let's get started. Go ahead and start up the editor from the `Edit` item in the `File` menu of the compiler. Select the `Hello` program from the standard file selector.

Help

At anytime, you can press ⌂?, which causes the editor to display a help file. This file has a brief list of the commands available in the editor. You can use the cursor movement commands, like the arrow keys, to scroll through the help file. RETURN or ESC returns you to the file you are editing.

Editor Modes

With the editor running, you should see an eighty-column screen with text displayed on twenty-two lines. At the bottom of the screen is a reverse-video banner containing the current editor mode. You should see the message `Mode: Edit`. The editor has several modes which, when activated, will allow you to use different editing features. There are five other modes in addition to the edit mode In this chapter we will only cover four modes: edit, escape, insert, and select. The features associated with these modes will be introduced later in this chapter. The two modes not discussed in this chapter are auto indent and word wrap. These are covered in Chapter 18.

Entering Text

The edit mode is where you will do most of the text entry. Also, most of the commands are active in the edit mode. Having just started the editor, the cursor is positioned at the start of line one. Go ahead and enter a few words. As you enter text, the cursor will over-strike any characters it encounters. When you get to the end of the line, the screen will scroll to the left to show you more text to the right.

Moving Through A File

There are many ways to move through a file. We should classify the range of movement in order to present the commands in a rational order. Let's say that:

Small movement	-	a few characters or lines
Medium movement	-	moving the range of the display screen (a page)
Large movements	-	from the start and end of the file

Let's start small and work up.

Small Movements

Press RETURN to move the cursor to the start of a line. Now press the TAB key. Notice the column position of the cursor. The tab stops are set at every eight columns. By pressing ⌂A, you can move back a tab stop. You can change the stops to best suit you. See Chapter 18.

The arrow keys are great for moving a few characters to the right and left, or moving between a couple of lines up and down. These keys are really easy to use: the direction of cursor movement is on the key. Hold an arrow key down for automatic repeat. Go ahead and experiment with these keys. You could use these keys to move anywhere in the file. But you don't have to – there are better ways!

Two commands let you move to the start or end of a line. To move the cursor to the end of a line, use the ⌂>, the end-of-line command. To move the cursor back to the start of a line, enter the ⌂<, the start-of-line command.

Medium Movement

In this range of movement, you will learn how to move to the top and bottom of a page. As stated earlier, a page is the amount of text visible on your display screen. There are two commands which will let you move the cursor to the top and bottom of the page.

The top of page command, ⌂↑, will move the cursor to the first line of the page. Go on and give it a try. Notice that the cursor remained in the column it started in. To move the cursor to the bottom of a page, use ⌂↓. This command will move the cursor to the last line of the page, the cursor again remaining in the column it started in. These two commands cause the cursor to retain its horizontal position. There is an extra feature associated with these two commands. Use the ⌂↓ command to position the cursor on the bottom line of a page. Now use the ⌂↓ command a second time. Examine the line count. It has increased by twenty-two lines! What happened is that use of ⌂↓ when the cursor is on the last line of a page will cause the next page to be displayed, and the cursor to be moved to the last line on that page (while preserving the horizontal position of the cursor). Position the cursor on line one of the current screen and try the ⌂↑ command. It works in the same way.

Large Movements

As files become larger, you will find the commands which perform large movements convenient. There are two methods to span a large distance in the file.

The first method is called scrolling. Scrolling is by a line or page. Let's start with line scrolling. To use the line scroll command, you need to be in escape mode. By the name, you can see it involves the escape key ESC. You enter escape mode by pressing ESC. You can exit the mode by pressing ESC a second time. This is called toggling the mode. When an escape mode command is used in this manual, we will assume that you will return to the edit mode when you finish the command. Back to the line scroll command. Go ahead and enter the command ESC⌂P. Did you notice what happened? The screen scrolled up one line. "Big deal," you say. "I could use the RETURN key for that." Well, here is where escape mode commands become interesting. Integrated with every escape mode command is the repeat count feature. Try the command ESC10⌂P. The cursor moved down ten lines! You can specify

any integer in the range 1 to 32767 in any escape mode command. Try it with the scroll up command, ESC10⏢O.

There are also commands that scroll one page without changing the position of the cursor. Try the command to scroll down one page, ⏢]. It has increased the line count by twenty-two lines. Let's try something completely different. This file we have been using is not long so let's try to scroll past the end of the file. If we give the command to scroll 300 pages, it would be past the end of the file. Go ahead and enter ESC 300⏢] and watch what happens. The editor would let us scroll past the end of the file. Now try scrolling past the start of the file. Enter the command ESC500⏢[and notice what happens. In this case the editor stopped us from going past the begining of the file, and placed the cursor on the first line of the file.

Another command for large movement allows you to move between eight even intervals in the file. The editor has divided our file into eight approximately equal sections. Using the screen move commands, you can move to the boundaries between these sections. Let's see how these commands work. Go ahead and enter the command ⏢1, and then the command ⏢9. The cursor first jumped to the first character in the file, and then jumped to the last character in the file. These two positions are the boundaries before the first section and after the eighth section. Try the command sequence ⏢2 through ⏢8 in order to get a feel for these commands. Notice the cursor position after each command. Also, note that these commands are not available in the ESC mode.

Modifying Text

Most of the work that you will do in the editor will be changing existing files. There are several methods available to you. The most often used features are deleting and inserting text. Using these two features, any type of modification to a file could be made.

Deleting Text

There several commands for deleting text. The amount of text you need to delete will determine the command you use. Let's go through some of the delete commands available to you, starting with minor deletions of a character or two and working up to many lines. Not all of the delete commands will be covered in this chapter. Read Chapter 18 for additional delete commands.

You may have already used the first delete command, which is the the delete key, or DELETE. When you press this key in the edit mode, the cursor will move one column to the left, removing the character which occupied the position to the left of the cursor. Similar to the delete command is the command to delete a character at the cursor. This command is ⌂F. When this command is given, the character covered by the cursor is deleted and the remainder of the line is moved one position to the left. Go ahead and try these commands.

The two commands covered above are fine for deleting a few characters; it would be nice to be able to delete an entire line. This next command deletes text from the cursor to the end of the line. Choose a line in the file which contains some characters. Position the cursor about half way in the line and press the command ⌂Y. The command caused all of the text from the cursor to the end of the line to be deleted. Now, position the cursor on the first position of the line and repeat the command. What happens? The editor deletes all of the characters from the line.

If you want to delete more than one line, use the delete line command. Go ahead and position the cursor at the start of the file. Enter the command ⌂T. Only one line is removed from the file. The command is also available from escape mode, where you may enter a repeat count number and remove as many lines as needed. Try this command out. Be sure and leave some of the file for the next sections!

Inserting Text

Our first insert command will allow us to insert spaces. Place the cursor between two characters and enter the command ⌂SPACE. A space will now separate the two characters. You could, of course, use the automatic repeat count in the escape mode and enter as many spaces as needed.

The second insert command will allow us to insert blank lines. To use this command, place the cursor on the line below the position where the new lines are expected. For example, if you wanted the new lines to appear after line one, you would position the cursor in line two. Now enter the command ⌂B. There is your blank line. Of course, you can us repeat counts to insert more blank lines.

One of the most common uses of the line insert capability is to make room for large additions to source files. The cursor is placed appropriately, then a few hundred lines are inserted, followed by a stream of text. There will undoubtedly be some extra lines left over. Typing ⌂R removes these lines, beginning at the cursor position and continuing to the first non-blank line. Let's give this a try. Insert fifty blank lines. Now move the cursor down a couple of lines, and type ⌂R.

The last method of inserting is using the insert mode. By using the insert mode, you can position the cursor at any position in the file and insert extra characters. Go ahead and position the cursor between some characters. Toggle insert mode by typing ⌂E. Now that you are in the insert mode, you can enter as much new text as needed. Exit insert mode by typing ⌂E.

ADVANCED FEATURES

Search / Search With Replace

These features can be a big time saver for you. Using these commands, you can have the editor search any portion of a file for a character string that you specify. You can even have the

editor replace the character string with another character string that you specify. For example, if you misspell the word "label" as "lable" throughout your file, you can have the editor replace the misspelled version with the correct version.

Search

This command is used to locate a sequence of characters specified by you. For example, let's suppose that you wished to locate every occurrence of the word PRINT in our file. First, position the cursor on the first character of the file. Now, enter the search command ⌂L. There are really two search commands. The one you just entered will search from the cursor's position to the end of the file. The second command, ⌂K, will search from the cursor's position to the start of the file. After you enter the search command, the prompt Search String: will appear in the banner. This is where you enter the character string you want the editor to locate. Should you make a mistake entering the search string, there are some line editing commands you can use to correct the mistake. Just about all of the character-oriented commands, such as the right and left arrow keys, are active, while the line-oriented commands, such as scroll down, are not. If you are interested in the details, see Chapter 18. Once you are satisfied with the search string, press RETURN. The editor will search for the first occurrence of PRINT. If the editor cannot locate a search string, it will respond with the message String Not Found. If the editor does locate the search string, the cursor will be positioned on the first character of the target string. To continue searching the file for the same string, use the editor macro OPTIONL. Editor macros are covered in detail in Chapter 18.

Search With Replacement

This feature is an extension to the search feature. It operates in an identical manner up to the point of entering the search string. The command for search down with replacement is ⌂J. The command for search up with replacement is ⌂H. Once you

enter the search string, this command will issue a prompt `Replace String:`. Enter your name, so that you can verify the replacement works. The same commands for editing the replacement string as mentioned in search will also work here. Once you press RETURN, a third prompt will be displayed: `Auto or Manual (A M Q)?` This prompt is asking you how you want the replacement made. Auto means the replacement should be made to every occurrence the editor encounters. Manual means you want the editor to locate each occurrence and to prompt you before the replacement is made. If you chose M, each time the search string is found, the prompt `Replace (Y N Q)?` is displayed. Y means make the replacement and continue search. N means to skip this replacement, but to continue the search. Q means to quit the search.

Cut, Copy, and Paste

These features can be used separately or combined. For example, if you have typed a section of program in your file, but you have typed the program block in the wrong place, you can use the cut commands to first cut (or remove) the block from the file. Then paste (or insert) the removed block at the desired location. As a second example, suppose you had a block of code which was to be used in several locations throughout the program. You could type every duplicate portion, or you could use the copy feature to create a copy of the duplicate block, then use the paste command to insert the duplicate block wherever it was needed.

Cut

When the cut command is issued, the editor will go into line-oriented select mode automatically. If you are interested in more information about this mode, see Chapter 18.

Let's try out the cut command. Enter the cut command, ⌂X. The editor goes into select (line-oriented) mode. You may use any cursor positioning commands we have talked about so far to move the cursor. Notice that as the cursor moves, the lines which the cursor moves over will be highlighted in reverse-

video. These are the selected lines. Go ahead and select a few lines. When you have finished selecting the text, press RETURN. The selected information is then copied into a file in the current folder called SYSTEMP. If you want to cancel the cut, press ESC instead of RETURN. The selected text is waiting in the SYSTEMP file, and can be put back into the file using the paste command (covered later in this section) or left as is.

Copy

The copy feature is similar to cut. Instead of removing the selected text, the editor will make a copy of the selected text and put it in the SYSTEMP file. Go ahead and enter the the copy command, ⌘C. The editor is in select mode. Select a few lines and press RETURN. The editor has placed a copy of the selected text in the SYSTEMP file. Notice that the use of the copy command will replace the selected text from the cut we just performed.

Paste

This feature is used to enter the text selected with the cut and copy commands back into our file. Try this command out. Position the cursor someplace in the file where it will be obvious that the selected text will be added, then press the paste command, ⌘V. The selected lines will then be inserted above the cursor.

Closing The File

When you are finished editing your text file, the command to exit the editor is ⌘Q. This will cause the editor to display a menu and prompt you as to the status of the file:

```
<1>  Save and continue editing.

<2>  Save and return to compiler.

<3>  Save and compile.

<4>  Abort edit and return to compiler.
```

In this sample session, you don't want to save the changes to the `Hello` file, so enter 4. If you were editing a program, you would want to save your changes. There are several options when you save a file. By choosing option 1, you replace the old version of the file on the disk with the new version, then return to the *editor*. Option 2, also replaces the old version of the file on the disk with the new version, then return to the *compiler*. Option 3 saves and returns to the compiler, but it compiles the file immediately using the current compiler options. The last option, number 4, does *not* save the file and returns to the compiler.

Chapter Summary

This chapter covered the most commonly used commands in the the editor. You can get by with most of your typing chores with what was covered in this chapter.

Having just completed this chapter, you are probably not very comfortable using all the editor commands. To get comfortable with the commands, you will need to use the editor. There are more commands available, as well as a complete keystroke macro facility. Once you are comfortable with the basic editing features, refer to Chapter 18 for more information about the editor.

CHAPTER 18

EDITOR REFERENCE

The editor allows you to write and edit source and text files. A brief introduction to the use of the editor was given in Chapter 17. This chapter provides reference material on the editor, including detailed descriptions of all editing commands.

The first section in this chapter, "Modes," describes the different modes in which the editor can operate. The second section, "Macros," describes how to create and use editor macros, which allow you to execute a string of editor commands with a single keystroke. The third section, "Commands," describes each editor command and gives the key or key combination assigned to the command. The fourth section, "Setting Editor Defaults," describes how to set the defaults for editor modes and tab settings.

MODES

The behavior of the editor depends on the settings of several modes, as follows:

- Insert.
- Escape.
- Auto Indent.
- Text Selection.
- Word Wrap.

Each of these modes has two possible states; you can toggle between the states while in the editor. All of these modes are described in this section.

Insert

When you first start the editor, it is in overstrike mode; in this mode the characters you type replace any characters the cursor is on. In insert mode, any characters you type are inserted at the left of the cursor; the character the cursor is on and any characters to the right of the cursor are moved to the right.

The maximum length of a line in the editor is 255 characters (including spaces); however, the editor can display only 80 characters of the line. If you continue to insert characters in a line after the line is 255 characters long, the new characters overstrike the old characters in the line. If the editor is both in insert and automatic-wrap modes, then when the characters being inserted reach the end-of-line marker (usually at column 255, see the section "Setting Editor Defaults" in this chapter), the editor inserts a carriage return before the word you are currently typing. The result is that the word that included column 255 and all remaining characters on the line (up to the 255th character) are moved to the next line down. See the section "Automatic Wrap" in this chapter for an example.

To enter or leave the insert mode, type ⌂E.

Escape

When you press the ESC key, the editor enters the escape mode. For the most part, the escape mode works like the normal edit mode. The principle difference is that the number keys allows you to enter repeat counts, rather than entering numbers into the file. After entering a repeat count, a command will execute that number of times.

For example, the ⌂B command inserts a blank line in the file. If you would like to enter fifty blank lines, you would enter the escape mode, type 50⌂B, and leave the escape mode by typing the ESC key a second time.

Earlier, it was mentioned that the number keys were used in escape mode to enter repeat counts. In the normal editor mode,

Ó followed by a number key moves to various places in the file. In escape mode, the Ó key modifier allows you to type numbers.

The only other difference between the two modes is the way CTRL_ works. This key is used primarily in macros. If you are in the editor mode, CTRL_ places you in escape mode. If you are in escape mode, it does nothing. In edit mode, ÓCTRL_ does nothing; in escape mode, it returns you to edit mode. This lets you quickly get into the mode you need to be in at the start of an editor macro, regardless of the mode you are in when the macro is executed.

The remainder of this chapter describes the standard edit mode.

Auto Indent

You can set the editor so that RETURN moves the cursor to the first column of the next line, or so that it follows indentations already set in the text. If the editor is set to put the cursor on column 1 when you press RETURN, then changing this mode causes the editor to put the cursor on the first nonspace character in the next line; if the line is blank, then the cursor is placed under the first non-space character in the first nonblank line above the cursor.

To change the return mode, type ÓRETURN.

Select Text

The Cut, Copy, and Delete commands require that you first select a block of text. The editor has two modes for selecting text: line-oriented and character-oriented selects. As you move the cursor in line-oriented select mode, text or code is marked a line at a time. In the character-oriented select mode, you can start and end the marked block at any character. Line-oriented select mode is the default.

While in either select mode, the following cursor-movement commands are active:

- bottom of screen
- top of screen
- cursor down
- cursor up
- start of line
- screen moves

In addition, while in character-oriented select mode, the following cursor-movement commands are active:

- cursor left
- cursor right
- end of line
- tab
- tab left
- word right
- word left

As you move the cursor, the text between the original cursor position and the final cursor position is marked (in inverse characters). Press RETURN to complete the selection of text. Press ESC to abort the operation, leave select mode, and return to normal editing.

To switch between character- and line-oriented selection while in the editor, type CTRLOX.

Automatic Wrap

In non-wrap mode, when you reach the end-of-line mark (at column 255 - see the section "Setting Editor Defaults" in this chapter), any additional characters you type overwrite the last character on the line. In automatic-wrap mode, when you type one character too many to fit on the line, the entire word that that character is part of is wrapped to the next line. For example, suppose you are typing the word pneumatolysis, and the letter *t* falls on column 255. In non-wrap mode, the additional characters overwrite the last character on the line and the line ends with pneumas; in automatic-wrap mode, the entire word pneumatolysis is moved to the beginning of the next line. Normally, when editing BASIC programs you will want to use the non-wrap mode.

The editor does not have "soft" carriage returns; that is, once a line is broken by the automatic wrap feature, there is a permanent carriage return at the end of the line. If you delete characters on the first line, the continuation line does *not* move back up to maintain the length of the first line. To remove the carriage return you must enter insert mode, then move the cursor to the beginning of the second line and type the DELETE key.

