Micol Advanced BASIC™

Structured Compiled Language System for the Apple IIGS

Version 4.0

Micol Advanced BASIC™

Structured Compiled Language System for the Apple IIGS

Version 4.0

Micol Systems Inc., 9 Lynch Road, Willowdale, Ontario, Canada M2J 2V6

Introduction

Limit Of Liability

While every precaution has been taken to ensure the correctness of the software and its accompanying manual, Micol Systems Inc. cannot assume any responsibility or liability for any damage or loss caused by our software. It is the responsibility of the user to make the necessary backups for the data and programs.

Apple Computer, Inc. makes no warranties, either express or implied, regarding the enclosed computer software package, its merchantability or its fitness for each 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.

GS/OS is a copyrighted program of Apple Computer, Inc. licensed to Micol Systems Inc. to distribute for use only in combination with Micol Advanced BASIC (GS version). Apple software shall not be copied onto another diskette (except for archival purposes) or into memory unless as part of the execution of Micol Advanced BASIC. When Micol Advanced BASIC (GS version) has completed execution, Apple software shall not be used in any other program.

Product Revision

Micol Systems Inc. reserves the right to make improvements to this software and manual at any time without notice.

The text file **INFO.DOC** on the **/MAB.SUPPORT** disk contains the latest information about this product which could not be included in the manual at the time of publication. Be sure to read this file into the editor for up-to-date information.

Copyright Notice

Published in Canada.

ISBN 0-921270-04-6

Software: Micol Systems Inc., Willowdale, Ontario

Documentation: Micol Systems Inc., Willowdale, Ontario and Redaction Electronique Enr., St-Hyacinthe, Quebec

FIRST EDITION, July 1988.

SECOND EDITION, revised, corrected, and enlarged.
First printing, February, 1992
Third printing, June, 1992

Table of Contents

Introduction

Part One: Overview of the Language

Part Two: The Programming Environment

Appendices

Part One: Overview of the Language

Chapter One

General Review

Comments on the Second Edition

We are proud to present the Second Edition of the *Micol Advanced BASIC* for the Apple IIGS reference manual. This manual has been completely reorganized to make it easier for everyone, especially the novice, to use.

If you are one of those who owns a First Edition copy of the manual, take the time to carefully look at the table of contents and the index to see where the changes were made. The table of contents and the index have been greatly expanded to make it easier for you to find the information you are looking for.

Take the time to read the manual through. You will find many programming tips written by people who have discovered and are already enjoying the power of the *Micol Advanced BASIC* Structured Language.

This reference manual has program examples throughout the entire manual. We recommend you study these program examples very carefully. You may also wish to compile and execute some of the more important ones. This way the explanations will become clearer to you and you will get practice in programming.

Send us your suggestions, comments and criticisms. We read all the letters we receive, even if we cannot reply to all of them. We will answer you if you include a self-addressed envelope with your letter.

Overview

The purpose of Part One is to give an overall look at *Micol Advanced BASIC* so you will get a general idea of what this language system has to offer.

Micol Advanced BASIC is a full-featured, compiled language system. Its purpose is to let you develop structured BASIC language programs for your Apple IIGS.

The BASIC program is created using the full-screen editor. Communication with the GS/OS operating system is done by means of the Command Shell. The Compiler and Linker translate BASIC source code into binary instructions which the microprocessor can directly execute.

Some Advantages of the Language

Micol Advanced BASIC needs only 768K of random-access memory to function, and yet all its components (the Command Shell, the Editor, the Compiler/Linker, and the

Run-Time Library) remain in memory during development. While developing your programs, no long delay will occur for one of the components to be loaded from disk.

The executable load files created by *Micol Advanced BASIC* use a special fast load format and may only be loaded by the loading software supplied with this product.

Micol Advanced BASIC may be used to produce Classic Desk Accessory files just by using a utility program provided. Stand-alone *Micol Advanced BASIC* programs use a common library located in a specific folder on the stand-alone disk.

A Stand-alone *Micol Advanced BASIC* program may also be executed on an Apple IIGS connected to an AppleShare network. *Micol Advanced BASIC* can also produce S16 (OMF) files that may be launched with the GS Finder.

Source code files created with the IIe/IIc version of *Micol Advanced BASIC* are highly compatible with those created with the GS version; only a few changes are needed to use the full power of the IIGS version.

Micol Advanced BASIC can use all the memory available to your Apple IIGS and is written in assembly language, the fastest code possible on your computer. Little time is spent compiling or linking, giving you more time do to what you can do best... program.

The Components of the Language System

1. The Command Shell

The Command Shell (or Shell, for short) allows the user to interface with the rest of the language system. Through the Shell, for example, it is possible to see the contents of a disk, invoke the text editor, compile a program, etc.

The Shell also has the capability of accepting commands from a file on disk. Utilities written by the user may also be added to the Shell. Because of these utilities, the possibilities of tasks the Shell can perform are almost unlimited. The Shell has the following features:

- Easy to remember commands
- Full complement of filing commands
- Test of compi' d programs
- Commands executed in a Shell Batch program
- AutoExec batch file
- Uses commands written in BASIC
- Easy-to-read help screen.

2. The Source Code Editor

The Source Code Editor lets you create, and modify BASIC source code files. The editor has word-processor-like features to ease the maintenance and revision of the source code files. The editor can read standard TXT ($04) or SRC ($B0) type files. The editor has the following features:

- 80-column, full-screen editor
- 128 kilobyte buffer (large enough for a file with about 4000 lines of code)
- Word-processor-like commands
- Fast and easy copy/movement of text
- Saves source code files in normal ASCII format
- Decimal to hex (and back) converter
- Easy-to-read help screen

3. The Full-featured Compiler and Linker

The Compiler reads the source code created using the source code Editor and generates an object code file which the Linker will convert to a machine usable format. The Compiler has the following features:

- Rapidly generates 65816 code
- Uses FastLoad to bring programs into memory fast
- Easy-to-remember Compiler Directives
- Ultra fast screen displays
- Support of source code libraries
- Link to assembly language programs
- Easy creation of large programs
- Easy creation of startup disk
- Utility to create Classic Desk Accessories

4. Full-featured Structured BASIC Language

With *Micol Advanced BASIC*, you can write programs that are more understandable than almost any other BASIC language. The use of meaningful variable names, indentation, structured loop control, improved data file handling, and many other features will make the creation of your programs a breeze. Now you can write those GOTOless programs that were impossible to do under Applesoft BASIC.

Micol Advanced BASIC can produce graphics and sounds that could never have been done before on an Apple II using Applesoft BASIC. Both Super High Resolution graphic modes (320 X 200 and 640 X 200) are supported with graphic text capability. *Micol Advanced BASIC* can also play back digitized noise, music or speech.

Micol Advanced BASIC gives you the ability to easily create applications that exploit multiple Windows and pull down Menus made famous with the Apple Macintosh™ computer. The *Micol Advanced BASIC* language systems offers the following features:

- Upward compatible with the Applesoft BASIC language
- Optional line numbers
- Dynamic character strings up to 1023 characters
- Simple variables and arrays of type boolean

- Ultra fast and sophisticated string manipulation
- True integer calculations (no conversion to real and back)
- Calculations of extended values: both integer and real
- MouseText character display
- **INKEY$** input and **PRINT USING** output
- **IF..THEN..ELSE, CASE_OF** conditional statements
- **REPEAT..UNTIL, WHILE..WEND** conditional loops
- Pascal language-like Functions and Procedures
- Support of recursive calls
- Low and Super High Resolution graphics
- Mixed text and graphics
- Great sound capabilities
- Complete and easy-to-use GS/OS file handling
- SuperTrace™ debugging command
- Easy Desktop program definition
- Complete and easy use of the Apple IIGS Toolbox
- Exclusive Controlled Uncertainty™

How this Manual is Organized

This manual is divided into eight distinct parts:

- First is the Copyright pages and Table of Contents. We have taken pains to make this Table of Contents as useful as possible. We hope you agree.
- Part One (this part) gives you a general overview of *Micol Advanced BASIC* (MAB), and how to use *Micol Advanced BASIC* with the usual equipment. There is a brief tutorial in Chapter Two all beginners should try.
- Part Two discusses the Programming Environment: what is needed to write and use a *Micol Advanced BASIC* program: Shell, Editor, Compiler/Linker, Library.
- Part Three is the most important section and describes the *Micol Advanced BASIC* language itself.
- Part Four describes the Desktop commands and some guidelines on how to write a Desktop-based program. This section should be ignored by beginning programmers as it is quiet involved.
- Part Five discusses the Apple IIGS ToolBox and how to access it from *Micol Advanced BASIC*. This section should be ignored by beginning programmers.
- Part Six discusses program management. Management includes debugging techniques, code segmentation, code optimization, and using assembly language routines with your *Micol Advanced BASIC* programs.
- Last come the Appendices, Glossary of words and Index. The Index is very complete, so if you have trouble finding something, feel free to consult it.

Special Note

> Special paragraphs marked "Programmers", "NOTE", "IMPORTANT", and "WARNING" will be contained within a paragraph such as this one. These paragraphs describe tricks of the trade, indicate some special things to watch out for or alert you to a potential dangerous situation. "Programmers" denotes advanced topics that novices may ignore.

The *Micol Advanced BASIC* System Disks

You have received with this product:

- The *Micol Advanced BASIC* GS Reference Manual, Second Edition
- One system disk labeled Master Disk.
- One system support disk labeled /MAB.SUPPORT
- A product registration card
- Information about the *Micol Advanced BASIC* Users Group (MABug)
- Other Product information

The first disk labeled Master Disk contains the *Micol Advanced BASIC* language system itself. The second disk labeled /MAB.SUPPORT holds folders containing example programs, utilities, tool sets, fonts, devices drivers, etc. normally unused with *Micol Advanced BASIC*.

IMPORTANT

> Please make backup copies of both system disks before starting your program development. Use the copied disks for your work and store the original disks somewhere safe.

The *Micol Advanced BASIC* language system consists essentially of four files (contained in folder MICOL.ADV.BASIC on the disk labeled Master Disk) and the Utilities folder: COMPILER.SHELL, EDITOR, LIBRARY, MICOL.ADV.BASIC and UTILITY/ folder. COMPILER.SHELL is the integrated Compiler, Shell and Linker. EDITOR is the source code Editor. LIBRARY is the run time Library and MICOL.ADV.BASIC is the Micol Loader, necessary to load stand-alone *Micol Advanced BASIC* programs.

The UTILITY folder will contain the external Shell commands you may write later to add more functionality to the Command Shell. The file AutoExec will tell you about any updates to the Language System or the Reference Manual.

The MAB.SUPPORT disk contains the following folders/files:

- folder Demo.Files

- folder MAB.TO.S16 (see Part Six, Chapter Five)
- folder MAB.TO.CDA (see Part Six, Chapter Five)
- folder SYSTEM (contains GS/OS files not needed in *Micol Advanced BASIC*)
- an optional text file named INFO.DOC

IMPORTANT

> The Demo.Files folder contains the source code of numerous example programs that could be very helpful in your understanding of this language. It is very important you look at these files in some detail. The fractal mountain generator is in the Fractal.Samples folder under the name Mt.Fractal. There is also a very nice demonstration game written in *Micol Advanced BASIC* in the folder MABug.DEMO (read the READ.ME file).

File INFO.DOC contains the latest product information which is not contained in this manual. If this file is absent, the manual is complete.

Micol Advanced BASIC also uses the Apple IIGS Tools in the proper folder. The other files or folders are used by the GS/OS operating system which *Micol Advanced BASIC* uses to communicate with your hardware.

What You Need to Know

Before you continue reading this manual, you should know:

- How to set up and use your Apple IIGS system (see the manuals that came with your computer)
- Some knowledge and understanding of the ProDOS file structure and use of Pathnames to access these files
- How to use GS/OS to manipulate disk files (see the Apple IIGS System Software User's Guide)
- Some knowledge of Applesoft BASIC or any other dialect of BASIC
- Advanced programmers should know about the Toolbox of the Apple IIGS.

Hardware Requirements

To use *Micol Advanced BASIC* for the Apple IIGS, you need one of the following computer systems:

- An Apple IIGS with ROM 01 (or later) and a minimum of 768K of RAM
- An Apple IIe with GS Upgrade and ROM 01 and a minimum of 768K of RAM

With:

- One 3.5 inch disk drive
- A monochrome monitor capable of displaying 80 columns
- GS/OS, the DOS required by *Micol Advanced BASIC*, is supplied on disk.

Suggested Additional Hardware

- A printer
- More memory on the expansion card (see below)
- A second or more disk drives
- A hard disk drive
- A color monitor

Run Time Memory Needs of Stand Alone Applications

- Text only application: .....512 kilobytes
- Text and picture files: .....768 kilobytes
- Text and animated graphics .....768 kilobytes
- Text, pictures, and sound files .....768 kilobytes
- Graphic text and picture files .....768 kilobytes
- Graphic text and animated graphics .....2048 kilobytes
- Graphic text, animated graphics, and sound files .....2048 kilobytes
- Simple Desktop programs (15-35K without Library) .....768 kilobytes
- Regular size Desktop programs (36-50K without Library) .....1024 kilobytes

The amount of memory may vary depending on the size of the program, the numbers of arrays, picture files, sound files, etc.

Setting up *Micol Advanced BASIC* on a Hard Drive

1. Boot a GS/OS System Disk which brings you into the GS Finder. The Finder has the facilities for this task.
2. Create a subdirectory called Micol.Adv.BASIC under the volume directory of your hard disk (choose New Folder under the File Menu or press <Apple>N).
3. Copy the *Micol Advanced BASIC* system files:
   a) Copy the files MICOL.ADV.BASIC, COMPILER.SHELL, EDITOR, LIBRARY, and the UTILITY folder from the Master Disk (folder Micol.Adv.BASIC) to the subdirectory Micol.Adv.BASIC you just created on your hard disk. Lock these files.
   b) Copy the file Micol.Icons from the Master Disk to the ICONS folder of your hard drive.
4. Put the original *Micol Advanced BASIC* disks away in your archive box.

Using *Micol Advanced BASIC* with the Finder

1. Open the Micol.Adv.BASIC folder by clicking on it twice or by clicking once and pressing Apple-O to open the folder.
2. Drag the Micol.Adv.BASIC icon onto the Desktop. Close the folder and Window. Dragging the icon onto the Desktop allows easier access to MAB.
3. Start the *Micol Advanced BASIC* language system by quickly clicking twice on the Micol.Adv.BASIC icon or by clicking once and pressing Apple-O to open the program.

Using *Micol Advanced BASIC* With a RAM Disk

Micol Advanced BASIC recognizes the RAM card created by the RAM Disk option of the Control Panel.

If *Micol Advanced BASIC* for the Apple IIGS detects a RAM disk with at least 192K of free space, it will use this RAM disk for its scratch work for compiling and linking. If there is no such RAM disk, the scratch work will be performed where the final linked program is created.

WARNING

> If *Micol Advanced BASIC* detects a RAM disk with a free space greater than 192K, it will use this RAM disk for its scratch files. These files are normally deleted at the end of compilation and/or linking. However, if an operating system error should happen during compilation or linking, these files will not be deleted. You should then delete these files yourself manually from the Shell, otherwise this RAM disk may not be used again.

Use of a RAM disk is highly recommended as it greatly speeds up the program development cycle: edit, compile/link, (execute), correct.

Using *Micol Advanced BASIC* With Your Printer

Micol Advanced BASIC allows you complete access to the Control Panel of the Apple IIGS. If you used the Control Panel's Printer or Modem Port controls to configure your printer, the printer should function properly because *Micol Advanced BASIC* uses the settings of the Control Panel. Refer to your Apple IIGS Owner's Manual to see how to change the settings.

Configuring Your Printer Using the Control Panel

You may use the Control Panel while working with *Micol Advanced BASIC* to alter the panels parameters. These parameters work with the built-in serial port or with a printer interface in the proper slot with the slot setting to "Your Card".

Because of the nature of laser printers, *Micol Advanced BASIC* will not work with them. If you are using the Network Version of *Micol Advanced BASIC*, the program output may be printed using the network spooler.

WARNING

> Some third party printers may need a device driver in the directory Drivers/; otherwise the printer may not function correctly. See your printer's manual if you are uncertain.

If You Need Assistance

Four good rules to follow are:

1. Don't panic. Take a deep breath and relax for a minute.
2. Go through the following checklist to delimit the problem
   a) See if you computer meets the minimum hardware requirements (see Hardware Requirements)
   b) Make certain that your hardware and peripherals are connected correctly and that all connections are secure. If a particular peripheral needs a device driver, make sure that it is installed on the boot disk
   c) Get your reference manual and consult
      — the Table of Contents and/or Index
      — find and read carefully the sections pertinent to your problem. More than sixty percent of all calls for technical support can be answered simply by reading the manual.
3. Ask a friend who has a computer to come and help you. Your friend may have enough experience to explain what you do not understand
4. Contact us at Micol Systems. You can communicate with us by mail or by phone. We provide free technical support to our registered customers:
   a) By mail, write to Micol Systems Inc. 9 Lynch Road, Willowdale, Ontario CANADA M2J 2V6. We will answer your letter by mail if you include a self-addressed envelope
      — Please include: a description of your hardware (computer brand and model, size of memory on expansion card), and the list of the peripherals in the computer
      — a complete listing (preferably on disk) of the program causing the problem. Determine where the problem is and clearly mark its location. If this is not done, we cannot help you.

b) By phone, call our office at (416) 495-6864. You can reach us during normal business hours Monday to Friday, 9:00 AM to 5:00 PM Eastern Time. There is no fee to pay except for the long distance call, if applicable. Sorry, we cannot accept collect calls.

Compatibility Overview

Applesoft BASIC

Micol Advanced BASIC is not a simple compiler of Applesoft BASIC programs and should not be thought of as such; it is much more than that. However, since *Micol Advanced BASIC* is a language system based upon Applesoft BASIC, you may convert your Applesoft BASIC programs to *Micol Advanced BASIC* programs with very little effort. Most programs written under Applesoft BASIC will run under *Micol Advanced BASIC* with modest changes. Please see Chapter Six, Part Six for more information.

You will have to modify the portions of code using:

- Disk filing
- Graphics
- Machine language routines
- Special memory locations (**PEEK**s and **POKE**s)
- Error handling
- Program segmentation

By making additional changes, you may take advantage of additional memory for programs or data, create better graphics and sounds, etc.

Micol Advanced BASIC for the Apple IIe/IIc

Micol Advanced BASIC for the Apple IIe and Apple IIc source code files are highly compatible with *Micol Advanced BASIC* for the Apple IIGS. You may use the same source files. Since you have a much greater programming space, you should be able to add the features you want in your programs.

You will have to modify the portions of code using:

- Graphics and Sound
- Machine language routines
- Special memory locations (**PEEK**s and **POKE**s)
- Error handling
- Program segmentation

NOTE

> Programs developed under the Apple IIe/c version must be recompiled under the Apple IIGS version.

Earlier Versions of MAB for the Apple IIGS

Most programs developed with *Micol Advanced BASIC* GS v2.0 to v3.7.2 are compatible with *Micol Advanced BASIC* v4.0. You may use the same source code. Of course, all programs developed with an earlier version of *Micol Advanced BASIC* for the Apple IIGS must be recompiled to execute under Version 4.0 of *Micol Advanced BASIC*. Please note that there has been a major change to the **WINDOW** command. Check Chapter Three, Part Four for details.

Syntactic Symbols Used in this Manual

Within this manual we will follow certain syntactic rules which you must know before reading this manual. The rules are:

Brackets [ ] are used when something is optional.

NOTE: Brackets are used in the syntax of some statements.

Braces { } are used to indicate that something is optional and may be repeated.

NOTE: Braces are also used to delimit comments.

Bold capital letters are used whenever a reserved word is denoted.

Aexpr is used to denote an arithmetic expression either integer or real An Aexpr may simply consist of an integer or real variable.

Alop is used to denote an arithmetic operator. An arithmetic operator may be a + - * / ^ MOD.

Relop means a relational operator. A Relop is a: <, >, <>, >=, <=, = and may also include the logical operators: AND, OR, NOT

Sexpr is used to denote a string expression. An Sexpr may simply be a string variable.

Expr is used to denote any expression, integer, string or real. In short, an Expr is an Aexpr or Sexpr.

Identifier is used to denote a Function, Routine, Procedure, Program or variable name. An identifier is made of letters, digits, underscore, ampersand, dollar sign, percent sign to a maximum of 62 characters.

Letters are either uppercase or lowercase and are case insensitive (no distinction is made between A and a).

Unop is a unary logical operator. It may be a plus sign, minus sign and NOT operator.

Filename is a string of alpha-numeric characters no longer than 15 characters in length.

Volume name is a string of alpha-numeric characters no longer than 15 characters in length. A slash (/) precedes the actual name.

Pathname is a string made of a volume name, directories (if any) and a file name. It may be no longer than 64 characters in length including slashes.

Chapter Two

Getting Started

A Brief History of BASIC

The original BASIC was written in 1964 under the direction of John Kenemy and Thomas Kurtz at Dartmouth College, New Hampshire, United States of America.

BASIC is the acronym for Beginners All-purpose Symbolic Instruction Code. It was intended to be relatively easy to learn and inexpensive to implement. The original BASIC was an interactive language, so that the programmer would get instant results. BASIC was originally intended as a teaching tool, so its capabilities were very limited.

Originally, a program line in a typical BASIC program had to begin with a line number. Subsequent implementations of the BASIC programing language required no line numbers and featured structured programming statements like **REPEAT..UNTIL** and **WHILE..WEND**.

Applesoft BASIC was installed in the Apple II+ computer in 1979 as the successor to the primitive Integer BASIC. Apple hadn't yet developed a disk operating system, so Applesoft had no built-in DOS commands, among many other limitations.

Micol BASIC was released in 1985 by Micol Systems as a structured and compiled BASIC language system based on Applesoft BASIC. *Micol BASIC* was designed to run on an Apple II+, IIe (64K) and IIc. Although *Micol BASIC* was much more powerful than Applesoft BASIC, it still was designed for a computer with limited abilities.

Micol Systems entirely rewrote *Micol BASIC* for the Apple IIGS and added numerous enhancements and improvements which became *Micol Advanced BASIC*, version 1.0, for the Apple IIGS in 1988. The next year, a special version for the Apple IIe (128K), IIc, and Laser 128 computers was released which took advantage of the better graphics and auxiliary memory in these computers and has most of the features found on the GS version.

Writing Your First Program in *Micol Advanced BASIC*

Okay, let's write a simple program in Micol Advanced BASIC. This program won't do much, but it'll be a start. Just follow these simple steps:

1. Insert a copy of the *Micol Advanced BASIC* system disk marked Master Disk into a 3.5 inch drive. Turn on the monitor and the computer.
   a) The GS/OS operating system (the program that tells the computer how to use the devices connected to the computer) will load and execute
   b) The *Micol Advanced BASIC* Language System will load and execute. The Command Shell prompt (]) will be displayed with the Command Shell waiting for a response from the user.
2. Enter **HELP**<CR> (<CR> means press the key marked Return). This command

lists all the commands known to the Shell. Take the time to read the commands that are available. Enter HOME<CR> to remove the Shell's Help display.

3. Insert a work disk into a drive:
   a) If you have a second disk drive, insert an already formatted disk with little or no information on it and go to step 4
   b) Remove the *Micol Advanced BASIC* (Master Disk) disk and insert an already formatted disk with little or no information on it. (The disk containing the sytem will not be needed for program development).
4. Enter **PREFIX** /Name.of.Disk<CR>. Replace Name.of.Disk by the name you gave the disk when it was formatted. **PREFIX** tells the Shell to use the disk Name.of.Disk as the default disk. The Command Shell does not care where the disk is, as long as GS/OS can find it; otherwise the message "Volume not found" will be displayed. Unless otherwise instructed, the system always uses the "prefixed" disk. To see which default directory the system is using, enter **PREFIX**<CR> without a disk name. To see the names of all of the volumes available in the system, enter **ONLINE**<CR>.
5. Enter **EDIT**<CR>. This Shell command will bring you into the *Micol Advanced BASIC* Source Code Editor.
6. Press <Apple>H (hold down the key with the white apple on it and the H key at the same time). This command shows the commands known to the *Micol Advanced BASIC* Source Code Editor. Press any key to make this screen disappear.
7. Enter the following program; be certain to press Return after each line. Press Delete to erase a character. Press the Arrow keys to move the cursor. Press Tab to make an indentation in a program line.

```
PROGRAM First_Program
HOME
INPUT "Hello, I'm your Apple IIGS, what's your name? "; Name$
PRINT "Nice to meet you "; Name$
PRINT "Watch me count from one to ten"
PRINT "But first, press any key so I can start"
GET Any_Key$
FOR Count% = 1 TO 10
   PRINT Count%
NEXT Count%
PRINT "Good-bye "; Name$; ", I hope we meet again"
END
```

Take the time to check and revise what you entered.

8. Press <Apple>S to save the program to disk. The Editor prompts for a program name. Enter any name (letters only) of no more than eleven characters and press Return. The program will be saved to disk.
9. Press <Apple>Q to quit the Editor and return to the Shell.

10. Enter **CATALOG**<CR>. The contents of the disk directory will be displayed on the screen. Notice the name of the file you just saved.
11. To compile your program, enter the word **COMPILE** followed by a space, followed by the name you gave the program in step 8, followed by a Return. The Compiler will display "Compiling...<Program name>". If you have entered the program correctly, your program will be transformed into a format that can be executed. If there is an error in the program, the message "Continue compilation, Edit program, or use Shell (C/E/S)?" will be displayed on the screen. Press "E" to return to the *Micol Advanced BASIC* Source Code Editor and correct the mistake. Continue with Step 8.
12. After the program has compiled without any errors, you will receive the message "Execute the program (Y/N)?". Press "Y" to cause the program to load and execute.
13. The program will ask you for your name. Enter your name followed by the Return key. Notice the action on the screen. That was all caused by the program you just wrote.

When the program has finished execution, control will be returned to the Shell. Congratulations! You have written and executed your first *Micol Advanced BASIC* program.

Entering Program Examples

Some program examples within this manual cannot fit in the manual's page the same way they would appear on the screen. If you see the Program Line Continuation character, the backslash (\), this indicates that the remainder of the line is continued on the next line (you may also enter the program lines exactly as they appear in the text if you wish, the Compiler can handle this syntax).

Example:

```
PROGRAM Example
HOME
INPUT "Enter name: ";Name$
INPUT "Enter age: ";Age%
INPUT "Enter any floating-point value: "; \
          Number&
END
```

Enter the line(s) containing a backslash as if the line(s) were continuous (do not enter the backslash, in this case). If the line has more than 80 characters, the Editor will follow you by scrolling the display from left to right. The Editor will reposition the display to its usual place when you press the Return key.

Suggested Manuals

Apple Computer Inc., Apple IIGS Toolbox Reference Volume I, Reading, Mass.: Addison-Wesley Publishing Co., 1988. 776 p.

Apple Computer Inc., Apple IIGS Toolbox Reference Volume II, Reading, Mass.: Addison-Wesley Publishing Co., 1988. 700 p.

Apple Computer Inc., Apple IIGS Toolbox Reference Volume III, Reading, Mass.: Addison-Wesley Publishing Co., 1988. 1100 p.

Apple Computer Inc., Human Interface Guidelines, Reading, Mass.: Addison-Wesley Publishing Co., 1987. 160 p. (This is the book needed to write software that conforms to Apples guidelines.)

Apple Computer Inc., Standard Apple Numerics Environment (SANE) Manual. 2nd ed., Reading, Mass.: Addison-Wesley Publishing Co., 1988. 320 p. (This manual contains the details of other SANE functions that can be implemented in *Micol Advanced BASIC*.)

Apple Computer Inc., GS/OS Reference Manual Vol. 1, Reading, Mass.: Addison-Wesley Publishing Co., 1990. 528 p. (This manual contains the details of other GS/OS calls that can be implemented in *Micol Advanced BASIC*.)

Little, Gary B., Exploring the Apple IIGS, Reading, Mass.: Addison-Wesley, 1987. 552 p. (The examples in this book are written using APW macros.)

Little, Gary B., Exploring Apple GS/OS and ProDOS 8, Reading, Mass.: Addison-Wesley Publishing Co. 1988. 369P (we frequently use this book for reference).

Gookin, Dan and Davis, Morgan, Mastering the Apple IIGS Toolbox, Greensboro, North Carolina,: Compute! Publications, Inc., 1987. 642 p. (outdated, but excellent for learning the Apple IIGS Toolbox. This book appears to be out of print, but if you should find a store that still has one in stock, we recommend you buy it.)

Acknowledgments

Micol Systems Inc. wishes to thank the following people for their generous assistance:

- All our beta testers, especially Peter Cameron.
- A special thanks to Michael Yost for his assistance and advice on Windows and his support software as well as help with this manual.
- Walter Torres-Hurt for his selfless dedication over the years. We also wish to thank him for the *Micol Advanced BASIC* Users Group (MABug).
- CodeSmith Software Inc. for its assistance in improving the 640 mode graphics.
- Michael Crawford for his generous support and assistance.
- Benoit Bernard of Programmation Sur Mesure Inc. for his help.
- Ann Hendersen, Eddie Drueding, and Bernard Claing for their time and effort.
- And all of those who took the time to write or phone to provide us with their comments, suggestions and constructive criticism.

Part Two: The Programming Environment

Chapter One

The Command Shell

Overview

The Command Shell is the control program. Through the Shell, you can do basic disk filing, enter the Source Code Editor or compile, link and execute a program. The Command Shell performs a similar function to the ProDOS 8 command interpreter, file BASIC.SYSTEM, performs under Applesoft BASIC.

The Right Brace character "}" is the prompt character of the Shell.

Line Editing Commands

These commands allow you to edit the commands entered from the keyboard.

Up and Down Arrow Keys (↑↓)

The Up and Down Arrow Keys are not used in the Shell.

Left and Right Arrow Keys (→←)

The Left and Right arrow keys will work only within the range of an input field.

The Return key

The key marked Return terminates a command and may be pressed anywhere in an input field without loss of characters.

The Delete key

The Shell recognizes two deletion modes, true delete and destructive backspace. By default, the Delete key performs a destructive backspace. To toggle between the two deletion modes, press <Apple>Delete.

The destructive backspace mode erases the character to the left of the cursor. The true delete mode erases the character under the cursor. All characters on the right of the cursor are moved to the left. The shape of the cursor is not changed.

The delete mode will remain until it is modified by another <Apple>Delete or until the system is restarted.

<Control>C (Break)

Pressing <Control>C will terminate a listing of a text file started with the **LIST** command.

<Control>C may also be used to interrupt the execution of a program while it is running.

<Control>R (Repeat)

Pressing <Control>R displays the last command executed. The command is not executed, but is displayed so it may be modified if necessary. Press Return to execute the command again.

The Shell "remembers" the previous command, even after using the Editor.

<Control>S (Space/Stop/Start)

Pressing <Control>S inserts a space character at the current cursor position, moving every character after the cursor one position to the right.

This command may also be used to stop and start a file listing or program execution.

<Control>X (Cancel)

Pressing <Control>X cancels the command being entered. A backslash character (\) appears as the last character on the line to indicate that the previous command has been cancelled.

Built-in Shell Commands

These commands allow you to perform the basic tasks of the Command Shell. Additional Shell commands may be written using *Micol Advanced BASIC*.

BATCH Pathname

The **BATCH** command allows Shell commands to be read from a text file on disk and executed as though the commands were entered from the keyboard. The Pathname is the name of a text file in a directory currently online.

The Batch file is usually created by the Source Code Editor, and is simply a text file containing the Shell commands described here which are to be executed by the

Command Shell. The commands are displayed as they are executed.

Any shell command except another **BATCH** command is a legitimate entry into a batch file. An **EDIT** or **COMPILE** command will execute, but will end the batch stream.

Any line in the Batch file beginning with a semicolon (;) will be considered a comment.

<Control>C will cancel the execution of a Batch file.

BATCH is particularly helpful to users who are doing their program development on a RAM disk and wish to set up their system to their own needs.

AutoExec File

When *Micol Advanced BASIC* is first booted, the system checks for a Batch file under the *Micol Advanced BASIC* folder called AutoExec. If this file is present, the Batch stream contained within AutoExec is executed, otherwise the system simply enters the Shell.

The system disk marked Master Disk has an AutoExec file on it, so you may wish to examine this file to better understand AutoExec files.

Example:

```
LIST INFO.DOC
;Erase or rename the AUTOEXEC file to stop
;INFO.DOC from appearing again.
```

The batch file AutoExec lists the INFO.DOC file on the screen.

CATALOG [Pathname]

CATALOG and its abbreviation **CAT** are used to display the contents of a volume or any of its directories. The directory information indicates if a file is locked or not, lists its name, type, size of the file in blocks, its date and time of creation, its date and time of modification and the size of the file in bytes. The quantity of blocks used and unused are listed after the list of the contents.

If a Pathname is stipulated, the directory will be read from the stipulated volume. If the Pathname does not begin with a slash (/), the default prefix will be used with the stipulated directory name. If a Pathname is not stipulated, the directory of the default prefix will be displayed.

Example:

```
CAT /RAM6
CATALOG SUBDIR/
CAT
```

COMPILE Pathname [, Pathname]

This command invokes the Compiler. The first Pathname is the source code Pathname of the file you wish to compile. If the source code Pathname cannot be found, an error will occur and the Shell prompt will return.

If the Pathname is followed by a comma and another Pathname, then the object code file will have this stipulated pathname with the appropriate extension added. After the compilation is completed, the filename containing the compiled program will end with a ".LNK" extension.

If a syntax error is detected, the BASIC source code line will be displayed in inverse video. You will be prompted "Do you want to Continue, Edit or return to the Shell (C/E/S)?. To continue the compilation, press "C". The Compiler will continue the program's compilation. To edit the error, press "E". The Editor will place the cursor on the line and approximate character where the compilation error occurred. To return to the Shell, press "S". The prompt of the Shell will appear.

COPY Pathname1 TO Pathname2

COPY duplicates the contents of the file Pathname1 by creating a new file and giving it the name Pathname2. If the original file and the duplicate file are in the same directory, Pathname1 must be different from Pathname2.

Example:

```
COPY /Disk/Old.File TO /RAM5/New.File
```

The file Old.File in volume /Disk will be copied to volume /RAM5 with the name New.File.

CREATE Pathname

CREATE generates a new directory file (folder) under the main or a subdirectory with the name stipulated by Pathname. The directory created is locked.

Examples:

```
CREATE /RAM6/DIRECT.1
CREATE /Library/Math/Trig
```

In the first example, the subdirectory Direct.1 will be created on volume /RAM6. In the second example, the subdirectory Trig will be created on volume /Library in the subdirectory Math/.

DELETE Pathname

DELETE erases a file from a directory. A subdirectory file must be empty before it can be deleted. The disk must not be write protected and the file must be unlocked.

Example:

```
DELETE /RAM6/Filename
```

EDIT [Pathname]

The **EDIT** command summons the Source Code Editor. The stipulated file must be of type TXT ($04) or SRC ($B0) to be edited.

If the command **EDIT** is entered without a Pathname and no file is being edited, the Editor will appear. No file name appears on the Data Line as there is currently no file being edited.

If **EDIT** is entered without a Pathname and a file is being edited, the Editor will appear to let you continue the editing process. The cursor will appear in the identical line and position as when you last left the Editor. The Pathname of the file is displayed on the Data Line.

If the **EDIT** command is followed by a Pathname and no file is being edited, or the file being edited has not been moodified since the last save, the stipulated file will be loaded from disk into the Editor's workspace. The file's Pathname will appear on the Data Line.

If **EDIT** is followed by a Pathname and the file currently being edited has been modified without being saved, the Command Line of the Editor will prompt "Save current file before loading new one (Y/N)?". If "Y" is pressed, the file currently in the Editor will be saved to disk, the workspace will be emptied and the specified file will be brought into the editor. If "N" is pressed, the file currently in the Editor will be erased from the editor's workspace and the specified file will be brought into the editor.

Example:

```
EDIT/RAM6/TXT.FILE
```

FORMAT Volume_Name

To initialize a storage device, use the **FORMAT** command. The initialized device will have the name stipulated as Volume_Name.

When this command is invoked, the Shell displays the location and the names of all devices connected to the computer. Select the appropriate device with the Up and Down Arrow keys and press Return to display the GS/OS Formatting Dialog Box.

Set the controls of the Dialog Box to "ProDOS" for the operating system (if necessary) and "800K 2:1" for the interleave. Press Return to start formatting the device.

For optimum performance with the white UniDisk 3.5 drive only, use 800K 4:1 interleave.

WARNING

> Be very careful with the **FORMAT** command. Once **FORMAT** is executed, any valuable information contained within the medium will be lost.

Example:

```
FORMAT Work.Disk
```

HELP

HELP lists the built-in Shell commands available with a brief description. User-written Shell commands are not listed.

Example:

```
HELP
```

HOME

HOME is simply used to erase the contents of the screen and place the cursor at the upper left corner.

Example:

```
HOME
```

LIST Pathname

LIST displays the specified source file on the screen, so you may preview it without entering the source code editor. Only files of type TXT ($04) will be displayed.

Pressing <Control>C ends the listing; pressing <Control>S pauses the listing. Pressing any key after that will restart the scrolling of the listing.

Example:

```
LIST /RAM6/INFO.DOC
```

LOCK Pathname

LOCK protects a file from being deleted or modified. When a file is locked, an asterisk (*) precedes the file name when a directory is displayed.

Example:

```
LOCK /RAM6/FILE
```

ONLINE

ONLINE displays the names of all the block devices such as floppies, hard drives, RAM drives and CD-ROM drives connected to the computer. **ONLINE** displays the names of the devices and the names of the volumes.

Example:

```
ONLINE
```

PREFIX [Directory_Name]

The command **PREFIX** indicates the path used by the system or sets a different default prefix. The default prefix contains part of the path leading to a specific file.

The default prefix is the prefix that is used unless another path is specified. If the Master Disk is booted, at startup, the (default) prefix is set to /Micol.Adv.BASIC/Micol.Adv.BASIC/.

The names of the volumes or directory files must be from online volumes. If not, the previous prefix will remain in use. The error message "Volume not found" will be displayed if the volume is not online.

If Directory_Name is preceded by a slash character (/), the prefix will be changed to this new volume name.

If Directory_Name is not preceded by a slash character, the current prefix will be used with the Directory_Name appended to form the path leading to the directory.

Examples:

```
PREFIX {Displays the current prefix}
{Adds System/Desk.Accs/ to the current prefix}
PREFIX System/Desk.Accs/
{Prefix becomes /Micol.Adv.BASIC/System/}
PREFIX /Micol.Adv.BASIC/System
```

PREFIX < [<]

This **PREFIX** command lets you move back one or more levels within a path by adding one or more less than symbols (<) with no separating spaces. One less than symbol (<) equals one directory level.

Use **PREFIX** with a Pathname to go "outside" any subdirectory.

Example 1:

```
PREFIX <        {Go back one level}
PREFIX          {Display the current prefix}
PREFIX <<       {Go back two levels}
```

Example 2:

If the current default prefix is /VOLUME/FIRST/SECOND/THIRD/FOURTH/, the command **PREFIX <<** will set the new default prefix to /VOLUME/FIRST/SECOND/.

PRINTER

PRINTER sets the port or slot number through which the printer output of the Editor and Compiler will be sent. The slot number entered must be a digit in the range one through seven.

The default output is set to slot/port one.

Example:

```
PRINTER
Send output through which slot? (1-7)?
```

QUIT [Pathname]

Use **QUIT** to leave the *Micol Advanced BASIC* language system. If **QUIT** is not followed by a Pathname, you will be prompted: "Are you certain you want to quit (Y/N)?". If "N" is entered, this command will be ignored and the Command Shell prompt will return. If "Y" is pressed, control will be returned to the operating system. Once you have entered "Y", you will leave *Micol Advanced BASIC*.

If **QUIT** is used with a Pathname, that GS/OS application will load and execute. No prompt requesting you to confirm your choice will appear, so make sure that the disk containing the program launcher is online; otherwise, you will receive an error.

Example:

```
QUIT
Are you certain you want to quit (Y/N)?
```

WARNING

> Some program selectors (the Finder included) require that the startup disk be online before the GS/OS application can successfully execute. Do not attempt to execute these files unless the startup disk is online, otherwise the system will crash!!

If *Micol Advanced BASIC* was already started using a program launcher, use **QUIT** without specifying an application name.

If you have only a single drive system, you may copy the Finder from the /MAB.SUPPORT disk to a RAM disk and access the Finder from the RAM disk with the *Micol Advanced BASIC* system disk in the drive.

Example:

```
QUIT /MAB.SUPPORT/SYSTEM/FINDER
```

If the second system disk, as well as the startup disk, is currently online, the Finder will load and execute.

RENAME Pathname1 TO Pathname2

RENAME changes the name of a file, directory or volume. If the paths are on the same volume, this command may even be used to move a file to another directory. To rename, Pathname1 must be unlocked and Pathname2 must not already exist.

Example:

```
RENAME /RAM6/FILE TO /RAM6/NEWFILE
```

RUN [Pathname]

RUN [Pathname] loads and executes the compiled and linked program specified in Pathname. The Pathname is usually the name of the source file of the program (the ".LNK" extension is added by **RUN**).

The **RUN** command may be entered without the Pathname to reexecute the previous program.

Whenever a program is **RUN**, the values of all booleans are set to false, numeric variables are set to 0 and all string variables to empty before executing the first line of the program.

Examples:

```
RUN /MICOL.ADV.BASIC/MT.FRACTAL
```

UNLOCK Pathname

UNLOCK removes the protection on a file, so it may be modified, deleted or renamed. A space rather than an asterisk will precede the filename when the proper directory is displayed.

Example:

```
UNLOCK /RAM6/FILE
```

Adding Your Own Commands to the Shell

When the Command Shell receives a command it does not understand, it assumes the command is the name of a Utility, a compiled *Micol Advanced BASIC* program, in the folder UTILITY directly under the *Micol Advanced BASIC* folder, and attempts to load and execute it.

If there is no such program name in the Utility folder, the Shell will display the message "Illegal command line". This filename is treated as equivalent to a built-in Shell command.

How to Write a Shell Utility

The first step in writing a Shell Utility is simply to write a *Micol Advanced BASIC* program, compile and link it.

After your Shell utility program is thoroughly debugged, take the compiled code and use the **RENAME** command to give the utility a meaningful name (no extension is necessary). Copy the completed program into the folder UTILITY.

To access this utility from the Shell, just enter the name of the command exactly as it appears in folder UTILITY.

Passing Parameters to the Utility

Micol Advanced BASIC Utilities may accept parameters. This parameter is a string entered by the user after the Utility name when the Utility is invoked. This parameter may not contain any spaces because a space is also a delimiter within the Shell (there must be a space between the Utility name and the parameter on the command line).

Example (from Shell command line):

```
Get_Help Help.File
```

The optional parameter, a simple ASCII string ended by a carriage return, will be placed into a buffer terminated by a zero (without the <CR>). The address of this buffer will be placed into locations 212 - 214 in the usual LSB, MSB, Bank format. To access this string, concatenate the values starting at the address contained in locations 212, 213 and 214 using the **CHR$** function until a zero is detected.

Example:

```
PROGRAM My_Utility
Param$ = ""
Address& = PEEK (212) + 256 * PEEK(213) + 65536 * PEEK(214)
REPEAT
    Number& = PEEK (Address&)
    IF Number& <> 0 THEN BEGIN
      Param$ = Param$ + CHR$(Number&)
    ENDIF
    Adress& = Adress& + 1
UNTIL Number& = 0 {Your utility code follows}
```

The parameter may then be used within your Utility program for any purpose you require.

See the INDENTER program on the /MAB.SUPPORT disk for a realistic example of a Utility. This program also is included as a Shell Utility on the disk labeled Master Disk. If you enter INDENTER<CR> from the Shell, this Utility will execute; you will then be able to get instructions on INDENTER's usage.

Chapter Two

The Source Code Editor

Overview

This full-screen Editor has word processor like features plus easy Compiler access and debugging assistance. The Editor has easy-to-remember, two-keystroke commands that ease the entry and revision of the source code.

The Editor retains the current file even while using the Command Shell, compiling or executing the program.

Entering and Quitting the Editor

Entering the Editor (EDIT [Pathname])

To summon the Editor, enter **EDIT** or **EDIT** Pathname at the Command Shell level. The Editor may also be entered by pressing "E" from the Compiler if an error is detected, or from a program if a run time error occurs while executing a program.

Quitting the Source Code Editor (<Apple>Q)

To leave the Editor and return to the Shell, press <Apple>Q. If the file being edited has been modified since the last save, the Editor will beep when <Apple>Q is pressed. If you hear this beep, you may wish to reenter the Editor (simply enter EDIT<CR>) to save the file before continuing.

Description of the Editor's Display

The screen display of the Editor consists of 24 lines. The Command Line is at the top of the screen. A reference ruler appears on the second line. Directly under the Reference Ruler is the Editing Display Area where your program will appear. At the bottom of the screen on line 24 is the Data Line.

The Command Line.

The Command Line displays prompts and messages when the Editor needs to get or return information.

The Editor's Command Line uses the following keys to edit the input to a command: Left Arrow, Right Arrow, Delete, <Control>S, and <Control>X.

The Reference Ruler

The second line displays a ruler. This line may be used to align text within the screen.

The Editing Display Area

The Editing Display Area is a window that uses 21 lines of the screen to show the text being edited. When necessary, this window moves up and down and from side to side to show text that cannot be entirely displayed within one screen.

The Data Line

This inverse video line gives information about the text file being edited:

- **Line Counter**
  - — This number represents the cursor's current line position in the text buffer. It is affected by up and down cursor movements and the Goto Line function (<Apple>G).
- **Column Counter**
  - — Entering characters or moving the cursor left or right causes the column counter to increase or decrease between 1 and 254.
- **Line Length**
  - — The Line Length counter shows the total number of characters in the current line.
- **Pathname Indicator**
  - — This area has a Pathname in it only after an existing file is loaded or after a new file is saved to disk. The Pathname will be truncated to fit the display if it is too long. This Pathname display remains until a new file is loaded or the text buffer is emptied.
- **Calendar/Clock Display**
  - — The date and time will be displayed on the lower right side of the screen. When a file is saved, the date and time are automa' :ally stamped on the file's directory entry.

The Sound Indicator

The editor will beep when the wrong command key is pressed.

Basic Editor Commands

Control Command Keys

These Control key commands allow editing on a single line of source code.

<Control>B Erase to start of line

<Control>B deletes the portion of the line from the cursor position to the beginning of the line.

<Control>X Erase current line

<Control>X deletes the line where the cursor is.

<Control>Y Erase to end of line

<Control>Y deletes the portion of the line from the cursor position to the end of the line.

The Apple and Option keys

The Apple key and the Option key are used in combination with another key to give commands to the Editor. Either the Apple or Option key plus the other key must be pressed at the same time for a command to be executed.

NOTE

> The Apple key is also called Command or Open-Apple. The Option key is also called Closed-Apple. On the Apple IIGS Upgrade, the Option key is called Closed-Apple. In this manual, <Apple> will refer to the White Apple key and <Option> will refer to the Black Apple key.

Escape key (Esc)

The Esc key may be used to cancel most commands at any time.

Return key

When the Return key is pressed, the cursor moves down to the beginning of the next line and the file is shifted down one line. If the cursor is in the middle of the line, the part to the right and under the cursor will be moved to the next line. The left side of the line will remain as it was.

Deletion Mode (<Apple>Delete)

The Editor recognizes two deletion modes: true delete and destructive backspace. To change the deletion mode, press <Apple>Delete. <Apple>Delete toggles from destructive

backspace to true delete. By default, the Delete key performs a destructive backspace.

The destructive backspace mode erases the character to the left of the cursor. The true delete mode erases the character under the cursor. All characters on the right of the cursor are moved to the left. The shape of the cursor is not changed. Destructive Backspace mode is shown by a Caret symbol (^) on the command line. True Delete mode is indicated by a Less Than symbol (<) on the command line. The Deletion mode character is displayed at the left of the Copyright notice on the Command Line.

The delete mode will remain until it is modified by another <Apple>Delete or until the system is restarted.

Delete Key

To delete a character, press the Delete key. The character will be erased and the line will move to fill the blank. If the cursor is over a line with no characters, this line will be erased and the following lines will move up one line. If the cursor is at the end of a line in the True Delete mode, or at the beginning of a line in Destructive Backspace mode, the previous and the current line will be merged and that section of the file will move up one line.

Help screen (<Apple>H or <Apple>?)

To see a summary of the commands available to you, press <Apple>Shift-/ or <Apple>H. The contents of the Editing Display Area will be replaced by the list of Editor commands. To remove the help screen and resume editing, press a key.

The key Help is supported on any ADB compatible extended keyboard.

Enter/Overstrike Mode (<Apple>E)

To alter the edit mode, press <Apple>E. Pressing these keys changes from Enter to Overstrike mode. Overstrike writes over existing characters without inserting other characters; Enter mode automatically inserts the character. The default setting is Enter. Enter mode is indicated by a flashing inverse space. Overstrike mode is shown by a flashing underscore.

Upper/LowerCase Mode (<Apple>X)

<Apple>X allows the user to enter uppercase characters without having to press the Shift key even when the Caps Lock key is in the Up position.

To activate this feature, press <Apple>X; the "C" in the copyright symbol on the command line will change to a lower case "c". The upper/lowercase entry will be reversed from what it was. To enter lowercase characters while using this feature, press the Shift key. To deactivate this feature, press <Apple>X again.

Moving in the File

Cursor Control (↑↓←→)

All arrow keys are functional. For any line greater than 80 characters, any attempt to move the cursor past the right edge of the screen will cause the display to shift to the left. If the screen has been shifted left, any attempt to move the cursor past the left most position of the screen will cause the display to shift right. Upward and downward motions work in the regular manner.

Think of the display as being an 80 column, 21 line window to the text file, with the cursor keys allowing you to move anywhere you want within the file.

When the cursor is moved up or down, you will eventually reach either the top or bottom of the screen display. When the cursor reaches the bottom, the file scrolls up. When the cursor reaches the top, the file scrolls down.

Move Down one screen (<Apple>↓)
Move Up one screen (<Apple>↑)

<Apple>Down-Arrow (↓) will move the cursor to the bottom of the screen, or if the cursor is already at the bottom of the screen, it will scroll the display one screen page (20 lines) up.

<Apple>Up-Arrow (↑) will move the cursor to the top of the screen, or if the cursor is already at the top of the screen, it will scroll the display one screen page (20 lines) down.

NOTE

> The screen scrolling commands may also be used while selecting a block of source code that will be moved, copied or deleted using the <Apple>C, <Apple>D or <Apple>M commands.

The keys Page Up and Page Down are supported on any ADB compatible extended keyboard.

Move To Beginning of Line (<Apple>←)
Move To End of Line (<Apple>→)

<Apple>Left-Arrow (←) will move the cursor to the first character of the current line, scrolling the display to the right if necessary. <Apple>Right-Arrow (→) will move the cursor one character past the end of the line, moving the display to the left if needed.

Move to Previous Word (<Option>←)
Move to Next Word (<Option>→)

<Option>Left-Arrow (←) moves the cursor to the first character of the previous word on the line, scrolling the display to the right if necessary.

<Option>Right-Arrow (→) moves the cursor to the first character of the next word on the current line, moving the display to the left if needed.

NOTE

> Pressing the <Apple> key instead of the <Option> key will not enable this command.

Relative Motion within the File

(<Apple>1 through <Apple>9)

Because a program source code file grows larger with every line you enter, the Editor "separates" the file into 9 parts. Each part is recalculated as you add lines to your file. Pressing <Apple> and a digit key will bring this "relative" portion of the file to the display window.

To move to the beginning of the file, press <Apple>1. To move to the middle of the file, press <Apple>5. To go to the end of the file, press <Apple>9.

The keys Home and End are supported on any ADB compatible extended keyboard.

Go to Program Line (<Apple>G)

To move quickly to a specific sequential program line, use <Apple>G: the Goto Line command. The command line prompts for an input. Give a line number and press Return. The line will be displayed on the first line of the display. This command helps locate the errors signaled by the Compiler.

WARNING

> Do not confuse the sequential program line numbers with the optional BASIC source code line numbers. The sequential program line numbers are created by the Editor and the Compiler and are in no way related to any line numbers the user may create.

Setting Tab Stops (<Apple>Tab)

To set tabulation positions, press <Apple>Tab. The current tabulation marks are

indicated by diamonds on the Command Line. The default tab settings are placed one every fifth position. Tab stops may be set only for the first 80 columns.

To set or delete tab stops, move to the desired position using the Right-arrow key (Left-arrow will move back to position one) and press the Tab key. The first Tab pressed will set the first position, the second pressed, the second tab position, and so on up to the 80th column. Press Return to confirm the new tab settings.

Tabbing (Tab key)

Use the Tab key to indent your source code. To tab to the next tabulation position, press the Tab key. The default tab settings are every fifth position and may be altered as desired by <Apple>Tab. If the cursor is past the current end of line, pressing Tab will expand the current line to one character less the required Tab position, then the cursor will move to the required position.

NOTE

> If the next Tab stop is currently occupied by text, pressing the Tab key will simply reposition the cursor without indenting.

Text Block Editing Commands

Copy Text Block from Buffer (<Apple>C)

This command is designed to copy a block of text from the copy buffer to the text area. You must have first moved the required lines to the copy buffer using the Move Block command (<Apple>M) described below, otherwise you will receive an error. Move the cursor to the line just after the position where you want to place the lines, then press <Apple>C. The lines will be copied from the copy buffer. You may copy a maximum of 32,767 characters (32K).

Delete Text Block from Code (<Apple>D)

To delete a block of text, press <Apple>D. Then press the Down arrow key to "mark" the lines to delete. The Up arrow key will unmark the lines. To confirm the deletion command, press the Return key. The marked text will be deleted.

This command operates on whole lines only: the Delete Block command cannot be used to remove a portion of a line.

WARNING

> The Editor cannot recover deleted text once this command is executed. Use the Move Text Block command (<Apple>M) instead if you wish a possible recovery later.

Move Text Block to Buffer (<Apple>M)

To move a block of text to the copy buffer for later copying, and optionally, to delete a block of text, press <Apple>M. To mark the lines to be moved, press the Down-arrow key. To unmark the lines, press the Up-arrow key. Press the Return key to move the marked text to the copy buffer. You will then be prompted if you wish to delete the marked text. Accepted input is "Y" for yes and "N" for no. A copy of the moved text will remain in the copy buffer until this command is used again or you leave the Text Editor.

The keys F2, F3 and F4 are supported on any ADB compatible extended keyboard.

Find/Replace Commands

Backward Find/Replace (<Apple>B)
Forward Find/Replace (<Apple>F)

The Backward Search and Forward Search commands are used to quickly move the cursor to a specific word or to search for and replace that word. A search always begins at the current cursor position.

These commands can search for a specific word or phrase (from 1 to 64 characters in length).

If the occurrence(s) of the word you want to search for is near the beginning of the file, use <Apple>F (Forward Search and Replace). Use <Apple>1 to start from the beginning of the file, if necessary. If the occurrence(s) of the word you want to search for is near the end of the file, use <Apple>B (Backward Search and Replace). Use <Apple>9 to start from the end of the file, if necessary.

We will use Forward Search (<Apple>F) in the examples (backward search works the same way). The Editor prompts: "Forward search: Find which string?". Enter the word(s) to find, then press Return. The text must appear exactly as it appears in the source code.

"Case sensitive search (Y/N)?". Press "N" to find all occurrences regardless of the case. Press "Y" to find only occurrences having the same upper and lowercase pattern as the one entered for the search string. A case sensitive search will look for word(s) with the exact combination of upper and lowercase letters that match the character string you are looking for.

The prompt "Replace with" asks for the string that will replace the word(s) you are looking for. If you are looking for a word, not replacing it, press Return without entering anything; otherwise, enter the replacement string.

Do an "Automatic replacement (Y/N)?" If "Y" is entered, all matches will be replaced

without user intervention. If "N" is entered, the user will be prompted to confirm the replacement of each occurrence as it is found.

If the Editor finds the word(s) you are looking for, it will show the occurrence in the center of the editing area displayed in inverse video. The editor will prompt if you want to "Continue the search (B/F/Q) ?". To continue the search forward, press "F". To continue the search backward, press "B". To quit the search, press "Q".

Example:

```
{Looking for a function}
Forward search: Find which string? FUNC
{Any case pattern}
Case sensitive search(Y/N)? N
{No replacement}
Replace with (Press Return)
{Prompt for every occurrence?}
Automatic replacement(Y/N)? N
```

WARNING

> Because this command may make extensive changes to your file, we recommend you save your file before using the automatic replacement feature, Until you are familiar with this feature, it is easy to make mistakes. Just reload the file to "undo" all the changes, if it did not do what you wanted.

Filing Commands

New Source Code File (<Apple>N)

To clear the text buffer and start anew press <Apple>N. You are prompted for confirmation. If you respond "Y", you will be as if you had just entered the Editor.

WARNING

> Once this command is executed, the text cannot be recovered unless it has been previously saved to disk.

Insert Source File from Disk (<Apple>I)

To insert or merge another text file into an already existing text file, move the cursor to the line preceding the insertion/merge position, then press <Apple>I. You will be prompted for a Pathname. Enter the Pathname and press Return. If the file does not exist, you will be notified. The text will be read from the disk one line at a time. Each

time a line is entered, the screen displays this new line. The cursor will remain on the line it was on before the command was given.

WARNING

> Never use <Apple>I to insert a file at the last line of the current file as Insert cannot be used to Append text. Create a dummy line as the last line and Insert to just before this line.

Save, Kompile and Execute File (<Apple>K)

This command will perform a Save (<Apple>S), compile, link and execute the file being edited without the operator's intervention as long as no compilation or linking error occurs.

If a compilation error occurs, the process is stopped, and the Compiler prompts: "Continue Compilation, Edit file or Shell (C/E/S) ?". An "E" entered here will return the user to the Editor at the position where the error occurred. A "C" will continue the compilation, and an "S" will take the user to the Shell.

If a run time error occurs during execution of the program, you will be prompted whether or not you wish to reenter the Editor to fix the problem. A "Y" will place the cursor at the line containing the error. An "N" returns control to the Shell.

IMPORTANT

> Regular use of this command is highly recommended as it greatly simplifies program development.

Load Source Code File (<Apple>L)

To load a text file into the Editor, press <Apple>L. This will bring up the command prompt line allowing a 64 character Pathname. Enter the Pathname and press Return to load the file. Loading a file into memory removes the previous file in the text buffer. After the file has been loaded, the Editor will display the first 21 lines starting from line one. The line and column counters will display one. The Pathname is shown on the data line before the clock display.

If you want to load a new file after having made changes to the current file, the Editor will prompt you to save the current file before loading the new file.

If you try to load a file larger than the text buffer can hold, the part which will not fit in the buffer will be cut.

IMPORTANT

> The <Apple>L command does not erase the text contained within the copy buffer. Use this command to copy text from one file to another, if necessary.

Save File as TXT type file (bit 7 on) (<Apple>S)

To save to disk the program you are currently editing, enter <Apple>S. This is the usual file save command. If you save to an already existing file, this file will be deleted first, then the new file will be saved in its place.

The Save command "remembers" the last Pathname entered. To reuse this previous Pathname, simply press "Y" to the prompt. The file saved with <Apple>S is of type TXT ($04).

WARNING

> The Compiler generates the object file from the file on the disk, not from the Editor buffer, so be certain to save your file before you call up the Compiler.

Save File as SRC Type File (bit 7 off) (<Apple>T)

<Apple>T saves the source code text file as an ASCII file. The file saved with <Apple>T is of type SRC ($B0). The text file created can be read by most word-processors. This command works the same way as <Apple>S.

Printing Commands

Print Source Code (<Apple>P)

To output a program listing to your printer, press <Apple>P. The command line will prompt you for the line number to start printing. Enter any positive number. Simply pressing Return is a line one. The command line will prompt you again for the line number to stop printing. Enter the second line number, or simply press Return as this is an implied last line. The printing of the listing will start immediately. To print the entire file, press the Return key twice. The Esc key may be used to cancel a print in progess.

Example:

```
First Line: 100<CR>
Last Line: 701<CR>
```

Text Window Printout (<Apple>W)

To print the text appearing in the text window, press <Apple>W. This command is most useful when you want a quick printout of the Editing Display Area.

The key F13 (Print Screen) is supported on any ADB compatible extended keyboard.

To cancel the printout in progress, press the Esc key.

Miscellaneous Commands

Convert Decimal to Hex (<Apple>#)

To convert a decimal number to hexadecimal, press <Apple># (<Apple>Shift-3). The command line will prompt you for input. Enter the decimal number to be converted to hexadecimal and press the Return key. Only valid numeric (0-9) characters will be converted properly as no error checking is done. Press any key to restore the command display.

Convert Hex to Decimal (<Apple>#)

To convert a base 16 number to base 10, press <Apple># (<Apple>Shift-3). The command line will prompt you for input. Enter a dollar sign ($) as the first digit to indicate that a base 16 number will be converted, then the base 16 number followed by the Return key. Only valid alphanumeric (0-9, A-F) characters will be converted properly. Press any key to restore the command display.

Version Information (<Apple>V)

By pressing <Apple>V, the Editor's Editing Display Area will clear and something like the following display will appear:

```
GS/OS Version                    3.3
Micol Advanced BASIC GS version  4.0
Last Modification Date           1 March, 1992
Bytes free in editor             23453
Bytes available in copy buffer   10009
Lines available for editing      2000
```

The Editors' maximum buffer size is almost 128 kilobytes: enough for about 3800 to 4000 lines of code. You may copy a maximum of 32,768 characters (32K) to the Editor's copy buffer.

Chapter Three

The Compiler

Overview

The *Micol Advanced BASIC* Compiler is a one pass compiler.; it reads the source code only once while generating the object code. The Compiler translates the ASCII file containing your BASIC program into an intermediate code which can be linked, then executed.

This chapter is short, but don't assume any lack of importance to the Compiler because of this chapter's short length. This chapter is simply a brief overview. The Compiler is the heart of the language system. Part Three, the longest Part, is a description of the language the Compiler can accept and in many ways is a description of the Compiler.

Invoking the Compiler

You may invoke the Compiler by using the Shell command **COMPILE** or by <Apple>K (Kompile) in the Text Editor (please see the appropriate section for details). If you do not use <Apple>K from the Editor, be certain to save your file before exiting the Editor as the Compiler works on the disk file, not the file in memory.

Example One:

```
{Default prefix is /Micol.Adv.BASIC/}
COMPILE DISK.UTIL
```

The file **DISK.UTIL** will be compiled onto the volume Micol.Adv.BASIC as file **DISK.UTIL.LNK**.

Example Two:

```
COMPILE DISK.UTIL,/RAM5/FILER
```

The file **DISK.UTIL** will be compiled as file **FILER.LNK** (a **.LNK** is always automatically appended) on volume RAM5.

WARNING

> Never forget that four characters are always appended to the object filename during compilation. If the total number of characters in the object filename results in more than 15 characters, you will receive an error at compilation time. To avoid this minor problem, always specify a source code filename of 11 characters or less.

During compilation, the Compiler generates three scratch files for its work. These scratch files are:

- <Filename.COD> the object code file
- <FileName.LIT> the file where literal constants are stored
- <FileName.LN> the file where forward references are stored.

The above three scratch files are then used by the Linker to create the executable load module, <FileName>.LNK

WARNING

As soon as the compilation and linking processes are completed, the three scratch files are deleted. If however, during compilation, you should receive a disk full message, it is because there is not enough storage for these scratch files as well as the other files on the disk. In this case, you will have to delete some files or direct compilation to another volume.

NOTE

Before the Compiler begins the compilation process, it checks for the existance of a RAM device with 192K free space or more. If such a device is detected, three scratch files are created: A.COD, A.LIT and A.LN on this RAM disk instead of the three scratch files described above. Compilation is much quicker if the scratch work can be done to a RAM device instead of a permanent storage device. If you have the available memory (more than one megabyte), it is recommended you configure the Control Panel RAM Disk to at least 192K.

Compiler Commands

The *Micol Advanced BASIC* Compiler has three Control key commands that may be used while a program is being compiled.

Aborting a Compilation

Pressing <Control>C stops the compilation in progress; control is returned to the Command Shell. If you use this command, you will probably notice several error messages generated by the Compiler. Simply ignore these messages as the compilation was not completed.

Compiled Listings to the Screen

If you press the letter "L" during compilation, the Compiler will send a compiled listing to the screen. This listing may be turned off by pressing the letter "L" again and may be paused by pressing <Control>S. Pressing any other letter will continue the compilation. This compiled listing is the same as that generated by the compiler option **LIST** described later in this manual.

Compiled Listings to the Printer

If you press the letter "P" during compilation, the compiled listing will be directed to your printer. This listing is the same as that sent to the screen described above.

WARNING

> The printer must be online at the time of compilation. By default, the printer must be connected to slot one or the system may hang. This slot number may be altered by the Shell command **PRINTER**.

Dealing with Syntax Errors

Unlike the Applesoft BASIC interpreter, *Micol Advanced BASIC* has dozens of different error messages, only one of which is the dreaded "Syntax Error". When the Compiler cannot make sense of a particular statement, it will send to the screen, in inverse video, the source code line as far as it could "understand" it, and relate what the Compiler "thinks" is the problem. The Compiler is sometimes wrong, but it is more often correct. In any case, you easily should be able to determine the real cause of the problem by taking time to read the error message and the line of code carefully.

You may be tempted to ask, when the Compiler gives you a message like "`' (' expected in line <line number>`", that if the Compiler knows what to expect, then why doesn't it simply insert t') character and continue?

Do not attribute any intelligence to the Compiler. It is little more than a very sophisticated pattern matcher and code generator. Some compilers do insert the character they "think" is missing, usually with very bad results.

The problem is that the Compiler often does not know what is really expected. With the information the Compiler has at the time, it is usually correct about what is needed. But maybe the cause of this error happened earlier.