If the editor is both in insert and automatic-wrap modes, then when the characters being inserted reach the end of the line, the editor inserts a carriage return before the word you are currently typing. The result is that the word that included column 255 is moved to the next line down.

Note that the line does *not* wrap when the last character in the line reaches column 255, but when the characters *being inserted* reach column 255.

The effect of inserting characters is the same whether you type them in or paste them in.

MACROS

You can define up to 26 macros for the editor, one for each letter on the keyboard. A macro allows you to substitute a

single keystroke for up to 128 predefined keystrokes. A macro can contain both editor commands and text, and can call other macros.

To create a macro, press ⌂ESC. The current macro definitions for A to J appear on the screen. The ← and → keys can be used to switch between the three pages of macro definitions. To replace a definition, press the key that corresponds to that macro, then type in the new macro definition. You must be able to see a macro to replace it — use the left and right arrow keys to get the correct page. Press OPTION ESC to terminate the macro definition. You can include CTRL*key* combinations, ⌂*key* combinations, OPTION*key* combinations, and the RETURN, ENTER, ESC, and arrow keys. The following conventions are used to display keystrokes in macros:

CTRL*key*	The uppercase character *key* is shown in inverse.
⌂*key*	An inverse A followed by *key*
OPTION*key*	An inverse B followed by *key* (for example, BK)
ESC	An inverse left bracket (CTRL [).
RETURN	An inverse M (CTRL M).
ENTER	An inverse J (CTRL J).
↑	An inverse K (CTRL K).
↓	An inverse J (CTRL J).
←	An inverse H (CTRL H).
→	An inverse U (CTRL U).
DELETE	A block

Each ⌥*key* combination or OPTION*key* combination counts as two keystrokes in a macro definition. Although an ⌥*key* combination looks (in the macro definition) like a CTRL A followed by *key*, and an OPTION*key* combination looks like a CTRL B followed by *key*, you cannot enter CTRL A when you want an ⌥ or CTRL B when you want an OPTION key.

If you make a mistake typing a macro definition, you can back up with OPTION DELETE. If you wish to retype the macro definition, press OPTION ESC to terminate the definition, press the letter key for the macro you want to define, and begin over. When you are finished entering macros, press OPTION ESC to terminate the last option definition, then press OPTION to end macro entry. The following prompt appears on the screen:

```
Write macros to disk?
```

Press Y to save the new macro definitions on disk. Press N to return to the file without saving the macros. Macros are saved on disk in the file SYSEMAC in the current folder.

To execute a macro, hold down OPTION and press the key corresponding to that macro.

COMMAND DESCRIPTIONS

This section describes the functions that can be performed with editor commands. The key assignments for each command are shown with the command description.

Screen-movement descriptions in this manual are based on the direction the display screen moves through the file, not the direction the lines appear to move on the screen. For example, if a command description says that the screen scrolls down one line, it means that the lines on the screen move *up* one line, and the next line in the file becomes the bottom line on the screen.

CTRLG **Beep the Speaker**

The ASCII control character BEL ($07) is sent to the output device. Normally, this causes the speaker to beep.

⇧, or ⇧< **Beginning of Line**

The cursor is placed in column one of the line it is in.

⇧↓ **Bottom of Screen / Page Down**

The cursor moves to the last visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the bottom of the screen, the screen scrolls down twenty-two lines.

CTRLC or ⇧C **Copy**

When you execute the Copy command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the current folder. (To cancel the Copy operation without writing the block to SYSTEMP, press ESC instead of RETURN.) Use the Paste command to place the copied material at another position in the file.

↓ **Cursor Down**

The cursor is moved down one line, preserving its horizontal position. If it is on the last line of the screen, the screen scrolls down one line.

← Cursor Left

The cursor is moved left one column. If it is in column one, the command is ignored.

→ Cursor Right

The cursor is moved right one column. If it is on the end-of-line marker (usually column 255), the command is ignored.

↑ Cursor Up

The cursor is moved up one line, preserving its horizontal position. If it is on the first line of the screen, the screen scrolls up one line. If the cursor is on the first line of the file, the command is ignored.

CTRLX or ⌂X Cut

When you execute the Cut command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the current folder and deleted from the file. (To cancel the Cut operation without cutting the block from the file, press ESC instead of RETURN). Use the Paste command to place the cut text at another location in the file.

⌂ESC Define Macros

The editor enters the macro definition mode. Press OPTION ESC to terminate a definition, and OPTION to terminate macro definition mode. The macro definition process is described in the section "Macros" in this chapter.

⌂DELETE — Delete

When you execute the delete command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. Use any of the cursor movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is deleted from the file. (To cancel the delete operation without deleting the block from the file, press ESC instead of RETURN.)

CTRLF or ⌂F — Delete Character

The character that the cursor is on is deleted and put in the Undo buffer (see the description of the Undo command). Characters to the right of the cursor are moved one space to the left to fill in the gap. The last column on the line is replaced by a space.

DELETE or CTRLD — Delete Character Left

The character to the left of the cursor is deleted, and the character that the cursor is on, as well as the rest of the line to the right of the cursor, are moved 1 space to the left to fill in the gap. If the cursor is in column one and the overstrike mode is active, no action is taken. If the cursor is in column one and the insert mode is active, then the line the cursor is on is appended to the line above and the cursor remains on the character it was on before the delete. Deleted characters are put in the undo buffer.

⌂T or CTRLT — Delete Line

The line that the cursor is on is deleted, and the following lines are moved up one line to fill in the space. The deleted line is put in the Undo buffer (see the description of the Undo command).

CTRLY or ⌂Y **Delete to EOL**

The character that the cursor is on, and all those to the right of the cursor to the end of the line, are deleted and put in the Undo buffer (see the description of the Undo command).

⌂W **Delete Word**

When you execute the delete word command, the cursor is moved to the beginning of the word it is on, then delete character commands are executed for as long as the cursor is on a nonspace character, then for as long as the cursor is on a space. This command thus deletes the word plus all spaces up to the beginning of the next word. If the cursor is on a space, that space and all following spaces are deleted, up to the start of the next word. All deleted characters, including spaces, are put in the Undo buffer (see the description of the Undo command).

⌂. or ⌂> **End of Line**

If the last column on the line is not blank, the cursor moves to the last column. If the last column is blank, then the cursor moves to the right of the last nonspace character in the line. If the entire line is blank, the cursor is placed in column 1.

⌂? or ⌂/ **Help**

Displays the help file, which contains a short summary of editor commands. Use ESC to return to the file being edited.

The help file is a text file called SYSHELP, found in the BASIC.OVERLAYS folder on the original compiler disk. Since it is a text file, you can modify it as desired.

⌂B or CTRLB — Insert Line

A blank line is inserted at the cursor position, and the line the cursor was on and the lines below it are scrolled down to make room. The cursor remains in the same horizontal position on the screen.

⌂SPACEBAR — Insert Space

A space is inserted at the cursor position. Characters from the cursor to the end of the line are moved right to make room. Any character in column 255 on the line is lost. The cursor remains in the same position on the screen. Note that the Insert Space command can extend a line past the end-of-line marker.

CTRLV or ⌂V — Paste

The contents of the `SYSTEMP` file are copied to the current cursor position. If the editor is in line-oriented select mode, the line the cursor is on and all subsequent lines are moved down to make room for the new material. If the editor is in character-oriented select mode, the material is copied at the cursor column. If enough characters are inserted to make the line longer than 255 characters, the excess characters are lost.

CTRLQ or ⌂Q — Quit

Exit to the editor Quit menu, which gives you the following options:

1. `Save and continue editing` — The file is saved to disk and you are returned to the same position in the file you were at when you quit it, in the same editing mode.

2. `Save and return to compiler` — Saves the file to the disk and returns you back to the compiler.

3 Save and compile — Saves the file to the disk and returns you back to the compiler at which time the file is compiled with the current compiler options.

4 Abort edit and return to compiler — If you have not yet saved the changes to the file you just aborted, you are asked to verify that you want to abort the editor without saving changes. You are returned to the compiler.

If you press RETURN without entering any other data in response to a prompt, the command is aborted and control returns to the menu.

CTRLR or ⌂R — Remove Blanks

If the cursor is on a blank line, that line and all subsequent blank lines up to the next non-blank line are removed. If the cursor is not on a blank line, the command is ignored.

1 to 32767 — Repeat Count

When in escape mode, you can enter a *repeat count* (any number from 1 to 32767) immediately before a command, and the command is repeated as many times as you specify (or as many times as is possible, whichever comes first). Escape mode is described in the section "Modes" in this chapter.

RETURN — Return

The RETURN key works in one of two ways, depending on the setting of the auto-indent mode toggle: 1) to move the cursor to column one of the next line; or 2) to place the cursor on the first nonspace character in the next line, or, if the line is blank, beneath the first nonspace character in the first nonblank line on the screen above the cursor. If the cursor is on the last line on the screen, the screen scrolls down one line.

If the editor is in insert mode, the RETURN key will also split the line at the cursor position.

⌂-1 to ⌂-9 — Screen Moves

The file is divided by the editor into 8 approximately equal sections. The screen-move commands move the file to a boundary between one of these sections. The command ⌂1 jumps to the first character in the file, and ⌂9 jumps to the last character in the file. The other seven ⌂*n* commands cause screen jumps to evenly spaced intermediate points in the file.

CTRLP or ⌂P — Scroll Down One Line

The editor moves down one line in the file, causing all of the lines on the screen to move up one line. The cursor remains in the same position on the screen. Scrolling can continue past the last line in the file.

⌂] or ⌂} — Scroll Down One Page

The screen scrolls down twenty-two lines. Scrolling can continue past the last line in the file.

CTRLO or ⌂O — Scroll Up One Line

The editor moves up one line in the file, causing all of the lines on the screen to move down one line. The cursor remains in the same position on the screen. If the first line of the file is already displayed on the screen, the command is ignored.

⌂[or ⌂{ — Scroll Up One Page

The screen scrolls up twenty-two lines. If the top line on the screen is less than one screen's height from the beginning of the file, the screen scrolls to the beginning of the file.

⌂L Search Down

This command allows you to search through a file for a character or string of characters. When you execute this command, the prompt Search string: appears at the bottom of the screen. If you have previously entered a search string, the previous string appears after the prompt as a default. Type in the string for which you wish to search, and press RETURN. Searches are not case sensitive, and include all occurrences of the string, whether it is imbedded in a longer string or not. Spaces that appear at the beginning and end of a string are significant. For example, if you search for the string NOT, any of the following strings could be found:

not

Note

prothonotary

The following editing commands are active when you are entering the search string:

←	cursor left
→	cursor right
⌂> or ⌂.	end of line
⌂< or ⌂,	beginning of line
DELETE	delete character left
⌂Y or CTRLY	delete to end of line
⌂Z or CTRLZ	undo delete
⌂E or CTRLE	toggle insert mode
ESC, CLEAR, or CTRLX	cancel command without saving changes
RETURN or ENTER	save changes and execute command

When you press RETURN, the editor looks from the cursor position toward the end of the file for the search string. If the string is found, the screen is moved so that the next occurrence of the string is on the top line. The cursor is placed on the first character of the target string. The search stops at the end of the

file; to search between the current cursor location and the beginning of the file, use the Search Up command.

Use ⌘L or ⌘K to continue searching

If the string is not found, the following message appears on the screen:

```
String Not Found
```

⌘K Search Up

This command operates exactly like Search Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the Search Down command.

⌘J Search and Replace Down

This command allows you to search through a file for a character or string of characters, and to replace the search string with a replacement string. When you execute this command, the prompt Search string: appears at the bottom of the screen. If you have previously entered a search string, the previous string appears after the prompt as a default. Type in the string for which you wish to search, and press RETURN. Searches are not case sensitive, and include all occurrences of the string, whether it is imbedded in a longer string or not.

When you enter the search string and press RETURN, the prompt Replace string: appears at the bottom of the screen. If you have previously entered a replacement string, the previous string appears after the prompt as a default. Enter the string with which you want to replace the search string, and press RETURN. The prompt `Auto or Manual (A M Q)?` appears.

A Type **A** and press RETURN to cause all occurrences of the search string from the cursor position to the end of the file to be replaced automatically. The cursor returns to the starting point when the replacement is done.

M If you type **M** and press RETURN, then when the search string is found, it is highlighted on the top line of the screen and the prompt `Replace (Y N Q)?` appears at the bottom of the screen. Type **Y** RETURN to replace the string and search for the next occurrence; **n** RETURN to leave this occurrence of the string unchanged and search for the next occurrence; or **Q** RETURN to leave the string unchanged and terminate the search and replace operation.

Q Terminate the search and replace operation and return to the file you are editing.

The following editing commands are active when you are entering text in response to the Search String and Replace String prompts.

←	cursor left
→	cursor right
⌂> or ⌂.	end of line
⌂< or ⌂,	beginning of line
DELETE	delete character left
⌂Y or CTRLY	delete to end of line
⌂Z or CTRLZ	undo delete
⌂E or CTRLE	toggle insert mode
ESC, CLEAR, or CTRLX	cancel command without saving changes
RETURN or ENTER	save changes and execute command

When you enter a replacement string and respond with A or M, the editor looks from the cursor position toward the end of the file for the search string. The search stops at the end of the file; to search between the current cursor location and the beginning of the file, use the Search and Replace Up command. If the string is not found, the following message appears on the screen:

String Not Found

ÓH **Search and Replace Up**

This command operates exactly like Search and Replace Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the Search and Replace Down command.

ÓTAB **Set and Clear Tabs**

If there is a tab stop in the same column as the cursor, it is cleared; if there is no tab stop in the cursor column, one is set.

TAB **Tab**

The cursor is moved to the next tab stop. If there are no more tab stops, then the cursor is moved to the end of the line. If the editor is in overstrike mode, the tab acts only as a cursor-control command; no characters are inserted between the last character on the line and the tab stop. If you type a character at the tab stop, however, space characters are inserted in the file between the last character and the new character. In insert mode, space characters are inserted from the cursor's starting location to the tab stop; any characters to the right of the cursor are moved to the right to make room.

CTRLA or ÓA **Tab Left**

The cursor is moved to the previous tab stop, or to the beginning of the line if there are no more tab stops to the left of the cursor. This command does not enter any characters in the file.

⌂RETURN — Toggle Auto Indent Mode

If the editor is set to put the cursor on column one when you press RETURN, it is changed to put the cursor on the first nonspace character; if set to the first nonspace character, it is changed to put the cursor on column one. Auto-indent mode is described in the section "Modes" in this chapter.

ESC — Toggle Escape Mode

If the editor is in the edit mode, it is put in escape mode; if it is in escape mode, it is put in edit mode. When you are in escape mode, pressing any character not specifically assigned to an escape-mode command returns you to edit mode. Escape and edit modes are described in the section "Modes" in this chapter.

When in escape mode, ⌂CTRL_ will return you to edit mode. In edit mode the command has no effect. From edit mode, CTRL_ will place you in escape mode, but the command has no effect in escape mode. These commands are most useful in an editor macro, where you do not know what mode you are in on entry.

CTRLE or ⌂E — Toggle Insert Mode

If insert mode is active, the editor is changed to overstrike mode. If overstrike mode is active, the editor is changed to insert mode. Insert and overstrike modes are described in the section "Modes" in this chapter.

CTRL⌂X — Toggle Select Mode

If the editor is set to select text for the Cut, Copy, and Delete commands in units of one line, it is changed to use individual characters instead; if it is set to character-oriented selects, it is toggled to use whole lines. See the section "Modes" in this chapter for more information on select mode.

CTRL⇧W Toggle Wrap Mode

If the editor is set to stop at the end of a line and ignore additional characters, it is changed to insert a carriage return after the last full word in the line and continue entering text on the next line. If it is set to wrap lines, it is changed to stop at the end of the line. The wrap mode is described in the section "Modes" in this chapter.

⇧↑ Top of Screen / Page Up

The cursor moves to the first visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the top of the screen, the screen scrolls up twenty-two lines. If the cursor is at the top of the screen and less than twenty-two lines from the beginning of the file, then the screen scrolls to the beginning of the file.

CTRLZ or ⇧Z Undo Delete

The last character or block of characters deleted using the Delete Character, Delete Character Left, Delete Line, Delete to End of Line, or Delete Word commands is inserted at the cursor position. If the cursor has not been moved, the file is restored to its state before the delete.

The Undo buffer functions as a stack, so multiple undos are possible. For example, suppose you delete the errors (shown in boldface) in the following text, in the order in which they appear (that is, first the e, then the l, and so on):

```
It**e** woulld appear that an appp**p**eal to reason would go un**A**answered.
```

When you execute the Undo command one time, the text deleted last is restored; in this case, an a. If you execute a second Undo command, the text deleted before that, pp, is restored, and so on. In this example, four Undo commands in a row would put the following text on the screen:

```
Apple
```

A maximum of 10240 characters can be stored in the undo buffer. No warning is issued if you delete more than 10240 characters.

⌘← Word Left

The cursor is moved to the beginning of the next nonblank sequence of characters to the left of its current position. If there are no more words on the line, the cursor is moved to the last word in the previous line or, if it is blank, to the last word in the first nonblank line preceding the cursor.

⌘→ Word Right

The cursor is moved to the start of the next nonblank sequence of characters to the right of its current position. If there are no more words on the line, the cursor is moved to the first word in the next nonblank line.

Setting Editor Defaults

When you start the editor, it reads the file named SYSTABS (located in the current folder), which contains the default settings for tab stops, return mode, and select mode. The SYSTABS file is an ASCII text file that you can edit with the editor.