For example, the programmer may have mistakenly entered a reserved word and used it as a variable name. The Compiler might expect a left parenthesis when what it actually found was an equal sign. If the Compiler had replaced the equal sign with a left parenthesis, the situation would be worse, not better.

Code Generation

As you probably know, the BASIC program you write is really only a representation of the actual code that is executed by the computer. This is true whether your program is compiled as under *Micol Advanced BASIC*, or interpreted as under Applesoft BASIC.

If you believe that Applesoft code you entered is what is actually executed, try this little experiment. Write a small program in Applesoft, then do a CALL -151 to get into the machine language monitor. Begin looking at the code starting at location $801 (2049 decimal). You will not recognize much; it is a special tokenized code.

The *Micol Advanced BASIC* Compiler scans your code and writes assembly language code as it goes. This is true of most (but not all) compilers.

With most language systems, code generation is regarded as a sort of black box. All you need to know is that a particular program will generate the necessary machine code to performs its task. You seldom get to see the code that is generated; you have to look upon it as a sort of magic.

Micol Systems takes a different approach. We believe that if you can see the code generated, you will better be able to understand what is going on and therefore write more efficient programs.

In order to speed compilation and save disk space, the Compiler writes an abbreviated assembly language code to disk. If you were to look at the file <FileName>.COD file generated by the Compiler, you would not recognize very much, even if you knew 65816 assembly language. However, if you specify the **CODE** compiler option at the top of your program, the Compiler will display this code in an assembly language format (see Part Three, Chapter One for additional information).

You will need a basic understanding of 65816 assembly language to understand this code, but as most of the detailed work of the compiled program is done by the run time Library routines, you won't need very much.

Most of the generated code is either setting encoded addresses and calling Library routines to perform the task, or generating code to control the flow of the program. Because of CPU limitations, most of the work performed by your programs must be performed by the Library routines.

Many Library routines used by the compiled program fall into one of three catagories: integer, real or string. The Compiler generates subroutine calls according to the following criteria: if the Compiler recognizes an operation to be integer, it appends an "I" to the function name stem. If it recognizes real arithmetic, it appends an "R", and it appends an "S" for string routines. If the Library routine R+ is being called, for example, real addition is being performed. Some important Library routines are:

LNOUT	Saves the line number information
MVARY	Used with array manipulations
FASS	Places **FOR** loop counter values onto its stack
FOR	**FOR** loop controls
NEXT	Decrements the **FOR** variable stack pointer
LDAC	Gets the boolean result from the stack

Chapter Four

The Linker

Overview

The *Micol Advanced BASIC* Linker will be summoned automatically if no error is detected during compilation. Because of this, the task of the Linker is mostly transparent to the user.

After the source code file has been compiled, the program is still not yet ready for execution. Three intermediate code files were created by the Compiler. These files contain all the information the Linker needs to generate the executable module.

The Linker will read files created by the Compiler from the volume where these files were written and create the file FileName.LNK in the appropriate folder.

How the Linker Works

First, the Linker reads the jump table (FileName.LN) that contains the names and relative addresses of all Functions, Procedures, Routines and other possible forward references in the source code.

Second, the Linker creates the binary load module FileName.LNK by reading the FileName.COD file. The Linker replaces the references to all the names of the Functions, Procedures, Routines and internally generated labels with their addresses, and generates the necessary code as it goes. The Linker sends a period to the screen for every 250 lines of code it has processed.

Third, after the generation of the executable code, the Linker converts the literal values written in the file FileName.LIT into binary and places this code at the top of the executable code module. These values will be loaded into their proper locations at initialization time (when the program is first executed).

After the linking process, the Linker will then try to delete the three scratch files generated by the compiler and used by the Linker, as they are no longer needed.

How to Use the Linker

As was already mentioned, the Linker is invoked automatically by the Compiler. The Linker does, however, require some user input after its task is finished.

If the Linker was summoned via the Shell using the **COMPILE** command, and the link is successful, you will receive the prompt, "Execute the file (Y/N)?". If you enter "Y", the program will load and execute. If you enter "N", you will be taken to the Shell.

If the Linker was called via the Editor with the Kompile (<Apple>K) command, the Linker will automatically load and run the executable object file after a successful link process.

Linking Errors

When the Linker detects an error, usually a non-existent Function, Procedure or Routine call (**FN** Module.Id or **GOSUB** Module.Id), the Linker displays "Undefined subroutine <ID>" in inverse video. <ID> refers to the name used to define the Function, Procedure or Routine in the program.

You are prompted to fix the error in the editor, "Edit the linker error (Y/N)?". A "Y" response to the prompt will bring the Editor to the screen with your file waiting to be edited. If you enter "N", you will be taken to the Shell.

Because the Linker does not know at which line the error occurred, the cursor is placed at the beginning of the source code file. Use the Forward Find/Replace command (<Apple>F) to locate the module call with the "undefined" subroutine <ID>.

Chapter Five

The Run Time Library

Reference Section

The run time Library, file **LIBRARY** on the system disk, is the workhorse of the compiled program. The Library contains all the routines needed by the compiled code to accomplish its tasks. The functions performed by the run time Library may be anything from doing integer multiplication to string garbage collection. The Library uses the floating point math routines of the Standard Apple Numerics Environment (SANE) contained in the Apple IIGS Toolbox as well as the Toolbox's graphics and sound capabilities, etc.

The run time Library is brought into memory when *Micol Advanced BASIC* is booted and remains in memory until you leave the language system. As is the case with the Compiler, the Editor and the executable load modules you will create under *Micol Advanced BASIC*, the Library is relocatable. This means the Library is loaded where the GS Memory Manager tells the Micol Loader memory is available. The Library is then referenced within a program by a jump location that is set at a fixed address in memory ($E100F0).

The Library consists of scores of run time routines and buffer memory. It comprises about sixty-four thousand bytes of code. Because most of the work the Library performs is done by internal routines, the speed of these routines is greatly increased.

The Micol Systems Licensing Agreement

The purchaser of *Micol Advanced BASIC* has the right to make backup copies of the *Micol Advanced BASIC* software for his/her one personal use. This software may not be given to another party except with express written permission of Micol Systems.

The purchaser of *Micol Advanced BASIC* has the right to make and distribute copies of the *Micol Advanced BASIC* Program Loader, the Micol icons, and the Run Time Library to execute a program developed by the legal owner of the *Micol Advanced BASIC* Language System if one of the two specifications below is followed. The Micol icons, the run time Library and the Micol System Loader (files **MicolIcons**, **LIBRARY** and **MicolAdv.BASIC**) consist of copyrighted code belonging to Micol Systems Inc.

That person or commercial entity owning a legal (non-pirated) copy of *Micol Advanced BASIC* is hereby granted a license to distribute free of charge compiled *Micol Advanced BASIC* programs provided one of the two conditions below is followed:

1. The Micol Systems Copyright notice is displayed while *Micol Advanced BASIC* is booting.
2. A negotiable, one time fee is paid to us before the release of the product on the commercial market. Once this fee is paid to us, you will receive a copy of a "Commercial Distribution License" from us to use the Run Time Library, the

Micol Icons, as well as the Micol Systems Loader which does not display the *Micol Advanced BASIC* Copyright notice, to be used with a specific product.

IMPORTANT

> You do not have the right to use the Micol icons, the *Micol Advanced BASIC* Run Time Library or the *Micol Advanced BASIC* System Loader with a program intended for commercial purposes unless you have met one of these two conditions.

Educational and Industrial Site Licenses

Micol Systems Inc. offers to companies and school districts and boards the possibility of making unlimited copies of *Micol Advanced BASIC* by purchasing a site license.

The site license package consists of:

- The *Micol Advanced BASIC* disks: Master Disk and /MAB.SUPPORT. These disks contain special, fully networkable versions of *Micol Advanced BASIC*, not otherwise obtainable
- Two copies of the *Micol Advanced BASIC* reference manual
- A site licensing agreement which allows you legally to make unlimited copies of the system disks and manuals for use with the specified site
- A product registration card
- The right to purchase additional manuals at a reasonable cost.

District and Board licenses are also available. For further details, contact the Micol Systems office during regular business hours.

Part Three: The Advanced BASIC Language

Chapter One

Compiler Rules and Directives

Overview

This chapter describes the general rules for writing *Micol Advanced BASIC* programs. You must pay special attention to this section as there is nothing in Applesoft of a similar nature. This chapter also describes special features of the language that can greatly aid you in your program development.

General Information

The programs you create with the *Micol Advanced BASIC* cannot be as free form as those created with Applesoft BASIC. You must follow certain rules regarding the sequential order of certain statements. This is something inherent to compiled languages.

A *Micol Advanced BASIC* program consists of a series of program lines. Each program line consists of one or more program statements. A program line may have a maximum of 250 characters and must end with a carriage return.

Multiple Statements per Line

A colon may be used to separate two or more program statements on the same line. Try to avoid this usage as it hinders program clarity.

Example:

```
TEXT:HOME
```

Line Numbers

If you wish, you may precede each program line with a line number as under Applesoft BASIC. Line numbers may range between 1 and 65535.

IMPORTANT

Line numbers are NOT required by *Micol Advanced BASIC* and their use is NOT recommended. Line numbers are no longer useful, and were retained solely for compatibility with Applesoft BASIC. Unless line numbers are referenced within a program, they will be ignored. Use of line numbers within a program is entirely up to the programmer.

IMPORTANT

When the Compiler or run time routine refers to a line in your program, it is referring to sequential line numbers given to the source code by *Micol Advanced BASIC*, not to any line numbers you have specified in your program.

Program Line Continuation Character (\)

The Editor and Compiler accept source code lines up to 250 characters long. The Editor's display will scroll from left to right when a source line of more than 80 characters is entered. To keep the program line within one screen, you may divide a source code line into two or more parts by terminating the line with a backslash (\). Enter the remaining source code line anywhere on the next line.

The backslash (\) must be the last character on the line and may appear only where extra spaces could appear. It may not be used to break reserved words or identifiers. The backslash may not be repeated on the same line, or you will receive an error.

Example:

```
PROGRAM Math
HOME
Number% = (1 * 6) + \
          (2 * 5)
PRINT Number%
END
```

Commenting Your Programs

Micol Advanced BASIC provides two ways to help you document a program: comment statements and comment delimiters.

Use annotations to better understand what the program does in order to make changes, corrections, or add new features to the program at a later time.

NOTE

> Unlike Applesoft BASIC, *Micol Advanced BASIC* does not generate any code for the comments in a program (except perhaps for line number information). Write whatever comments which aid in the understanding of the program.

Comment Statement (Old Method)

The **REM** (for remark) keyword instructs the Compiler to ignore all characters until the beginning of the next line. **REM** provides compatibility for programs originally written in Applesoft BASIC.

Example:

```
REM You may write comments like this as in Applesoft,
REM but the method described next is much better.
```

Comment Delimiter Characters [{ }] (Preferred Method)

Comments may also be enclosed within brace brackets [{ }], which may be placed anywhere in a program where extra spaces could be written. These comments may cover multiple lines if you wish.

NOTE

> Comment delimiter characters may be nested. An annotated section of code may be "commented out" without having to worry about the comments already written. "Commented out" code is treated like any other comment.

WARNING

> The right brace bracket (}) closes the comment and is extremely important. Do not forget to terminate the comment with a right brace bracket [}]; otherwise, the rest of the program will be considered a comment.

Examples:

```
PROGRAM Show_Comments
{This is a comment
   covering a couple of lines}
HOME
```

```
{{This FOR loop will not be in the program}
FOR Counter% {Comment here too} = 1 TO 100
   PRINT Counter%
NEXT Counter%}
END {Show_Comments}
```

Program Order

A *Micol Advanced BASIC* program must begin with a program name. Compiler options are the next statements to be included, if needed. **ALIAS**es, then **DATA** statements are declared thereafter. The optional identifier's type declaration follows next. Array declaration statements round up the program declarations.

Except for the program name, the lines just mentioned are optional, but if compiler directives, **DATA** statements or array declarations are used, they must not appear out of the order mentioned above, otherwise Compiler errors will arise.

Example:

```
PROGRAM Definition {Program Identifier}
{Compiler Options}
@ LIST, EXTEND
ALIAS "UNTIL 1 = 0" = "FOREVER"
DATA 1, 1.0, "1" {DATA statements}
{Identifier's Type Declaration}
INT (I - N): STR (S - Z)
DIM Alpha% (2), Beta (3), Coma$ (4) {Array declarations}
{Actual Program Start}
END
```

Program Name

The first line of each program must begin with the reserved word **PROGRAM** followed by a program identifier. The name of the program must begin with a letter and may only consist of letters (A-Z), digits (0-9) and underscores (_), and may not be a reserved word.

This line is not optional. If it is left out, the Compiler will return an error.

Note that a period (.) is not allowed in a program identifier.

Examples:

```
PROGRAM First_Program
PROGRAM Test.file {Not Allowed}
```

Compiler Directives

Compiler directives are special commands given to the Compiler to tell it to do a special task, such as sending a listing to the printer. Compiler directives consist of both compiler options such as **LIST**, and other instructions to the Compiler such as **ALIAS**.

The fact that the Compiler must see all the code before any program can be executed allows it to do certain things an interpreter is incapable of doing, such as giving more precise syntactic error messages. A thorough knowledge of these directives will help to get the most out of the compiled language and make programming more enjoyable.

Compiler Options

To use one or more compiler options, the line must begin with an at sign (@) followed by one or more options separated by commas (,). The compiler options may appear on separate lines, but the lines must be consecutive.

Example:

```
PROGRAM Example
@ LIST, CODE
<Program Code>
```

BANK_NO = Integer_Literal

This option is used to increase the memory allocated at run time for the string buffer and will probably only be necessary if you have very large string arrays.

The memory is reserved in memory banks of 64K. By default, a program has one bank for its string storage. Under most circumstances, one bank should be sufficient.

BANK_NO accepts integer values between 1 and 15. This means that as little as one bank (64K), or as much as 15 banks (just over a megabyte) may be allocated for string storage. If the computer does not have enough memory for the number of banks specified, there will be an error when the program begins to execute, not when it is compiled.

At the end of compilation, the Compiler indicates how many banks of memory for string buffer storage will be allocated for the program.

Example:

```
PROGRAM Example
@ BANK_NO = 4
```

In the example above, the program will allocate four banks (256K) of RAM for string buffer storage.

CODE

This option lets you see how assembly language code is generated by the Compiler as

it processes the program. Assembly language programmers will be able to see the code generated, and may be able to write better programs. **CODE** is included for the benefit of those who have an interest in learning more about how a compiler generates code.

To see the code generation displayed to the output device, use the **CODE** option. The Compiler writes the object code to disk in a compact assembly language-like format. With this option, the code will be expanded on the current output device to look like true assembly language.

Example:

```
PROGRAM Example
LIST, CODE
HOME
END
```

The Compiler produces this code for this simple program:

```
2   [0]  0        $0000      HOME
                              JSL LIBRARY
                              BYT INIT
                              BYT 00
                              WOR 0000
                              BYT 00
                              WOR 0000
                              JSL LIBRARY
                              BYT HOME
3 [0 ] 100        $0010      END
                              JSL LIBRARY
                              BYT LNOUT
                              BYT 00
                              WOR 0003
                              JSL LIBRARY
                              BYT END
```

ERROR

If a **RESUME** is used in a program which causes it to continue execution at the same line where a run time error occurred, the **ERROR** compiler option must have been specified to make the program function properly.

This option causes the Compiler to generate six (6) extra bytes of code for each line or loop. If you are short of memory, don't use it.

NOTE

> **ONERR GOTO** branches will work without this compiler option, but the program will not be able to **RESUME** execution.

See also the **RESUME** command in Part Three, Chapter Fourteen.

Example:

```
PROGRAM Example
@ ERROR
<Program Code>
```

EXTEND

Use this compiler option to increase the range and accuracy of all real simple variables and arrays from 7 or 8 places to 19 or 20 places. This is especially useful in scientific or technical programs in fields such as microbiology, engineering, and astronomy.

Normally, four bytes of memory are allocated for each real simple variable or real array element, but if the **EXTEND** compiler option is used, ten bytes of memory will be allocated for each real variable or array element. Each real value will have an accuracy to 19 or 20 places, with a range of $\pm 10^{\pm 4096}$. Because floating point calculations are always carried out using extended arithmetic, there will be little difference in execution speed if this option is used.

Example:

```
PROGRAM Example
@ Extend
<Program Code>
```

LIST

The **LIST** compiler option instructs the Compiler to generate a source code listing as the program is being compiled.

A compiled source code line consists of the sequential line number, the nesting level, the relative (not actual) address expressed in decimal notation where the first byte of this line will reside, this address expressed in hexadecimal notation, and the source code line. A symbol table dump of the variables followed by the memory usage information is displayed after the program lines. See "Compiled Listing" later in this chapter for additional information.

LONGINT

Use this compiler option to increase the range of all integer variables and integer

arrays from five to ten places. If you are not using extended real numbers, the accuracy (but not range) of long integers is actually greater than that of real numbers, but their execution is much faster. This may be an important consideration for you.

Two bytes of memory are normally reserved for each integer variable or integer array element. When the **LONGINT** compiler option is used, four bytes of memory are allocated for each integer variable or array element. Each integer value will have a range of ±2,147,483,647.

NOTE

> Because integer arithmetic is very fast, there will probably be little difference in execution speed if this option is used. However, twice as much memory is required for integer storage. This is only a factor if you have limited memory and very large integer arrays.

Example:

```
PROGRAM Example
@ LONGINT, CODE
```

NOGOTO

This compiler option is intended for teachers who wish to restrict their students to structured programming without using **GOTO**s or **POP**s. By specifying this option, **GOTO** and **POP** statements will become illegal and cause a compiler error if used. The reserved words **GOTO** and **POP** may then be used as variable names.

The **ONERR GOTO** statement is not affected by the **NOGOTO** compiler option.

Example

```
PROGRAM Example
@ NOGOTO
```

NOT_C

This compiler option turns off the <Control>C interrupt command ability during program execution. Pressing <Control>C from the keyboard during a program's execution will have no effect on programs if this option is used.

Example:

```
PROGRAM Example
@ NOT_C
```

IMPORTANT

> Do not use this compiler option until the program is thoroughly debugged.

OPTIMIZ

The compiler normally generates line information to let the programmer know where a run time error has occurred in the program.

This compiler option turns off the consecutive line information usually generated by the Compiler. This gives programs a small, but noticeable increase in execution speed. Use it to speed up the program once it is completely debugged.

IMPORTANT

> The most important function of **OPTIMIZ** is to conserve memory. A program using **OPTIMIZ** is about one-third smaller than one without it.

PRINTER

This option functions the same way as the compiler option **LIST**, except output is directed to the printer instead of the screen. Output is directed through slot one unless changed by the Shell **PRINTER** command. The listing is printed according to values set in the Control Panel.

Example:

```
PROGRAM Example
@ PRINTER
```

VAR2

This option restricts to two (or three if an exclamation mark (!), a dollar sign ($), an ampersand (&) or a percent sign (%) is at the end of the variable name) the number of significant characters in a variable name, as in Applesoft BASIC.

NOTE

> Use this compiler option only if you are compiling source code files converted from Applesoft BASIC programs and do not wish to modify the variable names.

Example:

```
PROGRAM Example
@ VAR2
```

Compiler Aliases

ALIAS "User statement" = "BASIC Expression"
~User Statement

The **ALIAS** compiler directive lets the programmer change a *Micol Advanced BASIC* statement or expression to another statement or expression of his/her choosing.

ALIAS definitions are placed after the compiler options and before the variable type declarations.

The purpose of Aliases is to give more meaning to your programs. For example, if you have a loop which you wish to execute as long as the computer is on, you may substitute **Forever** for the *Micol Advanced BASIC* code that actually creates this condition.

An Alias is defined by using the keyword **ALIAS** followed by the replacement statement, followed by an equal sign, followed by the statement that the Compiler will substitute. Both strings on either side of the equal sign must be enclosed in quotation marks ("").

To make the replacement within a program, use the tilde (~) character followed by the user replacement string (without the quotation marks). When the Compiler detects the tilde, it will search the **ALIAS** list (created at the top of the program) for a match and make the replacement during compilation.

An Alias substitution may not be the first executable line or the Compiler will issue an error.

IMPORTANT

> All string literals used with Aliases are case sensitive; the Alias definition and user statements must exactly match, or no change will occur. No error will be flagged, but as no substitution will occur, an error condition will undoubtably arise when the line is compiled.

Example:

```
PROGRAM Example
ALIAS "Pi" = "3.14159"
ALIAS "Forever" = "UNTIL 1 = 2"
ALIAS "Clear Screen" = "HOME"
INT (A - Z)
(Start of executable code follows)
Trig_Const = ~Pi
~Clear Screen
REPEAT
```

```
    PRINT "Trig_Const = ";Trig_Const
~Forever
END
```

NOTE

> If two Alias declarations beginning with the same letters are declared, the wrong match may be made. This problem may be avoided by declaring the longer Alias declaration first.

Example:

```
PROGRAM Example
@ List
{Note the order here, it's important,
if reversed, only first Alias matched}
ALIAS "Pi_Long" = "3.14159"
ALIAS "Pi" = "3.14"
ALIAS "Circumference" = "20.0"
{Note! Order here is unimportant}
Diameter = ~Circumference / ~Pi
Diameter = ~Circumference / ~Pi_Long
```

NOTE

> When the Compiler generates a compiled listing, the Alias substitution made during compilation will be displayed. If you are getting error messages that don't make sense to you, try generating a compiled listing.

Variable Type Declarations

INT(letter1-letter2) : STR(letter3-letter4)

The variable type declaration allows the programmer to write the integer and string identifier's of simple and structured data types (simple variables and arrays) without the percent (%) or string ($) character required by Applesoft BASIC. These statements are optional and are placed before the arrays are declared.

To declare a range of variables, specify the data type (**INT** for integer or **STR** for string) followed by a range of letters in parentheses. Separate the variable type declarations by a colon (:).

The range of letters used for integer variables must be different from the range used for string variables. If the declarations between the integer and string data types should

overlap, the Compiler will indicate that an error occurred.

Any possible implied declaration with the following characters, a "%" for integer "&" for real, "$" for string and "!" for boolean after the variable name, will override the declaration types mentioned above. These characters are still significant. Note that there is no implicit declaration for booleans.

NOTE

> A one letter range may be declared by specifying the same letter twice in the declaration.

Example:

```
PROGRAM Example1
INT (K-K): STR (S-S)
```

Variables beginning with the letter K and having no special character at the end will be integer variables, while variables beginning with the letter S and having no special character at the end will be string variables.

Example:

```
INT (I-R): STR (S-Z)
First$ = ""
Second = ""
Second$ = ""
Second% = 0
Third = ""
Forth = 0.0
Ninth = 0
Ninth& = 0.0
```

First$ is a string variable

Second is a string variable

Second$ is a string variable different from Second

Second% is an integer variable

Third is a string variable

Forth is a real variable

Ninth is an integer variable

Ninth& is a real variable

In the above example, all variables which begin with letters A through H will be real variables (unless followed by the character !, % or $). All variables which begin with letters I through R will be integers (unless followed by the character &, ! or $), and all variables which begin with letters S through Z will be string variables (unless followed by the character &, % or !). Second and Second$, although string variables in the above

example, are different variables.

Compiled Listing

Whenever you use the **LIST** or **PRINTER** compiler options, you generate what is called a compiled listing. This compiled listing contains much information that may be of use to you during your program development.

Here is an example of a compiled listing:

```
PROGRAM Example
Compiled listing of Example
3   [0]    0      $0000      HOME
4   [0 ]   16     $0010      FOR Counter = 1 TO 10
5   [1 ]   56     $0038           PRINT "Counter = ";Counter
6   [1 ]   84     $0054      NEXT Counter
7   [0 ]   100    $0064      END
```

No errors in compilation

SYMBOL TABLE DUMP

```
 1 R0205          10 R0209          Counter R0201

530   bytes required for variable storage
1     bank required for string storage
118   bytes code generated
```

Program Lines

The first position in the program line is occupied by the sequential line number. This is the number that is used whenever a line is referenced.

The second position in the program line is occupied by a number in square brackets ([]). This number is the level of nesting in which the program line appears. For example, this number tells you how many **FOR** loops or **IF** statements are currently active at the beginning of the line. This can be very valuable debugging information.

The third value displayed is the relative address in decimal, followed by the relative address in hexadecimal, followed by the actual program line itself.

Symbol Table Information

After the program lines, the Compiler displays the list of all types of simple and structured variables used in the program.

The Symbol Table contains the relative hexadecimal addresses of all simple boolean,

integer, real and string variables, numeric constants (literals), and arrays.

The capital letter in the address indicates the type of the variable. B indicates the address of a boolean, I indicates the address of an integer, R indicates the address of a real, and S indicates the address of a string

The local simple variables (accessible only to Functions or Procedures) are the first variables listed in alphabetical order. The values assigned to simple and structured data types are listed second, also in alphabetical order. The simple and structured data types are listed third, also in alphabetical order.

During compilation, the names of all variables have been converted into uppercase letters and so appear in the Symbol Table. The name of a local simple variable is preceded by a number sign (#). An array name is followed by a left parenthesis [(].

Statistical Information

After the Symbol Table has been displayed, there appear a few lines which give a bit of helpful information. These lines are:

```
530   bytes required for variable storage
1     bank required for string storage
118   bytes code generated
```

The first line indicates how many bytes of memory were used by the boolean, integer, floating point, and string variables and arrays, and all literals. The second line shows how many banks of memory are allocated for the string buffer. The third line shows how many bytes of program code were generated by the Compiler.

NOTE

> The program will actually occupy a bit more memory than is specified by the last statistical information line. This is because some memory will be occupied by code generated by the Linker to store initialization information. The first line of statisical information (bytes required for variable storage) will give you a rough idea of how much more.

Chapter Two

Basic Elements of the Language

Overview

In order to understand any computer language, you first have to learn the basic elements comprising the language. This chapter will deal with these basic elements that you will need to build upon to create *Micol Advanced BASIC* programs.

Basic Symbols

Micol Advanced BASIC uses letters of the alphabet, digits, and special characters to form the symbols of the language.

Digits (0 - 9)

Digits are used to form numbers, keywords, identifiers, and character strings.

Letters (A - Z, a - z)

These characters are used to make keywords, identifiers and character strings.

Special Characters

These characters (!, @, $, %, &, _ , ~ (,) {, }) may be used to give a specific meaning to identifiers, declare an array, specify a comment, etc.

Separators

Colon

The colon (:) separates two statements on a line.

Comma

The comma (,) separates two or more constants or variables on a line.

Parentheses

The parentheses [( )] separate complex string and math expressions as well as array element designators.

Space

A space specifies where one symbol ends and another symbol begins.

Variable Names

A variable name consists of letters, digits and the underscore character. A variable name may have up to 62 characters, but it is wise to limit its length to about 20 characters or less. Unless the **VAR2** compiler option is used, all characters are significant.

The variable name must begin with a letter of the alphabet. Characters may be either in upper or lowercase, but lowercase letters will be converted to uppercase during compilation.

A variable name may not be a reserved word and should be meaningful. By convention, variable names are easily destinquished from reserved words in that reserved words are entered in uppercase letters while variable names are in lowercase with only the first character in uppercase.

Unlike Applesoft, a variable name under *Micol Advanced BASIC* may contain a reserved word within it. For example Go_Home and For_Ctr are legal variable names.

Examples:

```
Factorial, General_Ledger, Tax, Price
```

instead of variables like

```
Z13, XYZ, A123
which are not meaningful.
```

These variables are not legal:

```
General.Ledger, 10%_Tax, Home
```

Variable Data Types

The data type defines the interpretation of values that simple variables, arrays, and expressions may have. *Micol Advanced BASIC* has four simple data types and four structured data types, one for each simple data type.

Simple Data Types

Micol Advanced BASIC supports boolean, integer, real and string variables as simple

data types.

Booleans

A boolean variable is assigned either a value of TRUE or FALSE. The function of a boolean variable is to be set to one state or the other, so that necessary action(s) may be taken later (this is often called a flag or switch). A boolean occupies only one byte of memory. The initial value of a boolean variable is FALSE.

The normal convention for variable names applies, but an exclamation mark (!) must be added at the end of the variable name to force the Compiler to type the variable as boolean.

Boolean variables may also hold an indefinite value if necessary. See Controlled Uncertainty in Chapter Twelve of this Part for details.

Examples:

```
Flag! = FALSE {Init flag for test}
Number = 10
IF Number > 6 THEN Flag! = TRUE
IF Flag! THEN BEGIN
   PRINT "Number is greater than Six"
ENDIF
```

NOTE

> The keyword **TRUE** or **FALSE** is displayed to the current output device when a boolean variable or relational expression is evaluated within a **PRINT** statement.

Example:

```
PRINT 1 <> 2 {Will print TRUE}
```

Integers

An integer value represents a numeric value that has no fractional part and has a limited range. The initial value of an integer variable is 0.

The normal convention for naming variables applies, but a percent sign (%) must be added at the end of the identifier to force the Compiler to type the variable as integer unless an **INT** variable type declaration is in force.

Example:

```
Dividend% = 1
Divisor% = 3
PRINT Dividend% / Divisor% {Result is 0}
```

Micol Advanced BASIC for the Apple IIGS has two ranges for integer values: Short

Integer and Long Integer.

Short Integers

Micol Advanced BASIC can represent short integer values in the range ±32767. Negative values are represented as two's complement numbers. A short integer occupies two bytes of memory.

Example:

```
Short_Integer% = 32000
```

Long Integers

Micol Advanced BASIC can represent long integer values in the range ±2,147,483,647. Negative values are represented as two's complement numbers. A long integer occupies four bytes of memory.

Long integer arithmetic is activated with the compiler option **LONGINT**. See also Chapter One in this Part under Compiler Options.

Example:

```
Long_Integer% = 2146493697
```

Real (Floating Point)

A real number represents a value that can represent a large range of values and may have a fractional part. The default number of significant digits that may accurately be represented is seven digits. The initial value of a floating point variable is 0.0.

The normal convention for naming variables applies, but an ampersand (&) may be added at the end of the identifier to force the Compiler to type the variable as a real to override an **INT** or **STR** variable type declaration.

Examples:

```
Dividend& = 1
Divisor& = 3
PRINT Dividend& / Divisor&
{Result is 0.3333333}
```

Micol Advanced BASIC IIGS has two ranges of precision for floating-point numbers: Single and Extended.

Single Precision

Single precision reals can represent values in the range $\pm 3.4 \times 10^{\pm 38}$. Seven digits are significant in calculations. A single precision real variable uses four (4) bytes of storage.

Examples:

```
PRINT EXP(1.0) {Prints 2.718282}
```

```
PRINT EXP(2.0) {Prints 7.389056}
```

Extended Precision

Extended precision reals can represent values in the range $\pm 1.0 \times 10^{\pm 4096}$. Nineteen digits are significant in calculations. A real variable occupies 10 bytes of storage in memory.

Extended precision arithmetic is activated with the compiler option **EXTEND**. See also Chapter One in this Part: Compiler Options.

Examples:

```
PRINT EXP(1.0) {Prints 2.718281828459045 22}
PRINT EXP(2.0) {Prints 7.389056098930650 22}
```

Scientific Notation

Large real values that are too large to be represented in decimal format (more than seven digits using single precision) may be represented using scientific notation. Scientific notation representation uses a multiple of 10 raised to a power of 10. Values may either be set or displayed using scientific notation.

Example:

```
Real& = 4E6 {Equivalent to 4,000,000 or 4 x 10^6}
Real& = 4E-6
```

Strings

A string is a sequence of characters including letters, digits, special characters, the space character and control characters.

The normal convention for naming identifier applies, but a dollar sign ($) must be added after the variable name to force the Compiler to type the variable as string. The dollar sign may be omitted if the **STR** Variable Type Declaration applies to the variable identifier in question.

The length of a string is equal to the number of characters inside it. Each string variable occupies three (3) bytes in data memory plus four (4) bytes of system information in addition to the characters in a separate string buffer. The maximum size a string can grow is 1023 characters. However, a string literal can only have 251 characters.

Micol Advanced BASIC uses two types of string storage: static and dynamic storage.

Static Storage

Static strings are used when a string of characters is encased in double quotation marks ("") within a program.

Example:

```
Name$ = "Steve"
```

Dynamic String Storage

Dynamic string storage is used in all other cases. A dynamic string variable holds the address where the actual string is in memory, but the actual string is stored in a special buffer area reserved for this purpose (see the compiler option **BANK_NO** for additional information).

Structured Data Types: The Array

Micol Advanced BASIC has four kinds of structured data types: Arrays of boolean, integer, real and string.

Declaring Arrays

**DIM Array_Name [!,%,&,$] (Size) **
[{, Array_Name [!,%,&,$] (Size) }]

Arrays are always declared and dimensioned at the beginning of the program after the optional compiler options, the **ALIAS** declarations, and **DATA** statements.

An array is a set of data of the same type. Each piece of information is called an element. Access to each element is made via a subscript (an index number to the array).

The **DIM** statement will allocate to the array the number of elements plus one, element 0 being the first array element.

NOTE

> Unlike Applesoft, all arrays, no matter how small, must be declared before they are used. If an array needs more memory than is available to the computer, an "Out of memory" error message will be displayed when the execution of the program begins. **DIM** sizes may only be numeric constants, not variables.