Each language recognized by the editor is assigned a language number. The SYSTABS file has three lines associated with each language:

1. The language number (currently 0 for BASIC).
2. The default settings for return, select, and word-wrapping modes.
3. The default tab and end-of-line-mark settings.

The first line in the SYSTABS file specifies the language. Languages can have numbers from 0 to 32767 (decimal). The language number must start in the first column; leading zeros are permitted and are not significant, but leading spaces are *not* allowed.

The second line of each set of lines in the SYSTABS file sets the defaults for various editor modes, as follows:

1. If the first column contains a zero, pressing RETURN in the editor causes the cursor to go to column one in the next line; if it's a one, pressing RETURN sends the cursor to the first nonspace character in the next line (or, if the line is blank, beneath the first nonspace character in the first nonblank line on the screen above the cursor).

2. If the second character is zero, the editor is set to line-oriented selects; if one, it is set to character-oriented selects.

3. If the third character is zero, the cursor stops when it reaches the end of a line; if one, the editor inserts a carriage return and wraps to the next line.

4. The fourth character is reserved for future enhancements.

5. The fifth character is reserved for future enhancements.

The third line of each set of lines in the SYSTABS file sets default tab stops. There are 255 zeros and ones in the first line of each pair, representing the 255 possible characters on a line. The ones (1) indicate the positions of the tab stops. A two (2) in any column of this line sets the end of the line. The column containing the two then replaces column 255 as the default right margin.

For example, the following lines define the defaults for AC/BASIC:

```
000
00000
00001000100010001000100010001000100010001000 ... 0102
```

If no defaults are specified for a language (that is, there are no lines in the SYSTABS file for that language), then the editor assumes the following defaults:

- RETURN sends the cursor to column one.
- Character-oriented selects.
- The cursor stops at the end of the line.
- There is a tab stop every eighth column.
- The end of the line is at column 80.

Note that you can change tabs and editing modes while in the editor.

Chapter Summary

This chapter covered the five modes of operation, defining customized editor macros, setting editor defaults, and the editor commands. As you can see, there are many commands and options used by the editor. If you become unsure of the use of an editor command, don't hesitate to refer back to this chapter.

Addendum to the Manual

This addendum contains permanent changes and corrections to the manual. For more up-to-date information about the compiler, you should use the editor to look at the READ.ME file in the BASIC folder of the compiler disk. The READ.ME file may contain important information pertaining to the function of the compiler. Note that AC/BASIC requires a computer with at least 512K of memory.

2-3 **IMPORTANT:** The file selector mentioned in step 4 does not automatically appear on the screen. After step 3, choose Open from the File menu or press ⌘-O to display the file selector.

3-5, 11-13
I-3 The font used for printing is Courier. If Courier is not found in the FONTS folder of the SYSTEM folder, then the system font is used. When using `WINDOW OUTPUT #filenumber`, the printer font may be changed with TEXTFONT.

4-2 There is an additional compiler option which brings up the initial screen as a 320x200 screen. This is the J option. When the J option is off, the initial screen is 640x200. The screen resolution may also be changed at run-time by using the SCREEN statement.

4-10 There is an additional metacommand OPTION which can be used to change the setting of various compiler options in the middle of a source file. They are as follows:

```
'$OPTION 8          'Read full 8-bits from COM1: port (no masking)
'$OPTION -8         'Mask the 8th bit from the COM1: port (default)
```

This metacommand is included since the Apple IIGS serial chip will set the 8th bit with the parity bit when the data size is chosen as 7 bits.

7-18 Example 8 will correctly print a "1" instead of a "0".

11-5 For `function-number` 0, the `filetype-string` is not ignored as stated. It is used as the default filename when the file selector appears. For example,

```
a$=FILES$(0,"Untitled")
```

will show the name `Untitled` highlighted in the file selector. For `function-number` 1, the `filetype-string` may be a numeric value 0-255 which specifies one file type by number.

11-12 The device CLIP: can optionally be opened specifically for text or pictures by specifying either TEXT or PICTURE after the colon. For example,

```
OPEN "CLIP:PICTURE" FOR OUTPUT AS #1
```

will open the system clipboard and specify that the data to be put on the clipboard is picture data such as from PICTURE$.

11-12 Input from the COM1: port may be masked or unmasked by using the '$OPTION metacommand described above.

12-5 The SCREEN statement, when specifying a screen type, can also be used to eliminate the default window and/or default menu. Adding 4 to the value of the screen type brings up the screen without the default window. Adding 8 to the value brings up the screen without the default menus. Here are some examples:

```
SCREEN 1+4        '320x200 screen without default window
SCREEN 2+4+8      '640x200 screen without window and without menus
```

12-2 The CONFIGURE program on the master disk may be used to remove the Scroll Bar and Grow Box from window type 1. Just compile and execute the program and follow the instructions.

12-6 The four window types used in the WINDOW statement may be negative to create a modal window which does not allow events such as mouse clicks outside of the window. A modal window is useful for displaying warning boxes which must be acted upon before any other user action takes place. There are eight valid window types: -4, -3, -2, -1, 1, 2, 3, 4.

12-7 A second form of the WINDOW OUTPUT statement allows the routing of all text and graphics to a device rather than the screen window. The only device currently supported is the printer.

```
WINDOW OUTPUT #filenumber
```

where filenumber is the filenumber of a previously opened file to the printer. For example, the following program will draw a rectangle on the printer:

```
OPEN "LPT1:PROMPT" FOR OUTPUT AS #2
WINDOW OUTPUT #2
LINE (0,0)-(100,100),,BF
CLOSE
```

12-27 Example 13 incorrectly dimensions the array to twice the needed size. Since the SCREEN() function returns bytes and there are two bytes in an integer, the size of the array in bytes should be divided by two. The dimension statement should read as follows:

```
DIM a%(SCREEN(0,0,96,8)/2)      ' Two bytes per element
```

If the array were single precision, the array size can be halved once again:

```
DIM a!(SCREEN(0,0,96,8)/4)      ' Four bytes per element
```

13-15 Buttons are always black with a white background.

13-16 EDIT FIELD type 2 is the same as type 1. Type 4 is the same as type 3

14-7 The HELLO program shows examples of manipulating the instrument array; move the slider controls to change the various effects. A custom waveform can be drawn by using the mouse and holding down the mouse button.

14-10
14-11 The PACK and UNPACK subprograms do not work properly with byte values over 127. Use the PACK and UNPACK subprograms in the HELLO program.

F-2 Error 13 is generated for a Type Mismatch Error instead of error 31.
Error 20 is generated for RESUME without error instead of error 146.
Error 160, will be generated if PICTURE ON is attempted while a window's output is being routed to the printer with the WINDOW OUTPUT #filenumber statement. Window output can only be sent to one device (screen,printer,string) at a time.
Error 164 is generated when attempting to open the printer device which is already open. This can occur when mixing LPRINT with OPEN "LPT1:".

G-1 When the compiler can not locate or load the editor, it beeps and does not display an error message. This can occur if there is not enough memory available or if the editor can not be found in the current search path given on page 4-7. When using the Finder, there may not be enough memory on a 512K machine to load the editor. If this occurs, launch AC/BASIC from the Launcher.

I-5 A text size greater than 66 point may cause system problems. This is not the fault of AC/BASIC.

APPENDIX A

INTERNAL DATA FORMATS FOR REAL NUMBERS

BINARY VERSUS DECIMAL NUMBERS

Chapter 4 of this manual showed you how to use the "D" option to control the type of numeric representation used by AC/BASIC for floating-point numbers. Floating-point numbers are "real" numbers in the mathematical sense. Mathematical "whole" numbers are called integers in AC/BASIC. Integers are stored and manipulated in the same manner, regardless of the floating-point representation being used. Integers are always stored and manipulated using two's-complement arithmetic. The two types of floating-point representations, called "binary" and "decimal", are explained in detail in this appendix.

Both floating-point representations store a number as a sign, an exponent, and a mantissa. The key difference affecting the user has to do with the different formats used for the mantissa. In the binary representation, the mantissa is stored as a base 2 number; in the decimal representation each digit of the mantissa is stored as a BCD (binary coded decimal) digit. Consequently, within the number of supported significant digits (a function of the storage size of the mantissa), decimal fractions are represented exactly. Because of this, no rounding errors will occur when real number calculations are done in a BASIC program that uses the BCD library. In accounting or other financial programs, the rounding of fractional cents can lead to erroneous results which are difficult to audit. There are four other considerations that should be taken into account when deciding which type of math to select for a program:

1. Default typing of variables: with decimal math the default variable type is double precision; with binary the default is single precision. Double-precision numbers occupy more memory than single-precision numbers and therefore increase the memory requirements of a program.

2. Default precision of mathematical operations and functions: in decimal, for all operations involving real numbers, the operation is carried out in double precision; in binary operations are carried out based on the highest precision of all the operands involved. Although less precise, single-precision computations are faster than double-precision.

3. Range of magnitude of representable numbers: with the decimal library the approximate range of real numbers is:

10^{63} > representable number > 10^{-64}

with the binary library the approximate range is:

10^{308} > representable number > 10^{-308}

4. Number of significant digits: with the decimal library, the maximum number of significant digits for a real number is 14; with the binary library, the maximum is close to 16.

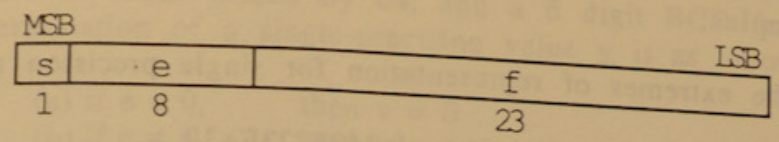
Note: Binary representation is used by most interpreters, and is the default representation used by AC/BASIC. In the majority of cases you will want to use this representation. If you have any doubt about which representation to use, then use binary.

The remainder of this appendix describes the internal representation of binary and decimal floating-point numbers. This information is presented for advanced programmers who need to know the internal structure of numbers for special purposes.

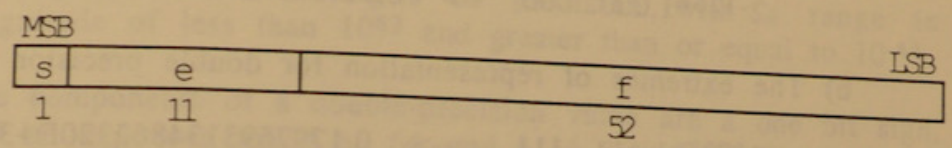
Single and Double Precision Binary Numbers

Single and double-precision binary floating-point values are maintained in the format described in *A Proposed Standard for Binary Floating-point Arithmetic* by IEEE Task P754 in *COMPUTER* magazine.

Single-precision format:



Double-precision format:



The largest possible exponent for both formats is reserved to represent the symbolic entity Not a Number (NaN).

The components of a single-precision value are a one bit sign, an eight bit exponent biased by 127, and a twenty-three bit fraction with an implied most significant bit. The interpretation of a single precision value v is as follows:

(a) if $e = 255$ and $f \neq 0$,	then $v = NaN$
(b) if $e = 255$ and $f = 0$,	then $v = (-1)^s$ infinity
(c) if $0 < e < 255$,	then $v = (-1)^s 2^{e-127} * 1.f$
(d) if $e = 0$ and $f \neq 0$,	then $v = (-1)^s 2^{-126} * 0.f$
(e) if $e = 0$ and $f = 0$,	then $v = 0$

This provides for 7 digits of precision with a range in magnitude of less than 10^{39} and greater than or equal to 10^{-37}.

The components of a double-precision value are a one bit sign, an eleven bit exponent biased by 1023, and a fifty-two bit fraction with an implied most significant bit. The interpretation of a double-precision value v is as follows:

(a) if $e = 2047$ and $f \neq 0$,	then $v = NaN$
(b) if $e = 2047$ and $f = 0$,	then $v = (-1)^s$ infinity
(c) if $0 < e < 2047$,	then $v = (-1)^s 2^{e-1023} * 1.f$
(d) if $e = 0$ and $f \neq 0$,	then $v = (-1)^s 2^{-1022} * 0.f$
(e) if $e = 0$ and $f = 0$,	then $v = 0$

This provides for almost 16 digits of precision with a range in magnitude of less than 10^{309} and greater than or equal to 10^{-307}.

Examples:

a) The extremes of representation for single precision are:

$2^{127}*1.111...111$ --> 0.3402823E+39

$2^{-126}*1.000...000$ --> 0.1175494E-37

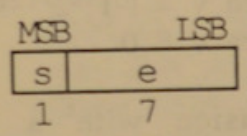
b) The extremes of representation for double precision are:

$2^{1023}*1.111...111$ --> 0.1797693134862320D+309
$2^{-1022}*1.000...000$ --> 0.2225073858507202D-307

Single and Double Precision Decimal Numbers

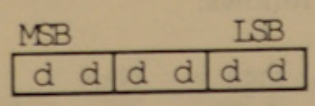
Single-precision and double-precision floating-point values are maintained in the format used by Microsoft Corporation for its interpreters and compilers which support decimal math. Both types of values share the same sign and exponent byte.

Single/Double-Precision Sign and Exponent Format:

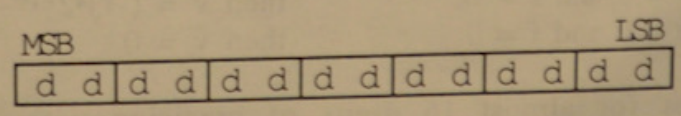


However, their mantissa formats vary. Shown byte-wise, each BCD digit (four bits per BCD digit) is represented by a "d":

Single-Precision Mantissa Format:



Double-Precision Mantissa Format:



The components of a single-precision value are a one bit sign, a 7 bit exponent biased by 64, and a 6 digit BCD fraction. The interpretation of a single-precision value v is as follows:

(a) if $e = 0$, then $v = 0$
(b) if $e \neq 0$, then $v = (-1)^s*(0.dddddd*(10^{e-64}))$

This provides for 6 digits of precision with a range in magnitude of less than 10^{63} and greater than or equal to 10^{-64}.

The components of a double-precision value are a one bit sign, a 7 bit exponent biased by 64, and a 14 digit BCD fraction. The interpretation of a double precision value v is as follows:

(a) if $e = 0$, then $v = 0$
(b) if $e \neq 0$, then $v = (-1)^s*(0.dddddddddddddd*(10^{e-64}))$

This provides for 14 digits of precision with a range in magnitude of less than 10^{63} and greater than or equal to 10^{-64}.

Examples:

(a) BCD Single-precision

Number	Internal Representation
-7.77777E+35	E4777777
7.77777E+35	64777777
0	00??????

(b) BCD Double-precision

Number	Internal Representation
-9.9999999999999D+35	E499999999999999
9.9999999999999D+35	6499999999999999
0	00??????????????

APPENDIX B

ASCII CODES

CHARACTER	DEC	OCT	HEX	NAME
NULL	0	000	00	null
SOH	1	001	01	start of heading
STX	2	002	02	start of text
ETX	3	003	03	end of text
ECT	4	004	04	end of transmission
ENQ	5	005	05	enquiry
ACK	6	006	06	acknowledge
BEL	7	007	07	bell code
BS	8	010	08	back space
HT	9	011	09	horizontal tab
LF	10	012	0A	line feed
VT	11	013	0B	vertical tab
FF	12	014	0C	form feed
CR	13	015	0D	carriage return
SO	14	016	0E	shift out
SI	15	017	0F	shift in
DLE	16	020	10	data link escape
DC1	17	021	11	device control 1
DC2	18	022	12	device control 2
DC3	19	023	13	device control 3
DC4	20	024	14	device control 4
NAK	21	025	15	negative acknowledge
SYN	22	026	16	synchronous idle
ETB	23	027	17	end of transmission blocks
CAN	24	030	18	cancel
EM	25	031	19	end of medium
SS	26	032	1A	special sequence
ESC	27	033	1B	escape
FS	28	034	1C	file separator
GS	29	035	1D	group separator
RS	30	036	1E	record separator
US	31	037	1F	unit separator

CHARACTER	DEC	OCT	HEX	NAME
SP	32	040	20	space
!	33	041	21	exclamation mark
"	34	042	22	quotation mark
#	35	043	23	number sign
$	36	044	24	dollar sign
%	37	045	25	percent sign
&	38	046	26	ampersand
'	39	047	27	apostrophe
(	40	050	28	opening parenthesis
)	41	051	29	closing parenthesis
*	42	052	2A	asterisk
+	43	053	2B	plus
,	44	054	2C	comma
-	45	055	2D	minus
.	46	056	2E	period
/	47	057	2F	slash
0	48	060	30	zero
1	49	061	31	one
2	50	062	32	two
3	51	063	33	three
4	52	064	34	four
5	53	065	35	five
6	54	066	36	six
7	55	067	37	seven
8	56	070	38	eight
9	57	071	39	nine
:	58	072	3A	colon
;	59	073	3B	semicolon
<	60	074	3C	less than
=	61	075	3D	equal
>	62	076	3E	greater than
?	63	077	3F	question mark
@	64	100	40	commercial at

CHARACTER	DEC	OCT	HEX	NAME
A	65	101	41	upper case letter
B	66	102	42	upper case letter
C	67	103	43	upper case letter
D	68	104	44	upper case letter
E	69	105	45	upper case letter
F	70	106	46	upper case letter
G	71	107	47	upper case letter
H	72	110	48	upper case letter
I	73	111	49	upper case letter
J	74	112	4A	upper case letter
K	75	113	4B	upper case letter
L	76	114	4C	upper case letter
M	77	115	4D	upper case letter
N	78	116	4E	upper case letter
O	79	117	4F	upper case letter
P	80	120	50	upper case letter
Q	81	121	51	upper case letter
R	82	122	52	upper case letter
S	83	123	53	upper case letter
T	84	124	54	upper case letter
U	85	125	55	upper case letter
V	86	126	56	upper case letter
W	87	127	57	upper case letter
X	88	130	58	upper case letter
Y	89	131	59	upper case letter
Z	90	132	5A	upper case letter
[	91	133	5B	opening bracket
\	92	134	5C	back slash
]	93	135	5D	closing bracket
^	94	136	5E	circumflex
_	95	137	5F	underscore
`	96	140	60	grave accent

CHARACTER	DEC	OCT	HEX	NAME
a	97	141	61	lower case letter
b	98	142	62	lower case letter
c	99	143	63	lower case letter
d	100	144	64	lower case letter
e	101	145	65	lower case letter
f	102	146	66	lower case letter
g	103	147	67	lower case letter
h	104	140	68	lower case letter
i	105	151	69	lower case letter
j	106	152	6A	lower case letter
k	107	153	6B	lower case letter
l	108	154	6C	lower case letter
m	109	155	6D	lower case letter
n	110	156	6E	lower case letter
o	111	157	6F	lower case letter
p	112	160	70	lower case letter
q	113	161	71	lower case letter
r	114	162	72	lower case letter
s	115	163	73	lower case letter
t	116	164	74	lower case letter
u	117	165	75	lower case letter
v	118	166	76	lower case letter
w	119	167	77	lower case letter
x	120	170	78	lower case letter
y	121	171	79	lower case letter
z	122	172	7A	lower case letter
{	123	173	7B	opening brace
|	124	174	7C	bar
}	125	175	7D	closing brace
~	126	176	7E	tilde
DEL	127	177	7F	delete

APPENDIX C

BASIC RESERVED WORDS

This is a list of all the reserved words in AC/BASIC. These words may not be used as variable names since they define the statements and functions in AC/BASIC.

ABS
AND
APPEND
AS
ASC
ATN

BASE
BEEP
BREAK
BUTTON

CALL
CASE
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVDBCD
CVI
CVS
CVSBCD

DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DIALOG
DIM

EDIT
ELSE
ELSEIF
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXIT
EXP

FIELD
FILES
FILES$
FIX
FN
FOR
FRE

GET
GOSUB
GOTO

HEX$

IF
IMP
INKEY$
INPUT
INPUT$
INSTR
INT

KILL

LBOUND
LCOPY
LEFT$
LEN
LET
LINE
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET

MACHINE
MENU

MID$
MKD$
MKDBCD$
MKI$
MKS$
MKSBCD$
MOD
MOUSE

NAME
NEW
NEXT
NOT

OCT$
OFF
ON
OPEN
OPTION
OR
OUTPUT

PALETTE
PEEK
PICTURE
POINT
POKE
POS
PRESET
PRINT
PSET
PTAB
PUT

RANDOMIZE
READ
READB
REM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN

SAVE
SCREEN
SCROLL
SEEKB
SELECT
SGN
SHARED
SIN
SOUND
SPACE$
SPC
SQR
STATIC
STEP
STOP
STR$
STRING$
SUB
SWAP
SYSTEM

TAB
TAN
THEN
TIME
TIMER
TO

UBOUND
UCASE$
USING
USR

VAL
VARPTR

WAIT
WAVE
WEND
WHILE
WIDTH
WINDOW
WRITE
WRITEB

XOR

PPENDIX D

ACHINE-LANGUAGE EXAMPLE

This appendix shows how a machine-language subroutine may be called from AC/BASIC; it is for advanced users. It assumes you know 65816 assembly language and have the APW or similar development environment with an assembler like the ORCA/M.

In certain specialized cases, you may wish to write a position-independent machine-language subroutine that an AC/BASIC program can call. In order for a procedure written in machine language to communicate with the calling program, certain program structure specifications must be followed. The two examples in this appendix show how to write the position-independent machine-language program and pass information between it and an AC/BASIC program. See the **MACHINE** statement in Chapter 16 for the direct-page addresses of the passed arrays.

Sieve of Eratosthenes

The Sieve of Eratosthenes is a popular algorithm for calculating prime numbers. It is written here in 65816 assembly language for the ORCA/M assembler from Byte Works. To assemble and link the assembly-language program in APW, type:

```
ASML SIEVE.S
```

This will produce the file `SIEVE` which is used by the BASIC program which is listed first. The object code header format is described in the APW manual.

```
REM AppendixD/Example1
REM The BASIC Sieve caller
REM Sieve of Eratosthenes using machine language.
REM $ABOUT "Eratosthenes test."
REM Requires the assembled and linked version
REM of the program SIEVE.S named SIEVE.

DEFINT A-Z
Size=8190
DIM A(21),Flags(Size+1),Misc(10)
OPEN "R",1,"SIEVE"  'Open as random-access file.

BLOAD #1,A,44        'Read object code header.
Lcode=A(4)           'Length of the code.
Ocode=A(21)+5        'Offset start of the code.

PRINT "Code length";Lcode;", Offset";Ocode
ERASE A : DIM A(Lcode/2)

X!=BSEEK(#1,Ocode,0)      'Position to start of code.
BLOAD #1,A,Lcode          'Read the code into array A.
CLOSE #1

Misc(1)=10                'Number of iterations.
Misc(2)=Size              'Size of the array.

PRINT Misc(1);"Iterations."
Start!=TIMER

MACHINE A,Flags,Misc      'Go execute.

Finish!=TIMER
PRINT misc(0);"primes."
PRINT "Sieve took";Finish!-Start!;"seconds."

PRINT "Click in window to end."
WHILE MOUSE(0)=0 :WEND
```

```
;The assembly-language Sieve program
     absaddr on
     gen off
     symbol off
     keep sieve
;Address of Flags in $24.
;Address of Misc in $28:
; Misc(0) is for return value Count.
; Misc(1) is number of iterations.
; Misc(2) is Size (=8190).
;The timing depends mostly on the k-loop below.
; As is 3.7 sec for 10 iterations (37 for 100).
;
sieve     start
     STZ  $00
     LDY  #$0002
     LDA  [$28],Y
     STA  $1C          ;Iterations to $1C
     INY
     INY
     LDA  [$28],Y
     INC  A
     STA  $04          ;Size+1 to $04

Itlp LDA  $04
     DEC  A
     ASL  A
     TAY
     LDA  #$FFFF
     STZ  $18          ;Count to $18, initialize to zero.

lab1 STA  [$24],Y      ;Set whole array Flags to -1.
     DEY
     DEY
     BNE  lab1
     STA  [$24]

     STZ  $14          ;Loop index i = $14
lab2 LDA  $14
     ASL  A
     TAY               ;Y = 2*i
     LDA  [$24],Y      ;Test Flags(i)
     BEQ  lab3         ;Jump if zero.
     TYA
     INC  A
     INC  A
     INC  A
     STA  $10          ;Else prime = $10 = 2*i + 3
     CLC
     ADC  $14          ;Start value A = k = i + prime.
     LDX  $00          ;Load 0 in X ($00 = 0, see above).
     BRA  lab5

lab4 ASL  A            ;k - loop
```

```
      TAY
      TXA                ; A = X = 0
      STA  [$24],Y       ;Flags(k) = 0
      TYA
      LSR  A
      CLC
      ADC  $10           ;k = k + prime
lab5  CMP  $04
      BCC  lab4          ;Go on till Size (in $04) reached.
      INC  $18           ;Increase count of primes.

lab3  LDA  $14           ;Next i
      INC  A
      STA  $14
      CMP  $04
      BCC  lab2          ;Go on till Size.

      DEC  $1C           ;Iteration count.
      BNE  Itlp
      LDA  $18           ;Return result in Misc(0).
      STA  [$28]
      RTL
      END
```

Using a Toolbox

This example show how a Toolbox call can be made from a machine-language subroutine. For a complete description of the Apple IIGS Toolbox calls, refer to the *Apple IIGS Toolbox Reference Manuals, volumes 1 and 2.*

This example program calls the QuickDraw Toolbox routine GetCursorAdr which returns the address of the current mouse cursor definition. The definition is copied into the second array argument of the **MACHINE** statement. The first element of the third array contains the number of words to transfer.

```
REM The BASIC caller for GETCURSOR
DEFINT A-Z
size=200
DIM code(100),cursor(size),Misc(1)
OPEN "R",1,"GETCURSOR"    'Open as random-access file.

BLOAD #1,code,44          'Read object code header.
Lcode=code(4)             'Length of the code.
Ocode=code(21)+5          'Offset start of the code.

X!=BSEEK(#1,Ocode,0)      'Position to start of code.
BLOAD #1,code,Lcode       'Read program into code.
CLOSE #1

Misc(0)=size              'Estimated size of cursor.

MACHINE code,cursor,Misc      'Go execute.
```

```
;The assembly-language GETCURSOR Toolbox sample
     absaddr on
     gen off
     symbol off
     keep getcursor
;Get cursor.
;
;Address destination array in $24.
;Address Misc in $28 (integers).
;Number of words to transfer in Misc(0).
;
getcursor start
     PEA  $0000
     PEA  $0000
     LDX  #$8F04
     JSL  $E10000    ;Toolbox call GetCursorAdr
     PLA
     STA  $2C        ;Address to $2C
     PLA
     STA  $2E
     LDA  [$28]      ;Get Misc(0)=word count.
     TAX
     LDY  #$0000
lab1 LDA  [$2C],Y    ;Transfer.
     STA  [$24],Y
     INY
     INY
     DEX
     BNE  lab1
     RTL
     END
```

APPENDIX E

DIFFERENCES BETWEEN VERSIONS OF AC/BASIC

AC/BASIC currently runs on three different machines: the Apple Macintosh (as Microsoft BASIC), the Commodore Amiga, and the Apple IIGS. These versions of AC/BASIC are very similar, and you will find it very easy to port programs between the three machines.

Each version of AC/BASIC has certain statements and functions designed to take advantage of special machine-dependent features. This appendix describes the major differences between the various implementations.

The descriptions given here are merely to highlight the differences between the implementations. For specific details about the the various statements and functions, refer to the chapters in the appropriate manuals.

This appendix is based on Amiga AC/BASIC version 1.2, Microsoft BASIC compiler for the Macintosh version 1.0, and Apple IIGS AC/BASIC version 1.0. If you have a more recent version of the Microsoft or Amiga AC/BASIC compiler, you should consult that reference manual for any of the statements or functions about which you have questions.

MACINTOSH-SPECIFIC STATEMENTS AND FUNCTIONS

The following statements and functions are specific to the Macintosh implementation of AC/BASIC and do not currently appear in the Apple IIGS version:

LIBRARY
SADD
SELECT CASE / END SELECT

AMIGA-SPECIFIC STATEMENTS AND FUNCTIONS

The following statements and functions are specific to the Amiga implementation of AC/BASIC and do not appear in the Apple IIGS version. The primary differences are that the Amiga has long integers and object graphics statements:

AREA
AREAFILL
CLNG
COLLISION ON/OFF/STOP
CVL
DECLARE FUNCTION
DEFLNG
MKL$
OBJECT (all forms)
ON COLLISION
PAINT
PATTERN
PEEKL
PEEKW
POKEL
POKEW
SADD
SAY
SCREEN CLOSE
SELECT CASE / END SELECT
SLEEP
STICK
STRIG
TRANSLATE

PPLE IIGS-SPECIFIC STATEMENTS AND FUNCTIONS

The following statements and functions are specific to the Apple IIGS implementation of AC/BASIC:

$ABOUT
BLOAD
BSEEK()
BSAVE
CONT
MACHINE
SCREEN()
SUB...SHARED
SWAP BYTES
TEXTCOLOR
TEXTBCOLOR

ACINTOSH STATEMENT AND FUNCTION DIFFERENCES

Several statements and functions work differently on the Apple IIGS than on the Macintosh. This section describes these differences.

- **CALL** — The IIGS uses the **MACHINE** statements to call machine-language subroutines rather than the **CALL** statement. The IIGS does not have libraries.

- **CIRCLE** — An additional parameter for the interior color of the circle is available on the IIGS.

- **CLEAR** — On the Macintosh, the `stacksize` parameter is used.

- **CONT** — This statement is not on the Macintosh.

- **FRE** — It returns values with different meanings.

- **LIBRARY** — Libraries are not supported on the IIGS.

- **LINE** — The color attribute may range from 0 to 255 on the IIGS, while it may be either 30 or 33 on the

Macintosh. Also, because the pixel shape varies between the systems, a square box on the Macintosh will appear as a tall and narrow rectangle on the IIGS. An additional parameter for the interior color is available on the IIGS.

- **NAME** — The file type on the IIGS is a three-letter string (or number 0 to 255) whereas the Macintosh has a four-letter file type

- **POINT** — Point returns either 30 or 33 on the Macintosh. The IIGS version returns a number that corresponds to a palette entry

- **PRINT USING** — The keyword **USING** must appear immediately after **PRINT** on the IIGS. A statement on the Macintosh such as:

```
PRINT TAB(5) USING "##";a
```

can be changed to the equivalent two statements:

```
PRINT TAB(5);
PRINT USING "##";a
```

- **PSET & PRESET** — The color attribute for these statements may be 30 or 33 on the Macintosh version. On the IIGS version, the color attribute corresponds to the color id in a **PALLETTE** statement.

- ROM calls — Many of the ROM calls on the IIGS support color. Most of the Macintosh calls operate the same with some notable exceptions like **PENPAT** and **SETCURSOR**.

- **SADD** — The **SADD** function is not available on the IIGS since strings are not necessarily stored internally as they appear when printed. For example, a string produced by **STRING$**(30,"*") does not allocate a 30 byte area of memory; instead it is represented by a 4 byte token. A string can be placed into an integer array by repeatedly using the **CVI** statement.

- **SELECT CASE** — The **SELECT CASE** construct is not part of the IIGS version of AC/BASIC

- **WAVE** — The syntax between versions is different:

  Macintosh:
  WAVE voice,wave-array,phase

  IIGS:
  WAVE voice,wave-array,size,address,instrument

  The phase specifier does not exist for the IIGS.

  The IIGS has the additional built-in waves **TRI** and **SQR** for specifying triangle and square waves. The wave size on the IIGS is not restricted to 256 bytes, and an optional instrument may be specified. The wave address may be modified for advanced waveform manipulation. A waveform on the IIGS can not contain the value zero — it causes the DOC to stop.

- **WINDOW** — Window type 4, which is a shadowed window on the Macintosh, appears as a single-border window.

- The maximum size of any single program block on the Apple IIGS is 64K which is 65536 bytes. This means the main program can not be larger than 64K and *each* subprogram can not be larger than 64K

MIGA STATEMENT AND FUNCTION DIFFERENCES

Several statements and functions work differently on the Apple IIGS than on the Amiga. This section describes these differences.

- **BEEP** — A **BEEP** statement on the IIGS will just sound the speaker. On the Amiga, the speaker will sound *and* the entire screen will be flashed.

- **CALL** — The IIGS uses the **MACHINE** statements to call machine-language subroutines rather than the **CALL** statement. The IIGS does not have libraries.

- **CIRCLE** — The syntax for this statement is identical on both versions. The differences lie in the color specifier and aspect ratio.

  The color specifier for the Amiga may range from 0 to 31, allowing up to 32 different colors. The IIGS color may be specified from 0 to 255.

  The IIGS has an additional optional parameter for the color with which the circle is filled.

- **CLEAR** — The heap size parameter on a **CLEAR** statement is significant on the IIGS. On the Amiga, the parameters are essentially ignored. Refer to Chapter 6 ("Compiler and Interpreter Differences") in the Amiga AC/BASIC Manual for information about the **CLEAR** statement on the Amiga.

- **CONT** — This statement is not on the Amiga.

- **DATE$** — **DATE$** can be used on both sides of the equals (=) sign in the IIGS version. That is, you can set the date by assigning a string expression to **DATE$**. In the Amiga version, you can only use **DATE$** on the right hand side of the equals sign.

- **DIALOG** — Dialogs are not available on the Amiga since the statements **DIALOG**, **BUTTON**, and **EDIT FIELD** are not in the Amiga version of AC/BASIC.

- **DIM** — The optional **SHARED** attribute has been added to the **DIM** statement on the Amiga. This attribute is not available on the IIGS. Refer to Chapter 8 of the Amiga AC/BASIC Manual for further information about **DIM SHARED**.

- **FRE** — It returns values with different meanings.

- **LIBRARY** — Libraries are not supported on the IIGS.

- **LINE** — The color attribute may range from 0 to 31 on the Amiga, while it may range from 0 to 255 on the IIGS.

An optional parameter on the IIGS allows a filled box to be filled with a color different from the border color.

- **MOUSE**(0) — Since the Amiga has two mouse buttons and the IIGS has only one, it is significant to note that **MOUSE**(0) returns information about the left mouse button on the Amiga version.

- **NAME** — The syntax between versions is different:

  IIGS:
  NAME old **AS** new [,filetype]

  Amiga:
  NAME old **AS** new

The filetype parameter is not valid on the Amiga.

- **OPEN** — The logical device CLIP: exists for the IIGS, but not for the Amiga.

- **PALETTE** — The **PALETTE** statement is very similar on the two machines, with the exception that the IIGS has 16 palettes and a special **PALETTE USING** form. Note that the palette number is specified *first* on the IIGS..

- **POINT** — It returns a palette entry on the IIGS. The Amiga version returns a value from 0 to 31.

- **PSET & PRESET** — The color attribute may be 0 to 255 on the IIGS, but only 0 to 31 on the Amiga.