To declare an array, use the reserved word **DIM**, give the array any legal variable name followed by its size between parentheses.

To declare more than one array, separate each array name and size by a comma.

Multi-dimensional Arrays

**DIM Name [!, %, &, $] (Size [{ , Size }]) **

[, Name [!, %, &, $] (Size [{ , Size }])]

A multi-dimensional array is an array having two or more dimensions. A different size may be used for each dimension.

To add another dimension to an array, enter a comma followed by another size value after the first size dimension. To declare more than one array, separate each array name and size declaration by a comma.

Example:

```
PROGRAM Month_Temp
DATA 23, 34, 32, 12, 11, 22, 20
DATA 18, 14, 17, 15, 16, 13, 12
DATA 11, 10, 7, 3, 0, -3, -6, -14
DATA -17, -19, -15, -12, -10, -8
DIM February (3, 6)
HOME
Temp_Total& = 0
FOR Week = 0 TO 3
   FOR Day = 0 TO 6
     READ Temperature% {Must read integer data}
     February (Week, Day) = Temperature%
     Temp_Total& = Temp_Total& + February (Week, Day)
   NEXT Day
NEXT Week
Aver_Temp = Temp_Total& / 28
PRINT "The average temperature for February is: ";Aver_Temp
END
```

Although it is possible to have an array with more than three dimensions, it is rare that one has to use such arrays. Review the logic of the program if such a large array is required.

Array Memory Usage

A boolean array uses one byte to hold the number of dimensions, two bytes per dimension size plus one byte times the number of elements plus one.

An integer array uses one byte to hold the number of dimensions, two bytes per dimension size plus twice the number of elements plus two bytes (four times the number of elements plus four bytes, if the **LONGINT** option is used).

A real array uses one byte to hold the number of dimensions, two bytes per dimension size plus four times the number of elements plus four bytes (ten times the number of elements plus ten bytes if the **EXTEND** option is used).

A string array allocates one byte to hold the number of dimensions, two bytes per

dimension size plus three times the number of elements plus three bytes.

Array Nesting

Under most circumstances, integer index variables should be used with boolean, integer and string arrays; real index variables should be used with real variables to reduce array access time.

WARNING

> If arrays are nested, that is, an array element is used as an array counter, you must nest arrays of the same type or an error will result. This means you may only nest real arrays within real arrays and integer arrays within string, integer and boolean arrays.

Operators

Micol Advanced BASIC has three types of operators: arithmetic, logical and relational.

Arithmetic Operators

Arithmetic operators are used with either integer or real variables. The arithmetic operators are addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^) and modulo (**MOD**). Here are some general rules to note:

1. An overflow error will be indicated when the result of any calculation is over the allowed range for that variable type.
2. Exponentiation works only with positive numbers; negative numbers will result in an error. Zero raised to any power is zero. Any positive number raised to the power of zero equals one.
3. The asterisk (*) is used in many programming languages as the operator for multiplication to avoid confusion with the capital letter X.
4. The unary minus sign (-) indicates the change of sign when it is used with one operand. Unary plus (+) is redundant and is ignored by the Compiler.

Relational Operators

A relational operator tests relationships between two conditions and produces a boolean result (**TRUE** or **FALSE**). It is this operation, more than anything else, that allows your programs to "think".

The relational operators are: less than (<), less than or equal to (<=), equal to (=), not equal to (<>), greater than or equal to (>=) and greater than (>).

Logical Operators

Logical operators operate on relational expressions to produce a boolean result of **TRUE** or **FALSE**.

The logical operators are: **NOT, AND, OR.**

Example:

```
IF (Real < 5.3) AND (NOT (Integer% > 20)) THEN \
   Flag! = TRUE
```

Evaluation of an Expression: Precedence Rules

The evaluation of an expression is done following a priority list established by math conventions. If the priority of the expressions is equal, the evaluation is done from left to right. The established math priorities are as follows:

1.	Expressions between parentheses	( )
2.	Unary operators	- , +
3.	Exponentiation operator	^
4.	Multiplication, Division, and **MOD** operators	* / **MOD**
5.	Addition and Subtraction operators	+ -
6.	Relational operators	>, >=, <=, <, <> ,=
7.	AND logical operator	**AND**
8.	OR logical operator	**OR**
9.	NOT logical operator	**NOT**

You may wish to use parentheses to make certain an expression is evaluated in the intended order. An expression may contain any number of parentheses.

Hexadecimal Literals

A hexadecimal number may be assigned to any integer or real variable. A hexadecimal number is a base 16 number and is always preceded by a dollar sign ($) and consists of the digits 0 through 9 and the letters A through F.

Example:

```
Hex_Number% = $12FF
Real = $FFFFFF
```

Mixed Arithmetic Expressions

What dictates how the Compiler evaluates a line of code? Basically, the Compiler determines the type of calculation to perform by the first data type (real or integer) it

encounters in a statement.

Micol Advanced BASIC handles mixed arithmetic very well, but extra code will need to be generated which requires extra time to execute. If possible, it is best to be consistent with your variable types when coding.

Expressions with Simple Variables

Example:

```
Real_Var& = Integer% * 3 + Real&
```

Because this assignment is made to a real variable, the above formula will be treated as a real formula. The integer value in variable Integer% will be converted to real.

Example:

```
Integer% = Real1& * Real2& / Real3& + Real4&
```

In this example, each real value must be converted to its integer equivalent before the expression can be evaluated. It would be better to assign the formula to a real variable, then reassign the real variable to an integer variable in another statement.

Example:

```
Real& = Real1& * Real2& / Real3& + Real4&
Integer% = Real&
```

Expressions with Arrays

As with simple variables, the Compiler determines the type of calculation by the first variable type it encounters. What is different with arrays is that the array counter is also effected. It is best to maintain the same type of array and array counter. Integer arrays should have integer counters, and real arrays should have real counters. String and boolean arrays should use integer counters.

Example:

```
Array& (Real&) = 3
Array% (Int%) = Integer%
Array$ (Int%) = "String"
Array! (Int%) = TRUE
```

Any other choices from the above examples will force a conversion to the other type before the correct array element can be accessed.

Simple Variable Declaration

In *Micol Advanced BASIC*, simple variables may be declared in one of two ways: implicitly and explicitly. Implicit declarations are done simply by using the variable.

The Compiler determines whether a variable has not been used before and automatically allocates space for it if need be. This is the method used by Applesoft BASIC.

Micol Advanced BASIC also offers the option of explicitly declaring a simple variable, similar to the way arrays are explicitly declared. This means, you must state within your program, that you are using this particular variable. This is similar to the system used in Pascal and C. This method almost completely eliminates the possibility that you will later enter this variable incorrectly.

Explicit variable declarations are also a very good idea for documentation purposes, as you can easily determine all variables used within the program. You may wish to include comments to better explain the variable's usage.

Although the explicit declaration adds some complexity to the language, it is probably preferable to use implicit declarations as program maintainance is made easier.

DECLARE Boolean!, Integer%, Real&, String$

To explicitly declare a variable, enter the reserved word **DECLARE** followed by a list of simple variables separated by commas. A program may have as many **DECLARE** statements as needed, but they must be the first and only statement on a program line.

IMPORTANT

> If no **DECLARE** statement is encountered in the program, all simple variables will be placed automatically into the Symbol Table. Once a **DECLARE** statement is detected in the program, all subsequent variables, not already defined, must be declared by a **DECLARE** statement; otherwise, the Compiler will signal an error. If you attempt to **DECLARE** a variable a second time, you will receive an error at compile time.

Example:

```
PROGRAM Declaration
DECLARE Real, Integer%, String$
Real& = 5.0
Integer% = 25
String$ = "This variable has been declared"
Any_Thing% = 23 (Error here, not in DECLARE list)
```

Variable Assignments

[LET] Avar = Aexpr
[LET] Svar = Sexpr

The assignment instruction is the equal sign (=) and is used to assign an expression to a variable. The equal sign also implicitly declares this variable if it has not been used before (if **DECLARE** is not being used). The expression is always located on the right side of the equal sign. The result is stored in the variable to the left of the equal sign.

The reserved word **LET** may be used to specify an assignment. **LET** was retained solely for compatibility with Applesoft BASIC and is ignored by the Compiler. Use **LET** only if you wish to stay within the original definition of Dartmouth BASIC.

Examples:

```
Number& = 35.1
Number% = 10 * 2 / 5
String$ = "This is a small message"
Boolean! = TRUE
```

Initializing the Data Space

CLEAR

CLEAR will reinitialize all simple and structured variables. All numeric variables will be set to zero, all strings will be set to empty and booleans will be set to **FALSE** as was the case when the program was first executed.

Example:

```
Variable = 10
PRINT Variable {Value is 10}
CLEAR
PRINT Variable {Value is 0}
```

WARNING

> An implicit initialization is done at the first line of executable code. Branching to this line of code will reset all variables to zero or null as if the program restarted. Do not use **CLEAR** from any segment other then segment zero or you will crash the computer.

Chapter Three

Mathematical Functions

Overview

The mathematical functions under *Micol Advanced BASIC* have been classified into two categories: general purpose functions and trigonometric functions. All use integer or real arguments and yield integer or real results.

All calculations are made using single precision arithmetic unless the **EXTEND** and/or **LONGINT** compiler options are used (all examples use single precision).

General Purpose Functions

ABS (Aexpr)

ABS (Absolute) returns the absolute (positve) value of the argument. The argument may be negative, zero or positive.

Example:

```
Number% = ABS (-10)
PRINT Number% {Will print 10}
```

EXP (Aexpr)

EXP (Exponent) yields the value of the constant e (2.718281828) raised to the power of the argument. An argument smaller than zero always returns zero. An argument of zero returns one.

Example:

```
Exponent = EXP(10)
PRINT Exponent
```

INT (Aexpr)

INT (for integer) returns the whole number portion of the argument, discarding the fractional part, if any.

NOTE

> **INT** does not convert a real argument to an integer as the function name implies, but simply truncates the value. A real value remains a real value after **INT** has performed its work. In *Micol Advanced BASIC* there are no functions to convert values from real to integer and integer to real, but rather this conversion is done automatically and need not concern the user.

Examples:

```
Real_Num& = INT (95.9)
```

LOG (Aexpr)

LOG (for logarithm) yields the natural logarithm base e (e = 2.718282) of the positive argument passed to it. If an argument equal to zero or negative is passed, a run time error will occur.

Example:

```
Logarithm = LOG (10)
```

MOD

MOD (for modulo) returns the remainder of the real or integer division of the nominator by the denominator.

Example:

```
Nominator% = 25
Denominator% = 4
Remainder% = Nominator% MOD Denominator%
PRINT Remainder% (Writes a 1)
```

ROUND (Aexpr)

ROUND returns the rounded value of the argument. For a positive value, if Aexpr is between x.5 to x.9, the result is rounded upward. If the value is between x.0 to x.4, the number is rounded downward.

For a negative value, if Aexpr is between -x.5 to -x.9, the number is rounded downward. If the value is between -x.0 to -x.4, the value is rounded upward.

If the number to be rounded is assigned to an integer result, the value will be returned unchanged.

Example:

```
Kappa& = 1.8
Delta& = ROUND (Kappa&) {Delta& will = 2}
Kappa& = 1.4
Delta& = ROUND (Kappa&) {Delta& will = 1}
```

SGN (Aexpr)

SGN returns the sign of the argument. A negative argument returns a negative one. If the argument equals zero, **SGN** returns a zero. A positive argument returns a one.

Example:

```
Result = SGN (0) {Equals zero}
Result = SGN (-123) {Equals negative one}
Result = SGN (123) {Equals positive one}
```

SQR (Aexpr)

SQR returns the square root of the argument. The argument must be a positive, real or integer expression, otherwise a run time error will occur.

If the value returned by **SQR** is multiplied by itself, the result may be less than the initial value. The loss of precision occurs because of truncation.

Example:

```
FOR Count% = 1 TO 10
    Product% = Count% * Count%
    PRINT Count%, Product%, SQR (Count%)
NEXT Count%
```

Trigonometric Functions

Micol Advanced BASIC has four trigonometric functions. All arguments or results are expressed in radians (not degrees).

ATN (Aexpr)

ATN yields the arc tangent (inverse tangent) of the parameter. The value returned represents an angle expressed in radians in the range $\pm\pi / 2$.

Example:

```
Tangent& = TAN (Radians)
Inv_Tan& = ATN (Tangent&)
```

COS (Aexpr)

COS returns the cosine of the argument. The cosine is the ratio of the length of the adjacent side to the length of the hypotenuse (in a right-angled triangle). The argument is the angle as expressed in Radians.

Example:

```
Cosine& = COS (30 * Pi& / 180)
```

SIN (Aexpr)

SIN yields the sine of the argument. The sine is the ratio of the length of the opposite side to the length of the hypotenuse (in a right angled triangle). The argument is the angle as expressed in Radians.

Example:

```
Sine& = SIN (60 * Pi& / 180)
PRINT Sine&
```

TAN (Aexpr)

TAN returns the tangent of the argument, (a number between 0 and the accuracy limit of the data type used). The tangent of 90 degrees is infinity.

Example:

```
Tangent& = TAN (Radians&)
```

Radian/Degree Conversion Functions

Most of you are used to working with degrees instead of radians. You may find the following conversion Functions useful to use within your programs.

```
{Take Degree as input}
FUNC DegreeToRadian [Degree&]
   Pi& = 3.14159265
   Radian& = Degree& * (Pi& / 180)
ENDFUNC [Radian&] {Return Radian as output}

{Take Radian as input}
FUNC RadianToDegree [Radian&]
   Pi& = 3.14159265
   Degree& = Radian& * (180 / Pi&)
ENDFUNC [Degree&] {Return Degree as output}
```

Chapter Four

Strings

Overview

A string may be thought of as text. Each word or sentence of this manual may be thought of as a string. All data sent to the screen or the printer are sent as strings.

Under *Micol Advanced BASIC*, strings are dynamically stored. This means that string lengths do not have to be declared in advance.

This section deals with strings and string manipulation functions at your disposal under *Micol Advanced BASIC*. You must pay special attention to this chapter as some of the string functions operate somewhat differently than under Applesoft. Also, there are several additional string functions that give the string handling abilities of *Micol Advanced BASIC* much greater power than any other language you have probably seen.

String garbage collection, a topic not well understood by many users, is also discussed in this chapter.

String Function Notes

Here are some things to pay special attention to:

1. No string shaping function such as **LEFT$** may be used until the string argument has been explicitly given a value.
2. String shaping functions assume integer arithmetic and will make the conversions from real to integer as needed. The sole exception is **STR$** which assumes a real value as its parameter and will make the conversion from integer to real as needed. Therefore, any real number within string functions, except **STR$**, will be converted to integer before the manipulation is done. Since the type conversion delays the programs a bit, use integer values whenever practical.
3. Strings may grow to a maximum length of 1023 characters. However, static strings such as "This is a string" may only have a maximum length of 251 characters.

The ASCII Character Set

Each character has a numeric value, and this numeric value is used in order to evaluate strings.

"A" < "B" and "B" > "A" are true. If you look at the ASCII chart (Appendix F), you will see that "A" has the numeric value 65 and "+" has the numeric value 43. These numbers are used to evaluate string expressions.

String Comparisons

Strings are compared using relational operators to determine if, for example, one string is the same or is different from another string. Comparisons are made using the ASCII numeric value of each character in both strings.

Examples:

```
"Ronald" = "Ronald"
"Ronald" <> "RONALD"
"Ronald" < "Steve"
"Walter" > "Steve"
```

By comparing one string with another, strings may be sorted in alphabetical order or inverse alphabetical order. See also the **ASC** and **CHR$** conversion functions.

String Concatenation

Concatenation is the act of merging two or more strings into one. The concatenation operator is the plus sign (+). The maximum length a string can grow under concatenation is 1023 characters. Any attempt to create a string greater than 1023 characters will result in an error during program execution.

Examples:

```
String$ = "This is " + "one big " + "string"
String1$ = String1$ + String2$
```

Conversion Functions

The following functions are used to return numeric results for string arguments or string results for numeric arguments.

ASC (Sexpr)

ASC returns the ASCII value of the first character of the string argument. If the string is empty (has no characters in it), a value of zero will be returned.

The value returned is always between 0 and 127. Most characters, however, are actually stored internally with a value greater than 127. To know the true value of the character, **PEEK** at location 202 (**True_Value**) in direct page immediately after using the **ASC** function. (See Appendix F: the ASCII chart.)

Example:

```
Letter$ = "A"
ASCII = ASC (Letter$) {Prints 65}
```

CHR$ (Aexpr)

CHR$ takes the numeric argument and returns the character corresponding to its ASCII value. The argument must be between 0 and 255 or a run time error will occur. Values greater than 128 will repeat the text mode character set. (See Appendix F: the ASCII chart.)

Example:

```
Letter$ = CHR$ (65)
PRINT Char$ {Prints the letter A}
```

LEN (Sexpr)

LEN (Length) returns the number of characters within a string or string variable. If no character appears within Sexpr, **LEN** will return a zero. All strings have a length of zero initially. You may need to use **LEN** to check the length of a string when using a string shaping function, as a possible error condition may arise.

Example:

```
String$ = "Micol Systems Inc."
PRINT "Number of string characters is:"; LEN (String$)
```

LEN returns a value of 18.

STR$ (Aexpr)

STR$ converts the numeric argument into its string equivalent.

Example:

```
String1$ = STR$(12.34)
```

NOTE

> The string "12.34" and the real number 12.34 will appear the same when they are displayed; however, inside the computer's memory, they are stored quite differently.

VAL (Sexpr)

The **VAL** function converts the contents of the string argument into its numeric equivalent. **VAL** removes any leading spaces from the string argument before doing the evaluation.

If **VAL** evaluates an argument with non-numeric characters, **VAL** will convert and return all the digits appearing before the non-numeric character or space. If the first character in the argument is non-numeric, **VAL** yields a zero.

Example:

```
String$ = "12.34"
Real& = VAL (String$)
```

String Searches

The following function is very useful and has no equivalent in Applesoft. Its purpose is in searching for sub-strings within a string, but this has very many applications seemingly unrelated to string searches. Examples throughout this manual will demonstrate some of these uses.

INDEX (SubString$, String$, [Aexpr])

INDEX will return the position number of the first character where SubString$ occurs in String$ from one to the length of String$. If SubString$ does not appear within String$, a zero will be returned.

An optional occurrence value ranging from 1 to 255 may also be specified. The match will not be made unless the stated instance of SubString$ exists.

Example 1:

```
String$ = "This is a string"
PRINT INDEX (" is ", String$)
```

The **PRINT** statement will display 5. The first space character is the fifth character of the string.

Example 2:

```
Alpha$ = "abcdebxyz"
Beta$ = "b"
PRINT INDEX (Beta$, Alpha$, 1)
PRINT INDEX (Beta$, Alpha$, 2)
```

The first **PRINT** will show that the first occurrence of "b" is at the 2nd position and the second occurrence will show the second "b" at the 6th position in the string.

Example 3:

```
Allowed$ = "AEIOUaeiou"
REPEAT
   GET Char$
UNTIL INDEX (Char$, Allowed$) > 0
PRINT Char$
```

This code will allow only a vowel to be entered.

String Manipulation

The following functions will allow you to manipulate strings in any manner required by your program. This string shaping ability is one advantage BASIC has over almost any other language and *Micol Advanced BASIC* has more than most BASICs.

INSERT$ (String1$, String2$, Pos_Number)

To write over a portion of a string using the contents of another string, use **INSERT$**. Both string arguments must be string variables. The contents of String1$ will be used to write over the characters of String2$ starting at the specified position. Each character will be copied over String2$ until all characters are copied or the end of either string is reached.

Example:

```
String1$ = "Italy"
String2$ ="The rain in Spain falls mainly on the plain."
INSERT$ (String1$, String2$, 13)
PRINT String$
```

This code will print "The rain in Italy falls mainly on the plain."

LEFT$ (Svar, Aexpr)

LEFT$ yields the number of characters specified by Aexpr starting from the left side of Svar. If the number of characters requested is greater than the string length, a run time error will occur. If in doubt, check the string length with the **LEN** function before executing this function.

Example:

```
String$ = "Micol Systems Inc."
PRINT LEFT$ (String$, 5)
```

The word "Micol" will be printed.

LOWER$ (Svar)

LOWER$ changes all the uppercase characters of a string into lowercase characters. All other letters in the string variable are left unaltered. A string variable is the only argument accepted.

Example:

```
String$ = "ABCDEFGHIJ"
Low$ = LOWER$ (String$)
PRINT Low$ {Will print abcdefghij}
```

MID$ (Svar, Aexpr1 [,Aexpr2])

MID$** returns a substring of Svar starting at Aexpr1. If Aexpr2 is not present, the entire string is returned from Aexpr1 to the end of Svar, otherwise **MID$ returns the number of characters specified. If the starting character position is beyond the last character of Svar, a run time error will occur.

Example:

```
String$ = "Micol Systems Inc."
PRINT MID$ (String$, 7, 7)
```

The word "Systems" will be printed.

RIGHT$ (Svar, Aexpr)

RIGHT$ returns the characters specified by Aexpr starting from the right side of Sexpr. If the number of characters requested is greater than the length of Svar, a run time error will occur. If in doubt, check the string length with the **LEN** function before executing this function.

Example:

```
String$ = "Micol Systems Inc."
PRINT RIGHT$ (String$, 12)
```

The words "Systems Inc." will be printed.

UPPER$ (Svar)

UPPER$ will change all lowercase characters of a string into uppercase characters. All other characters in Svar are left unaltered. A string variable is the only parameter accepted.

Example:

```
String$ = "abcdefghij"
Up$ = UPPER$ (String$)
PRINT Up$ {Will print ABCDEFGHIJ}
```

WARNING

> Avoid writing string manipulation functions on both sides of a comparison operator, where both sides return a string result. A problem arises because a single string manipulation buffer is maintained for all string manipulation functions which allows only one function to be performed at a time. This greatly increases the speed of the operations as string transfers are minimized.

System String Functions

These functions let you use some system functions by converting the information into a character string. You may manipulate these string data as any other string.

DATE$

DATE$ returns the date in the format stipulated by the Control Panel settings.

Example:

```
Day$ = DATE$
PRINT Day$
```

Something like 25/Feb/92 will be displayed.

PREFIX$

PREFIX$ returns a string with the name of the current default prefix.

Example:

```
Volume_name$ = PREFIX$
PRINT Volume_Name$
```

TIME$

TIME$ returns the time from the AppleIIGS clock in the format set by the control panel, for example HH:MM:SS.

Example:

```
Clock$ = TIME$
PRINT Clock$
```

The time is displayed like this: 10:24:23

String Garbage Collection

Garbage is memory which was once used for a purpose, but is now unused and lost to the system.

When a string is reassigned another value, the new string must be built in another area of memory. The pointer (or address) to the old string is changed to point to the new string, and the area in memory to which the string variable originally pointed becomes lost, or garbage. Eventually, most of the string memory will become garbage and need to be reclaimed. This reclaiming is done using a process called "String Garbage Collection".

FRE (0)

FRE (for Free) forces a collection of all unused character strings and returns the number of bytes available to the system for building further character strings.

The argument may be any legal mathematical expression, but a value of zero is used by convention. The parameter has no effect on the result, but is required by the Compiler, otherwise an error will occur.

If you assign **FRE** (0) to a real variable, the entire number of bytes remaining will be returned. If you assign **FRE** (0) to an integer variable, the number of banks available to the program will be stored in Direct Page location **True_Value** (202) and the rest of the address will be returned in the assigned variable (in two's complement notation). If you are using the integer **FRE** (0), then retrieve the bank number immediately after executing **FRE** (0) as location 202 is also used for other purposes.

If **FRE** (0) returns an unacceptably small number of bytes, use the **BANK_NO** compiler option, described in Part Three, Chapter One, to allocate more memory for string storage, if possible. By default, a program is allocated one bank (64K) for string storage.

Example:

```
Free_Bytes% = FRE (0)
Free_Banks% = PEEK (202)
Total_Bytes& = Free_Bytes% + 65536 * Free_Banks%
```

or

```
Total_Bytes& = FRE (0) {Same result}
```

Micol Advanced BASIC uses an efficient, double-linked garbage collection algorithm that seldom produces, if ever, any noticable delay.

Programmers

> Toolbox Note: Do not confuse the results of **FRE** (0) with the functions of the Tool Memory Manager which returns the amount of free memory in the computer.

Chapter Five

Making Decisions

Overview

We all have to make a large number of decisions in our daily lives. The vast majority of programs also have to make decisions, and actions have to be taken based on these decisions.

We have discussed relational operations earlier in this manual. In this chapter you will learn to use these relational operations and have your programs take action based on the results of these relational operations.

Decision making is probably the most important aspect of computer programming. It is important you have a complete understanding of this topic if your programs are to function as intended.

Program Indentation

It is important that your program source code reflect the logic within your programs. The logic within your programs can best be represented by line indentation. Once a statement falls under a particular control structure, this statement should be indented one Tab. Once this control structure is resolved, the Tab should be removed. There should be one Tab for each active control structure.

If you are confused, simply look to the examples within this manual. Each example reflects the standard indentation.

Single Choice Decisions

As we have stated earlier in this manual, a relational operation yields a result of TRUE or FALSE. Based on this result, we may wish to have a certain set of actions taken. In addition, we may also wish that an alternate set of actions will be taken in the event the first set of actions is not taken. That is, we have a choice to make, one set of actions or another. It is in this circumstance that we will wish to make use of the most important statement in computer programming, the **IF** statement.

The IF Statement

Simple IF

**IF Relop THEN Statement [(: Statement)] **
[ELSE Statement [(: Statement)]]

Relop is evaluated and produces a boolean result (TRUE or FALSE). If the result is TRUE, the statement(s) following the keyword **THEN** until the end of the line or optional **ELSE** keyword are executed. If the **ELSE** statement is present and Relop is FALSE, the statements following the **ELSE** until the end of the line will be executed. In both cases, when the instructions have been executed, the flow of execution continues on the next line of instructions.

The **IF..THEN..ELSE** statement is designed to provide an **ELSE** option to the Applesoft **IF..THEN** structure. This statement works correctly when the statements to be executed after the **THEN** or the **ELSE** are on a single line of code. More than one statement may be written after the **THEN** or the **ELSE** by preceding the second and following statements by a colon (:).

Example:

```
Op$ = "-"
IF Op$ = "+" THEN Num = 2 ELSE Num = 3
```

Block IF..THEN..ELSE

IF Relop THEN BEGIN
  Statement
  [(: Statement)]
[ELSE BEGIN
  Statement
  [(: Statement)]]
ENDIF

Relop is evaluated and produces a boolean result (TRUE or FALSE). If the result is TRUE, the statements following the keywords **THEN BEGIN** until the **ELSE** (if present) or **ENDIF** are executed. If an **ELSE BEGIN** block is present and Relop is FALSE, the statements following the **ELSE BEGIN** until the **ENDIF** will be executed. In either case, when the instructions have been executed, the flow of execution continues after the **ENDIF**.

To allow more than one line of code for either the **IF** or **ELSE** statement, add the **BEGIN** keyword. The **BEGIN** keyword encloses other *Micol Advanced BASIC* statements within the **IF..THEN..ELSE..ENDIF** block structure.

ENDIF is used to close an **IF BEGIN** or **ELSE BEGIN** (if present). **ELSE** or **ELSE BEGIN** also close an **IF BEGIN**. If no **BEGIN** is present, the end of line will terminate the conditional statement. If confused, just study the examples that follow.

Example:

```
IF 1 = 2 THEN BEGIN
   PRINT "This line will never be executed"
   PRINT "Neither will this line"
ELSE BEGIN
   PRINT "This line will be executed"
```

```
    PRINT "And so will this one"
ENDIF
END
```

The **IF..THEN** also accepts a boolean variable as part of the expression.

Example:

```
Flag! = TRUE
IF Flag! THEN Num_Of_Truck% = 10
```

or

```
IF Flag! = TRUE THEN Num_Of_Truck% = 10
```

It is preferable, however, to use the first method because if the boolean has been set to an uncertain value, the expression may never evaluate to TRUE.

An **IF** block may contain one or more **IF** blocks within it. There may be as many as 20 **IF** blocks nested within another.

Example:

```
IF Outer_Flag! THEN BEGIN
   IF Middle_Flag! THEN BEGIN
     IF Inner_Flag! THEN BEGIN
          PRINT "All conditions met"
     ELSE BEGIN
          PRINT "Inner_Flag! not true"
     ENDIF
   ELSE BEGIN
     PRINT "Middle_Flag! not true"
   ENDIF
ELSE BEGIN
   PRINT "Outer_Flag! not true"
ENDIF
```

Consider using the multi-choice construct **CASE_OF** if more than two **IF..THEN..ELSE** structures are nested.

Multi-Choice Decisions

Multi-choice decisions occur whenever there are several possible actions that may be taken based on a particular situation. Suppose, for example, an office manager has to base the bonus situation of the salespeople in his office on the number of products sold by each salesperson in a month. If there are several categories of bonuses, determining the correct bonus can get very difficult using **IF** statements. One solution is a **CASE_OF** statement that functions in many ways as an **IF** statement, but allows for many possible choices.

The CASE_OF Statement

CASE_OF Aexpr
DO Label1, Label2
Statement(s)
ENDDO
[{DO Label3, Label4
Statement(s)
ENDDO }]
[ELSE_DO
Statement(s)]
ENDCASE

CASE_OF allows the user to choose one option among many without having to make use of multiple single conditional statements.

The **CASE_OF** statement evaluates Aexpr and selects one **DO..ENDDO** block from the other **DO..ENDDO** blocks using the result of the evaluation. If Aexpr yields a real result, only the whole number portion is used.

A **CASE_OF** statement must have at least one **DO..ENDDO** block of statements, and may have as many **DO..ENDDO** blocks of statements as is necessary.

The **DO..ENDDO** structure is made of a list of CASE labels followed with a block of statements to be executed on the lines of code below. When a label within a **DO..ENDDO** block matches the result of the arithmetic expression, the statement(s) in the **DO...ENDDO** block of statements will be executed.

The **DO** list may have from one to twenty labels separated by commas. The label is always an integer constant ranging from ±32767. A label may be preceded by a lesser than (<) or greater than (>) symbol to make a range of labels. No label should be repeated as only the first match is used.

If a match is <u>not</u> made and an **ELSE_DO** appears after the last **DO..ENDDO** block, the statement(s) following the **ELSE_DO** until the **ENDCASE** will be executed. The **ELSE_DO** must be the only statement on the line of code. The control of flow will continue at the line of code after the **ENDCASE**. It is always a good practice to have an **ELSE_DO** block to handle the unexpected conditions.

Example:

```
Number% = -100
REPEAT
   CASE_OF Number%
     DO 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, > 80
          PRINT Number%;" is positive"
     ENDDO
     DO -1, -2, -3, -4, -5, -6, -7, -8, -9, < - 79
          PRINT Number%;" is negative ";
```

```
            PRINT "isn't it?"
          ENDDO
        ELSE_DO
          PRINT Number%;" is not in range"
        ENDCASE
        Number = Number + 1
    UNTIL Number% > 100
```

If a match is not made and an **ELSE_DO** does not appear after the last **DO..ENDDO** block, control of flow continues at the line of code after the **ENDCASE** statement.

NOTE

> A static string may also be used as a label within a **DO** line. Only the first character of the string will be used, and is the same as if the label had been entered as the ASCII value of the first character instead.

Example:

```
String$ = "Aardvark"
Ascii% = ASC (String$)
CASE_OF Ascii%
    DO "A","a"
      PRINT "Letter was upper or lower case A"
    ENDDO
ENDCASE
```

CASE_OF statements may be nested within other **CASE_OF** statements. The maximum level of nesting allowed is 8 levels deep. The nested CASE statement is placed in a **DO..ENDDO** structure.

Chapter Six

Basic Input/Output of Information

Overview

Virtually all programs accept information from some source, process this information, and send this processed information to a storage or display device.

Principal sources for input are through the computer keyboard and a storage device such as a disk drive. Less often, the input of information is from the program itself. The output from the program is usually sent to a display device such as a monitor or the printer or to a long term storage device such as a disk drive.

Data Input

Input is anything that can be entered into the computer using an input device, usually the keyboard, or read from a storage device such as a disk drive.

Internal Data Entry

DATA Var [{,Var}]

DATA statements are used to place specific values into memory that may later be retrieved during execution of the program.

DATA statements are placed at the beginning of the program after the optional compiler directives. The **DATA** statement must be placed in the correct position in the program in order to be compiled. Please see the Program Order section in Chapter One of Part Three.

Only integer, real and string literals are accepted as datum for a **DATA** statement. Each datum is separated from the next by a comma. The length of a **DATA** statement is limited only by the length of the program line. The number of **DATA** statements is limited only by the memory available.

Real literals must be distinguished from integer literals by having the terminating fraction written in decimal form (i.e 13.0). Integer literals greater than 65535 will be considered real. String literals must be enclosed between double quotes. Booleans may not be used in a **DATA** statement.

Example:

```
PROGRAM Data_Example
DATA 1, 1.0, 1.0E25, "One"
```

DATA statements may not be empty (have a non-definite value) as in an Applesoft BASIC program or be followed by any other statements on the same line. A **DATA** line

must have a literal between each comma otherwise the Compiler will signal an error.

Example:

```
{Missing values are illegal and will
cause errors during compilation}
DATA "TEXT",,"MORE TEXT",,0,0,,0
```

READ Var [{,Var}]

The function of the **DATA** statement is to give a method to store constant information that may be used each time the program is executed. These data are accessed within a program by means of a **READ** statement. A **DATA** statement only has meaning when used in conjunction with a **READ** statement.

To **READ** data, a loop of some kind is usually used. The **DATA** values are read one by one, starting from the first line of **DATA**. The **DATA** pointer cannot turn back or skip any values, but may be moved back to the beginning using the **RESTORE** command.

If the program tries to read more values than are available, an error will occur. Leaving values unread does not produce an error.

If the data types in the **DATA** and **READ** statements do not match, an error will occur when the program tries to read in the datum.

Example 1:

```
PROGRAM Read_Data
DATA 1, 1.0, "One"
{Main Program}
READ Integer% {Read integer datum}
READ Real& {Read real datum}
READ String$ {Read string datum}
END
```

Example 2:

```
PROGRAM Read_Numbers
DATA 1, 2, 3, 4
DATA 5.0, 6.0, 7.0, 8.0
DIM Number% (3), Number (3)
{Main Program}
FOR Counter% = 0 TO 3 {Read first DATA line}
   READ Number% (Counter%)
   PRINT Number% (Counter%)
NEXT Counter%
FOR Counter = 0 TO 3 {Read second DATA line }
   READ Number (Counter)
```

```
    PRINT Number (Counter)
NEXT Counter
END
```

RESTORE

RESTORE places the **DATA** pointer back to its starting position. This means the values in the **DATA** statements may be reread.

Example:

```
PROGRAM Read_Numbers
DATA 1, 2, 3, 4
DATA 5, 6, 7, 8
DIM Number% (7)
{Main Program}
HOME
{Read values in DATA statements}
FOR Counter% = 0 TO 7
    READ Number% (Counter%)
    PRINT Number% (Counter%)
NEXT Counter%
RESTORE {Bring DATA pointer to position one}
{Reread values in DATA statements}
FOR Counter% = 0 TO 7
    READ Number% (Counter%)
    PRINT Number% (Counter%)
NEXT Counter%
END
```

Keyboard Entry

GET Svar

GET is used to read one character from the keyboard and place it into a string variable. The character entered is not echoed on the screen.

The program continues execution with the next statement without waiting for a press of the Return key. The cursor is displayed until a character is entered.

GET accepts only a string variable as its argument. The Compiler will issue an error if a numeric variable is used. Use the **VAL** function to convert the digit if required.

NOTE

> <Control>C will not interrupt the execution of **GET**. All Control characters may be read from the keyboard with **GET**.

See also the next chapter for another use of **GET**.

Example:

```
REPEAT
   GET Vowel$
   IF INDEX (Vowel$, "AEIOUaeiou") > 0 THEN PRINT Vowel$
UNTIL INDEX (Vowel$, "AEIOUaeiou") > 0
```

INKEY Svar

INKEY scans the keyboard to determine if a key has been pressed. **INKEY** is similar to **GET** except **INKEY** does not wait for a key press and does not display a cursor.

If no key has been pressed, an empty string is returned in Svar. If a key has been pressed, a one byte string representing the key pressed is created in Svar.

NOTE

> To be effective, **INKEY** must be used within a loop.

Example:

```
REPEAT
   INKEY Character$
   IF Character$ <> "" THEN PRINT Character$
UNTIL Character$ <> ""
```

INPUT ["Prompt string";] Var [{, Var }]

INPUT accepts data from the current input device (usually the keyboard). An optional message, enclosed in quotation marks, may be displayed prompting the user for the necessary input.

The prompt must appear after the keyword **INPUT**, and be followed by a semi-colon (;), and the list of variables. If no prompt is specified, **INPUT** automatically displays a question mark (?) as the prompt.

NOTE

> No question mark is displayed when the prompt string is present but empty; use this to hinder any prompt.

INPUT may have any number of variables, each separated by a comma.

INPUT accepts simple variables and arrays of type integer, real and string. Boolean variables are not accepted.

The **INPUT** statement will ask for the second, and any subsequent input on a separate line by displaying a question mark (?) for each missing input.

WARNING

> Pressing the Return key for each piece of information is the only way to accept data from an **INPUT** with multiple variables. The comma (,) and semi-colon (;) are not accepted as delimiters as under Applesoft BASIC.

In order to make programs easier to understand, use one **INPUT** statement for each piece of information.

INPUT accepts <Control>S to insert a space. An input may be terminated by pressing <Control>C only if the **NOT_C** compiler option is not used. The Delete key erases a character during response to an input (the delete mode may be altered during execution, see Appendix A).

NOTE

> Memory locations 4 and 5 in the Library's Direct Page control the maximum number of characters that may be entered using **INPUT**; 255 is the default. This value is stored as a hexadecimal number in least significant byte, most significant byte order. Do not **POKE** a value greater than a 3 into location 5 or an error will occur when the next **INPUT** is encountered.

The bell will ring if the maximum number of characters allowed in an **INPUT** line has almost been reached.

String Input Rules

Characters with ASCII codes from 32 to 127 may be entered from the keyboard. Control characters will be ignored.

Numeric Input Rules

If, during a numeric input, the user enters something other than a numeric value, the message "?Reenter" will be displayed. A question mark prompt will appear on the next line and the computer will wait for the appropriate input. For a real input, all non-numeric characters except a capital "E", a period (.), a comma (,), a plus sign (+), and a minus sign (-) will be rejected. For integer input, only digits, a comma (,) and the plus and minus signs are allowed input. The commas are for user convenience and are ignored.

A numeric expression, such as "3 * 4 / 6", is not accepted as numeric input.

Examples:

```
INPUT "Enter name: "; Name$
INPUT "Enter age: "; Age%
INPUT "Enter any real value: "; Number&
```

See also the next chapter for other uses of **INPUT**.

Entry from Other Devices

INSLOT (Slot_Number)

INSLOT is used to get characters from the device connected to the slot or port number specified. The argument may be any integer literal between 0 and 7; a 0 is used to return input to the keyboard. Any negative value or a value greater than 7 will return an error.

IMPORTANT

> **INSLOT** is best used in conjunction with a **GET**. **INPUT** may be used after an **INSLOT**, but because **INPUT** expects a carriage return to terminate an entry, **INPUT** is only suitable in limited situations.

Example:

```
INSLOT (2) {Input from slot 2}
GET Char$ {Reads character from port 2}
INSLOT (0) {New input from keyboard}
```

Data Output

Output is information that can be sent from the computer, usually to a screen display or printer, or to a disk device for long term storage.

Screen Display Control

The following commands control the manner in which text is output to the screen.

DELAY = Aexpr

DELAY pauses the program the stipulated time. One increment equals about 0.01 seconds for a normal Apple IIGS. If you have an accelerator card installed, the delay will be that much quicker.

Example:

```
DELAY = 100 {Pause about one second}
```

HOME

HOME erases the contents of the text window and places the cursor at the top left corner of the screen.

Example:

```
FOR Line% = 1 TO 23
   PRINT "This fills part of the screen"
NEXT Line%
HOME
PRINT "Now the screen is almost clear"
```

NOTE

> To move the cursor to the top left corner of the screen without erasing the screen, use **VTAB** (1): **HTAB** (1).

INVERSE

INVERSE causes the subequent character(s) sent to the screen to be displayed in inverse video (reversing the black and white of a character block).

INVERSE will stay in effect until a **NORMAL** command is encountered.

Example:

```
INVERSE
PRINT "This is an inverse display"
NORMAL
PRINT "This is a normal display"
```

MS_TEXT

MS_TEXT (for MouseText) allows the ability to send MouseText characters to the screen.

MouseText characters are a set of graphical characters designed specifically for the Apple II computer. This character set has the ASCII range 64 ($40) through 95 ($5F).

Example:

```
{Display keycap symbols}
MS_TEXT {Turn on MouseText}
PRINT "@ H U J K M"
MS_TEXT {Turn off MouseText}
```

IMPORTANT

> A second **MS_TEXT** turns off the effect of the previous **MS_TEXT**.

NORMAL

NORMAL restores the display to the standard text characters. **NORMAL** turns off the previous **INVERSE**. **NORMAL** character display is the default mode.

See the example for **INVERSE**.

SPEED = Aexpr

SPEED controls the rate at which the characters appear on the screen. Aexpr must be between 1 and 255; the minimum speed being 1 and the maximum speed being 255. The default display rate is set to 255, the maximum speed. A speed of zero is equal to a speed of 255.

Example:

```
SPEED = 100
PRINT "This line will print slowly"
SPEED = 255
PRINT "Now printing at normal speed"
```

Unformatted Text Output

PRINT [Expr] [;] [,] [Expr]

PRINT is used to display all data types including boolean.

Any legal math or string expression, literal or variable may appear inside a **PRINT** statement. Each expression will be evaluated when it is executed. If a logical expression

is in a **PRINT** statement, the result of the comparison (**TRUE** or **FALSE**) is printed.

When a semi-colon (;) is placed at the end of a statement, the semi-colon prevents a Carriage Return (ASCII #13), needed to move the cursor to the next line. Any subsequent output following the semi-colon is printed on the same line. The cursor remains to the right of the last character printed. The next item to be printed will appear at the current cursor position.

A comma (,) at the end of a statement places the cursor at the next tab column (1, 16, 32, 40, 48, 56, 64, 72 or 80). The contents of the next **PRINT** is displayed starting at that position.

Anything other than a semi-colon (;) and a comma (,) as the last character in a **PRINT** statement will generate a carriage return (ASCII #13) as the last character output and place the cursor at column 1 of the next line. If the cursor is already on a new line, an empty blank line will be displayed or printed. The screen will scroll if necessary.

TAB and **SPC** may also be used within a **PRINT** to format the display.

NOTE

> A question mark (?) may not be used as a shorthand notation for **PRINT** as under Applesoft BASIC.

Examples:

```
PRINT "Your name is "; Name$;" your age is "; Age%
PRINT {Only sends a <CR>}
PRINT "1 + 2 + 3 = "; 1 + 2 + 3
PRINT 1, 2, 3, 4, 5
PRINT 1.5 > 9.3 {Will print FALSE}
```

See also the next chapter for other uses of **PRINT**.

See also Part Six, Chapter One for debugging uses of **PRINT**.

Formatted Text Output

PRINT USING Mask$; [Expr] [;] [,] [Expr]

PRINT USING is used to display real values to the current output device using a particular format. Formatting is made to both sides of the period of the real value.

Except for the real value formatting ability, **PRINT USING** functions just like **PRINT**. **TAB** or **SPC** statements may be used within **PRINT USING** if needed.

A mask is used to define the format of the output. The mask may be a string literal or string variable. Rules for the mask are as follows:

1. Only dollar signs ($), number signs (#), commas (,) and a single period (.) are allowed within a mask.

2. Commas may appear only to the left of the period. If digits are to be output, commas will appear in the printed output in the same position they appear in the mask.
3. Number signs may appear on either side of the period. Every occurrence of the number sign will be replaced with digits or padded with spaces on the left of the period and by digits or padded with zeros (0) on the right of the period.
4. Dollar signs are allowed only on the left side of the period. Each occurrence of a dollar sign will be replaced with a space until just before a digit would appear, then a single dollar sign will be printed. Additional dollar signs will be replaced by the appropriate digits.
5. A fraction will be truncated, not rounded.
6. If the number should require more places on either side of the period than are specified in the mask, the digits will not be displayed. Make sure to allow enough room in the mask for all possible values.

NOTE

> The character value of the comma and period may be changed to conform to the non-English speaking world. The comma and period may be changed to other characters by modifying the appropriate memory locations listed in Appendix A.

To print monetary values, use a mask similar to this: Mask$ = "$,$$$,$$$.##".

To print numeric values, use a mask similar to this: Mask$ = "#,###,###.##"

Example:

```
Number& = 1234.567
PRINT USING "$$$,$$$,$$$.##"; "The value is"; Number&
```

The line above will print: The value is $1,234.56 (with five leading spaces).

Example:

```
Mask$ = "###,###,###.#"
Number& = 123456.78
PRINT USING Mask$;"The value is "; Number&
```

The line will indicate The value is 123,456.7 (with four leading spaces).

NOTE

> To format the output of an integer value, then simply assign this integer value to a dummy real variable, and use the dummy real variable in the **PRINT USING** statement.

Cursor Positioning

The following commands affect the movement of the screen cursor, and sometimes the printer head. Cursor positioning is affected by the borders of the screen which may be altered during execution of the program making it possible to create text windows. Please see Appendix B for specific information.

POS (Aexpr)

POS (for Position) returns the current horizontal position of the cursor at the moment **POS** is executed. The value returned is from one to 80. One is the left-most side and 80 is the right-most side of the screen.

The argument is ignored, and has no effect on the result of the evaluation of **POS**, but must be present, otherwise an error will occur during compilation.

Example:

```
HOME
PRINT "Position: ";POS (0)
```

This statement returns the number 11 for the position of the cursor.

SPC (Aexpr)

SPC (for space) prints the specified number of spaces to the current output device and may only be used inside a **PRINT** statement.

Aexpr may be any valid arithmetic expression. **SPC** must be in the range one to 255 otherwise an error occurs at run time. If Aexpr is real, its value will be truncated.

SPC moves the cursor or print head the number of spaces specified starting from the current cursor position. If the cursor is moved past the right margin, it continues spacing on the line below.

IMPORTANT

> Semi-colons must be used after each **SPC**, otherwise a carriage return will be generated destroying the effect of **SPC**.

Example 1:

```
PRINT SPC(15);"The total is:";Total$
```

TAB (Aexpr)

TAB (for Tabulation) is used to position the cursor to the specified position on either the screen or printer and may only be used inside a **PRINT** statement. The position values range from 1 to 80. The first horizontal position (1) being on the left margin and

the last one (80) on the right margin.

Aexpr may range from one to 255. Values from 81 to 255 will tab on lower lines of the screen.

If Aexpr is real, only the whole number portion will be used.

If a **PRTON** statement is in effect, **TAB** will move the print head at the position specified, in a forward direction only.

IMPORTANT

> Semi-colons must be used after each **TAB** statement, otherwise a carriage return will be generated, destroying the effect of the **TAB**.

Example 1:

```
PRINT TAB (15);Total$
```

HTAB (Aexpr)

HTAB (for horizontal tab) moves the cursor to the horizontal position specified by Aexpr. The cursor may be moved from left to right or right to left.

Aexpr may range from one to 80. Any values outside this range will result in a run time error. If Aexpr is real, only the whole number portion will be used.

Example:

```
PROGRAM Demo_HTAB
HOME
HTAB (36)
PRINT "is the";
DELAY = 50
HTAB (31)
P INT "This";
DELAY = 50
HTAB (43)
PRINT "proper order."
END
```

VTAB (Aexpr)

VTAB (for Vertical tab) moves the cursor vertically to a specific line on the screen.

The argument may be any valid arithmetic expression with a result ranging from one to 24. Any values outside this range will result in an error at run time. If Aexpr is real, only the whole number portion will be used.

The cursor may move in either vertical direction.

Example:

```
PROGRAM Demo_VTAB
HOME
VTAB (4)
PRINT "On line four"
END
```

Output to Other Devices

OUTSLOT (Slot_Number)

OUTSLOT is used to send subsequent output through a device connected to the specified slot number. The argument must be a digit between 0 and 7; any negative value or value greater than seven will cause an error.

IMPORTANT

A 3 is used to return output to the screen.

NOTE

None of the screen formation statements such as **TAB** will work when used in conjunction with **OUTSLOT**.

Example:

```
OUTSLOT (2) {Output through slot 2}
PRINT String$ {Sends character(s) to port 2}
OUTSLOT(3) {Sends output to the screen}
```

PRTON

PRTON (Printer On) turns on the communication link to the printer and redirects all output to it. **PRTON** assumes the printer is connected to slot one (printer port) of the computer. If this is not the case, use **OUTSLOT**.

PRTON does not interrupt the execution of the program if the computer is connected to a serial printer even if the printer is turned off. However, the program may hang if a parallel printer is turned off.

Example:

```
PRTON
PRINT "This line is written on the printer"
```

```
TEXT
PRINT "This line is written on the screen"
```

TEXT

TEXT turns off the communication link to the printer and restores the screen as the current output device.

Example:

```
PRTON
PRINT "This line is sent to the printer."
TEXT
PRINT "This line is sent to the screen."
```

NOTE

> **TEXT** may only be used to turn the printer off and the screen display back on if the printer was originally turned on with a **PRTON**.

Chapter Seven

Disk Filing

Overview

It is often the case that data generated by a program must be stored in some long term device for later usage. Also, data stored from some outside source often must be read in from a long term storage device for immediate usage. Such data are usually stored as disk files.

A typical example of such file usage is in a word processor. Once the text is generated within the word processor, it must be saved, or all the work would be wasted once the computer is turned off. Conversely, this text may have to be read back into the word processor at a later time for further modifications.

Disk filing commands are necessary to maintain and access these files. Access and maintenance of disk files is the topic of this chapter.

File Management

These commands allow you to manage the disk files on your system.

CAT$

CAT$** is designed to get file information from a directory. Each use of **CAT$ returns a string containing a file directory entry from the default directory, just as it is displayed using the **CATALOG** command under the Shell (minus the heading).

The volume information is returned on the last line, concatenated with the last file name and information, separated by a carriage return (ASCII 13).

CAT$ must be contained in a loop. If more directory information can be read, **True_Value** (memory location 202) will contain a zero. If the last line has been read, **True_Value** will be non-zero. Remember that **True_Value** is used for other purposes and should be tested immediately after each use of **CAT$**.

Example:

```
PROGRAM Show_Directory
{Display directory header}
HOME
PRINT "Filename"; TAB(21); "Type"; TAB(27); \
      "Blocks"; TAB(36); "Created"; TAB(43); \
      "Time"; TAB(55); "Modified"; TAB(64);"Time"; \
      TAB(74);"EOF"
```

```
PRINT
{Get directory listing}
REPEAT
   String$ = CAT$
   IF PEEK(202) <> 0 THEN BEGIN
     PRINT String$
   ENDIF
UNTIL PEEK(202) <> 0
END
```

IMPORTANT

> The entire directory file must be read at one time, otherwise the directory file will remain open unnecessarily, which will probably cause problems at a later time. You may have to read the directory entries into a string array.

NOTE

> If you wish the contents of a directory other than the default directory, you will have to change the default prefix with the **PREFIX** command. You may first have to save the current directory with use of the **PREFIX$** command, then reinstate the original directory after the directory has been read.

COPY Svar1 TO Svar2

COPY duplicates the file defined in Svar1 into a file with name Svar2. Svar is the Pathname of the files and may be either a string variable or a string literal.

If Svar2 is assigned an empty string, the file specified in Svar1 will only be read. If an error occurs during the read, **True_Value** (location 202) will contain a non-zero value. This allows you to verify a file without generating an error.

Example:

```
File1$ = "/RAM5/File"
File2$ = "/RAM6/New.File"
COPY File1$, "" {First Verify File1$}
IF PEEK (202) = 0 THEN COPY File1$ TO File2$
```

CREATE Svar

CREATE will generate a directory file (type DIR). Svar is the Pathname of the new directory and may be either a string variable or a string literal. Svar must not already exist or an error will be generated.

CREATE locks the newly created directory.

Example:

```
CREATE "/Micol.Adv.BASIC/New.Dir"
```

DELETE Svar

DELETE will erase the file specified from the appropriate directory. Svar is the Pathname of the file to be deleted and may be either a string variable or a string literal.

A file may not be deleted if it is open or locked. A directory file may only be deleted if it is empty. Use this command when a specific file is no longer needed.

Example:

```
DELETE "/RAM6/FILE"
```

FLUSH

FLUSH will empty all open file buffers to their respective files.

The main function of **FLUSH** is in file security. If any program runs a significant time with open files and the program malfunctions, without periodic use of **FLUSH**, the information in the buffer(s) may be lost. This command ensures that all data inside the file buffer(s) will be transferred to their respective disk files.

A program using the command **FLUSH** will be slightly slower because of the time needed to copy the information to disk, but you will be certain to have all the information saved should a power surge or interruption occur.

Example:

```
FLUSH
```

FORMAT Svar

FORMAT is designed to initialize a disk device such as a floppy or a RAM disk. Svar is the volume name the disk will have once formatted and may be either a string variable or a string literal.

The **FORMAT** command displays the location and the names of all devices connected to the computer. The user will select the appropriate device with the Up and Down Arrow keys and press Return to display the GS/OS Formatting Dialog Box.

The user should set the controls of the Dialog Box to ProDOS for the operating system and 800K 2:1 for the interleave, if necessary then presses Return to start

formatting.

Example:

```
FORMAT "/Work.Disk" {The disk will be named Work.Disk}
```

WARNING

> The user must be certain he/she wishes to format the specified device as once the final Return is pressed, all information on the device will be erased. It is recommended a warning message be displayed, and possible exit allowed, before **FORMAT** is executed.

LOCK Svar

LOCK is used to protect a file from being deleted or modified. Svar is the Pathname of the file to be locked and may be either a string variable or a string literal.

When a file is locked, an asterisk (*) precedes the filename when a directory is displayed to show that the file is protected.

Example:

```
LOCK "/RAM6/FILE"
```

ONLINE$

ONLINE$ returns a string which contains all the current online volume names.

Each volume name is separated by a Return character (ASCII 13). This Return character may be used to isolate each online volume name within your program.

Example:

```
OnLine_Name$ = ONLINE$
PRINT OnLine_Name$
```

PREFIX Svar

PREFIX uses Svar to set the default prefix. Svar is the Pathname to a directory and may be either a string variable or a string literal.

If Svar contains an empty string (""), the system will only display the default prefix to the screen. If Svar is not empty, the default prefix will be set to Svar. The volume must be online when this command is executed; otherwise, an error will occur.

Example:

```
PREFIX "/RAM6/Directory"
```

RENAME Svar1 TO Svar2

RENAME will change the name of a file, directory or volume. Svar1 and Svar2 may be either string variables or string literals.

Svar1 is the Pathname to the original file and Svar2 is the Pathname the file will have. If Svar1 and Svar2 are on the same volume, but in different directories, Svar1 will be moved to the directory stipulated in Svar2.

Svar1 must be unlocked, and Svar2 must not already exist.

Example:

```
RENAME "/RAM6/File" TO "/RAM6/Newfile"
```

UNLOCK Svar

UNLOCK removes the protection on a file so that it may be erased, modified or renamed. Svar is the Pathname of the file and may be either a string variable or a string literal.

A space rather than an asterisk indicating that the file is unprotected will precede the filename when the appropriate directory is displayed.

Example:

```
UNLOCK /RAM6/FILE
```

Direct Access to the Operating System

GS_OS (Operation_Code, PathName$, Integer_Array% ( )

The **GS_OS** command makes it possible to communicate directly with the operating system of the Apple IIGS, GS/OS.

GS_OS is designed to call individual operations within the operating system. These calls can perform a whole assortment of things such as getting volume information, or erasing a volume directory, etc; whatever GS/OS is capable of. All of the disk access commands executed by *Micol Advanced BASIC* are done by such calls to GS/OS.

To make use of this command, you will need a GS/OS reference manual. The one by Gary Little mentioned in Suggested Manuals in Part One is quiet suitable.

GS_OS requires three parameters: a GS/OS call number, a string variable whose contents may or may not be required, and an integer array which will contain the parameter list required by the GS/OS call. The three parameters are:

1. The call number is the value required by GS/OS to determine which operating system command is needed. This value is an integer literal (either decimal or hexadecimal) and must be a value greater than $2000 hexadecimal as the **GS_OS** command only supports GS/OS class one calls.
2. A Pathname is not required by all GS/OS calls, but PathName$ must appear in the **GS_OS** command. If an Integer_Array% element contains a negative one (-1),

the string contained within PathName$ will be used for this call. PathName$ may be any legal *Micol Advanced BASIC* string which is also a legal GS/OS Pathname, according to the call.

3. The list of parameters required by the call is provided to GS/OS using Integer_Array% starting with element zero. One integer array element is equal to one word. The size of the integer array must be at least as large as the maximum number of words sent or returned by the call and must be so dimensioned. The left parenthesis is required in the syntax of this command.
4. If an error occurs as a result of the call, the GS/OS error value will be returned in location 202 (**True _Value**). A zero indicates that the call was made correctly. Any other value signals that an error occurred (or that the call was made improperly). Please see Appendix D for GS/OS error codes.

IMPORTANT

> When long integers are used, the most significant bytes of each array element are ignored, and the array is compressed into short integers before the contents of the array are passed to GS/OS. The array is decompressed after the results are returned.

Example:

```
PROGRAM OS_Example
@ LIST
INT (A - Z)
DIM Array (40)   {Large enough for any purpose}
Array (0) = 12   {12 class one parameters}
Array (1) = -1   {Pathname in element 1}
PathName$ = "/RAM5/File"
{Make the call to GS/OS}
{$2006 GetFileInfo}
GS_OS ($2006, PathName$, Array ( )
IF PEEK (202) = 0 THEN BEGIN {No error}
   FOR Ctr = 1 TO 20 {Display the result returned}
     PRINT Array (Ctr)
   NEXT Ctr
ELSE BEGIN
   PRINT "GS/OS error" ;PEEK (202)
ENDIF
END
```

NOTE

> The **GS_OS** command can easily handle GS/OS calls with one string. If the GS/OS call you wish to make requires two or more strings, you will have to create the GS/OS class one strings yourself in some buffer area, and pass these addresses within the **GS_OS** call. But don't worry, very few calls require more than one string.

General File Access

File Access Number

The commands within this section require a File Access Number. This is simply a digit (no variables allowed), from one to eight, that you give the file when it is opened. This value, rather then the Pathname, is used to access the file for further operations.

APPEND (File Access Number)

APPEND moves the file pointer to the end of the open file. Any future reads or writes to the file will be from this position. The File Access Number must be the same one that was used under the **OPEN**, **ROPEN** or **WOPEN** command.

Example:

```
ROPEN (1) "File" {Open an existing file}
APPEND (1) {Write after end of file"
PRINT (1) "After old end of file"
CLOSE(1)
```

CLOSE (File Access Number)

CLOSE will close the file specified by the File Access Number. The File Access Number must be the same one that was used when the file was opened with an **OPEN**, **ROPEN** or **WOPEN** command.

All files must be closed after having been used. The closure of the files ensures that all data have been transferred from memory buffers to their disk files. An **END** or **STOP** will also close all files currently opened.

Example:

```
WOPEN (1) "FILE"
CLOSE (1)
```

FILE (Svar)

FILE verifies that a file with the corresponding Pathname exists. Svar is the Pathname of the file, and may be either a string variable or a string literal.

FILE is a boolean function which returns TRUE if the file exists or FALSE if there is no such file. The **FILE** state may also be assigned to a boolean variable: Flag! = FILE (File$).

Example:

```
IF FILE ("/RAM6/HELLO") THEN BEGIN
   ROPEN (1) "/RAM6/HELLO"
ELSE BEGIN
   WOPEN (1) "/RAM6/HELLO"
ENDIF
```

The type of file may be determined by **PEEK**ing into memory location **True_Value** (202) right after using the **FILE** command. This value is a number representing the file type.

In addition, if **FILE** is TRUE, the file size, in blocks of 512 bytes, will be returned in locations 204 and 205 in LSB, MSB order; in location 212 and location 213 is stored the Auxiliary file type.

Example:

```
File_Exists! = FILE (InputFile$)
IF File_Exists! THEN BEGIN
   FileType% = PEEK (202)
   IF FileType% = 4 THEN BEGIN
     PRINT "The file "; InputFile$; " is of type TXT"
   ELSE BEGIN
     IF FileType% = 176 THEN BEGIN
         PRINT "The file " ; InputFile$; " is of type SRC"
     ENDIF
   ENDIF
ELSE BEGIN
   PRINT InputFile$;" does not exist"
ENDIF
```

GET (File Access Number) Svar

GET will read characters, one at a time, from disk and place the character into Svar. The File Access Number must be the same one that was used when the file was opened.

If the end-of-file marker is encountered during a **GET**, the variable waiting for a

value will be undetermined, whereas the end-of-file flag will be set to TRUE.

Example:

```
IF FILE ("File") THEN BEGIN
   ROPEN (1) "File"
   REPEAT
     GET (1) Char$
     IF NOT EOF (1) THEN PRINT Char$;
   UNTIL EOF (1)
   CLOSE(1)
ENDIF
```

INPUT (File Access Number) Var [{,Var}]

INPUT functions like the keyboard based **INPUT** statement except it accepts data coming from a file instead of the keyboard.

The File Access Number must be the same number that was used when the file was opened. Var may be any simple or array variable type except boolean.

As with the keyboard **INPUT** command, the data read from the device must correspond to the type required by the variable in the variable list.

WARNING

> **INPUT** is only suitable for reading text files. Note that the only delimiter for a string input is the carriage return (ASCII 13). Commas (,) and semicolons (;) are regarded as data for this purpose. If more than 1023 characters are read before a carriage return is encountered, an error will be generated.

Example:

```
IF FILE ("/RAM6/File") THEN BEGIN
   ROPEN (1) "/RAM6/File"
   REPEAT {Read from disk}
     INPUT (1) String$
     INPUT (1) Real
     INPUT (1) Integer%
     PRINT String$, Real, Integer%
   UNTIL EOF (1)
   CLOSE (1)
ENDIF
```

OPEN (File Access Number) Svar

OPEN establishes a link between the file specified in Svar and future commands directed at the file. Svar may be either a string variable or a string literal.

OPEN will check for the existence of the file stipulated in Svar. If the file exists, it will simply open the file (perform an **ROPEN**). If the file doesn't exist, **OPEN** will create a new file with the stipulated name, then open it (perform a **WOPEN**). In both cases, the file pointer will be pointing to the beginning of the file.

Example:

```
OPEN (1) "/RAM6/FILE"
PRINT (1) "String"
CLOSE (1)
```

PRINT (File Access Number) [USING Mask$;] Var[{,Var}]

PRINT and **PRINT USING** function exactly like their screen-based counterparts except they send their data to the disk instead of the screen or printer.

The File Access Number must be the same number that was used when the file was opened. Var may be an integer, real or string variable or array.

NOTE

> **TAB**s will not produce spaces in a text file.

WARNING

> If the data created with a **PRINT** are to be read by an **INPUT** statement, then be certain not to suppress the carriage return by using a comma(,) or a semi-colon(;) after each variable list. It is best to have one variable per **PRINT** statement.

Example:

```
WOPEN (1) "FILE"
PRINT (1) "Output to file"
FOR Loop_Ctr% = 1 TO 10
   PRINT (1) Loop_Ctr%
NEXT Loop_Ctr%
CLOSE(1)
```

The end-of-file marker is pushed forward as each variable's contents are written to disk.

ROPEN (File Access Number) Svar

The **ROPEN** command will open an already existing file and will position the file pointer to the beginning of the file. The File Access Number used with the **ROPEN** command must be used with all the commands referencing the file being accessed later.

Svar is the Pathname of the file and may be either a string variable or a string literal. The Pathname of the file being read must exist on the disk being accessed. Any attempt to **ROPEN** a non-existent file will cause a run time error.

ROPEN establishes a relationship between the File Access Number and the Pathname. Without this relationship established, the system cannot know which File Access Number belongs to which file.

IMPORTANT

> File Access Number 8 will provide much faster access to sequential files than File Access Numbers 1 thorough 7. However, because File Access Number 8 maximizes file access by reading several file blocks into internal memory from which the file information is then accessed, it is unsuitable for random access files.

Example: (See **GET**)

WOPEN (File Access Number) Svar

WOPEN will erase any existing file with the same Pathname stipulated by Svar, and create an empty file with the specified Pathname. If the file already exists, and that file is locked, an error will be generated. Svar may be either a string variable or a string literal.

The File Access Number used with the **WOPEN** command must be used with all the commands referencing the file being accessed.

WOPEN establishes a relationship between the File Access Number and the Pathname. Without this relationship established, the system cannot know which File Access Number belongs to which file.

Example: (See **PRINT**)

Sequential File Access

EOF (File Access Number)

EOF is used to detect the end-of-file marker when a sequential file is being read. The File Access Number must be a digit between 1 and 8 and must be the same value used when the file was opened.

EOF is a boolean function and may be assigned to a boolean variable as: Flag! = **EOF** (1). This boolean variable may then be tested like any boolean variable.

If the end-of-file is encountered while reading a variable's value, the value of the variable is undetermined, but the **EOF** flag will be set to TRUE.

If you try to test the end-of-file on a file which has not been opened, you will receive a run time error.

Example:

```
ROPEN(8) "/RAM6/FILE" {Get fast access with 8}
REPEAT
   INPUT (8) String$
   IF NOT EOF (8) THEN PRINT String$
UNTIL EOF(8)
CLOSE(8)
```

Random Access Files

SEEK (File Access Number) Record Number, Record Size

SEEK is used to move the file pointer within a random access file. **SEEK** will move the end-of-file marker if the position is past the current end-of-file. You may then read or write to this file location as you require.