- **PRINT USING** — The keyword **USING** must appear immediately after **PRINT** on the IIGS. A statement on the Amiga such as:

```
PRINT TAB(5) USING "##";a
```

can be changed to the equivalent two statements:

```
PRINT TAB(5);
PRINT USING "##";a
```

- **PUT** — The syntax between versions is different:

  IIGS:
  PUT (x1,y1)-[(x2,y2)],array[(ind,[ind...])[,action]

  Amiga:
  PUT [**STEP**] (x,y),array[(ind,[ind...])[,action]

  The IIGS has the ability to scale the graphic data, based on the size of the destination rectangle. The Amiga cannot perform an equivalent function, so only the upper left coordinates can be specified.

- ROM calls — The ROM calls described in Appendix I are specific to the Apple-line of computers.

- **SADD** — The **SADD** function is not available on the IIGS since strings are not necessarily stored internally as they appear when printed. For example, a string produced by **STRING$**(30,"*") does not allocate a 30 byte area of memory; instead it is represented by a 4 byte token.

- **SCREEN** — There are two screen modes on the IIGS compared to the four available on the Amiga. The IIGS has no interlaced modes.

- **TIME$** — **TIME$** can be used on both sides of the equals (=) sign in the IIGS version. That is, you can set the time by assigning a string expression to **TIME$**. In the Amiga version, you can only use **TIME$** on the right hand side of the equals sign.

- **WAVE** — The syntax between versions is different:

  Amiga:
  WAVE voice, wave-definition

  IIGS:
  WAVE voice,wave-array,size,address,instrument

  The IIGS has the additional built-in waves **TRI** and **SQR** for specifying triangle and square waves. The wave size on the IIGS is not restricted to 256 bytes, and an optional

instrument may be specified. The wave address may be modified for advanced waveform manipulation. A waveform on the IIGS can not contain the value zero — it causes the DOC to stop.

The Amiga produces stereo output. A voice of 0 or 3 indicates the left channel is to be used while 1 or 2 indicates the right channel.

- **WIDTH** — The IIGS compiler allows a form of the width function which returns the width of a string expression in terms of pixels. This function does not exist in the Amiga version.

- **WINDOW** — The syntax between versions is different:

  IIGS
  WINDOW `windownum[,[title][,[rect][,type]]]`
  WINDOW OUTPUT `#filenumber`

  Amiga
  WINDOW `windownum[,[title][,rect][,[type][,id]]]`

  The window type on the IIGS refers to a particular kind of window that has certain attributes. On the Amiga, the type refers to a set of attributes that are activated when the window is defined.

  The Amiga's **WINDOW** statement also has an additional parameter. The screen specifier associates the new window with a currently defined screen. The IIGS has no equivalent to a screen specifier.

  The **WINDOW OUTPUT** `#filenumber` statement is not supported by the Amiga implementation.

- The maximum size of any single program block on the Apple IIGS is 64K which is 65536 bytes. This means the main program can not be larger than 64K and *each* subprogram can not be larger than 64K

PROGRAMMING TECHNIQUES

Virtually all the programs written in Microsoft BASIC on the Macintosh can be compiled and run on the Apple IIGS. For some programs, no conversion will be required; for others, more extensive changes may have to be made. This section discusses how to write programs that will run on the Macintosh and both screen modes of the Apple IIGS and even on future Apple computers running AC/BASIC.

Screen Resolution

The Apple IIGS currently has two screen resolutions — 640x200 and 320x200. The Macintosh may have any screen resolution since video boards and monitors are available from a variety of manufacturers. What this means is that you can not rely upon the absolute size of the screen. A statement like,

```
LINE (0,0)-(630,100),68,BF
```

would be just fine on the 640x200 screen of the Apple IIGS but is too large for the 320x200 screen and many Macintosh monitors. A better approach would be to have variables which define the size of the screen; then all graphics operations would be based on those variables. For example,

```
top=0                ' Assume Apple IIGS 640x200 screen
left=0
bottom=199
right=639
WINDOW 1,"",(left+5,top+13+5)-(right-5,bottom-5),3
```

This small program will draw a window on the screen leaving a 5 pixel border on each side. To change the program for a different screen resolution, only the numbers in the first four lines must be changed. The menu height assumption of 13 pixels only applies to the Apple IIGS. To account for this, the following line can be added, and the 13 in the **WINDOW** statement can be replaced with `menuheight`.

```
menuheight=13
```

When working with coordinates that are within a window, use the **WINDOW**(2) and **WINDOW**(3) functions to get the width and height of the window. The next example divides the window width into ten equal parts and draws a circle in the center of each.

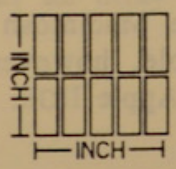
```
w=WINDOW(2)                    ' Get window width
h=WINDOW(3)                    ' Get window height
FOR x=0 TO w-w/10 STEP w/10
   CIRCLE (x+w/10/2,h/2),10
NEXT x
```

The **WINDOW**() function may also be used to estimate the total screen size since the initial default window in AC/BASIC is just smaller than the total screen size

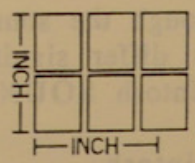
Aspect Ratio

A screen's aspect ratio is defined as: "The number of horizontal pixels per inch divided by the number of vertical pixels per inch." This is best shown by a picture,

Pixels in 640x200 mode
Aspect ratio 2.4:1



Pixels in 320x200 mode
Aspect ratio 1.2:1



The aspect ratio on the Macintosh is usually 1:1 which means the pixels are square.

Only the **CIRCLE** statement automatically compensates for the aspect ratio. To draw a square which takes into account the aspect ratio,

```
aspect=2.4
LINE (x,y)-(x+w*aspect,y+h),,BF
```

The aspect ratio for the Apple IIGS can be derived from the mode from the **SCREEN** statement:

```
res=1                    ' Resolution, either 1 or 2
SCREEN res
aspect=1.2 * res
```

Colors

The Macintosh has two colors: black and white. The Apple IIGS has either 4 or 16 true colors depending upon the resolution. Usually, programmers represent colors on a two-color screen with shades or patterns.

Since BASIC programs on the Macintosh have not dealt with color until the introduction of the Macintosh II, most Macintosh programs were written strictly for two-color screens. Furthermore, on the two-color Macintoshes, odd color numbers appear as black and even numbers are white. The standard convention is to use 33 for black and 30 for white. On the Apple IIGS these are represented as 0 and 255, respectively.

Sounds

Although the sound statements on the Macintosh and the Apple IIGS differ significantly, it is possible, however, to convert a Macintosh **SOUND** statement for the Apple IIGS.

Macintosh

```
SOUND frequency, duration, volume, voice
```

Apple IIGS

```
SOUND voice, interval, duration, tone, volume
```

The following conversion formulas will result in similar sounds:

Apple IIGS		Macintosh
voice	=	voice
interval	=	duration * 3.297
duration	=	duration * 3.297
tone	=	**INT** (9+12*LOG(frequency/13.75)/**LOG**(2))
volume	=	volume

Sound effects produced on the Macintosh by changing the frequency slightly may be replicated by changing the instrument array on the the Apple IIGS.

The **WAVE** statement is also different on the two machines. But once again, the Macintosh statement may be converted for the Apple IIGS. The wave definition array on the Macintosh contains values from -128 to 127 where on the Apple IIGS the values must be from 1 to 255. This means each array element must be processed by the following formula:

```
element = INT(element + 129 * (255 / 256))
```

Note that a wave definition element can not have the value zero. This causes the sound generator to stop.

The Macintosh **WAVE** statement syntax may be used on the Apple IIGS except for the phase parameter which may be specified as an array index instead.

AC/BASIC Reference Manual

APPENDIX F

RUN-TIME ERROR MESSAGES

Loading Errors

Some run-time errors may occur while loading before the program begins execution:

`Chained file has wrong format` — The program referenced in a **CHAIN** or **RUN** statement must be a compiled AC/BASIC program file.

`Failing load of system tools` — The AC/BASIC compiler and the compiled programs load many of the Apple Toolboxes from disk. This error occurs when the Toolboxes are not found on the disk or are outdated. The AC/BASIC compiler disk contains the the required tools which may be copied to your own disks. See Appendix H for a complete installation guide of the system Toolboxes.

`Failing read of run-time libraries` — The required run-time libraries called `BASIC.1L` and `BASIC.2L` were not found using the normal search path of folders. See the section "Link Run-time Library" in Chapter 4 for a description of the search path and using the "R" option to link the run-time libraries to the compiled program.

`Not enough memory available` — The AC/BASIC compiler and compiled programs require at least 512K of memory. A large compiled program may require even more memory. Note that the RAM disk reduces the amount of available memory; delete some files from the RAM disk or use the Control Panel to reduce the RAM disk size.

Execution Errors

Here is a list of all of the errors which may be reported during the execution of a program. The number associated with each message is the number which will be reported to the screen at run-time (no error trapping), or stored in the function variable **ERR** (with error trapping).

0 User break — `Quit` was selected from the `File` menu or ⌂-. was pressed with break event trapping off. This is the only run-time error not trap-able with the **ON ERROR** statement. **ON BREAK** *can* trap it.

3 RETURN without GOSUB — A **RETURN** statement has been encountered during execution, and there was no pending **GOSUB** statement.

4 Out of data — A **READ** statement has been attempted with no data available from a **DATA** statement.

5 Illegal function call — A call to a function has been attempted with too many arguments or arguments not within the expected range.

6 Overflow — A numeric operation has been attempted that returned a number too large or too small for the machine to represent, or to assign to a destination data type (e.g., `B%=30000 : A%=B%+B%`).

9 Subscript out of range — A reference to an array has been attempted with a subscript beyond its current dimension.

10 Duplicate definition — An attempt was made to redimension an array without erasing it first, or a redimensioned array in a **COMMON** block does not match exactly with the original declaration.

11 Division by zero — Occurs when zero is used as the denominator for a division operation, or when zero is raised to a negative exponent.

14 Out of heap space — There is no more memory available at run-time.

15 String too long — The string in this operation is too long. The maximum string length is 32767 bytes.

16 String formula too complex — AC/BASIC can not do all the requested string operations in a single statement. To fix, divide the operation into two functionally equivalent ones.

19 RESUME statement missing — No **RESUME** statement occurred in the error handling routine.

24 Too many dimensions — The maximum number of array dimensions is 7.

28 Syntax error — A syntax error not caught at compile-time may be the result of a function (FN) reference..

31 Type mismatch — A string was expected when a number was given, or vise versa.

38 Undefined array — A reference has been made to an array which has not been dimensioned.

50 FIELD overflow - The total of all fields in a **FIELD** statement is greater than the record size of the specified file.

52 Bad file number — A reference has been made to a file number which is not associated with any open file.

53 File not found — A reference has been made to a file that does not exist. Check to make sure the file is in the specified directory.

54 Bad file mode — An invalid mode specifier has been given in an **OPEN** statement, or an invalid I/O statement has been attempted on a file opened for a certain mode, e.g. an **INPUT** statement on a file opened **FOR OUTPUT**.

55 File already open — An attempt has been made to open a file which is already in use.

57 Device I/O error — An error has occurred in the system I/O utilities.

58 File already exists — An attempt has been made to create a file which already exists.

59 ProDOS error — An error has occurred in the Apple IIGS ProDOS which is unexpected and could not be processed by AC/BASIC. Consult the *Apple IIGS ProDOS 16 Reference Manual* for more details.

61 Disk full — There is no more room to write to the output disk.

62 Input past end — An attempt has been made to read past the end of a file.

63 Bad record number — A reference has been made to a record number of zero or greater than the maximum permissible record number.

64 Bad file name — An illegal file name has been given.

67 Too many files — There are too many files open simultaneously.

68 Device unavailable — A reference has been made to a device which is not currently available to the system.

69 Internal error — This error should not occur. If it does, try to determine which statement causes this error and then call the Absoft Technical Support number printed on the disk label.

70 Disk write protected — The write-protect notch or hole is open meaning that the disk data can not be changed. Either close the write-protect hole if you want to write to the disk, or do not try writing to the disk.

74 Unknown volume — The volume name referenced can not be found among the devices available to the Apple IIGS. The first part of a filename prefix is the volume name. This error probably means that a floppy disk was ejected but is still being used by the AC/BASIC program.

80 Binary number in decimal operation — Binary numbers can not be used in a math operation with decimal numbers since the internal formats are different. Use the conversion routines in Chapter 12.

117 Illegal ordering — The variable must appear after its use in a **COMMON** statement.

129 Illegal co-processor call — This error should not occur. If it does, try to determine which statement causes this error and then call the Absoft Technical Support number printed on the disk label.

132 Subprogram overflow — Too many subprogram calls with too many arguments were made. Try to reduce the number of nested subprogram calls or reduce the number of arguments in the subprogram calls.

134 Array too large — The maximum size of a single array is 65536 bytes.

139 Stack error — The internal AC/BASIC stack has been corrupted. Try to determine which statement causes this error and then call the Absoft Technical Support number printed on the disk label.

143 Can not convert — The conversion to binary numbers while using decimal math is illegal.

144 Too many files opened — Apple IIGS ProDOS currently allows a maximum of eight files to be opened at once.

146 RESUME without error — A **RESUME** statement was encountered for which no error was trapped (**ON ERROR GOTO**).

147 RESUME error — Can not find the statement for the **RESUME NEXT** operation. This can only occur if both the "C" and "N" compiler options are off which means there is no internal indication as to where the next statement begins. Turn on the "N" option when using error trapping.

148 Stack error — The internal AC/BASIC stack has been corrupted. Try to determine which statement causes this error and then call the Absoft Technical Support number printed on the disk label.

150 to 153 Internal consistency checks — The library files `BASIC.1L` and `BASIC.2L` are checked for correctness when used. If this error occurs, re-copy these two library files from the original compiler disk.

154 Graphics array invalid — The array-based graphic image which **PUT** is trying to use is invalid. A probable cause would be that a portion of the array was over-written.

155 Resolution error — A graphic picture or recorded from the 320x200 screen can not be displayed on the 640x200 screen and vise-versa.

156 PICTURE string format error — The string which contains a picture recorded by **PICTURE** is invalid. A probable cause would be that a portion of the string was over-written.

157 Out of string space — All the text used for buttons, window titles, and menus can not exceed 4096 characters total.

158 EDIT FIELD error — Can not create the edit field. Most likely, the field size is too small on the screen to hold the text.

159 SOUND error — Notes can not be added to an existing **REPEAT** sequence. First stop the sequence with **SOUND STOP**.

All error values between 1 and 255 that are not documented above are either reserved for future implementation or are internal error checks which should not occur. Try to determine which statement produces the error and then call the Absoft Technical Support number printed on the disk label.

AC/BASIC Reference Manual

APPENDIX G

COMPILE-TIME ERROR MESSAGES

Alert Box Errors

Some compile-time errors refer to file I/O errors or internal compiler problems rather than to specific program lines. These appear in alert boxes:

`Can not Create List File or Temporary Files` — The list file or one of the temporary files created by the compiler can not be written to the disk. This is probably due to unavailable space on the disk or a write-protected disk.

`Can not Open Source File` — The program to be compiled can not be found on the disk or can not be read because it is damaged.

`Failure Creating File` — The compiled program file can not be written to the disk. This is probably due to unavailable space on the disk or a write-protected disk.

`Label/Symbol Table Overflow` — The compiler can store up to 64K of label and variable names. If this error occurs, shorten some of the label and variable name or divide the program into two parts which may be CHAINed.

Program Errors

This is a comprehensive alphabetical list of the AC/BASIC compile-time errors.

Array name was used before with another type — An array may only have one type for all its elements.

Array or subprogram with wrong number of arguments — An array reference or subprogram call has fewer or more arguments than the declaration.

Binary number occurring in decimal operations — A variable assigned in binary math can not be referenced as a decimal-math number. To fix, use only binary math or only decimal math in a program, or use the proper numerical I/O conversions explained in Chapter 11.

CALL statement without subprogram name or missing () — When calling a subprogram, the name or parentheses are missing. To fix, put in the name of the subprogram or put in the required parentheses when using the **CALL** statement.

Can not declare a SUB within a SUB — Subprogram declarations can not be nested.

Can not open Include file — The compiler is unable to open the specified **$INCLUDE** file. The problem may be a disk read error or the file is not found in the specified folder.

Can not write work file — The compiler creates temporary work files to store intermediate data while compiling. One of these could not be written; probably due to a full disk, locked disk, or a disk write error.

Can't have Include metacommand in an Include file — Include files can not be nested. Only the main program file can have include statements.

Compiler error — There is an internal error in the compiler. Try to determine which statement causes the compiler error, then please call the Absoft technical support phone number on the compiler disk.

DEF statement syntax error — One of various syntax errors in a specification statement (**DEFINT**, **DEFSNG**, **DEFDBL**, **DEFSTR**, **DEF FN**).

Division by 0 — Occurs when zero is used as the denominator for a division operation, or when zero is raised to a negative exponent.

Double use of variable as function argument — A variable can only be used once as an argument in a user-defined function declaration. For example, **DEF FN**`a(x,x)=x+2`.

Duplicate definition — An attempt was made to redimension an array without erasing it first or a redimensioned array from a **COMMON** block didn't match the original array..

Duplicate label — Same line number or label previously defined in the same program.

Duplicate subprogram name — A subprogram name appeared in two **SUB** statements in the same program.

ELSE or END IF without IF — There are too many **ELSE** or **END IF** statements which do not have corresponding **IF** statements.

END SUB missing — The subprogram must contain the statement **END SUB** at the end of the subprogram.

END SUB while not in subprogram — The statement **END SUB** can not appear in the main program.

Exceeding allocated stack space — The stack space allocated for use with user-defined functions is limited to 256 bytes. This means that any user-defined function can not have more than 256 bytes worth of parameters: 32 double-precision, 64 single-precision, or 128 integer parameters.