The **SEEK** command must be used before any read or write operation to a random access file, otherwise the next read or write operation will be done right after the previous read or write. Be certain not to leave out this command if a random access file is used.

You must decide what record size you wish; the record may be any size. Once the record size is specified, any record may be accessed within the file; even sub-records within the file may be accessed by specifying the correct record size.

To access a specific field within a certain record, you may skip the previous fields using dummy **INPUT**s. To do so, each field must end with a c riage return. If the Return characters at the end of each field have been suppressed, then the **INPUT** statement(s) will not be able to read the data since **INPUT** expects the Return character as the end-of-field delimiter.

NOTE

> The use of a File Access Number 8, reserved for use with sequential file access, will result in an error during compilation.

When calculating the record size, remember that the Return character also requires one byte.

NOTE

> **SEEK** may function the same way as the **POSITION** statement of the Applesoft BASIC interpreter by specifying a record size of 1.

```
PROGRAM Random_Access
HOME
WOPEN (1) "/Volume2/File"
INPUT "Enter record size" ;Size
REPEAT
   INPUT "Enter record number" ; Record
   INPUT "Enter any number" ; Number
   SEEK (1) Record, Size
   PRINT (1) Number
UNTIL Number = 100
CLOSE(1)
HOME
ROPEN (1) "/Volume2/File"
PRINT "The values entered were:"
REPEAT
   INPUT "Enter record number ";Record
   SEEK (1) Record, Size
   INPUT (1) Number
   PRINT Number
UNTIL Number = 100
CLOSE(1)
END
```

NOTE

> From the programmer's standpoint, the only difference between a sequential file and a random-access file is the use of the **SEEK** command.

Chapter Eight

Control of Flow

Overview

Unless special action is taken, each program statement will execute after the preceding statement has finished execution. Very few programs would have any real worth if this linear program flow could not be altered.

It is the purpose of this chapter to discuss the methods available under *Micol Advanced BASIC* to direct program flow in an appropriate manner. In this regard, *Micol Advanced BASIC* is one of the most powerful languages for any computer. Use these commands wisely and your programs will be something to be proud of.

Program Termination

The termination statements are designed to end the execution of a program; control passes out of the program.

External Flow

RUN Pathname

To execute another *Micol Advanced BASIC* program or a GS/OS application, use the **RUN** command. Pathname must be the Pathname, including the ".LNK" extension, if any, of the program. Pathname may be a string literal or string variable. The file must be online at execution time or an error will be issued.

Examples:

```
RUN "MAB.LNK"
Path_Name$ = "FINDER"
RUN Path_Name$
```

Flow Interruption

END

END terminates the program's execution, and returns control to the Command Shell (if the program was entered from the programming environment).

END may be placed anywhere in a program. **END** closes all open files, frees all memory, and sets the screen to text mode.

NOTE

> Although the Compiler automatically generates an **END** at the end of the program code, it is recommended to conclude all programs with **END** for documentation purposes.

IMPORTANT

> In any program that was launched with the Finder, **END** returns control to the Finder. If the program was started as a TurnKey system, a System Death will be performed.

Example:

```
PROGRAM EXAMPLE
PRINT "This is a sample program"
END
```

STOP

STOP is identical to **END** except it prints the line number where the program halted. **STOP**'s primary function is in debugging.

Example:

```
PROGRAM Example
PRINT "This is a simple program"
STOP
```

BYE

BYE terminates the execution of the program and returns control to the program launcher (even if the program was started from the Command Shell). **BYE** performs what is called a ProDOS QUIT.

NOTE

> When a TurnKey system terminates, it is usually to a System Death, and the computer will have to be rebooted to continue running. If you are creating a TurnKey system, and a **BYE** should execute, the booting program will re-execute instead of a System Death.

Example:

```
PROGRAM Hello
```

```
HOME
PRINT "Hello"
BYE {Return to the Program Launcher}
```

Branching

Branching consists of unconditional and selective branching. With unconditional branching, the flow will be altered exactly as specified by the control structure. With selective branching, the branch will be based upon a condition previously determined. With selective branching, if the conditions are not right, the flow will not be altered at all.

With branching, control is directed to another area of the program. Careless use of this construct may cause havoc in your programs. For this reason, it is recommended you avoid branching as much as possible. Ideally, branching should only be done in error handling.

The Routine Declaration

ROUTINE Id

Before we can discuss branching, it is necessary to discuss a little about Routine declarations. This topic will be covered again in the next chapter in more detail.

Whenever you wish to branch to another line with the use of **GOTO**s, it is possible to branch to a mnemonic name instead of a line number. In order to do this, you must first declare the area of code you wish to branch to with a **ROUTINE** name. The syntax is simply the keyword **ROUTINE** followed by a unique identifier. This identifier has the exact same syntax as a simple variable and may be an existing variable name.

During compilation, the Compiler checks for the existance of duplicate **ROUTINE** names. If a second **ROUTINE** name is detected, an error will be issued.

WARNING

> If you are segmenting your program, then you may not give any **ROUTINE** name that has already been declared within a previous segment. The Compiler has no way to distinguish between identical **ROUTINE** names in different segments.

Unconditional Branching

Unconditional branching takes the program flow to the statement indicated. The abusive use of unconditional branching may considerably reduce the legibility of a program, so its use should be avoided whenever possible.

The Dreaded GOTO

GOTO Identifier
GOTO Line_Number

GOTO forces the program flow to the line indicated. If the reference line does not exist, the linker displays the message "Undefined line or subroutine". When a **GOTO** makes a reference to a line number (not recommended), the line number is treated as a **ROUTINE** identifier.

NOTE

> The use of **GOTO**s is recommended only in recovery from an error. To disable **GOTO**, use the **NOGOTO** compiler option.

Example:

```
IF Number = 5 THEN GOTO Routine_Name
END
ROUTINE Routine_Name
END
```

Selective Branching

Selective branching may be used when three or more selections are needed. The use of this option is not recommended as it can lead to problems in determining the program flow, if errors arise. The multi-decision **CASE_OF** is probably a more appropriate structure, and its use is recommended.

The ON..GOTO Statement

ON Aexpr GOTO Identifier [{,Identifier}]
ON Aexpr GOTO Line_Number [{, Line_Number}]

ON..GOTO branches to a specific statement or line depending on the value of Aexpr between the words **ON..GOTO**. If Aexpr is real, the value is truncated before the branch is taken.

Aexpr is evaluated. If the value is less than one or greater than the number of identifiers or line numbers, the program flow will continue with the statement following the **ON..GOTO**. Otherwise, the flow will be directed to the sequential label or line determined by the result.

Example:

```
PROGRAM Example
HOME
REPEAT
   PRINT "Enter a number from 1 to 3 ";
   GET Digit$
   PRINT Digit$
UNTIL INDEX (Digit$, "123") > 0
Digit% = VAL (Digit$)
ON Digit% GOTO One, Two, Three
{Exit point for program}
ROUTINE Finish
END {End of Program Execution}
{Selection is handled below}
ROUTINE One
   PRINT "One chicken soup"
GOTO Finish
ROUTINE Two
   PRINT "Two Fetucinni entrees"
GOTO Finish
ROUTINE Three
   PRINT "Three turkey breasts"
GOTO Finish
```

Loops

Repetitive statements are used to repeat an action until a condition is met. *Micol Advanced BASIC* has four repetitive contro' statements: **FOR..NEXT**, **FOR.UNTIL**, **REPEAT..UNTIL**, and **WHILE..WEND**.

Finite Loops

The statements in this section are useful for loops that have a predetermined number of iterations or repeat execution until another condition arises.

FOR .. NEXT Loops

FOR Loop Counter = Initial TO Terminal [STEP Increment]

This statement begins with the keyword **FOR** followed by an integer or real variable

as the Loop Counter. The Loop Counter is assigned the value in Initial and then verified to see if its value is greater than Terminal. If Loop Counter's value is greater than Terminal's, the statements within the loop will not be executed and control will continue to the statement after the following **NEXT**. If Loop Counter's value is smaller than or equal to Terminal's value, the statements within the **FOR** loop will be executed.

When all the statements in the loop have been executed, Loop Counter will either be incremented or decremented and the **FOR** statement will continue until Terminal's value is exceeded.

When there is a **STEP** Increment, if the result of Increment is positive, the value of Increment is added to the Loop Counter. If the result of Increment is negative, the positive value of Increment is subtracted from Loop Counter. If **STEP** is not specified, the increment is always a positive 1.

NEXT Loop Counter

NEXT followed by a Loop Counter signals the end of a **FOR** loop. The Loop Counter must match the one used in the previous **FOR** statement.

If, during compilation, a **FOR** statement is without its matching **NEXT**, the Compiler will issue an appropriate error message at the end of compilation.

Example:

```
FOR Loop_A% = 1 to 10
    FOR Loop_B% = 1 TO 10
      PRINT "Loop_B = "; Loop_B%
      PRINT "Loop_A = "; Loop_A%
    NEXT Loop_B%
NEXT Loop_A%
```

Please note the following rules for **FOR..NEXT** loop construction:

1. The loop will not be entered if the loop counter's value is already satisfied.

Example:

```
FOR Loop_Counter% = 10 TO 9
    PRINT Loop_Counter%
NEXT Loop_Counter%
```

2. The **NEXT** statement must contain the same variable used as the loop counter in the previous **FOR** statement, otherwise an error will occur during compilation.
3. A loop cannot be "exited" by changing the value of the loop counter. The value of the loop counter cannot be changed since the actual loop counter's value is maintained elsewhere. If any attempt is made to reassign the loop counter within the **FOR..NEXT** loop, the loop counter will be reassigned the value it otherwise would have at the top of the next iteration of the loop.
4. There may be only one **NEXT** for each **FOR**. A line of code like **IF Value = 10 THEN NEXT Ctr** is not allowed in *Micol Advanced BASIC*.
5. The terminal expression is calculated each time at the top of the loop. The **FOR**

loop may end prematurely if a variable is used for the Terminal value and this variable is being reassigned inside the loop. Watch out for an unintentional reassignment. Also, if the terminal expression is somewhat complicated, it may eat up valuable execution time. It is preferable to assign that expression to a dummy variable just outside the loop, and use this dummy variable as the terminal value within the **FOR..NEXT** loop.

Example:

```
FOR Ctr1% = 3 TO 32000 STEP 2
   Dummy% = SQR (Ctr1%)
   FOR Ctr2% = Ctr1% TO Dummy% STEP 2
     IF Ctr% MOD Ctr2% = 0 THEN BEGIN
          Dummy% = 1 {Stop the inner loop}
     ENDIF
   NEXT Ctr2%
   IF Dummy% > 1 THEN PRINT Ctr2%
NEXT Ctr1%
```

Be certain the variable (Dummy%) is not unintentionally changed within the active **FOR..NEXT** loop as the loop may not act as desired.

6. Never use a **GOTO** to exit a **FOR..NEXT** loop, otherwise the pointers necessary for the functioning of **FOR..NEXT** statements will not be reset correctly. The program may malfunction if this loop is used again. If a **FOR..NEXT** loop must be left prematurely, use the **FOR..UNTIL** loop structure instead.
7. The use of integer loop counters is recommended, where practical, as they execute much faster than their real counterparts.

FOR .. UNTIL Loops

FOR Loop Counter = Initial TO Terminal UNTIL Relop

The **FOR..UNTIL** structure repeats one or more statements a precise number of times or until the specific condition is TRUE.

This statement begins with the keyword **FOR** followed by a Loop Counter. The Loop Counter is assigned the value in Initial. The Loop Counter is then verified to see if its value is greater than the Terminal value.

If the Loop Counter's value is greater than the terminal value, the statements in the loop will not be executed and control will be directed to the statement following the next **NEXT**. If the Loop Counter's value is smaller than or equal to the Terminal value, a test is made to see if the **UNTIL** condition is TRUE or FALSE. If the condition evaluates to TRUE, control is passed to the statement after the **NEXT** statement. If the **UNTIL** condition is FALSE, the loop is entered.

When all the statements in the loop have been executed, Loop Counter will have one added to its current value and the **FOR** statement will continue until the value of Loop Counter is greater than Terminal or until Relop become TRUE. Loop Counter is always

incremented by one.

As with the **FOR..NEXT** loop construct, this statement must also be closed by a **NEXT** statement with a matching Loop Counter. The pertinent rules described above for **FOR** loops also apply here.

Example:

```
FOR Loop_Ctr% = 1 TO 10 UNTIL Animal$ = "cat"
    INPUT "Enter any animal's name ";Animal$
    PRINT "The ";Animal$;" is a fine animal"
    Animal$ = LOWER$ (Animal$) {Need lowercase for test}
NEXT Loop_Ctr%
```

FOR..NEXT and **FOR..UNTIL** loops may be nested. The maximum nesting is 20 levels deep.

Examples: {Notice the nesting order}

```
FOR Out_loop_Ctr% = 1 TO 10
    FOR In_loop_Ctr% = 1 TO 10
      PRINT "In_loop_Ctr = ";In_loop_Ctr%
      PRINT "Out_loop_Ctr = ";Out_loop_Ctr%
    NEXT In_loop_Ctr%
NEXT Out_loop_Ctr%
```

This second example will show an alternate way of writing nested **FOR..NEXT** loops, but the logic is also more difficult to follow.

```
FOR Out_loop% = 1 to 10: FOR Inloop% = 1 TO 10
PRINT "Inloop = ";Inloop%:PRINT "Out_loop = ";Out_loop%
NEXT Inloop%: NEXT Out_loop%
```

Examples of what NOT to do are:

```
FOR i = 1 TO 50  FOR j = 1 TO 10
      PRINT i,j  NEXT i
NEXT j {Misplaced loop variables}
```

Conditional Loops

Conditional loop structures will execute the statements inside the structure until a particular condition does or does not arise.

REPEAT Loops

REPEAT
Statement
[{: Statement }]
UNTIL Relop

The **REPEAT..UNTIL** structure executes the statement(s) enclosed between these keywords until Relop is TRUE. The statement(s) in the loop will always be executed at least once. The program flow continues after the **UNTIL** statement.

Example:

```
REPEAT
    INPUT "Enter any animal's name: ";Animal$
    Animal$ = LOWER$ (Animal$)
    IF Animal$ <> "cat" THEN BEGIN
      PRINT "The ";Animal$;" is a fine animal"
    ENDIF
UNTIL Animal$ = "cat"
```

WHILE Loops

WHILE Relop
Statement
[{: Statement }]
WEND

The **WHILE..WEND** structure executes the statement(s) enclosed between these keywords as long as Relop is TRUE. The statement(s) in this loop will not be executed if the expression is not initially TRUE. The program flow continues after the keyword **WEND** (for WhileEND).

Example:

```
Animal$ = "" {Make certain loop is entered}
WHILE Animal$ <> "cat"
    INPUT "Type any animal's name";Animal$
    Animal$ = LOWER$ (Animal$)
    IF Animal$ <> "cat" THEN BEGIN
      PRINT "The ";Animal$;" is a fine animal"
    ENDIF
WEND
```

The **REPEAT..UNTIL** and **WHILE..WEND** structures may be nested to a maximum of 20 levels each.

Chapter Nine

Modularization

Overview

When a project becomes a large programming task, it becomes necessary to break this task into smaller portions, making this project easier to conceive. This method applies the old maxim: "Divide and Conquer."

A large program may be divided into modules. A module is like a small program that may be executed whenever needed. Each module performs a specific task. Breaking a program into small, easy-to-maintain portions is called modularization.

Not only does modularization simplify the programming task, it also has the advantage of creating routines that may be re-used by other programs.

A module is a very important construct to the concept of structured programming. Once control is passed to a module, unless an unforeseen circumstance occurs, control will return to a known location.

Advantages of Modularity

1. **Ease of conception.** It is easier to create an ensemble of short and simple modules than a long and linear program. Each module will perform a certain, well-defined task.
2. **Maintainance**. Because each module performs a single well-defined task, it is relatively easy to debug and modify this module as the need arises.
3. **Portability**. The modules written may be as independent as possible from other modules. Thus a module may then be used in another program with no or very few changes.
4. **May be written by different programmers**. Once the task to be done is well defined, the modules may be written by more t' an one person. After the modules are written, they also may be individually tested.

Module Types

Micol Advanced BASIC has three different types of modules: the Routine, the Function and the Procedure.

A Routine is probably what you are already familiar with. A Routine is the typical BASIC "subroutine". All variables are global (available to the entire program), and parameters are not passed to it. Control is passed to the Routine with a **GOSUB** or **PERFORM** statement and control is returned through a **RETURN** statement placed ideally at the end of the Routine. Unlike most BASICs, a Routine in *Micol Advanced BASIC* may be given a name with which the Routine may be later referenced.

A Function is a module which returns a numeric result. The Function may have both local and global simple variables, accepts one or more parameters and always returns a single numeric value. A Function is given a name and is implicitly called within an **FN** statement. Control is not returned until the end of the Function is encountered.

A Procedure, like a Function, has both local and global simple variables and accepts parameters. Control is passed to a Procedure by means of a **GOSUB**, and control is returned following the Procedure call. Values that need to be shared between a Procedure and the main body of the program are shared by means of parameters passed by address or by global variables declared earlier in the main program body.

Module Identification

As described under **ROUTINE** names in the previous chapter, all Routines, Procedures and Functions may have distinct identifiers.

The Compiler saves the module names declared after a **FUNC**, **PROC** or **ROUTINE** reserved word during compilation. If duplicate module identifiers are found, the Compiler will report an error.

If you attempt to reference a Function, Procedure or Routine within your program which you have not defined, during the linking phase, you will receive the message, "Undefined Line or Subroutine" error. Since the Linker has no way of knowing at which line this error occurred, you will need to use the Source Code Editor to locate the undefined Routine.

Program Order with Modules

```
PROGRAM Identification
ALIAS "UNTIL 1 = 2" = "FOREVER"
INT (I-N): STR (S-Z)
DATA statements
DIM statements
DECLARE Boolean!, Integer%, Real&, String$
Function Declarations
Procedure Declarations
Main Program Body
Routine Declarations
END
```

The above list of declaration statements should be followed to ensure a structured program.

Routines

ROUTINE Identifier
[{ Statement(s)}]
RETURN

A Routine is declared by using the reserved word **ROUTINE** followed by an identifier.

The body of the Routine may contain any legal executable statements: **DIM, DATA** and compiler directives are not executable statements.

RETURN marks the end of the Routine, and tells the program to return to the statement following the **GOSUB** which caused the branch to this Routine. Only one **RETURN** should appear in a Routine.

RETURN must never be used to end a Procedure or Function as the Compiler will return an error if so attempted.

A Routine module is called by means of a **GOSUB** statement followed by the identifier of the Routine.

If the return stack is empty when the **RETURN** is executed, the message "RETURN without GOSUB error" is displayed when the error occurs at run time.

All variables included in a Routine are global and may be used by other Routines.

WARNING

> If the normal program flow reaches a Routine, the Routine will execute. This must be avoided. For this reason, Routines should be placed after the main program body, so they will not be executed without being explicitly called. There should be an **END** statement at the end of the main program body to stop the program flow.

Example:

```
GOSUB Box
END
ROUTINE Box
    PRINT "In subroutine"
RETURN
```

Functions and Procedures

As in the Pascal and C languages, *Micol Advanced BASIC* has the concept of Procedures and Functions that are separate from the main body of the program and that may receive values as parameters.

General Rules

A program may have a maximum of 127 Functions or Procedures. The Functions and Procedures may reside anywhere in the program, but it is best to declare them all at the top of the program.

Unlike a Routine, a Procedure or Function will not execute by simply letting the normal program flow reach the Procedure or Function: it must be called. Also, unlike a Routine, a Function or Procedure may have both local and global variables and accept values as parameters.

Nesting of Procedures and Functions is not allowed.

Global and Local Variables

Global Variables

A global variable is a variable that may be used and modified by any part of the program. Any variable declared at the top of the program outside a Procedure or Function is always global. Arrays are always global. This means the entire program is able to access any array element.

It is sometimes necessary for the entire program, including Procedures and Functions, to be able to "see" certain variables. Whenever a variable is declared outside of a Procedure or Function, but before this Function or Procedure, any subsequent code, including Functions and Procedures, will have access to this variable. The variable is declared simply by being used; initializing the variable(s), or placing it in a **DECLARE** statement is all that's necessary.

Example:

```
PROGRAM Global_Test
{Variable Global& may be used by the Procedure}
Global& = 567.89
PROC Example [Real&, Integer%]
   PRINT Real&
   PRINT Integer%
   PRINT Global&
ENDPROC
GOSUB Example [100.1, 123]
END
```

Local Variables

Any variable declared within a Procedure or Function is local to that Procedure or Function only if that variable has not been declared globally before this Procedure or Function.

By local, we mean that only the Function or Procedure in which the variable is used will have access to it. Neither the main body of the program, nor another Function or Procedure can see the variable. Two variables within two Functions or Procedures may look the same, but in reality these variables are different.

Using local variables has the great advantage that the value of a variable with the same name outside the Function or Procedure is not accidentally changed by the program.

Values may be shared outside the Function or Procedure only if a parameter is passed by address or a variable has been declared earlier as global.

If the **LIST** or **PRINTER** compiler option is in effect, a number sign (#) will precede the names of local variables in the Symbol Table listing (displayed after the compilation).

Example:

```
PROGRAM Global_Test
PROC Example [Number%]
   PRINT Number%
ENDPROC
Number% = 567
GOSUB Example [123]
PRINT Number%
```

In this example, the local variable Number% within the Procedure will have a value of 123, and the global variable Number% outside the Procedure will have a value of 567.

The Optional Parameter List

Values may be passed to a Function or Procedure by means of parameters. Parameters are variables within a Function or Procedure that will contain a value passed to it after it has been called. A parameter list is the series of values sent to the Function or Procedure when the Function or Procedure is being called. Both parameters and parameter lists are enclosed in brackets.

The rules for the declaration of the parameters are the same as those for any other variable. For all practical purposes, the number of parameters that may be passed is unlimited.

Each parameter will have a corresponding value passed to it when the Function or Procedure is being called. A strict one to one correspondence exists between the type of value passed and the receiving parameter; they must be of the same data type.

Parameters may be simple variables of boolean, integer, real, or string. Parameter lists may be arithmetic expressions or variables, string variables and literals or booleans which may also be the reserved words **TRUE** and **FALSE**.

A real literal, if passed in the parameter list, must have its fractional part explicitly written, so that the Compiler knows whether a real or an integer literal is intended. If the real value has no fractional portion, you must specify a .0 as in 123.0.

If a mismatch occurs between the parameter type and the passed value type, an error

will occur during execution. For example, if a real expression is passed as the first value to a Function, the first corresponding parameter must be a real variable; the same applies to integer, string or boolean parameters.

Ways of Passing Parameters

Each parameter that is passed to either a Procedure or Function may be passed in one of two ways: pass by address or pass by value.

It is important to understand the difference, as this can affect the program's logic. People familiar with either the Pascal or C languages should already have a good understanding of these concepts.

Passing by Value

To declare explicitly that a parameter is passed by value, use the reserved word **VALUE** before the parameter declarations. Passing a parameter by value is the default. Every parameter encountered up to an **ADDRESS** reserved word or the end of the parameter declarations will be passed by value.

If a parameter is passed by value, only the value in the passing variable is given to the Procedure or Function. This means, that under no circumstance will the passing variable have its value changed within the receiving subroutine.

Example:

```
PROGRAM Example
{Passing by Value is default}
PROC Add [VALUE Gamma]
   Gamma = Gamma + 1
ENDPROC
Upsilon = 10
Gamma = 25
GOSUB Add [Upsilon]
PRINT Upsilon, Gamma
```

The values printed are 10 and 25. Thus, the value of the parameter passed was not modified by the Procedure. When the Procedure Add was called, the variable Gamma was created and the value of the parameter (25) was assigned to it. The incrementation Gamma = Gamma + 1 was done with this new variable and not to the variable Upsilon where the value was unchanged. The value of Gamma is 25 outside the Procedure because the one is added to the local variable Gamma, not the global (but declared after the Procedure) variable Gamma.

Passing by Address

When a parameter is passed by address, the address of the passing variable is also passed to the Procedure or Function so that the passing variable will be modified if the parameter is altered within the called Procedure or Function.

WARNING

> When an integer or real literal is passed by address, that value is made vulnerable to change within the program. For this reason, never pass a literal as a parameter when it is passed by address as the literal's value in memory may also change.

To pass a parameter by address, use the reserved word **ADDRESS** followed by the parameters to be passed by address. All parameters up to the end of the parameter declaration or the reserved word **VALUE** will be passed by address.

Example 1:

```
PROGRAM Example
PROC Add [ADDRESS Gamma]
   Gamma = Gamma + 1
ENDPROC
Upsilon = 10
Gamma = 25
GOSUB Add [Upsilon]
PRINT Upsilon, Gamma
END
```

The values printed are 11 and 25 respectively. The value of the passed parameter Upsilon was modified by the Procedure.

Note that the local Gamma and the global Gamma still have different values.

Function Definition

FUNC Identifier [Parameter list]
Statement(s)
ENDFUNC [Variable]

To define a Function, use the reserved word **FUNC** followed by any unique, legal identifier. The Function identifier may be followed by an optional list of parameters encased in brackets ([]).

The body of the Function may contain any legal executable statements, the same as a Routine.

A Function is terminated with an **ENDFUNC**. Following the reserved word **ENDFUNC** must appear brackets enclosing a simple variable which contains the value which needs to be returned by the Function. The variable must be of the same type, either integer or real, as the calling formula with the **FN** statement; otherwise, an error will occur at run time.

A Function is implicitly called within a formula by preceding the Function identifier

and an optional parameter list by the reserved word **FN**.

WARNING

> Do not attempt to access a Function with a **GOSUB**. If you do, you cannot access the value returned by the Function. Also, do not use a parameter variable as the variable used to return the Function value as the result may become corrupted.

If the Function which you try to access does not exist, you will be informed during the linking phase.

Example:

```
FUNC Square [Param]
   Variable = Param * Param
ENDFUNC [Variable] {Square}
INPUT "Calculate the square of what number?"; Digits
{Function call follows}
Number = 2 * FN Square [Digits] + 1
```

If you enter 5, for example, the Function Square will return 25.

Procedure Definition

PROC Identifier [Parameter list]
Statement(s)
ENDPROC

To declare a Procedure, use the reserved word **PROC** followed by a Procedure identifier. The Procedure identifier may be followed by an optional parameter list encased in square brackets ([]).

The body of the Procedure may contain any legal executable statements: **DIM**, **DATA** statements and compiler directives are not executable statements.

The Procedure must be terminated by an **ENDPROC**, which ends the Procedure and generates an automatic return to the statement following the Procedure call. The Compiler will inform you if an **ENDPROC** has been omitted at the end of compilation.

NOTE

> If you attempt to use a **RETURN** in a Procedure, the Compiler will issue an error.

A Procedure may be called only with a **GOSUB** followed by the Procedure identifier

and the optional parameter list. The **GOSUB** must not branch to a line within the Procedure as unexpected results will occur. If the Procedure does not exist, a message will be displayed during the linking phase.

Explicit Variable Declarations

If a **DECLARE** is used in a program containing Functions and Procedures, every subsequent Procedure and Function which contain local variables will need a **DECLARE**. Include the **DECLARE** following the Procedure or Function definition. There is an implicit **DECLARE** within the parameter declarations, so no **DECLARE** is required there.

Example:

```
PROGRAM Declare_Test
PROC Example [Parm1, Parm2%]
   DECLARE Real&, Integer%
   Real& = Parm1
   Integer% = Parm2%
ENDPROC
GOSUB Example [100.1, 123]
```

Passing Control to a Subroutine

FN Identifier [Parm-1, Parm-n]

A Function cannot be called explicitly as a Routine is called, but must be called implicitly within a mathematical formula.

In order to call a Function and have it return a value, within the formula where the value is required, insert the keyword **FN** followed by the Function name followed by the optional parameter list. In effect, the Function is treated as a sort of variable.

Example:

```
Number = 100 + 32 * FN Square [Parm] / 22
```

GOSUB Identifier [Parm-1, Parm-n]
GOSUB Line_Number [{, Line_Number}]

GOSUB is used to pass control to either a Routine or Procedure. If control is given to a Routine, control is returned with a **RETURN** statement. If control is given to a Procedure, control is only returned at the end of the Procedure by an **ENDPROC**. In both cases, the execution will continue after the statement following the calling **GOSUB**.

Example:

```
GOSUB Label
PRINT "Program will resume here"
END
ROUTINE Label
   PRINT "Now in subroutine"
RETURN
```

POP

POP is the enemy of structured programming. **POP** removes the latest **GOSUB** address from the stack. This can be very dangerous making it difficult to determine where an error occurred.

Although some use for **POP** can be found, the use of **POP** is not encouraged as it may lead to chaos in your programs. **POP** was retained solely for compatibility with Applesoft BASIC.

The **NOGOTO** compiler option may be used to disallow the use of **POP**.

PERFORM Routine_Id UNTIL Relop

A **PERFORM** executes a Routine continuously until the Routine sets the Relop following the **UNTIL** to TRUE.

As with a **GOSUB**, a **RETURN** is expected at the end of the called Routine to cause a return to the **PERFORM** statement.

Example:

```
PERFORM Animals UNTIL Animal$ = "cat"
END {This statement is necessary}
ROUTINE Animals
   HOME {No offense to cat lovers}
   INPUT "Type in any animal's name ";Animal$
   Animal$ = LOWER$ (Animal$)
   IF Animal$ <> "cat" THEN BEGIN
     PRINT "The ";Animal$;" is a fine animal"
   ENDIF
RETURN
```

Computed Routine Selection

ON Aexpr GOSUB Routine_Id1 [{,Routine_Id(n)}]

The **ON..GOSUB** structure works in a similar manner to the **ON..GOTO** structure. **ON..GOSUB** also allows you to use named Routines. Based upon the result of Aexpr, the proper module identifier will be used.

If the result of the expression is one, the first label in the list will be used. If expression is two, the second label in the list will be used, etc.

If the value is none of the above possibilities, the first sequential statement following **ON..GOSUB** will be taken. As with any **GOSUB** to a Routine, when the system encounters a **RETURN**, the next statement following the computed **GOSUB** will be executed.

Example:

```
INPUT "Enter a value between 1 and 3 ";Integer%
ON Integer% GOSUB One, Two, Three
END
ROUTINE One
   PRINT "One"
RETURN
ROUTINE Two
   PRINT "Two"
RETURN
ROUTINE Three
   PRINT "Three"
RETURN
```

Module Library Usage

A library of modules is a collection of often used Functions and Procedures that may be used in several programs.

Why create a library of modules? Because you don't want to keep reinventing the wheel. Using a library of modules in your programs give them consistency and makes your programs easier to develop and maintain because the modules are already written and debugged.

Creation of a Library of Modules

First, you must decide what Procedures or Functions you require for future use. Be certain each subroutine is completely reliable and thoroughly commented.

Create a subdirectory on a suitable volume and save the source code of the

subroutines to a suitable filename under this directory.

When you wish to use a Function or Procedure from this library, make use of the **INCLUDE** statement described below.

INCLUDE Pathname

To include a module in a program, add the line **INCLUDE** Pathname in the source code file. Pathname indicates the path to a source code file (of type TXT or SRC). Pathname may only be a string literal.

The **INCLUDE** statement may appear anywhere in a program after the compiler directives. An **INCLUDE** file may have **DATA** and array declaration statements but the **DATA** and **DIM** statements must still appear in their established order.

The file being read in must be available (online) at compilation time. When the Compiler detects an **INCLUDE** statement, it looks for a file with the specified Pathname and starts reading it as though it were included inside the program itself. The Compiler displays the message "INCLUDING pathname" each time it detects an **INCLUDE** statement.

Using the **INCLUDE** statement also has the advantage of having only the necessary program code in the Editor, saving the Editor's work space for the code specific to your application.

IMPORTANT

> Make sure your module has been thoroughly debugged before you include it in your program as the sequential line number information is frozen at the line of the **INCLUDE** statement and resumes only after the module has been read. Run time errors may be difficult to detect.

Example:

```
INCLUDE "/Micol.Adv.BASIC/Library/Math.Routines"
```

Recursion

Recursion is an important topic in computer science. Those of you who have studied computer science at the college or university level are already well aware of this fact. Those of you who are planning to study computer science will soon be finding this out for yourselves. What is recursion, and why is it so important?

Recursion is the act of stipulating something in terms of itself. We have all heard it said, "a rose is a rose is a rose". This, in a way, is a recursive definition of a rose. The rose is defined in terms of itself.

The concept of recursion is not something we deal very often with in our daily lives as the previous definition of a rose proves. Not many things around us can be defined in terms of themselves.

Mathematics has some use for recursion though. The most common example of a use for recursion in mathematics is the definition for the factorial of a number: N! = N * (N - 1)!

This formula translated is: the factorial of a number N is equal to the number N times the factorial of the number N minus one. As you can see, the factorial of a number is defined in terms of lower orders of itself. If we add to this the definition that when N reaches its lowest allowed value of one, that N! is equal to 1, we have a complete recursive definition for factorial.

There is much in computer science that can be defined in terms of itself. This programming language, *Micol Advanced BASIC*, was designed with a parse table that has many features defined in terms of themselves.

As with the definition of factorial above, the definition must be complete, or our recursive definition is worthless. If factorial had been left undefined for its smallest value of one, we could not have made use of it. One minus one is zero, and anything multiplied by zero is zero.

Because much of what is defined in computer science is defined recursively, it is only natural that computer scientists would like programming languages that allow them to express the solution in the manner in which they have laid out the problem in question. This is the principal reason recursion in programming languages is so stressed in computer science.

But, recursion in programming languages suffers some severe problems which we will now demonstrate. Let us take the definition for factorial just given and program it in *Micol Advanced BASIC* making use of recursion. You will soon see why recursion might be desirable, and also why it is often not the best way to solve a problem.

Example:

```
PROGRAM Recursion
FUNC Factorial [N]
    IF N <= 1 THEN BEGIN
      Factorial = 1
    ELSE BEGIN
      Factorial = N * FN Factorial [N - 1]
    ENDIF
ENDFUNC [Factorial]
{Start of Program}
HOME INPUT "Take the factorial of what number ? ";Number
Factor = FN Factorial [Number]
PRINT "The factorial of ";Number; " is ";Factor
END
```

As you can see, the function Factorial looks very much like the mathematical definition for factorial. This function will continue to call itself until N is less than or equal to one, at which time it will simply unwind the stack, successively returning another value for Factorial [N - 1].

One problem has to do with implementation of recursion under the programming language being used. How is the parameter N treated by the language? If the programming language does not reinstate the previous value of N as the return stack unwinds, as *Micol Advanced BASIC* does, the recursive function will not act as desired.

Another problem is that we are only looking at the theory and not at the real world of programming. In the real world, there is much that goes on behind the scenes in the execution of the programming language to maintain these calls. For example, each time the **FN** statement is executed, a run time stack must be saved and then reinstalled after the return from the call. There is also a certain overhead with the passing of each parameter, etc. Factorial could be programmed more effectively using a simple loop instead of recursion.

A question once asked on a final exam in a computer science class was: "True or false, anything that can be programmed in a loop can be programmed using recursion?".

The author of this question was looking too much at the theory of recursion, and not enough at the reality. Recursion is, itself, simply a type of controlled looping, so that the question had little real meaning. Use recursion when it is practical, but do not lose sight of reality.

Chapter Ten

Graphics

Overview

Micol Advanced BASIC has commands for making great graphics using the 32K Super High Resolution graphics screen built into your Apple IIGS.

Micol Advanced BASIC supports Low Resolution graphics with four lines of text (40 x 40 blocks) similar to Applesoft BASICs. *Micol Advanced BASIC*'s Super High Resolution commands control both of the Apple IIGS's Super High Resolution graphics modes: 320 X 200, and 640 X 200.

Low Resolution Graphics

The Low Resolution graphics mode with text (40 x 40 blocks in 16 colors) is supported in *Micol Advanced BASIC*.

GR

GR sets a Low Resolution screen of 40 blocks x 40 blocks. **GR** must be executed before any other Low Resolution commands are executed; otherwise further Low Resolution graphics commands will have no effect.

When **GR** executes, the Low Resolution screen is established and cleared to black. The text cursor is moved to the twenty-fourth text line.

The point of origin of the coordinate system starts at the upper left corner of the screen: 0,0 is the upper left corner 39, 39 is the lower right corner

Four lines of text at the bottom of the screen may be displayed.

Example: see under **HLIN**.

COLOR = Color_Number

Sixteen colors may be displayed in Low Resolution graphics. Color_Number ranges from black (0) to white (15). If no color is specified, color 0, black, is used by default. This means that if the Low Resolution color is not set, your graphics will be invisible.

Table 3.10.1 - Low Resolution Colors

Value	Color	Value	Color
0	black	8	brown
1	magenta	9	orange
2	dark blue	10	gray2
3	violet	11	pink
4	dark green	12	green
5	gray1	13	yellow
6	medium blue	14	aqua
7	light blue	15	white

Example: see under **HLIN**.

HLIN X-Coord1, X-Coord2 AT Y-Coord

HLIN stands for Horizontal LINe. **HLIN** draws a Low Resolution horizontal line using the most recently defined color from point X-Coord1, Y-Coord to X-Coord2, Y-Coord. The X co-ordinates may not be negative or greater than 39. The Y co-ordinate may not be negative or greater than 47. If these coordinate values are exceeded, an error will occur during execution. Any of the values above may be either integer or real expressions.

Example:

```
PROGRAM Show_HLIN
INT (A-Z)
GR
FOR Loop1 = 0 TO 39
    FOR Loop2 = 0 TO 39
      Y_Coord = RND (47)
      X_Coord1 = RND (39)
      X_Coord2 = RND (39)
      COLOR = RND (14) + 1
      HLIN X_Coord1, X_Coord2 AT Y_Coord
    NEXT Loop2
NEXT Loop1
END
```

PLOT X-Coord, Y-Coord

PLOT places a Low Resolution block at the location specified. The X coordinate may not be negative or greater than 39 under Low Resolution. The Y coordinate may not be negative or greater than 47; if they are, an error will occur during execution.

Example:

```
PROGRAM Show_Blocks
INT (A-Z)
GR
FOR Down = 0 TO 47
   FOR Across = 0 TO 39
     COLOR = RND (15)
     PLOT Across, Down
   NEXT Across
NEXT Down
END
```

SCRN (X-Coord, Y-Coord)

SCRN stands for screen. It returns the color number of the block specified at the location X-Coord, Y-Coord. X-Coord and Y-Coord must be within the limits specified under **PLOT**.

Example:

```
PROGRAM Show_Pos
GR
COLOR = 13
PLOT 10, 20
Color_Num% = SCRN (10, 20)
VTAB(24)
PRINT "Colour Number = ";Color_Num%
GET Wait$
END
```

TEXT

TEXT turns off the Low Resolution graphics mode and turns on the 80 column text screen. Follow **TEXT** with a **HOME** to remove any garbage text characters left over from the Low Resolution screen.

VLIN Y-Coord1, Y-Coord2 AT X-Coord

VLIN stands for Vertical LINe. **VLIN** draws a vertical line using the most recently defined color from point Y-Coord1, X-coord to Y-coord2, X-coord. The X coordinate may not be negative or greater than 39 in Low Resolution mode. The Y coordinates may not be negative or greater than 47 otherwise an error will occur during execution. Any of the values above may be either integer or real expressions.

Example:

```
PROGRAM Show_VLIN
INT (A-Z)
GR
FOR Loop1 = 0 TO 39
    FOR Loop2 = 0 TO 39
      X_Coord = RND (39)
      Y_Coord1 = RND (47)
      Y_Coord2 = RND (47)
      COLOR = RND (14) {Don't want black} + 1
      VLIN Y_Coord1, Y_Coord2 AT X_Coord
    NEXT Loop2
NEXT Loop1
DELAY = 1000
TEXT:HOME
END
```

Super High Resolution Graphics

The usual Applesoft BASIC High Resolution graphics commands have been implemented in *Micol Advanced BASIC* using the much improved Super High Resolution (SHR) graphics modes of the Apple IIGS. While Applesoft's High Resolution graphics had a maximum High Resolution of 260 X 192, *Micol Advanced BASIC* has a maximum Super High Resolution of 640 X 200. In addition, the colors under *Micol Advanced BASIC* are vastly improved over their Applesoft counterparts.

Please note these important differences between Super High Resolution graphics under *Micol Advanced BASIC* GS and High Resolution graphics under Applesoft BASIC:

- SHR has:
  - — only one display area
  - — a much greater resolution display
  - — a greater number of colors
  - — a greatly improved color quality
  - — a choice of background colors

— an easy mix of text and graphics
— the ability to use the Apple IIGS ToolBox for drawing
— no bit-mapped shape tables

HGR and HGR2

HGR must be used to set the 320 X 200 Super High Resolution graphic mode. The pixels (picture elements) are numbered from 0 to 319 horizontally, and from 0 to 199 vertically. A maximum of 16 different colors may be displayed at one time. **HGR** must appear in the program for the 320 X 200 graphic mode to be turned on.

HGR2 must be used to set the 640 X 200 Super High Resolution graphic mode. The pixels (picture elements) are numbered from 0 to 639 horizontally, and from 0 to 199 vertically. A maximum of 16 different colors may also be displayed at one time. **HGR2** must appear in the program for the 640 X 200 graphic mode to be turned on.

With both graphic modes, the graphics color as well as the background color is set to black. Use **BKCOLOR** and **HCOLOR** to change the shade of the background and graphics colors respectively.

The point of origin of the co-ordinate system starts at the upper left corner of the screen: 0,0 is the upper left corner for both graphics modes. 319, 199 is the lower right corner for 320 (**HGR**) mode and 639, 199 is the lower right corner for 640 (**HGR2**) mode. Initially, the position is set to 0,0 in both modes.

Images appearing in 320 mode are twice as large as the 640 mode, but with half the resolution. For example, a box that appears square in 320 mode will appear as a vertical rectangle in 640 mode.

Example: (See example under **HPLOT TO**.)

BKCOLOR = Color_Number

This command allows you to change the background color to any color currently available (please see table 3.10.2). The commands (**HGR** or **HGR2**) that start up the Super H' h Resolution graphic modes also set the background color to 0 (black).

NOTE

> The effect of **BKCOLOR** is not immediate.

BKCOLOR also changes the background color of the characters displayed using **DRAWSTR**.

Example: (See example under **HPLOT TO**.)

HCOLOR = Color_Number

HCOLOR sets the current Super High Resolution color according to Table 3.10.2. The default color when an **HGR** or **HGR2** is issued is black, which under most circumstances means invisible. You will probably wish to set another color using **HCOLOR** before you actually begin to draw.

In both graphics modes, you have a maximum of 16 colors. We have tried to keep the colors in both modes as close to each other as possible.

Programmers

> By a skillful use of the Apple IIGS ToolBox, using Tool number 4 (QuickDraw), it is possible to have as many as 4096 colors in 320 mode (16 with each pixel). This is not possible in 640 mode as special action had to be taken to give you 16 colors (4 is the usual number of colors in 640 mode). Please see Part Five on ToolBox usage for further information.

Table 3.10.2 SHR Colors

Number	HGR	HGR2
0	Black	Black
1	Dark Gray	Dark Gray
2	Brown	Light Yellow
3	Purple	Purple
4	Blue	Blue
5	Dark Green	Dark Green
6	Orange	Orange
7	Red	Red
8	Beige	Pink
9	Yellow	Yellow
10	Green	Green
11	Light Blue	Light Blue
12	Lilac	Lilac
13	Periwinkle	Light Purple
14	Light Gray	Light Gray
15	White	White

DRAWSTR (Svar)

To add text to the SHR graphics, use **DRAWSTR**. Svar may be a string literal or a string variable. The string stipulated in Svar will be displayed starting at the current graphics position. You may wish to use **HPLOT** to move to the position you desire.

The text will be colored using the latest **HCOLOR** set. **BKCOLOR** may be used to set the background color of the text.

If SHR is active, **LEN** may be used to find the graphics length of the string. **PEEK True_Value** (locations 202 and 203) after taking the **LEN** to find the size in pixels.

Example: (See example under **HPLOT TO.**)

HPLOT X-Coord, Y-Coord

HPLOT moves the high resolution pointer to the X coordinate and Y coordinate stipulated and plots a single point in the latest **HCOLOR**. The maximum X-Coord and Y_Coord values are those discussed under the **HGR** and **HGR2** commands. Values too large will plot off the screen and no error will be generated.

Example: (See example under **HPLOT TO.**)

HPLOT TO X-coord, Y-coord

HPLOT TO will draw a straight line from the last graphics position to the position stipulated using the latest **HCOLOR**. No errors are generated by this command; values greater than the screen will display to the limits of the screen. You may wish to use **HPLOT** to change the graphics position.

Example:

```
PROGRAM Draw_Box
HGR2 {set 640 mode}
HCOLOR = 4 {Blue}
BKCOLOR = 15 {White}
HPLOT 10, 10 {Draw box starting here}
HPLOT TO 600, 10
HPLOT TO 600, 190
HPLOT TO 10, 190
HPLOT TO 10, 10
HPLOT TO 600, 190
HPLOT 600, 10
HPLOT TO 10, 190
{Write message on box}
HPLOT 190, 50 {Start the message here}
```

```
DRAWSTR ("This is a big box")
DELAY = 1000
TEXT
END
```

This will plot a blue box crossed with an X for about 10 seconds.

Super High Resolution Shapes

Once the Super High Resolution mode is set with either the **HGR** or **HGR2** command and the colors set using the **HCOLOR** and **BKCOLOR** commands, it is then very easy to draw most figures using the **TOOLBOX** command.

The four types of shapes you can create are: rectangle, oval, arc, and rounded rectangle.

The three different modes of drawing are: paint (draws solid figure), frame (draws the outline of the figure only), and erase (draws a figure using the current background color).

For example, to draw a circle, first select the proper graphic mode using **HGR** or **HGR2**, set the desired color(s), then use the **TOOLBOX** command to draw the circle.

When using four basic shapes (Rectangle, Oval, RRect and Arc), the shapes are drawn by Tool number 4, QuickDraw II, in an invisible rectangle to determine their size and shape. QuickDraw II gets the dimensions of the rectangle through the following parameters:

- Min_X (the X value of the left side)
- Min_Y (the Y value of the top)
- Max_X (the X value of the right side)
- Max_Y (the Y value of the bottom).

The Procedures **Draw_Rect** and **Draw_Arc** below may be used to draw four different types of shapes in three different modes in any size. **Draw_Rect** is used to draw the rectangle and the oval while **Draw_Arc** draws the arc and the rounded rectangle.

When using the Procedure **Draw_Arc** to do arcs, **Start_Angle** is the angle in degrees which the arc starts, with 0 degree being a vertical line , and **Angle_Length** is the length of the arc in degrees.

To create all this variety with the code presented below, select the correct Procedure and change the first parameter passed to the desired value based upon the table on the next page.

Many, many more figures are possible using very similar techniques detailed in the manual Programming the Apple IIGS Toolbox listed in Part One.

Table 3.10.3 Pre-Defined Shape Drawing Functions

	Rectangle	Oval	RRect	Arc
Frame	83	88	93	98
Paint	84	89	94	99
Erase	85	90	95	100

Example:

```
PROGRAM Shape_Examples
INT (A - Z)
DIM Buffer (10)
PROC Draw_Rect [Func_Num, Min_X, Min_Y, Max_X, Max_Y]
   LSB = ADDR (Buffer ( ) {Address of buffer}
   MSB = PEEK (202) {Bank of Buffer}
   TOOLBOX (4, 74: MSB, LSB, Min_X, Min_Y, Max_X, Max_Y)
   TOOLBOX(4, Func_Num: MSB, LSB )
ENDPROC

PROC Draw_Arc [Func_Num, Min_X, Min_Y, Max_X,\
                Max_Y, Start_Angle, Angle_Length]
   LSB = ADDR (Buffer ( ) {Address of buffer}
   MSB = PEEK (202) {Bank of buffer}
   TOOLBOX (4, 74: MSB, LSB, Min_X, Min_Y, Max_X, Max_Y)
   TOOLBOX (4, Func_Num: MSB, LSB, Start_Angle, Angle_Length)
ENDPROC

{Example program begins here}
HGR {Start 320 Graphics Mode}
HCOLOR = 15 {White}
GOSUB Draw_Rect [88, 5, 5, 250, 190]
DELAY = 1000
TOOLBOX ($04, $15: $0) {Clear SHR screen for 2nd drawing}
HCOLOR = 2 {Brown}
GOSUB Draw_Arc [98, 5, 5, 250, 190, 50, 180]
DELAY = 1000
END
```

Joystick and Paddle Controls

Since joystick or paddle controls are very often used with graphics, this topic is best covered in this section.

PDL (Paddle_Number)

A paddle is a game controller having only one axis, either X or Y. A joystick combines two paddles to control both axis simultaneously.

PDL returns a value (either integer or real) which is generated by the paddle number specified. Paddle_Number must be a value from 0 to 3 inclusive, otherwise a run time error will occur.

A paddle uses a paddle number from 0 to 3. A joystick uses paddle numbers 0 and 1 or 2 and 3. The value returned varies from 0 to 255.

To do consecutive readings on the hand control(s), use real variables with **PDL** to ensure accurate readings. This delay can insure a more accurate reading.

Paddle and Joystick Buttons

The button(s) on a game controller may be read by **PEEK**ing the appropriate locations. Buttons on game controllers return a value > 127 if the button is pushed, and < 127 if the button is not pushed.

PEEK (49249) for game controller 0 (this is also the Open Apple key). **PEEK** (49250) for game controller 1 (also the Closed Apple key). **PEEK** (49251) for game controller 2.

Example:

```
PROGRAM Joystick_Check
HOME
REPEAT
   VTAB (1,: HTAB (1)
   PRINT "X = "; PDL (0.0),
   PRINT "Y = "; PDL (1),
UNTIL PEEK (49249) > 127 OR PEEK (49250) > 127
IF PEEK(49249) > 127 THEN BEGIN
   PRINT "Button 1 was pressed"
ELSE BEGIN
   PRINT "Button 2 was pressed"
ENDIF
```

This program will execute until one of the buttons is pushed.

Chapter Eleven

The Sound of Music

Overview

The Apple IIGS has perhaps the best sound capabilities of any microcomputer on the market today and *Micol Advanced BASIC* allows you to create some very delightful sounds. Music may also be created quite easily by using just a few simple commands. This is the topic of this chapter.

Audio Output

BELL

Use **BELL** to provide an aural feedback to the user when the program is being used improperly or as a warning to a possibly dangerous situation.

BELL will produce a beep sound through the speaker of your Apple. You may control the volume of the beep through the Control Panel of your computer.

Example:

```
BELL: BELL {Ring bell twice}
```

Sound

Each sound comes from a specific sound wave. For example, the sound wave of a saxophone is different from the sound wave of a piano.

A sound is made by the compression and expansion of the air around us. This invisible movement is the vibration of the air. When the vibrations reach our eardrums, our eardrums vibrate which become impulses our brain interprets as sound.

The number of times a vibration occurs per unit time is called a frequency. The note Middle C played on a piano, for example, generates a sound with a frequency of 246 Hz.

A sound is represented graphically by a waveform. A wave starts with increasing positive values up to a limit specified and then decreases to negative values with the same limit, increases again up to positive values and so on.

The vertical axis represents the amplitude or the loudness of the sound. The horizontal axis represents the time. A waveform has a minimum and maximum frequency and amplitude. One up and down wave motion is called a cycle. A note is measured by the number of cycles per second.

The sound produced with *Micol Advanced BASIC* is digitized; that is, numbers are fed to the computer and the computer then uses those numbers to make sounds.

Waveforms

The Apple IIGS generates sounds using three kinds of waves: square waves, triangle waves, and sine waves. Square waves create buzzing sounds, triangle waves produce another type of buzzing effect. A sine wave makes a pure, clear sound. *Micol Advanced BASIC* uses a sine wave as the default waveform.

The Default Waveform

The default waveform is established when *Micol Advanced BASIC* is booted. This waveform is used by the sound making commands of *Micol Advanced BASIC* and is a full wave made of 255 parts numbered from 1 to 255.

The default waveform is based on a sine function times 255 from zero to 360 degrees. This default waveform produces a pleasant sound and will be suitable for most purposes.

Creating your own Waveform

If you decide that the default waveform is not suitable for your purposes, then it may be replaced with another waveform using the **WAVE** command.

WAVE = Wave_Numbers

WAVE places a new waveform into the waveform buffer. This buffer is used by the system to store the waveform needed for the **NOISE** and **MUSIC** commands described later.

The Waveform buffer is 256 bytes long and each value in the waveform may range from 1 to 255. A value of zero will be changed to one because a value of zero turns off the internal sound generators.

WAVE must be placed within a loop in order to set the entire Wave buffer. Initialize the Waveform Counter by assigning location 42 to zero, then keep assigning values to **WAVE** until location 42 is zero again; i.e. the Wave buffer is full.

WAVE may be used at any time within the program, but the new waveform will not be used until the next **NOISE** or **MUSIC** command is executed.

Example:

```
PROGRAM Make_Waveform
INT (A - Z)
Number = 0
Flag! = TRUE
POKE 42,0 {Init buffer counter}
REPEAT {Make the wave form table}
   IF Flag! THEN BEGIN
     Number = Number + 4
     IF Number > 250 THEN Flag! = FALSE
```

```
    ELSE BEGIN
      Number = Number - 4
      IF Number < 5 THEN Flag! = TRUE
    ENDIF
    WAVE = Number
  UNTIL PEEK (42) = 0 {Until buffer is full}
```

Be careful when using **WAVE**, as in the following example:

```
WAVE = SIN (Radians%) * 100
```

will almost always place a zero into the waveform buffer as the integer sine is almost always zero. Here is an example on how to work around this:

```
WAVE = 100.0 * SIN (Radians%)
```

The first thing seen by the Compiler after the equal sign is a real number, hence real arithmetic will be performed.

Experiment with different values to find the sound you are looking for. Unfortunately, some values are unsuitable for the system causing it to crash, so do not be surprised if the system should happen to do just that. Try different functions such as cosine multiplied by a factor to get an integer value between 0 and 255.

Making the Sound

NOISE (Generator, Pitch, Volume)

NOISE is used to generate a simple sound. It should be suitable for most sound effects.

Generator is an integer value from 1 to 15 and is the actual generator number used internally by the computer.

Pitch and Volume are integer values that may vary from 1 to 255. Each parameter may be either an integer literal or an integer variable. A real parameter will be rejected by the Compiler.

The pitch is the starting frequency used. The volume is the loudness of that sound.

If a **NOISE** statement with an active generator is given new pitch and volume values, that generator will be turned off and restarted using the new pitch and volume.

NOISE may be stopped with the **QUIET** or **SILENCE** command described later.

Example:

```
Generator% = 1
Pitch% = 80
Volume% = 100
NOISE (Generator%, Pitch%, Volume%)
```

Music

Micol Advanced BASIC not only allows you to make simple sounds, as just described, you can also create rather delightful music.

As you might guess, music requires more knowledge and preparation than just making simple sounds.

Instruments

All music is performed by musical instruments. These musical instruments may be violins, flutes, or even human voices. Each instrument has its own unique sound quality.

In order to make music under *Micol Advanced BASIC*, a musical instrument must be defined. This instrument may be almost anything you would see in a band or orchestra.

Default Instrument

A default instrument has already been defined into *Micol Advanced BASIC* and should be adequate for most purposes. You will have to experiment to determine if this instrument is suitable for you.

Creating Other Instruments

If the default instrument isn't suitable for your purposes, you may create an instrument of your own. *Micol Advanced BASIC* is capable of storing the envelope definition of any instrument using the **INSTRUM** command.

The instrument buffer is 44 bytes long. This means that 44 values must be read before the new instrument may be used. The proper values for an instrument range from 0 to 127.

Unfortunately, it is not a simple task to understand the data that needs to go into the Instrument Buffer. Study Table 3.11.1 to understand the values that are needed.

INSTRUM = Aexpr

The **INSTRUM** command is used to place a new instrument into the Instrument Buffer with the different values needed to duplicate the sound produced by any musical instrument. This Instrument Buffer is used by the system whenever you execute the **MUSIC** command

WARNING

> Be certain of the values you are assigning **INSTRUM**. **INSTRUM** does no error checking, and if any value is inappropriate, the system may crash when the next **MUSIC** command is executed.

To fill the Instrument Buffer, set memory location 42 to zero. Then, within a loop, keep assigning **INSTRUM** a value until location 42 is zero again.

(See the example after Table 3.11.1)

Table 3.11.1. Instrument Data Structure

Envelope

An envelope represents what the sound would look like if it were drawn on paper. It may have up to eight segments numbered 0 to 7, each having a breakpoint and an increment value pair.

The breakpoint value is a byte with a value from 0 to 127. It specifies the volume level. The volume should not be set to zero before the end of the segment or the sound is considered done. The last breakpoint is always zero. A difference of 16 in the breakpoint value represents a change of 6 decibels (db) in amplitude.

The increment value is a measure of time indicating the time needed to get to the breakpoint volume that uses two bytes. Both bytes range from 0 to 127. The value of 1 in the low-byte represents a 1/256th in value. An increment value of 0 is equivalent to a sustained note. The note will play until no generator can play it and the original generator producing the note is allocated another note.

Release Segment

This integer number from 0 to 7 indicates at which breakpoint the note will start to fade away. The release may occupy several segments, but the last breakpoint is always zero.

Priority Increment

This value from 0 to 127 indicates at which moment a generator will drop the oldest note it is currently playing before playing the new one.

Pitchbend Range

This number indicates the number of semitones a note may be raised. The accepted values are 1, 2 and 4 semitones.

Vibrato Depth

This number from 0 to 127 indicates the vibrato effect for a note. No vibrato effect occurs with a value of zero. The vibrato should always be set to zero.

Vibrato Speed

This number from 0 to 127 controls the rate of vibrato. Higher values produce faster vibrato. Set it to zero when vibrato Depth is zero.

Spare

This byte is not documented and it may be reserved for future use. Set it to zero.

aWaveCount and bWaveCount

A *Micol Advanced BASIC* instrument has only one wave list per oscillator. Wave A and Wave B should always be set to one.

The default wave count could change in a future version of the language.

aWaveList and bWaveList

A wavelist is an array structure of 6 bytes. Since a *Micol Advanced BASIC* waveform has only one wave in each list, set the topKey value to 127.

WaveList Array Structure

- **topkey**
  The Note Synthesizer always plays the note with a topKey value equal or higher than the preceding one. The waveform should be stored in increasing topKey value. The value ranges from 0 to 127. The last waveform should have a topKey value of 127.
- **Wave Address**
  This is the high byte of the waves address in the Digital Oscillator Chip (DOC) RAM.
- **Wave Size**
  Wave Size is set to 256 bytes. Set the number to zero.
- **DOC Mode**
  This number represents the size of the sound wave measured in bytes. One wave is used per instruments defined with *Micol Advanced BASIC*. The mode of the DOC chip should always be set to 0. Do not modify.
- **relPitch**
  This value is used to tune the waveform. The high byte value represents whole semitones, the low byte represents fractions of semitones. A value of 1 in the low byte equals 1/256th of a semitone.

Example

```
PROGRAM New_Instrum
INT (A-Z)
{Instrument Definition}
{Envelope}
{Noise}
DATA 127          {Breakpoint 0}
DATA 0,127        {Increment value 0}
DATA 120          {Breakpoint 1}
DATA 20,1         {Increment value 1}
DATA 120          {Sustain at 120}
DATA 0,0          {Zero increment is sustain}
{Segment}
DATA 0            {Release to 0 volume}
DATA 60,120       {Slowly}
DATA 0,0,0        {Pad with extra breakpoint}
DATA 0,0,0        {Increment pairs until the}
DATA 0,0,0        {total is eight}
DATA 0,0,0        {End of envelope definition}
DATA 3            {Release starts at 3rd segment}
DATA 32           {Priority increment}
{Pbrange, vibdep, vibf, spare, A, B}
DATA 2,80,90, 0,1,1
{topkey, addr, size, ctrl, pitch}
DATA 127,7,2,6,0,12 {Halt b, to be swapped in by a}
DATA 127,7,2,1,0,12
{End of instrument definition}
POKE 42,0         {Init Instrument Buffer}
{Initialize the loop}
REPEAT
   READ Number
   INSTRUM = Number {Fill one buffer entry}
UNTIL PEEK (42) = 0 {zero when done}
{Program continues}
```

NOTE

> The **INSTRUM** command is an integer command. This means, that unless the value directly after the equal sign is explicitly a real variable or a real number, integer math will be used.

Making the Music

Now we come to the section where you actually generate the music. Once suitable waveforms and instruments have been defined by you, the actual music is generated by the **MUSIC** command.

MUSIC (Generator, Pitch, Volume)

MUSIC is used to generate musical sounds. By a proper use of the **WAVE** and **INSTRUM** commands, virtually any instrument may be simulated. In addition, because there are 15 generators available, you may have several instruments going at once.

Generator is a number from 1 to 14. This number is a relative generator number established by the system, and not the actual generator as in the **NOISE** command.

Pitch and Volume may vary from 1 to 127. Each parameter may be either an integer literal or an integer variable.

The pitch is the starting frequency based upon the values you placed into the wave table. The volume is, of course, how loud the music will be made.

If you do not wish the default waveform with the **MUSIC** command, then be certain to set the new waveform buffer using the **WAVE** command.

If another **MUSIC** command is issued using the same generator, the new sound will replace the old one.

NOTE

> The **NOISE** command cannot be used simultaneously with **MUSIC**. If **MUSIC** is active, **NOISE** will be ignored, and if **NOISE** is active, **MUSIC** will be ignored.

Example:

```
MUSIC (1, 40, 80)
```

This example uses logical generator 1 with a pitch of 40 and a volume of 80.

IMPORTANT

> The **MUSIC** command requires the use of an Apple IIGS Tool located on the system disk. Be aware that when the **MUSIC** command is first executed, the booting disk must be online so that the approriate Tool may be loaded. If this disk is not online, you will receive a request to insert it. The *Micol Advanced BASIC* system disk marked Master Disk comes with this Tool installed.

Stopping Sounds

It is not enough to simply create sound or music, you must also be able to turn these sounds off. Very few programs would be suitable with sound running all the time.

QUIET (Generator)

QUIET is used to turn off the specified generator and may be used to create pauses in noise or musical sequences. Generator is the generator that was used when the **NOISE** or **MUSIC** command was executed.

Example:

```
MUSIC (1, 40, 80) {Start generator one}
DELAY = 1000
QUIET (1) {Silence generator one}
```

Turn Them All Off

SILENCE

The **SILENCE** command turns off all sound generators currently playing. This command has no parameters.

The **END** and **STOP** commands also produce the same effect as **SILENCE**.

Example:

```
MUSIC (1, 40, 80) {Start generator one}
DELAY = 1000
SILENCE {Shutdown generators and tools}
```

Chapter Twelve

Creating The Human Element

Overview

Unlike computers, human beings are not regulated by On and Off. What makes humans special is the ability to see the different shades of gray, to make a decision based on related information or on a hunch. Programming languages try to imitate this randomness using pseudo-random numbers. *Micol Advanced BASIC* takes this one step further by introducing Controlled Uncertainty™.

Pseudo Random Numbers

Pseudo random numbers are not really random, but only appear to be. The only random number in the sequence is the first, or the seed as it is called. After that, the generator goes through a complex set of calculations to get what appears to be a random result.

Micol Advanced BASIC has two pseudo random number generators: one for integers, one for reals, both activated by the **RND** function.

Be cautious with the use of **RND**. It is easy to call the real pseudo random number generator by mistake when you want to use the integer generator or vice versa. Be careful not to call the wrong one since they behave differently.

Integer Pseudo Random Numbers

Integer% = RND (Aexpr)

The integer pseudo random number generator is invoked when the assignment is made to an integer variable. The integer **RND** function yields a pseudo random number between 0 and Aexpr inclusive. Thirty-two thousand (32,000) is the largest argument that may be passed to **RND**.

If an **INKEY$**, **INPUT** or **GET** is executed within a program, the integer random number generator will be reseeded. This reseeded value is an actual random number.

To use the integer random generator, do something like this:

```
FOR Ctr% = 1 TO 6
    Dice% = RND (5) + 1 {Random values between 1 and 6}
    PRINT "Throw # "; Ctr%; " of the dice is a "; Dice%
NEXT Ctr%
```

Real Pseudo Random Numbers

Real& = RND (Aexpr)

The real **RND** function yields a floating point pseudo random number between zero and one inclusive. The argument is ignored but must be included, otherwise an error will occur during compilation.

To use the real random generator, do something like this:

```
FOR Ctr% = 1 TO 100
    Real_Random& = INT (RND (1) * 100)
    PRINT "Pass # "; Ctr%;" is "; Real_Random&
NEXT Ctr%
```

Controlled Uncertainty™

Programming languages usually deal in absolutes of logic. Something is either true or false, and actions are always taken depending on this condition.

Micol Advanced BASIC goes one step further and gives the programmer the possibility to set conditions that may or may not take a certain action based on this condition. We feel this is a feature that has many possibilities if intelligently used.

We call this feature Controlled Uncertainty. It is uncertain because there is the possibility an alternate decision will be made. It is controlled because the decision is being made within one of the structured constructs of *Micol Advanced BASIC*.

When could Controlled Uncertainty be useful? Anytime you wish to program human uncertainty within a program. Many things in life are based on assumptions, not facts. Any condition that is not absolutely true or false may use this feature.

Setting the Uncertain Condition

Controlled Uncertainty may be set using certain settings of boolean variables. Usually a boolean variable is set to **TRUE** or **FALSE**. Under *Micol Advanced BASIC*, a boolean variable may also be set to **DUNNO**, **DOUBT** or **BELIEVE**. **BELIEVE** is used if the condition is probably true, but there is a chance it is false. **DOUBT** is used if the condition is probably untrue, but there is a possibility it is true. There also exists **DUNNO**. **DUNNO** is the logical equivalent to a random number generator and will randomly select one of the other four possibilities.