Expecting equals sign — In a case such as `A(I,J)=expression` the equals sign (=) is missing or expected.

Expecting further text in this statement — The statement needs more information past the keyword such as a variable name or a parameter.

Expression of the form operand-NOT-operand — The **NOT** operator is unary meaning that it one has one operand which follows the **NOT**.

Function not defined previously — An attempt was made to use a user-defined function which has not yet been declared.

Illegal argument in SUB statement — An invalid argument such as a constant can not appear as an argument in the definition of a subprogram.

Illegal assignment statement — An assignment statement is expected which contains an equals sign. This may be caused from having an illegal character after a variable name.

Illegal character in number — A number must contain only digits, a decimal point, and an optional signed exponent "E" for exponentiation.

Illegal function argument in function definition — An invalid argument such as a constant can not appear in a function declaration.

Illegal function call — A call to a function has been attempted with too many arguments or arguments not within the expected range.

Illegal occurrence coordinate — A coordinate can only be used in a graphics drawing statement in the proper statement position.

Illegal statement in subprogram — A statement such as **DEF FN**, **COMMON**, **CLEAR**, **ERASE**, or **SUB** can not appear within a subprogram.

Illegal subprogram name — A subprogram name must begin with a letter and be followed by up to 39 letters, digits and periods.

Illegal use of reserved name — The reserved word can not be used in that context. Reserved words can not be used as variable names or labels. See Appendix C for a list of reserved words.

Illegal: ELSE statement IF — When the **ELSE** statement is used in a block-**IF** statement, the **ELSE** must be the only statement on the line.

Improperly nested single-line IF — Check the **IF** syntax in Chapter 10 to see how each **ELSE** must correspond to an **IF**.

Incorrect nesting: previous FOR needs NEXT — A **FOR** statement exists without a matching **NEXT**.

Incorrect nesting: previous IF needs END IF — An **IF** statement exists without a matching **END IF**.

Incorrect nesting: previous WHILE needs WEND — A **WHILE** statement exists without a matching **WEND**.

Items in the list must be variables or array elements — Only variable and single array elements are permitted as parameters to input statements. For example, the statement **INPUT** 5 can not assign a typed value to the constant 5.

Left side of an assignment must be a variable or array — To assign a value to a variable, place it on the left side of the equals sign (=). An expression and most functions can not be used for the result on the left side of an equals sign.

Line buffer overflow (more than 256 characters) — A BASIC line can not be more than 256 characters in length. To fix, divide the statements into smaller lines.

Main program must precede all subprograms — All subprograms must appear after the main program. The utility program SORTSUBS is included on the compiler disk which will rearrange a program so that all the subprograms are at the end.

Memory overflow: too many variables — The program has too many variables, use fewer. This error should not occur since there can be up to 65536 variables.

Missing array name (UBOUND/LBOUND) — The UBOUND and LBOUND functions need an array name for a parameter.

Missing left parentheses — A left parentheses is assumed to be missing from the expression.

Missing operand — An operand is expected in a case such as "A=B+".

Missing operator such as + or AND — A simple operator is expected. This occurs when two variable names appear next to each other without an operator in between.

Missing or wrong keyword — A keyword is missing or invalid. Check the statement syntax definition for the appropriate keyword(s).

Missing right parentheses — A right parentheses is assumed to be missing from the expression.

Missing separator such as comma — A comma is expected.

Nesting depth WHILE, FOR, IF overflow — The nesting level of **WHILE**, **FOR**, and **IF** expressions is 16 levels.

NEXT without FOR — A **NEXT** statement was encountered without a previous **FOR** statement.

No statements allowed beyond END IF — The **END IF** statement must appear on a line by itself.

One or more of the following missing: WEND, NEXT, END IF — A **WHILE**, **FOR** or **IF** structure needs its companion terminator.

Only addition is a legal operator for strings — An arithmetic operation other than addition (+) was used on string variables. The only operation that may be used on strings is concatenation (+).

Only dynamic arrays can be erased — When an array is declared with **DIM STATIC**, the array can not be erased. Even though AC/BASIC on the Apple IIGS does not support static arrays, this error is included for compatibility reasons.

Overflow — A numeric operation has been attempted that returned a number too large or too small for the machine to represent, or to assign to a destination data type (e.g., A%=40000).

Program exceeding 64 kb — The length of the main program can not be larger than 65536 bytes in length. If this occurs, you can break your program into smaller units and use **CHAIN** or **RUN** to connect them. Or, you may use the **SUB...SHARED** subprogram declaration to move large sections of the main program into a subprogram which can access and use all the variables from the main program.

Spelling error or name not array name — A number has a "spelling" error such as letters or an array name is expected in the statement.

Statement legal only in subprogram — The **SHARED** and **EXIT SUB** statements may only be used in subprograms.

Statement or function with wrong number of arguments — The number of arguments specified does not match the number in the function definition or statement syntax.

String formula too complex — The compiler can not handle the multiple string operations. To fix, break the string operations into smaller separate ones.

Syntax error — Indicates an error in the statement grammar or in the format of a statement or function being used.

Syntax error: probably error in rectangle designation — Check the rectangle syntax which should be `(x1,y1)-(x2,y2)`.

TO keyword missing in FOR statement — The **FOR** statement must have the **TO** keyword.

Too many array dimensions — There are too many dimensions in an array declaration or reference. To fix, make arrays with fewer dimensions.

Too many NOT operators — The expression has too many **NOT** statements. To fix, simplify the expression with fewer **NOT**s or divide into two expressions.

Too many WHILE, FOR, IF statements — There can be approximately 256 **WHILE**, **FOR**, and **IF** structures in a single nesting level. To fix, move some of the structures to a subprogram or to a higher nesting level by inserting an "always true" **IF** statement

Too much code generated by one statement. Simplify — The statement is too complex for the compiler. To fix, divide the statement into two or more functionally equivalent parts.

Type mismatch — The wrong type of variable was passed to a function or subprogram.

Undefined array — This statement requires an array that already exists. Generated by **GET**, **PUT**, and **WAVE** statements.

Undefined label or function (FN) — A label referred to in a **GOTO** or **GOSUB** statement or function reference (FN) does not exist in the program.

Undefined subprogram — A subprogram with the specified name does not appear in this program.

Variable already defined before in subprogram — A variable occurring in a **SHARED** statement must not be referenced or assigned before its declaration in the **SHARED** statement.

Variable must appear after COMMON statement — Variables defined in a **COMMON** statement must only be referenced after their definition. Put **COMMON** statements first in a main or subprogram.

Variable or array not found in main program — A variable used in a **SHARED** statement within a subprogram can not be found in the main program.

WEND without WHILE — A **WEND** statement was encountered without a previous **WHILE** statement.

AC/BASIC Reference Manual

APPENDIX H

AC/BASIC INSTALLATION GUIDE

Chapter 2 of this manual showed you how to make a backup of the AC/BASIC compiler disk and use the compiler from the Launcher. As shipped, the compiler disk is configured for ease of use on either a one-drive or two-drive Apple IIGS. This appendix presents other methods of installing AC/BASIC that may be better, depending on your equipment and needs.

This appendix includes an overview of all the files on the AC/BASIC compiler disk and it includes three separate installation guides for:

- computers with one or two floppy-disk drives.
- computers with a floppy-disk drive and a hard disk.
- computers with a megabyte or more of memory and want to use the built-in RAM disk.

Files on the AC/BASIC Compiler Disk

The root directory of the AC/BASIC compiler disk contains the following file and folders:

PRODOS — This is part of the Apple IIGS ProDOS licensed from Apple Corporation.

SYSTEM — Contains the files licensed from Apple Corporation which represent a "minimum" System Disk version 3.1. The Toolboxes are in this folder.

BASIC — Contains the AC/BASIC compiler, the folder of overlay libraries and editor files, and the HELLO program. These are the three filenames:

```
AC.BASIC
BASIC.OVERLAYS
HELLO
```

EXAMPLES Contains the example programs from the chapters and appendices in the manual. Each chapter and appendix which contains examples has its own folder

Copying Files

In this appendix, you need to copy files and folders. This can be done using the Desktop or Finder by dragging a file to the place where you want it. For example, to copy the EXAMPLES folder to another disk on a two-drive system, drag the folder to the icon of the other disk and release the mouse button. Please refer to the *Apple IIGS System Disk User's Guide* for instructions on how to copy a file, format a disk, and manipulate windows.

One or Two Disk Installation

This section describes how to install AC/BASIC on a single disk for use with an Apple IIGS that has either one or two floppy disk drives.

The AC/BASIC compiler disk is already configured to be used from the Launcher on a one-drive system. The disk contains a subset of the System Disk; enough to boot the computer and use the compiler.

There is not enough free space on the compiler disk to compile more than just a few of the example programs. Therefore, you may want to copy the EXAMPLES folder to a blank formatted disk and then delete it from the compiler disk. (Remember, only use the compiler disk backup!)

For users with two disk drives, you may want to put just the BASIC folder onto a blank disk which doesn't have a SYSTEM folder. You can then use your own system disk to boot your computer. AC/BASIC requires System Disk version 2.1 or later. To get the most recent version of the System Disk, see your Apple dealer.

For users with one disk drive, you should copy the BASIC folder to your own stripped-down system disk which must be version 2.1 or later. These folders and files can be deleted from a system disk to free space for the compiler:

BASIC.SYSTEM	(file)
SYS.UTILS	(directory of system utilities)
BASIC.LAUNCHER	(file)
SYSTEM/P8	(file for ProDOS 8)
SYSTEM/DESK.ACCS	(folder of desk accessories)
SYSTEM/FONTS/*file*	(all files in the FONTS folder, but *not* the folder itself)

Hard Disk Drive Installation

This section describes how to install the AC/BASIC compiler onto an Apple IIGS system equiped with a hard-disk drive.

Most likely, you will want to create a new folder on your hard disk drive which will contain the AC/BASIC compiler, overlay libraries, and the editor files. You can simply copy the BASIC folder from the compiler disk to your hard disk.

Since you will always be booting the computer from the hard disk, the AC/BASIC library and editor files may be "hidden" in the SYSTEM folder of the hard disk by moving the folder BASIC.OVERLAYS into the SYSTEM folder. The compiler and the run-time applications always look in the SYSTEM folder of the boot disk for the libraries. Thus, only one copy of the libraries needs to be on your hard disk. The complete search path for the BASIC overlays and the editor is in Chapter 4 in the description of the "R" option.

AC/BASIC requires the installation of System Disk version 2.1 or later on your hard disk drive. To get the most recent version of the System Disk, see your Apple dealer.

RAM Disk Installation

This section describes how to install and use the AC/BASIC compiler on the built-in Apple IIGS RAM disk. Using the RAM disk on an Apple IIGS, which has at least one megabyte of memory, greatly speeds all disk operations by using a block of memory as if were another disk drive.

First, the RAM disk must be set-up from the Apple IIGS Control Panel. To do this use the three-key combination Ó-CONTROL-ESC and select the Control Panel menu by using the up arrow and pressing the RETURN key when it is highlighted. On the Control Panel menu, select RAM Disk and press RETURN. With the left and right arrows, choose a maximum RAM disk size of at least 256K and a minimum size of 32K, then press RETURN. Select QUIT and press RETURN twice. You should be back to the point before you had entered the Control Panel. Now, for the changes to take effect, you must turn the computer off and back on. A RAM-disk icon should always appear on the desktop.

The following files should be copied from the AC/BASIC compiler disk or the hard disk to the RAM disk. This has to be done every time the computer is turned on since the content of the RAM disk is erased when the power is turned off.

```
AC.BASIC
BASIC.OVERLAYS          (folder)
```

In addition, copy your BASIC source programs to the RAM disk. When finished editing and compiling your program, copy the program and the compiled application back to a floppy disk or the hard drive since the content of the RAM disk will be lost after turning the power off.

To run the compiler, launch it from the RAM disk rather than from a floppy disk or the hard disk

APPENDIX I

APPLE IIGS TOOLBOX ROUTINES

Much of the power of the Apple IIGS is contained in the QuickDraw Toolbox which is stored in Read-Only Memory (ROM) and on the System Disk. This appendix describes how many of the useful QuickDraw routines may be called to enhance text fonts and styles, graphics, and other Apple IIGS features.

The QuickDraw routines are accessed using **CALL** as if they are built-in subprograms. Either syntax may be used:

```
CALL routine-name (argument1,argument2,...)
routine-name argument1,argument2 ...
```

The routine-name is usually the same name used in the Toolbox; however, in some cases a different name is used for clarity. The rest of this appendix is used to describe each routine.

Text Routines

Various text attributes may be changed using these QuickDraw Toolbox routines. When a text attribute is changed, it affects all subsequent AC/BASIC text operations like **PRINT** and **INPUT** and even the error messages.

TEXTBCOLOR `color-number`

Sets the background text pen color for the text. The initial text background color is white. The possible values for `color-number` are the palette entries for the current screen:

320x200 screen			
0	black	8	beige
1	dark gray	9	yellow
2	brown	10	green
3	purple	11	light blue
4	blue	12	lilac
5	dark green	13	periwinkle blue
6	orange	14	light gray
7	red	15	white

640x200 screen	
0	black
1	purple
2	green
3	white

These colors can be changed by using the **PALETTE** statement. The four colors on the 640x200 screen are a combination of the colors in the four mini-palettes as described in the section "Understanding the Colors" in Chapter 12.

The background text pen color may also be changed with the **COLOR** statement described in Chapter 12.

TEXTCOLOR `color-number`

Sets the foreground text pen color of the displayed text. The values for `color-number` are the same as for **TEXTBCOLOR**. For example, to change the current text appearance to be blue letters on a yellow background on the 320x200 screen use,

```
TEXTBCOLOR 9
TEXTCOLOR 4
```

The foreground text pen color may also be changed with the **COLOR** statement described in Chapter 12.

TEXTFACE `style-number`

The style of the displayed text is set by **TEXTFACE**. The `style-number` is calculated by adding the numbers for the desired styles from the table below. For example, if you want text which will have a bold face and be underlined, use 1+4=5.

style-number	Description
0	Plain Text
1	**Bold**
2	*Italic*
4	Underline
8	Outline
16	Shadow

Note that there is a problem with some Apple IIGS machines in that underlined text is not underlined. See your Apple dealer for the latest System Disk and ROMs which should correct this problem

TEXTFONT `font-number`

Sets the font to one of the available fonts contained in the `SYSTEM/FONTS` folder or in ROM. The `font-number` may be any number from -32768 to 32767, but only when a font with that number exists will the text font change. Do not use font number -1.

font-number	Description	Best point size(s)
-2	System Font	8
0	System Font	8
3	Geneva	10,12
5	Venice	14
20	Times	10,12
21	Helvetica	10,12
22	Courier	10,12

This example displays the six fonts on the system disk. For all the fonts to appear, they must be in the `SYSTEM/FONTS` folder on the boot disk.

```
FOR x=1 TO 6
   READ font$,number,size
   TEXTSIZE size
   TEXTFONT number
   PRINT font$,number,"Size:"size
NEXT x
DATA "System",0,8,"Geneva",5,12
DATA "Venice",5,14,"Times",20,12
DATA "Helvetica",21,12,"Courier",22,12
```

TEXTMODE mode-number

Normally, text is drawn on top of any existing graphics in a window. Using **TEXTMODE**, text may be graphically combined with existing graphics on the window. The following table show the possible value for mode-number.

mode-number	Description
0	PSET
1	OR
2	XOR
3	BIC
4	Inverted PSET
5	Inverted OR
6	Inverted XOR
7	Inverted BIC

The PSET, OR, and XOR modes function the same as described in the **PUT** statement in chapter 12. The initial text mode is 0 which draws the text on top of any existing graphics. The BIC mode is the "Black Is Changed" which places the foreground text onto the window in the background color.

The four inverted modes invert the text before displaying it on the screen using one of the modes.

Mode-number should be an integer value. Do not use the keywords "OR", "AND", etc. for the mode.

For example,

```
TEXTMODE 2     ' Exclusive-OR mode
```

TEXTSIZE size

Set the point size of the text. There are 72 points in an inch. For example,

```
TEXTSIZE 36
```

will set the text height to be about half and inch. The actual size of the text will vary from device to device. Fonts are usually only defined for certain point sizes (i.e. 10 and 12). When a different point size is specified, the text is scaled accordingly. Values from 1 to 255 are valid.

Initially, the text size is 8 point on the Apple IIGS.

Color Routines

These two routines set the background and foreground colors of the graphics pen. The colors may also be set with the **COLOR** statement described in Chapter 12.

BACKCOLOR color-number

This call sets the background graphics pen color which is used by the **CLS** statement and the **ERASE**-object Toolbox calls and when a window is scrolled by **SCROLL** or **PRINT**. This example clears the screen with the color 14:

```
SCREEN 1        ' Use 320x200 screen
BACKCOLOR 14    ' Set background color to light gray
CLS             ' Clear the screen with light gray
BACKCOLOR 15    ' Reset the background color to white
```

FORECOLOR color-number

This call sets the foreground graphics pen color which is used by the object drawing QuickDraw calls.like **PAINTRECT**.

Graphics Pen Routines

These routines may be used to change the pen which is used by the QuickDraw II Toolbox routines. AC/BASIC maintains its own graphics pen which is unaffected by these Toolbox routines. The AC/BASIC text pen, however, is affected.

GETPEN VARPTR(location%(0))

The vertical and horizontal locations of the graphic pen are returned as the first and second elements, respectively, of the integer array `location%`. For example,

```
DIM hold%(1)     ' Assume OPTION BASE 0
GETPEN VARPTR(hold%(0))
y = hold%(0)     ' Get vertical location
x = hold%(1)     ' Get horizontal location
```

Some AC/BASIC statements, like **PRINT**, will change the QuickDraw pen location

HIDEPEN

Hides the drawing pen so that any subsequent drawing will not appear on the screen. **SHOWPEN** will turn the drawing pen back on.

MOVE x,y

Moves the graphics pen location by adding x to the current horizontal value and y to the current vertical value. Thus, if the current pen location is (100,100),

```
MOVE 100,-40
```

will make the current pen location (200,60).

MOVETO `x,y`

Moves the graphics pen location to the point (x,y).

PENMODE `mode-number`

Sets the graphic pen drawing mode to the one specified by `mode-number` which must have a value 0 to 7. The eight drawing modes are described under the **TEXTMODE** section a couple pages earlier.

PENNORMAL

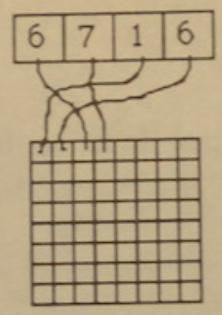
PENNORMAL returns the graphic pen to its original state:

- Solid pen pattern (black)
- Solid pen mask (all 1's)
- Single dot size (1,1)
- PSET drawing mode

BACKPAT VARPTR `(pattern-array(0))`
PENPAT VARPTR `(pattern-array(0))`

PENPAT sets the drawing pen pattern which is used by the **PAINT-** object drawing routines. **BACKPAT** sets the background pattern which is used by the **ERASE-** object drawing routines and the statements **CLS** and **SCROLL**. The `pattern-array` defines the color for each pixel in the 8 pixel by 8 pixel pattern. As described in the section "Understanding the Colors" in Chapter 12, a pixel color is defined by either a 2-bit or 4-bit palette entry value depending on the screen resolution.

When declared as an integer array, `pattern-array` must have at least 16 elements which occupy 32 bytes. Because the Apple IIGS has a byte-swapped microprocessor, the pixel mappings of the 320x200 screen to the pattern in hexadecimal are as shown where the first four pixels are dark green, orange, orange, and red represented by the integer `&H6716`:



The second word of a 320x200 pattern defines the next 4 pixels on the same line; there are two words per line.

The 640x200 screen has only one word per line since each pixel is represented by a 2-bit value 0 to 3. Once again, the bytes are swapped. The QuickDraw Toolbox uses all 32 bytes even on the 640x200 screen.

This example defines an 8 by 8 pattern of alternating light gray and black pixels for the background pattern. The foreground pattern is diagonally striped with blue and green:

```
REM AppendixI/Example1
SCREEN 1                    ' Use the 320x200 screen
DEFINT a-z                  ' Use integer arrays
DIM rect(3),back.pattern(15),fore.pattern(15)
FOR count=0 TO 15
   READ back.pattern(count)
NEXT count
FOR count=0 TO 15
   READ fore.pattern(count)
NEXT count
BACKPAT VARPTR(back.pattern(0))
PENPAT VARPTR(fore.pattern(0))
CLS
WHILE MOUSE(0)=0
   rect(0)=MOUSE(2)         ' Y coordinate
   rect(1)=MOUSE(1)         ' X coordinate
   rect(2)=rect(0)+32
   rect(3)=rect(1)+40
   PAINTOVAL VARPTR(rect(0))
WEND
```

```
DATA &HE0E0, &HE0E0      :REM Checkerboard pattern
DATA &H0E0E, &H0E0E
DATA &HE0E0, &HE0E0
DATA &H0E0E, &H0E0E
DATA &HE0E0, &HE0E0
DATA &H0E0E, &H0E0E
DATA &HE0E0, &HE0E0
DATA &H0E0E, &H0E0E

DATA &HAA44, &HAA44      :REM Diagonal pattern
DATA &H4AA4, &H4AA4
DATA &H44AA, &H44AA
DATA &HA44A, &HA44A
DATA &HAA44, &HAA44
DATA &H4AA4, &H4AA4
DATA &H44AA, &H44AA
DATA &HA44A, &HA44A
```

PENSIZE width,height

Sets the width and height of the graphic pen which is used for all drawing. The initial pen is one pixel wide and one pixel high. A common use for **PENSIZE** is to make the size of the pen match the aspect ratio of the screen:

```
PENSIZE 2,1    ' For 640x200 mode
```

SHOWPEN

Turns the drawing pen on after a **HIDEPEN** had turned it off. Initially, the drawing pen is on.

Line Drawing Routines

The two line drawing routines draw lines using the QuickDraw pen settings.

CALL LINE (x,y)

Draws a line in the current foreground color from the current graphics pen location to the point at a *displacement* of (x,y) in the current output window. For example,

```
LINE -5 10
```

will draw a line from the current pen location to a point 10 pixels below and 5 pixels to the left of the current pen location.

Note that the keyword **CALL** must be used along with parentheses to distinguish this ROM routine from the **LINE** statement.

LINETO x,y

Draws a line in the current foreground color from the current graphics pen location to the point specified by (x,y) in the current output window.

Object Drawing Routines

Five types of objects can be drawn using QuickDraw. There are also five drawing operations for each type of object. This leads to 25 QuickDraw Toolbox routines which draw objects in one way or another using the current pen settings.

Operation Syntax
ERASE
FILL
FRAME
INVERT
PAINT

Type of object	Syntax
Arcs	**ARC**
Ovals	**OVAL**
Polygons	**POLY**
Rectangles	**RECT**
Rounded rectangles	**ROUNDRECT**

The operation syntax and object syntax may be combined to form the name of the QuickDraw routine. For example, to paint an oval, use **PAINTOVAL**.

The operations are as follows:

ERASE Paints the object with the current background color and pattern.

FILL Fills the interior of the object with the specified pattern. All **FILL** routines require an additional pattern parameter which is identical to the pattern used for **PENPAT**. See the **PENPAT** for the pattern description.

FRAME Draws the boundary of the object with the current pen settings.

INVERT Inverts the pixels in the interior of the object.

PAINT Fills the interior of the object with the current pen pattern.

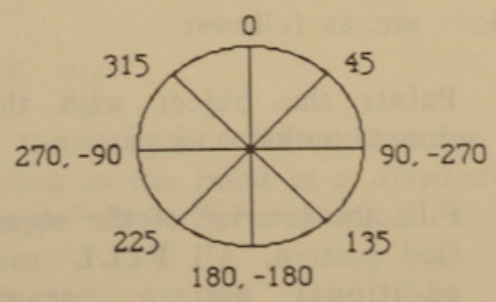
Most of the object drawing routines use an integer array `rect%` which defines a rectangle as follows:

Element	Description
rect%(0)	y coordinate of upper-left corner
rect%(1)	x coordinate of upper-left corner
rect%(2)	y coordinate of lower-right corner
rect%(3)	x coordinate of lower-right corner

ERASEARC VARPTR(rect%(0)),start,end
FILLARC VARPTR(rect%(0)),start,end,**VARPTR**(pat%(0))
FRAMEARC VARPTR(rect%(0)),start,end
INVERTARC VARPTR(rect%(0)),start,end
PAINTARC VARPTR(rect%(0)),start,end

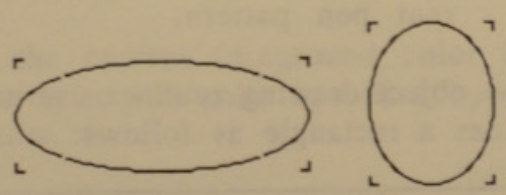
Draws an arc from angle `start` to angle `end` of the oval defined by the rectangle in the array `rect%`. **FRAMEARC** draws a true arc while the other routines use the wedge defined by the arc

an the center of the oval. The angle definition differs from the one used for the **CIRCLE** statement in Chapter 12. Each angle is specified in degrees and is offset as shown in the diagram:



ERASEOVAL VARPTR(rect%(0))
FILLOVAL VARPTR(rect%(0)),**VARPTR**(pat%(0))
FRAMEOVAL VARPTR(rect%(0))
INVERTOVAL VARPTR(rect%(0))
PAINTOVAL VARPTR(rect%(0))

Draws an oval whose edges are defined by the specified rectangle. The oval is drawn to fit in the rectangle as shown:



ERASEPOLY VARPTR(poly%(0))
FILLPOLY VARPTR(poly%(0)),**VARPTR**(pat%(0))
FRAMEPOLY VARPTR(poly%(0)
INVERTPOLY VARPTR(poly%(0))
PAINTPOLY VARPTR(poly%(0))

Draws a polygon using the specified integer array poly% with each element having the meanings described in the table (assuming an **OPTION BASE** of 0). The rectangle defined in the array is the rectangle enclosing the whole polygon. Note that the coordinates (y then x) are in the reverse order than usual.

Element	Description
0	Number of array elements times 2
1	y coordinate of upper-left rectangle corner
2	x coordinate of upper-left rectangle corner
3	y coordinate of lower-right rectangle corner
4	x coordinate of lower-right rectangle corner
5	y coordinate of first vertex
6	x coordinate of first vertex
.	.
.	.
n	y coordinate of last vertex
n+1	x coordinate of last vertex

The last coordinate should be the same as the first to complete the polygon. This is a requirement for **FRAMEPOLY**.

The array I/O statements described in Chapter 16 can be used to save the polygon array to disk.

ERASERECT VARPTR(rect%(0))
FILLRECT VARPTR(rect%(0)),**VARPTR**(pat%(0))
FRAMERECT VARPTR(rect%(0))
INVERTRECT VARPTR(rect%(0))
PAINTRECT VARPTR(rect%(0))

Draws a rectangle with the upper-left and lower-right corners specified in the integer array rect%.

ERASEROUNDRECT VARPTR(rect%(0)),width,height
FILLROUNDRECT VARPTR(rect%(0)),w,h,**VARPTR**(pat%(0))
FRAMEROUNDRECT VARPTR(rect%(0)),width,height
INVERTROUNDRECT VARPTR(rect%(0)),width,height
PAINTROUNDRECT VARPTR(rect%(0)),width,height

Draws a rectangle which has rounded corners with the upper-left and lower-right corners specified in the integer array rect%. The width and height are the horizontal and vertical radii of the oval used to draw the rounded corners.

Mouse Cursor Routines

HIDECURSOR

Hides the mouse cursor from view. To show the cursor again, call **SHOWCURSOR**.

INITCURSOR

This call resets the cursor to its initial state:

- Arrow cursor
- Shown

OBSCURECURSOR

Hides the mouse until the mouse location changes. **SHOWCURSOR** is not needed to show the mouse cursor; moving the mouse will show it.

SETCURSOR VARPTR(cursor%(0))

Sets the mouse cursor to a new image. The array `cursor%` is defined as follows:

cursor%(0)	cursor height in pixels
cursor%(1)	cursor width in words
cursor%(2)	first word of cursor image
.	
.	
cursor%(2+w*h)	first word of cursor mask
.	
.	
cursor%(2+2*w*h)	y coordinate of hot spot
cursor%(3+2*w*h)	x coordinate of hot spot

The w and h are the width and height. The height in `cursor%(0)` is the height of the cursor in screen pixels. You must derive the width in `cursor%(1)` from the following formula for 320x200:

```
cursor%(1)=INT((width in pixels + 7)/4)
```

or for 640x200:

```
cursor%(1)=INT((width in pixels + 15)/8)
```

The cursor image is defined with colors in much the same way that color appears on the screen. For a 320x200 screen cursor, every 4-bits represents a color pixel.

An example for the 320x200 screen is the best way to explain how the cursor is displayed. The following image and mask data will define a 4 by 4 pixel cursor which has a yellow border and a red center:

```
DATA &H6666, 0      :REM Image data
DATA &H8668, 0
DATA &H8668, 0
DATA &H6666, 0

DATA &HFFFF, 0      :REM Mask data
DATA &HFFFF, 0
DATA &HFFFF, 0
DATA &HFFFF, 0
```

The extra word of 0 is needed for the Toolbox call; every line of image and mask data must have an extra word which is 0.

The mask data which has all the bits set, will cause the color in the image data to be displayed directly on the screen using the palette `15-col` where `col` is the 4-bit color specified in the image data. If a pixel is defined with the image data and mask data equal to 0, then the pixel displayed is the same as the pixel on the screen — the screen pixel is unaffected. For example, this image/mask data defines a 4 by 4 pixel cursor which has a yellow border and a clear center:

```
DATA &H6666, 0      :REM Image data
DATA &H0660, 0
DATA &H0660, 0
DATA &H6666, 0
```

```
DATA &HFFFF, 0      :REM Mask data
DATA &H0FF0, 0
DATA &H0FF0, 0
DATA &HFFFF, 0
```

Each word is *byte-swapped* on the Apple IIGS. That is why a group of four pixels which appear on the screen with colors:

6	8	8	6

are defined by &H8668.

The following example creates a cursor which looks like the American flag:

```
REM AppendixI/Example2
SCREEN 1
DEFINT a-z
w=5                 ' 5 words wide
h=13                ' 13 stripes high
size=3+2*w*h        ' Size of array
DIM a(size)
FOR count=0 TO size
   READ a(count)
NEXT count
SETCURSOR VARPTR(a(0))      ' Set the new cursor
WHILE MOUSE(0)=0 : WEND     ' Wait for click
```

```
DATA 13          :REM Height -- 13 original colonies!
DATA 5           :REM Width
DATA &HBBBB,&H88BB,&H8888,&H8888,0  :REM B = blue,8 = red
DATA &HBBBB,&H00BB,&H0000,&H0000,0  :REM 0 = white
DATA &HBBBB,&H88BB,&H8888,&H8888,0
DATA &HBBBB,&H00BB,&H0000,&H0000,0
DATA &HBBBB,&H88BB,&H8888,&H8888,0
DATA &HBBBB,&H00BB,&H0000,&H0000,0
DATA &HBBBB,&H88BB,&H8888,&H8888,0
DATA &H0000,&H0000,&H0000,&H0000,0  :REM Full white stripe
DATA &H8888,&H8888,&H8888,&H8888,0  :REM Full red stripe
DATA &H0000,&H0000,&H0000,&H0000,0
DATA &H8888,&H8888,&H8888,&H8888,0
DATA &H0000,&H0000,&H0000,&H0000,0
DATA &H8888,&H8888,&H8888,&H8888,0  :REM 13th stripe

DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0  :REM Normal mask
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0
DATA &HFFFF,&HFFFF,&HFFFF,&HFFFF,0

DATA 12          :REM Y hot spot (lower-right)
DATA 15          :REM X hot spot
```

The array I/O statements described in Chapter 16 can be used to save the cursor array to disk.

SHOWCURSOR

Shows the mouse cursor which was hidden with HIDECURSOR.

AC/BASIC Reference Manual

APPENDIX J

BIBLIOGRAPHY

AC/BASIC Compiler Reference for Amiga, Absoft Corporation. (1987)

American National Standard for Minimal BASIC, X3.60-1978, ANSI, 1430 Broadway, New York, N.Y. 10018. (1978)

Amiga Basic, Rev C, Commodore-Amiga, Inc. and Microsoft Corporation. CBM (1985)

Apple IIGS Owner's Guide, Apple Computer, Inc. (1986)

Apple IIGS ProDOS 16 Reference, Apple Computer, Inc. (1987)

Apple IIGS System Disk User's Guide, Apple Computer, Inc. (1986)

Apple IIGS Toolbox Reference, Volumes 1 and 2, Apple Computer, Inc. (1987)

COMPUTER, A Proposed Standard for Binary Floating-Point Arithmetic, Draft 8.0 of IEEE Task P754, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 (1981)

David Eyes and Ron Lichty, *Programming the 65816*, Brady Communications Company, Inc. (1986)

Introduction to Amiga, Rev D, Commodore-Amiga, Inc., CBM (1986)

Microsoft BASIC (Macintosh), Microsoft Corporation (1984)

Microsoft BASIC Compiler User's Guide (Macintosh), Microsoft Corporation (1986)

Microsoft QuickBASIC (IBM), Microsoft Corporation (1987)

Tom R. Halfhill and Charles Brannon, *Advanced Amiga BASIC*, COMPUTE! Publications, Inc. (1986)

INDEX

Symbols

" 7-1
! (PRINT USING) 11-35
! 7-7
(PRINT USING) 11-36
7-7
$ 7-7
$$ (PRINT USING) 11-37
$ABOUT 4-11, 13-19
$IGNORE 4-11
$INCLUDE 4-9, 4-12
$PAGE 4-15
& (PRINT USING) 11-35
% 7-7
' 6-5, 11-59
* 8-3
** (PRINT USING) 11-37
**$ (PRINT USING) 11-38
+ (PRINT USING) 11-36
+ 8-3
, (PRINT USING) 11-37
- (PRINT USING) 11-37
- 8-3
. (PRINT USING) 11-36
. 7-4
/ 8-3
: 6-5, 6-8
< 8-3
<= 8-3
= 8-3
> 8-3
>= 8-3
? 11-28
\ 8-3
\\ (PRINT USING) 11-35
^ 8-3
^^^^ (PRINT USING) 11-38
_ (PRINT USING) 11-35

A

"A" option 4-3
about box 4-11
ABS 8-9
AC/BASIC 1-2, E-1, E-10
  Amiga specific E-2
  Apple IIGS specific E-3
  differences E-3, E-5
  installation H-1
  licensing 4-8
  Macintosh specific E-1
  RAM disk installation H-4
  versions of E-1
AND 8-8
Apple IIGS Owner's Guide 1-2
Apple IIGS System Disk User's Guide 1-2
Apple menu 4-11, 13-19
Apple menu logo 12-21
Applesoft interpreter 1-2
arcs 12-13
  conversion errors 8-5
  errors 8-7
  mixed-type 8-1, 8-4
  subexpression 8-5
arithmetic
array-based graphics 16-8
arrays 7-11, 7-14
  declaring 7-15
  erasing 7-16
  memory layout 12-26
ASC 9-3
ASCII 3-1
ASCII codes B-1
aspect ratio, circle 12-14
aspect ratio, screen E-11
assembly language D-1
ATN 8-10

B

BACKCOLOR I-5
BACKPAT I-7
backup, making a 2-1
BASIC 1-1
BASIC folder H-1
BASIC.1L 4-8
BASIC.2L 4-8
BASIC.OVERLAYS folder 4-7
batch compilation 5-1
baud rate 11-13
BCD A-1
  double-precision format A-4
  single-precision format A-4
BEEP 14-1, 14-2
bibliography J-1
binary numbers A-2
binary representation A-2
bit 12-32

BLOAD 16-7
block IF statements 10-13
Boolean 8-8
boxes 12-11
Bprogr.1f 3-9
BREAK ON/OFF/STOP 13-29
break, event trapping 13-29
BSAVE 16-8
BSEEK 16-9
BUTTON 13-14
byte swapping 16-10

C

"C" option 4-4, 13-2
CALL 10-11, 15-9
CALL LINE I-10
case sensitive 6-8
CDBL 8-6, 8-11
CHAIN 7-10, 10-20
character set 6-3, B-1
CHDIR 11-3
CHR$ 7-1, 9-4
CINT 8-11
CIRCLE 12-13, E-11
circles 12-13
CLEAR 16-4
CLIP: 11-12
CLOSE 11-14, 16-5
CLS 12-24, I-5
COLOR (statement) 12-22
color 12-17
    320x200 palette entries 12-33
    640x200 palette entries 12-35
    background 12-22
    dithering 12-34, 12-35
    foreground 12-22
    intensity 12-17
    master 320x200 table 12-19
    pixel 12-23
    redefinition 12-17
    tables 12-3
    striped 12-3
colors
colors, Macintosh E-12
column 11-39, 11-41, 11-42
COM1: 11-12, 11-58
commands 18-7
comments, program 6-5
COMMON 10-21
compiler options 4-1
    batch compile 5-1
compiling a program 3-7
COMPUTER magazine A-2
concatenation (+) 7-1
constant folding 7-3
constants 7-1
    exponential notation 7-2
    floating point 7-2
    hexadecimal notation 7-2
    integer 7-2
    numeric 7-2
    octal notation 7-3
    string 7-1
CONT 4-4, 13-2
Control Window 3-6, 4-2
control statements 10-1
conventions in manual 6-2
COS 8-12
CSNG 8-6
CSRLIN 11-40
CTRL-Q 3-11
CTRL-S 3-11
current palette 12-17
CVD 11-50
CVDBCD 11-53
CVI 11-50
CVS 11-50
CVSBCD 11-53

D

"D" option 4-5, 8-6
DATA 11-59, 11-61
data bits 11-13
DATE$ 9-5
decimal math 4-5
decimal numbers A-1
DEF FN 7-4, 7-20
DEFtype 7-7
Developers 4-8
DIALOG() 13-10
DIM 7-11, 7-15
directory 11-2
distribute 4-8
dithering 12-34, 12-35
double precision 7-6
Double-precision format A-3
duration (sound) 14-3

E

"E" option 4-9
EDIT FIELD 13-16
Edit (menu item) 3-4

EDIT$ 13-16
editor 17-1, 18-1
editor reference 18-1
ellipses 12-13
ELSE 10-11
END 10-24
END IF 10-13
END SUB 15-6
EOF 11-55
EQV 8-9
ERASE 7-16
ERASEARC I-11
ERASEOVAL I-12
ERASEPOLY I-12
ERASERECT I-13
ERASEROUNDRECT I-13
ERL 13-24
ERR 13-24
ERROR 13-27
error handling 13-23
error list, generating 4-9
error trapping 13-23
  compile-time 3-8, G-1
  run-time F-1
  Toolbox F-1
errors
errors, evaluation 8-7
evaluation errors 8-7
events 13-1
example of using the compiler 2-2
EXAMPLES folder H-2
EXIT SUB 15-7
EXP 8-12
exponent A-1
expressions 8-2

F

FastCopy 2-1
FIELD 11-43
File menu 3-3
file buffers 11-9
file names 11-2
file pointer 11-8
FILES (statement) 11-4
files 11-4
  buffers 11-9
  maximum open 11-12
  random access as sequential 11-9
  random-access 11-9
  random-access I/O 11-42
  sequential 11-8
  sequential input 11-15
  sequential output 11-24
FILES$ 11-5
FILLARC I-11
filled boxes 12-11
FILLOVAL I-12
FILLPOLY I-12
FILLRECT I-13
FILLROUNDRECT I-13
FIX 8-13
flat ASCII 3-1
FN 7-20
FOR...NEXT 10-16
FORECOLOR I-5
FRAMEARC I-11
FRAMEOVAL I-12
FRAMEPOLY I-12
FRAMERECT I-13
FRAMEROUNDRECT I-13
FRE 16-5
functions, math 8-1
functions, user-defined 7-4, 7-20

G

GET (files) 11-45
GET (graphics) 12-25, 12-29, 12-30
GETPEN I-6
GOSUB 10-3
GOTO 10-2
graphics pen 12-4

H

Hello demo program 2-2
help 18-11
HEX$ 9-6
HIDECURSOR I-14
HIDEPEN I-6
hierarchy, operation 8-3

I

"I" option 4-9
I/O utility functions 11-55
IEEE A-2
IF...GOTO 10-13
IF...THEN blocks 10-13
IF...THEN...ELSE 10-11
ImageWriter II 12-25
IMP 8-9
include files 4-12
INITCURSOR I-14

INKEY$ 11-21
INPUT # 11-17, 11-33
INPUT 11-15, 12-7
INPUT$ 11-22
installing AC/BASIC H-1
INSTR 9-7
instrument table 14-9
INT 8-14
integer 7-5
integer division (\) 8-3, 8-4
  DEFtype 7-9
interpreter differences
interval (sound) 14-3
INVERTARC I-11
INVERTOVAL I-12
INVERTPOLY I-12
INVERTRECT I-13
INVERTROUNDRECT I-13

K

keys 18-7
KILL 11-6
KYBD: 11-12, 11-56, 11-58

L

"L" option 4-9
label 6-4
label rules 6-8
labels 6-6, 6-8
LBOUND 7-18
LCOPY 12-24
LEFT$ 9-8
LEN 9-8
LET 7-13
licensing 4-8
LINE 12-11, E-12
LINE INPUT # 11-20
LINE INPUT 11-19, 12-7
line formats 6-4
line numbers 6-2, 6-6
line order 6-6
line width 11-24
lines 12-11
LINETO I-10
link run-time library 4-6
list files 4-8
listing (source) 4-9
LOC 11-58
LOCATE 11-39
LOF 11-56
LOG 8-15
logical operators 8-8
LPOS 11-42
LPRINT 11-26
LPRINT USING 11-34
LPT1: 11-12, 11-56
LSET 11-48

M

"M" option 4-6
MACHINE 16-6
machine language D-1
Macintosh BASIC E-1
macros 18-5
mantissa A-1
manual conventions 6-2
math functions 8-1
maximum array dimensions 7-11
maximum array elements 7-15
maximum array size 7-15
maximum file record 11-45
maximum line length 6-4, 7-2
maximum line number 6-6
maximum program size 15-2
maximum string length 7-2, 7-5
memory-related statements 16-1
MENU (statement) 13-18
MENU ON/OFF/STOP 13-21
menu bar 12-21
  custom 13-17
    event trapping 13-21
    initialization 13-19
menu
MENU() 13-19
metacommands 4-10, 6-5
Microsoft BASIC 12-7, E-1, E-10
Microsoft BASIC Interpreter 6-1
MID$ 9-9, 11-44
mixed-type arithmetic 8-1, 8-4
MKD$ 11-49, 11-53
MKDBCD$ 11-51
MKI$ 11-49
MKS$ 11-49, 11-53
MKSBCD$ 11-51
MOD 8-3
MOUSE ON/OFF/STOP 13-7
  double-click 13-4
  event trapping 13-7
  events 13-2
    pointer coordinates 13-5
  status 13-3
  triple-click 13-4

mouse
MOUSE() 13-3
MOUSE(0) 13-3
MOUSE(1) 13-5
MOUSE(2) 13-5
MOUSE(3) 13-5
MOUSE(4) 13-5
MOUSE(5) 13-5
MOVE I-6
MOVETO I-7
music production 14-3

N

"N" option 4-6, 13-2
NAME 11-7
New (menu item) 3-4
NOT 8-8
Note to Developers 4-8
note table (sound) 14-5
null strings 7-4
numeric range 7-3, A-4

O

OBSCURECURSOR I-14
OCT$ 9-10
ON BREAK GOSUB 13-29
ON ERROR GOTO 13-23
ON MENU GOSUB 13-21
ON TIMER GOSUB 13-28
ON...GOSUB 10-7
ON...GOTO 10-7
OPEN 11-10
Open (menu item) 3-4
OPTION BASE 7-15, 7-17
options 3-6, 4-1
OR 8-8
OR, graphics 12-28

P

page breaks 4-11, 4-14
PAINTARC I-11
PAINTOVAL I-12
PAINTPOLY I-12
PAINTRECT I-13
PAINTROUNDRECT I-13
PALETTE 12-17
parity 11-13
passes 3-9
path names 11-2
PEEK 16-1
PENMODE I-7
PENNORMAL I-7
PENPAT I-7
pens 12-4
  graphics 12-4
  text 12-4
PENSIZE I-9
PICTURE 12-29
PICTURE$ 12-29
pixel 11-31
pixel, screen E-11
POINT 12-23
POKE 16-2
POS 11-41
precedence, order of 8-3
precision of operands 8-4
PRESET 12-10
PRINT # 11-26
PRINT 11-26
PRINT USING 11-34
Print (menu item) 3-5
print zone 11-24
printer 11-12
printing a file 3-5
programming techniques E-10
PSET 12-10
PTAB 11-31
PUT (files) 11-46
PUT (graphics) 12-27, 12-30

Q

QuickDraw Toolbox D-5, I-1
Quit (menu item) 3-5

R

"R" option 4-6
radius, circle 12-13
RAM disk H-4
random-access files 11-9
random-access I/O 11-42
RANDOMIZE 8-16
range, numeric 7-3, A-4
READ 11-60, 11-61
reading program data 11-58
real numbers A-1
recursion 15-5
REM 6-5
rename files 11-7
reserved words C-1

RESET 16-5
resolution, screen 12-5
RESTORE 11-59, 11-61
RESUME 13-26
RETURN 10-6
RGB 12-17
RIGHT$ 9-11
RND 8-16
ROM routines I-1
row 11-39, 11-40
RSET 11-48
RUN 7-10, 10-23
run files 3-8
run-time events 4-4, 4-6

S

SCREEN E-12
SCREEN statement 12-5
screen dump 12-24
screen resolution 12-5, E-10
SCREEN() function 12-29
screens 12-1
SCRN: 11-12, 11-56
SCROLL 12-16
search path, run-time 4-7
semitone 14-3
sequential files 11-8
sequential input 11-15
sequential output 11-24
serial port 11-12
SETCURSOR I-14
SGN 8-18
SHARED 15-5, 15-7
SHOWCURSOR I-17
SHOWPEN I-9
Sieve of Eratosthenes D-1
SIN 8-18
single precision 7-5
single-precision format A-3
SORTSUBS 15-3
SOUND 14-1, 14-3
SOUND RESUME 14-3
SOUND voice,STOP 14-3
SOUND WAIT 14-3
SOUND(voice) 14-3
sound, Macintosh E-12
source listing 4-9
SPACE$ 9-12, 11-29
SPC 11-29
SQR 8-19
square E-11
STATIC 15-5
statistics, compilation 3-9
STOP 10-24
stop bits 11-13
STR$ 9-12
string literal 7-1
STRING$ 9-13, 11-29
strings 7-1, 7-4, 9-1
  comparison 9-2
  concatenation 9-1
  maximums 9-2
  relational operations 9-2
SUB / END SUB 15-3
SUB 15-2
subprogram names 15-3
subprograms 15-1
subroutines 10-3, 15-1
SWAP 7-14
SWAP BYTES 16-10
SYSTEM 10-24
SYSTEM folder H-1
system folder 4-7

T

TAB 11-30
TAN 8-20
text pen 12-4
TEXTBCOLOR I-1
TEXTCOLOR I-2
TEXTFACE I-3
TEXTFONT I-3
TEXTMODE I-4
TEXTSIZE I-5
TIME$ 9-14
TIMER ON/OFF/STOP 13-28
timer, event trapping 13-28
Toolbox routines I-1
Transfer (menu item) 3-5
tutorial 2-1

U

"U" option 4-8
UBOUND 7-19
UCASE$ 9-15
Unix 16-6
user-defined functions 7-4, 7-20

V

VAL 8-20
variables 7-1, 7-3

appropriate types 7-10
array 7-11
assigning 7-12
COMMON declaration 10-21
declaring 7-6
execution speed 7-10
names 7-4
numeric 7-5
sizes 7-15
string 7-4
types 7-4
VARPTR 16-3
voice (sound) 14-3
volume 14-3

W

WAVE 14-1, 14-3, 14-7
waveform 14-7
waveforms 16-9
WHILE...WEND 10-19
WIDTH 11-24
WIDTH LPRINT 11-24
WINDOW CLOSE 12-8
WINDOW E-10
WINDOW OUTPUT 12-7
WINDOW statement 12-6
creation 12-6
current output 12-2, 12-6, 12-7, 12-9
currently active 12-2, 12-6, 12-7, 12-9
default type 12-7
function parameters 12-9
gadgets picture 12-2
window
WINDOW() function 12-9, E-11
windows 12-1
WRITE # 11-32
WRITE 11-32

X

XOR 8-8
XOR, graphics 12-28

Z

zone width 11-24

Quick Keyword Reference

ABS	8-9
AND	8-8
ASC	9-3
ATN	8-10
BEEP	14-2
BLOAD	16-7
BSAVE	16-8
BSEEK	16-9
BUTTON	13-14
CALL	15-9
CDBL	8-11
CHAIN	10-20
CHDIR	11-3
CHR$	9-4
CINT	8-11
CIRCLE	12-13
CLEAR	16-4
CLOSE	11-14
CLS	12-24
COLOR	12-22
COMMON	10-21
COS	8-12
CSNG	8-11
CSRLIN	11-40
CVD	11-50
CVDBCD	11-53
CVI	11-50
CVS	11-50
CVSBCD	11-53
DATA	11-59
DATE$	9-5
DEF FN	7-20
DIALOG()	13-10
DIM	7-15
EDIT FIELD	13-16
EDIT$	13-16
END	10-24
END SUB	15-6
EOF	11-55
EQV	8-9
ERASE	7-16
ERL	13-24
ERR	13-24
ERROR	13-27
EXIT SUB	15-7
EXP	8-12
FIELD	11-43
FILES	11-4
FILES$	11-5
FIX	8-13
FOR	10-16
FRE	16-5
GET (graphics)	12-25
GET (files)	11-45
GOSUB	10-3
GOTO	10-2
HEX$	9-6
IF	10-11
IMP	8-9
INKEY$	11-21
INPUT	11-15
INPUT#	11-17
INPUT$	11-22
INSTR	9-7
INT	8-14
KILL	11-6
LBOUND	7-18
LCOPY	12-24
LEFT$	9-8
LEN	9-8
LET	7-13
LINE	12-11
LINE INPUT	11-19
LINE INPUT#	11-20
LOC	11-58
LOCATE	11-39
LOF	11-56
LOG	8-15
LPOS	11-42
LPRINT	11-26
LSET	11-48
MACHINE	16-6
MENU	13-18
MENU()	13-19
MID$	9-9
MKD$	11-49
MKDBCD$	11-51
MKI$	11-49
MKS$	11-49
MKSBCD$	11-51
MOUSE()	13-3
NAME	11-7
NEXT	10-16
NOT	8-8
OCT$	9-10
ON...ERROR	13-23
ON...BREAK	13-29
ON...DIALOG	13-13
ON...GOSUB	10-7
ON...GOTO	10-7
ON...MENU	13-21
ON...MOUSE	13-7
ON...TIMER	13-28
OPEN	11-10
OPTION BASE	7-17
OR	8-8
PALETTE	12-17
PEEK	16-1
PICTURE	12-29
POINT	12-23
POKE	16-2
POS	11-41
PRESET	12-10
PRINT	11-26
PRINT USING	11-34
PRINT#	11-26
PSET	12-10
PTAB	11-31
PUT (graphics)	12-27
PUT (files)	11-46
RANDOMIZE	8-16
READ	11-60
RESET	16-5
REST	11-48
RESTORE	11-61
RESUME	13-26
RETURN	10-6
RIGHT$	9-11
RND	8-16
RUN	10-23
SCREEN	12-5
SCREEN()	12-29
SCROLL	12-16
SGN	8-18
SHARED	15-7
SIN	8-18
SOUND	14-3
SPACE$	9-12
SPC	11-29
SQR	8-19
STOP	10-24
STR$	9-12
STRING$	9-13
SUB	15-2
SWAP	7-14
SWAP BYTES	16-10
SYSTEM	10-24
TAB	11-30
TAN	8-20
TIME$	9-14
UBOUND	7-19
UCASE$	9-15
VAL	8-20
VARPTR	16-3
WAVE	14,7
WEND	10-19
WHILE	10-19
WIDTH	11-24
WINDOW	12-6
WINDOW()	12-9
WRITE	11-32
WRITE#	11-32
XOR	8-8