If a boolean variable is set to **BELIEVE** and then tested, there is about an eighty percent chance the condition will be TRUE, about twenty percent chance the condition will be FALSE. If a boolean variable is set to **DOUBT** and then tested, there is about a twenty percent chance the condition will be TRUE, and about eighty percent chance the condition will be FALSE.

In addition, booleans set to an uncertain condition may be **AND**ed or **OR**ed with other booleans which will often make one of the other alternatives. We have collected all

the possibilities into an uncertainty table which we display here.

Table 3.12.1. Uncertainty Table

AND

	False	Doubt	Believe	True
False	False	False	False	False
Doubt	False	Doubt	Doubt	Doubt
Believe	False	Doubt	Believe	Believe
True	False	Doubt	Believe	True

OR

	False	Doubt	Believe	True
False	False	Doubt	Believe	True
Doubt	Doubt	Doubt	Believe	True
Believe	Believe	Believe	Believe	True
True	True	True	True	True

Example:

```
PROGRAM Human_Computer
HOME
PRINT "Hello, I'm your Apple computer, ";
PRINT "I've been turned off for a while."
PRINT "I do remember the time and the date, ";
PRINT "but not your name."
INPUT "What is it again? "; Name$
Mood! = BELIEVE
IF Mood! THEN BEGIN
   PRINT "Im feeling well today, and ";
   Health! = DOUBT
   IF Health! THEN BEGIN
     PRINT "hope you're feeling fine too."
   ELSE BEGIN
     PRINT "certainly hope you're not feeling poorly."
   ENDIF
ELSE BEGIN
   PRINT "I'm sorry, I'm not well today, can't talk anymore."
   Polite! = DUNNO
```

```
    IF Polite! THEN BEGIN
      PRINT "Have a nice day "; Name$
    ELSE BEGIN
      PRINT "Get lost "; Name$; " and don't call again!!"
    ENDIF
  ENDIF
  END
```

WARNING

> The statements **IF** Flag! **THEN** and **IF** Flag! = **TRUE THEN** do not have the same effect when Controlled Uncertainty values such as **DOUBT** or **BELIEVE** are used. If the variable Flag! is assigned to **DOUBT** and Flag! is tested as **IF** Flag! = **TRUE THEN**, the variable Flag! will never be true, while if the variable Flag! is tested as **IF** Flag! **THEN**, the variable Flag! will be true about 20 percent of the time.

NOTE

> The condition at which a boolean variable is currently set may be determined by using the **PRINT** <Boolean!> statement to print the boolean value of **FALSE**, **DOUBT**, **BELIEVE** or **TRUE**.

Chapter Thirteen

Direct Memory Access

Overview

This chapter discusses how to look at and change the contents of specific memory locations, and manage blocks of memory within a *Micol Advanced BASIC* program.

Examining and Changing Memory

PEEK (Aexpr)

To see the value of a particular memory location, use the **PEEK** command where Aexpr is the address to be referenced.

Your computer has memory addresses at least in the hundreds of thousands, probably over a million. Unfortunately, if you are using (default) short integers, the maximum value an integer can have is 65535. This means that integer **PEEK**s may only be used within bank zero, which usually is to locations in direct page. If you wish to access memory locations in higher memory locations, be certain to assign **PEEK** to a real variable.

NOTE

> The Direct Page area used by the run time Library (not by the computer firmware) will be accessed if the value passed to **PEEK** is less than 256. Zero page used by Applesoft BASIC and Direct Page are not the same.

Example:

```
Integer% = PEEK (Location%) {Can only access bank zero}
Real& = PEEK (Location&) {Can access any memory}
PRINT PEEK (Location&) {Can access any memory}
```

POKE Aexpr1, Aexpr2

POKE may be used to change the contents of the memory location specified. Aexpr1 is the address in memory. Aexpr2 is the value to be stored in the memory location and cannot be greater than 255, otherwise, an error will occur at run time.

If a negative integer address is used, **POKE** will convert the address into a two's complement address.

If the address passed to **POKE** is less than 256, the direct page area used by the run time Library (not the zero page area used by the computer) will be accessed.

IMPORTANT

> **POKE** cannot change memory locations 224 to 255 that are reserved in the Direct Page for system usage. If a **POKE** is made to any of these addresses, an error will occur during program execution.

Example:

```
POKE Location%, Number% {Addresses in bank zero only}
POKE Location&, Number {Can access almost the entire memory}
```

Finding the Address of a Variable or Array

ADDR (Variable [(])

The **ADDR** (Variable) command returns the address of any variable. If the variable is an array, the left parenthesis must be included to inform the Compiler that an array is being referenced.

The address returned is the actual address obtained during execution of the program, NOT the relative address displayed by the Symbol Table Dump at the end of compilation (if the **LIST** or **PRINTER** option is used).

NOTE

> **ADDR**, when assigned to an integer variable, will only return addresses between ±32767 ($0000-$FFFF) unless the **LONGINT** compiler option is used. If you are assigning the result of **ADDR** to a short integer variable, you may fetch the bank number of the address of the variable by **PEEK**ing **True_Value** (location 202) of the direct page immediately after executing **ADDR**. The actual address in the bank, if greater than +32767, will be represented as a negative number. Add 65535 to a real variable get the positive value.

If you are using real values with **ADDR**, you will get the full address of the specified variable.

ADDR is often used to find the address of a buffer used by a particular Tool. (See Allocating Toolbox buffers in Part Five.)

Example

```
This_Address& = ADDR (Variable)
This_Address% = ADDR (Variable)
Bank_Number = PEEK (202)
Array_Address& = ADDR (Array ( )
```

Memory Images and Files

Sometimes it is necessary, within a program, to be able to bring information from a disk file directly into memory. Also, the opposite may be true, memory locations must be saved to disk to be used sometime later, perhaps even by another program.

Micol Advanced BASIC has two commands to accomplish these tasks. You must however be very careful, as there is no protection, any part of memory may be accessed.

BLOAD Svar, Start_Address, Bytes_to_Load

BLOAD stands for **B**inary **LOAD**. Use **BLOAD** to bring binary data (a non-compressed picture or sound information file) into memory. Because there is no checking on the file type, any uncompressed file may be loaded with **BLOAD**.

Svar is the Pathname of the file. Svar may be either a string variable or a string literal. Start_Address is the address of the first memory location to which the file will be loaded. Bytes_to_Load represents the size of the file in bytes. Start_Address and Bytes_To_Load may be either a variable or literal of type integer or real.

All parameters must be present to be accepted by the Compiler. The disk which contains the file must be online upon execution of the statement, otherwise a run time error will be generated.

BLOAD will load the file in the specified memory area. If Start_Address is zero, the file will be loaded to the address specified by the file information on disk, otherwise the file will be loaded to the address specified. If Bytes_to_Load is zero, the entire file will be loaded, otherwise only the specified number of bytes will be loaded.

Example: (using an uncompressed picture file):

```
BSAVE "PICTURE", 14753792, 32768
BLOAD "PICTURE", $E12000, $8000
```

32,768 bytes of memory holding a Super High Resolution picture will be saved and then loaded with these commands. Note that the addresses in both lines are identical This example saves then loads the entire Super High Resolution screen (14753792-14786560 \ $E12000-$E19FFF) without any decompression.

If your paint program can save the picture in binary format, you will be able to load this picture into memory with *Micol Advanced BASIC* using the following code:

```
{Change to HGR2 for pictures drawn in 640 x 200 mode}
```

```
PROGRAM Load_Picture_320
HOME
INPUT "Enter full pathname of picture "; Picture$
HGR
BLOAD Picture$, $E12000, $8000
GET A$
```

Picture files will not load correctly if they were saved in a compressed format.

BSAVE Svar, Start_Address, Bytes_to_Save

BSAVE stands for **B**inary **SAVE**. Use **BSAVE** to save any information from memory into a binary file on disk. The file will be saved as type BIN ($06).

Svar is the Pathname of the file. Svar may be either a string variable or a string literal. Start_Address is the address of the memory location whose memory image will be saved. Bytes_to_Save represents the size of the file in bytes. Start_Address and Bytes_To_Save may be either of type integer or real in a variable or literal.

All parameters must be present to be accepted by the Compiler. **BSAVE** will save the Bytes_to_Save number of bytes from Start_Address.

Example: (See **BLOAD**)

Memory Management

Micol Advanced BASIC has commands that allow you to allocate and deallocate memory as your program requires. The memory may contain anything. Some memory may contain a graphics picture, data for the program, etc.

The User ID Number

All memory within your Apple IIGS is managed by a Tool called the Memory Manager. When *Micol Advanced BASIC* is started up, the Memory Manager assigns an identification number to be used for all calls to it. This application ID number is active until you quit *Micol Advanced BASIC*.

This ID number is stored in memory locations 232 and 233 in the direct page and may be retrieved at any time using **PEEK**.

GET_MEM (Handle&, Location&)

This statement is used to request a block of memory from the Memory Manager. The memory may be used to allocate a Direct Page area or to allocate other memory for just about any purpose. Once allocated, you have exclusive use of this memory.

Handle& is a 4-byte memory location which will contain information necessary to free the memory when you are finished with it. Location& is the actual address of the allocated memory block. Both parameters must be real variables.

GET_MEM allocates a block of memory that is in a fixed memory bank with a fixed, page-aligned address that does not use Special memory (graphics memory, etc). The block is fixed, and locked.

Before you can call **GET_MEM**, there are certain values you must establish:

1. Memory location 202 (**True_Value**) indicates to **GET_MEM** where the memory will be allocated. A value of zero tells **GET_MEM** to get the block from anywhere in memory. A non-zero value indicate that the block will be allocated from a specific bank.
2. The variable Location& must indicate from which bank number the memory should be allocated.
3. The value assigned to variable Handle& must indicate the amount of memory needed in bytes.

After the call to **GET_MEM** is finished, the variable Handle& will contain the handle of the memory block allocated by the Memory Manager. The actual address of the block will be in Location&.

If the memory is allocated successfully, **True_Value** (location 202) will contain a zero, otherwise **True_Value** will contain the error number returned by the Memory Manager.

Handle& is required by **FREEMEM** to deallocate the memory block.

Example:

```
POKE 202,1 {Get memory from a specific bank}
Location& = 0 {Get memory from bank zero}
Handle& = 256 {Get 256 bytes (one page)}
GET_MEM (Handle&, Location&)
IF PEEK (202) <> 0 THEN BEGIN
   PRINT "Error in memory allocation"
ELSE BEGIN
   PRINT "The address of the block is: ";Location&
ENDIF
```

WARNING

> If a memory block is allocated directly by using the **TOOLBOX** command instead of **GET_MEM**, the block of memory must be deallocated using the proper Memory Manager call.

FREEMEM (Handle&)

To deallocate a block of memory, use the **FREEMEM** command. The argument is the handle of the block obtained with **GET_MEM**. Handle& must be a real variable.

The memory may also be deallocated by letting the program finish. All memory will be released automatically.

Example:

```
FREEMEM (Handle&)
```

MOV_MEM Start_Addr, Num_of_Bytes AT Dest

To move memory from one location to another, use the **MOV_MEM** command. The arguments may be either of type real or integer.

Start_Addr is the address of the first byte that needs to be moved. Num_of_Bytes is the total number of bytes to be moved, and Dest is the address to where these bytes need to be moved. The maximum number of bytes that may be moved at one time is 65535 bytes (one bank). The locations may not overlap, or the memory may copy over itself.

One practical use of **MOV_MEM** is to save a part of the current text screen display, for example. **MOV_MEM** may be used to move 1024 bytes starting at 1024 to a safe location, and 1024 bytes starting at 66560 to another safe location. When returning the screen to its original display, move the memory back to the screen.

Example:

```
DIM Array% (1026) {Allocate a buffer}
Array_Addr = ADDR (Array%( ) + 3
Screen_Bank0 = $400
Screen_Bank1 = $10400
MOV_MEM Screen_Bank0, 1024 AT Array_Addr
MOV_MEM Screen_Bank1, 1024 AT Array_Addr + 1025
HOME
{To restore screen just saved}
MOV_MEM Array_Addr, 1024 AT Screen_Bank0
MOV_MEM Array_Addr + 1025, 1024 AT Screen_Bank1
```

Chapter Fourteen

Run Time Error Handling

Overview

Error handling, or error trapping as it is also called, is the art of dealing with unexpected situations. These situations may be, for example, bad user input, an empty disk drive, improper data, or even an intentional user response which causes an error condition, such as pressing <Control>C.

When an error occurs, control is usually passed to an error handling routine. An error handling routine, for example, may allow the user to recover from the error by giving precise instructions on how to correct the situation. After the error has been corrected, the program usually resumes execution at a suitable point.

IMPORTANT

> Do not confuse error trapping with debugging. Error handling is a normal operation of almost every properly functioning program and is simply dealing with unexpected situations. Never use any of the commands described in this chapter until your program is properly debugged (unless, of course, you are debugging the error trap itself).

Handling the Error

During the program development phase, whenever an error condition arose, a message was displayed on the screen describing the error and the line where the error occurred. You more than likely went to the Text Editor to fix the problem. This situation was carefully devised to help you debug your program.

Now, you have gone beyond this phase so that your program operates as it should, or at least as close as possible. Unfortunately, unforseen conditions may arise during the execution of the program and the system sending a message to the screen isn't adequate anymore.

Now, the program error must be dealt with internally, and usually the program must continue on with its work. That is, the error must be handled.

The *Micol Advanced BASIC* commands described in this section are all you should need to take care of these unexpected situations. However, this is a topic where creativity is required, so actually designing what happens in your error handling routine is largely up to you.

ONERR GOTO Module_Id

If an error occurs during program execution, **ONERR GOTO** deactivates the normal debugging capability of *Micol Advanced BASIC* and transfers control to an error handling routine. **ONERR GOTO** also passes information to the program to help determine what the problem is and where it happened.

When an error occurs during the execution of a program, the error number is placed into one of two memory locations (location 154 or 155).

Location 154 holds the error number returned by the run time routine. Location 155 holds the error number returned by the operating system. Under no circumstance can both error conditions arise at the same time. The list of the error codes from the run time routines is in Appendix C. The list of the error codes from the operating system is in Appendix D.

Place the **ONERR GOTO** at a location prior to where you believe the error is likely to happen; in practice, this is often at the beginning of the program.

To deactivate an **ONERR GOTO**, place zeroes into the direct page locations 191 and 192 using a **POKE**. This will enable the normal debugging capability of *Micol Advanced BASIC*.

It is often very useful to know on which sequential line number the error happened. The sequential line number where the error occurred is stored as a binary value in locations 204 and 205 in LSB, MSB order. The following program line will determine at which sequential line the error occurred:

Line_Error& = PEEK (204) + 256 * PEEK (205)

It may be desirable to place the error handling routine as the last portion of code before the final **END** statement. This will help avoid confusion with the normal program code.

To avoid an infinite error loop, you may want to deactivate the **ONERR GOTO** if execution errors should occur within the error handling routine. Don't forget to reactivate the **ONERR GOTO** by placing another **ONERR GOTO** as the last line of the error handling routine, if necessary.

Example:

```
PROGRAM Error_Example
ONERR GOTO Error_Trap
{<Program code>}
END
ROUTINE Error_Trap
POKE 191,0
POKE 192,0 {Turn off future ONERR GOTOs}
IF PEEK (154) > 0 THEN BEGIN
   PRINT "Language error # ";PEEK (154);
ELSE BEGIN
   PRINT "GS/OS error # ";PEEK (155);
```

```
ENDIF
PRINT " in line "; PEEK (204) + 256 * PEEK (205)
END
```

RESUME

RESUME instructs the program to continue execution at the same line or structured statement in which the error was encountered.

RESUME restores the previous **FOR** loop stack pointer as well as the stack pointer used for Procedures, Functions and Routines. If you intend to use a **RESUME**, then the error handling routine should contain neither **FOR** loops nor calls to subroutines (**GOSUB**s) as the values on the stack(s) may become corrupted.

If **RESUME** is used in a program, the **ERROR** compiler option must be specified in the program. If **ERROR** is not specified, an error will occur at run time when **RESUME** is encountered.

Example:

```
PROGRAM Example
@ ERROR {Required for RESUME}
ONERR GOTO Error_Trap
HOME
Divisor = 0
Dividend = 100
Quotient = Dividend / Divisor
PRINT "Quotient is: ";Quotient
END {END needed to terminate program before error trap}
ROUTINE Error_Trap
HOME
PRINT "In Error Trap"
Divisor = 10 {Stop another division by zero error}
PRINT "Press Return to resume program"
GET Wait$
RESUME {Will execute the error line again}
```

Part Four: Creating the Apple IIGS Desktop

Chapter One

Desktop Programming

Overview

This chapter explains the Desktop metaphor created by Apple and shows what is needed in a Desktop program written under *Micol Advanced BASIC*.

The Desktop Environment

What is the Apple IIGS Desktop? The Desktop is a metaphor used by Apple to help individuals use computers without having to learn hard-to-remember and often difficult-to-use commands. This metaphor uses objects used in everyday life to conceptualize computer operations.

It is not necessary to remember commands when a Desktop program is used; the operations appear on the screen in a manner the user is already familiar with. The user only has to make a selection to perform the action. If you wish to learn more about the Desktop metaphor, get a copy of the Human Interface Guidelines from Apple Computer, Inc..

Desktop programming is somewhat difficult. It requires a lot of planning and attention to details. A Desktop application does a lot of little things in the background that take little time to write into code.

The Desktop commands used in *Micol Advanced BASIC* will control the vast majority of the functions needed by a Desktop program written by the average *Micol Advanced BASIC* programmer.

Essentially, there are three types of displays on the Apple IIGS Desktop: Menus, Windows and Dialog Boxes. We will explain what each of these is in detail in its respective chapter. For now, it is sufficient to know that they exist.

In general, information is passed to the appropriate Desktop command using two arrays: an integer array and a string array. The arrays must be large enough to hold the elements of the largest Dialog Box, Menu, or Window. The arrays are used to define the Dialog Boxes, Menus, and Windows but may be used for other purposes if memory is short. In addition, it is not necessary to have a different set of integer and string arrays for each Menu, Dialog Box or Window; they may be reused from call to call.

The Desktop uses the Super High Resolution graphics screen, either in 320 mode or 640 mode. Before any of the Desktop commands may be used, an **HGR** or **HGR2** command must have been previously given to start the proper graphics mode. A **TEXT** command is used if you wish to erase the Desktop and return to normal text input.

Define the Menus, Dialog Boxes and Windows by filling in the appropriate arrays

and use the appropriate Desktop command to display (and activate) the Menus, Dialog Boxes and Windows.

The values returned by the movements of the Mouse and clicking in the Menu bar, Dialog Boxes, and Windows are caught in a loop, called the Event loop, that handles all the commands and the choices the program's user makes while using the program. The actions performed by the user are often handled outside the loop in a sub-section of the program. A **CASE_OF** statement does this job nicely.

Desktop Commands

Micol Advanced BASIC understands four commands to let you write applications that use the desktop. The **MENU** command controls the Menu bar. The **WINDOW** command directs how Windows open and close. The **DIALOG** command manages all aspects of Dialog Boxes. The **MOUSE** command controls the actions performed by the program's user (the Event).

A Desktop program must have the **MOUSE** command and at least one of the other three Desktop commands (**DIALOG, MENU** or **WINDOW**) to function properly; otherwise the program will not be able to respond to the user.

Desktop capability under *Micol Advanced BASIC* is adequate for most of the applications written with *Micol Advanced BASIC*. The Toolbox may also be called directly using the **TOOLBOX** command if you require a more advanced Desktop.

Monitoring the Desktop

MOUSE (Integer_Array ( )

The three types of Desktop displays: Menus, Windows, and Dialog Boxes, are all monitored by the **MOUSE** command.

First, you have to create one of the Desktop displays using the **MENU**, **WINDOW** or **DIALOG** command. Once the display is as you wish, you must use **MOUSE** to allow the user to respond to the display.

The only parameter required by **MOUSE** is an integer array dimensioned to at least 20 elements. This array may have any value before **MOUSE** is executed, but will contain the value(s) needed to respond when control is returned to your program.

You may have to place **MOUSE** in a looping situation, and access the values described in subsequent chapters. If you are accessing a Dialog Box, **MOUSE** need not be in any looping structure, as no response is returned until the user has responded to the Dialog Box. However, both **MENU** and **WINDOW** require **MOUSE** to be contained within a loop with repeated checks for the necessary values returned by the particular command.

The individual aspects of the **MOUSE** command will be explained in more detail in the three chapters that follow.

Example:

```
PROGRAM Desktop_Demo
DIM EventRecord% (59)
DIM DeskTopArray$ (42)
{Dialog Boxes, Menus, and Windows are defined here}
{Program Start}
HGR {Set 320 x 200 mode for Desktop, required}
GOSUB MenuBar {Define Menu Structure}
GOSUB EventLoop {Handle the Users Actions}
END {Desktop_Demo}
{eof}
```

The example programs on the disk /MAB.SUPPORT, in the subdirectory Demo.Files/Desktop.Samples/ demonstrate the use of these commands.

Programmers.

A Desktop application written in *Micol Advanced BASIC* uses the following tool sets: QuickDraw II, Event Manager, Window Manager, Control Manager, Menu Manager, LineEdit, Dialog Manager and Scrap Manager. These tools will be loaded and started automatically when one of the Desktop commands is executed by the program. They will be shut down when the Desktop is erased from the screen or the program finishes execution.

Chapter Two

Menus

Overview

This chapter describes the commands needed to create and monitor Menus.

Menu Specifics

Pull-down Menus allow a user to make a single selection from a list of selections (a Menu List), among a set of lists (the Menu Bar), and perform a task based on this selection.

Menu Lists may be easily created, enabled (made selectable), disabled (made non-selectable), and removed. Each selection within a Menu List is called an Item. Items within a Menu List may be enabled, disabled, and removed just as easily.

A distinction must be made between the Menu Bar, Menu List, and Menu Item. A Menu Bar, the white rectangle that appears on the top of the Menu display, contains the Menu Lists. When a List in the Menu Bar is selected, a pull-down List of Items is displayed. The pull-down List is the entire collection of Menu Items for this Menu List. A Menu Item is a command.

Defining a Menu

The Menu Lists are defined in the reverse order in which they are displayed: the Menu List appearing on the right must be created first; the Menu appearing on the left must be defined last. This means, that the first defined Menu List (with the smallest element number) will be the right most Menu List displayed, and the last defined Menu List (with the highest element number) will be the left most appearing Menu List.

The Items in a pull-down List should be listed with the most often used Item at the top and the least often used one at the bottom.

Do not forget to define the Super High Resolution graphics screen with either an **HGR** (320 mode) or **HGR2** (640 mode) before creating your Menus.

Menu Definition Syntax

```
{ [ MenuArray$ (Subscript) ="]>> Menu List \ [Attr_Char ] N Menu_ID"]}"
{ [ MenuArray$ (Subscript) = "::Menu Item\ [ Attr_Char ] N Menu_ID"]}
{ [ MenuArray$ (Subscript) = "." ]}
{ [MenuArray$ (Subscript) = ""]}
```

The definition of a Menu Bar is assigned to a string array which will be passed to the

command which actually makes the Menu. The Menu Lists and Items are assigned to the individual elements of the string array. One string array element holds one Menu List or Item definition. The string array must be dimensioned to a little more than the total number of Items in the Menu definition.

Be sure to number the string array elements exactly. If the subscript number of the Menu array is repeated, the contents (Menu Item) of these elements will not be displayed.

A Menu List definition begins with any two title characters, which indicate the start of a Menu List, followed by the actual Menu List title between spaces. These List characters may be any visible characters, but the greater than symbol (>) is suitable, so we will use it exclusively in our examples. The Menu List Title is simply a text string which describes the list of Items to follow. Two spaces should appear before and after the Menu titles in 320 mode (**HGR**), and one space in 640 mode (**HGR2**); otherwise, the Menu List Titles will appear stuck one against the other when they are displayed.

A Menu Item definition begins with any two item characters (different from those used in the List, we will use a colon (:) in our examples) which indicates a Menu Item. Spaces before and after the Menu Items are not necessary: they will be done automatically. Following these two characters must appear the Menu Item Name, which is the text which will appear on the screen informing the user what the selection is.

Following the Menu List Title or Menu Item Name is a backslash character (\) which indicates that the Menu List ID or Menu Item ID follows, followed, perhaps, by special attribute definitions.

Menu Title and Item Identification Numbers

A unique number must be assigned to each Menu List and Menu Item; otherwise, the program will not be able to determine which List or Item was chosen by the user. Menu List and Item ID numbers should be listed in ascending order. The identification numbers must be allocated as shown in Table 4.2.1.

Table 4.2.1 ID Number Allocation Table

Menu List ID #	Description
0	Internal use to indicate first Menu List
1-255	Preferred ID numbers for Menu Lists
256-65534	May be used by user's application
65535	Internal use to indicate last Menu

If the Apple Menu is included, it must have an ID number of one (1).

Menu Item ID #	Description
0	Internal use to indicate first Menu Item
1-249	Used by Desk Accessories
250-254	Reserved for the Edit Menu Items
250	Undo
251	Cut
252	Copy
253	Paste
254	Clear
255	Reserved for Close command (in File Menu)
256-65534	Used by program's Menu Items
65535	Internal use to indicate last Menu Item.

The Menu List ID or Menu Item ID is defined by using one of the unique numbers above preceded by one of the following letters:

N - Number

The letter N is followed by a unsigned decimal (ASCII) number. This number defines the unique Menu List ID or Menu Item ID in decimal. This characteristic will probably appear in every Menu List or Menu Item definition.

H - Hexadecimal

This letter is used to specify the Menu List ID or Menu Item ID as a hexadecimal value. You will probably wish to use N instead of H.

Example:

```
Menu$ (1) = ">> Title \N3"          {Menu title defined}
Menu$ (2) = "::Item 1\N301"         {Menu item defined}
Menu$ (3) = "::Item 2\N302"
Menu$ (4) = "::Item 3\N303"
Menu$ (5) = "."                     {End of Menu list definition}
{Other Menus would be defined here}
Menu$ (99) = ""                     {End of Menu bar definition}
```

Menu Attribute Characters

These characters (-, *, B, C, D, I, O, S, U, V, X) may be included with the Menu List ID or Menu Item ID and are used to give one or more specific features to a Menu List or Item such as:

- Give a keyboard equivalent to a Menu Item
- Indicate a default setting to a Menu List or Item
- Separate Menu Items
- Give a specific style to a text Item
- Restore the colors of a Menu List or Item

The attribute characters may be entered in upper or lowercase, by convention, uppercase characters are used.

* - Keyboard Equivalent

The asterisk tells **MENU** to display an Apple logo and a character to the right of the Menu Item to indicate that a keyboard equivalent is available. Menu Lists may not have a keyboard equivalent.

Some keyboard equivalents should be used for specific Menu Items (see Table 4.2.2).

When using a letter as a key equivalent, be certain to define both an uppercase and lowercase character. When a special character (?, #, etc.) is used (especially where the Shift key must be pressed) as key equivalent, be sure to enter both characters in the definition; otherwise, the user may think that the key equivalent does not function properly.

The key equivalent is automatically trapped by the **MOUSE** command.

Example 1:

```
Menu$ (11) = "::New\*NnN259"
```

This definition allows you to use Apple-N (Apple-Shift-n) or Apple-n as key equivalents.

Example 2:

```
Menu$ (32) = "::Help...\V*?/N257"
```

This definition allows you to use Apple-? (Apple-Shift-/) or Apple-/ (Apple-/) as key equivalents.

Table 4.2.2 Reserved Keyboard Equivalents

Apple Menu		File		Edit	
Help	?	New	N	Undo	Z
		Open	O	Cut	X
		Close	W	Copy	C
		Save	S	Paste	V
		Quit	Q		

We strongly recommend you program some keyboard equivalents to offer an alternative to using the Mouse. Not everyone loves the Mouse.

Specifying Defaults

Attribute characters C and D are used to show a specific setting at the creation of the Menu. Attribute C is used with Menu Items only. Attribute D may be used on Menu Lists and Items.

D - Disable and Dim a Menu Title/Item

The letter D prevents the user from employing a Menu List or Item until a specific event occurs; the name of the Menu Title or Item appears in grey. In the case of a Menu List, the entire Menu List is deactivated.

Use this attribute to disable a Menu List or Item in a Menu definition before displaying the Menu bar. To activate a disabled Menu List or Item, use the appropriate Menu Control Number and **MENU** command described later in this chapter.

NOTE

> Never disable the Apple Menu; otherwise, New Desk Accessories will not be available to the user.

Example 1:

```
{Water Menu is disabled}
Menu$ (30) = ">> Water \DN3"
Menu$ (31) = "::Salt\N301"
Menu$ (32) = "::Fresh\VN302"
Menu$ (33) = "::Poisonous\N303"
Menu$ (34) = "."
Menu$ (35) = ""
```

Example 2:

```
{Items 301 and 303 are disabled}
Menu$ (30) = ">> Water \N3"
Menu$ (31) = "::Salt\DN301"
Menu$ (32) = "::Fresh\VN302"
Menu$ (33) = "::Poisonous\DN303"
Menu$ (34) = "."
Menu$ (35) = ""
```

C - Item Selection Indicator

This attribute places the character following the C before the Menu Item Name. Use this character to indicate that a Menu Item has a default or current setting.

The Check mark (character code 18) or the Diamond (character code 19) are the

usual characters indicating a default or current setting.

Example 1:

```
Menu$ (22) = "::Hot Pepper\C" +CHR$ (18) + "N451"
```

A check mark will appear in front of the word Hot Pepper.

Separating Groups of Menu Items

The attribute characters V and - (Dash) draw a line in a Menu List to separate groups of Items. Use these characters to group Items that are independent of other sets, but related to the Menu under which they appear. They may not be used for Menu Lists.

V - Underline

This letter is used to place an underline between two Menu Items.

Example:

```
Menu$ (21) = ">> File \N2"
Menu$ (22) = "::New\*NnN2001"
Menu$ (23) = "::Open...\*OoN2002"
Menu$ (24) = "::Close\V*WwN255"
Menu$ (25) = "::Save\*SsN2003"
Menu$ (26) = "::Save As...\N2004"
Menu$ (27) = "::Revert to Saved\VN2005"
Menu$ (28) = "::Print Setup...\N2006"
Menu$ (29) = "::Print...\V*PpN2007"
Menu$ (30) = "::Quit\*QqN2008"
Menu$ (31) = "."
```

- (Dash) - Dividing Line

This character provides a dividing line that makes more space between Items. The dash character must have its own Item Definition and Number. The dividing line should always be displayed in grey (dimmed). This attribute separates Item Names with descenders to have a better looking Menu List.

Example:

```
Menu$ (06) = ">> Edit \DN2"
Menu$ (07) = "::Undo\*ZzN250"
Menu$ (08) = "::-\DN9999"
Menu$ (09) = "::Cut\*XxN251"
Menu$ (10) = "::Copy\*CcN252"
Menu$ (11) = "::Paste\*VvN253"
```

Part Four: Creating the Apple IIGS Desktop

```
Menu$ (12) = "::-\DN9999"
Menu$ (13) = "::Clear\N254"
Menu$ (14) = "::-\DN9999"
Menu$ (15) = "::Show Clipboard\N260"
Menu$ (16) = "." {End of Menu List}
```

Font Style Menu Item Characters

The following attribute characters usually appear in the Font Style or Size Menu List. As with the **INVERSE** command, use these characters only to attract the attention of the user.

The font used must be capable of representing the desired character style; otherwise, the text will appear in normal text.

B - Boldface

This letter makes the Menu Item appear in boldface characters. This attribute is often used in the Size Menu List to indicate that a true font size is available by displaying the point size in bold face.

I - Italics

This letter makes the Menu Item appear in italic characters. This style is available only when the QuickDraw II Auxiliary Tool set is active.

O - Outline

This letter draws an outline of the Menu Item string. This style is available only when the QuickDraw II Auxiliary tool set is active.

S - Shadow

This letter adds a shadow to the name of the Menu Item. This style is available only when the QuickDraw II Auxiliary Tool set is active.

U - Underline

This letter underlines the name of the Menu Item.

The Shaston font, 8 points, (the current system font at the time of publication), does not support Underline.

X - Restore Menu or Item Color(s)

This attribute restores the color of a Menu Title or Item Name if it is displayed in a color other then black or white. The character **X** is used especially to restore the color of

the Apple logo; otherwise, the logo will turn green when the Apple Menu is selected.

Example:

```
Menu$ (40) = ">>@\XN1"
```

The color table for menus is set to the default colors (black and white). To alter the default colors, use the **TOOLBOX** command.

Apple Menu Items

The About Program_Name Item

This Menu Item must be the first item in the Apple Menu List. It is used to display a Dialog Box containing the name of the program, version number, copyright information as well as any information the application programmer wants to display. The name of the program follows the word "About".

Example: (see below)

The Help... Item

This optional Menu Item, if present, must be one of the top Items in the Apple Menu List. It is used to display a Dialog Box containing helpful information (hints, suggestions, etc) about the program being used.

Example:

```
Menu$ (40) = ">>@\XN1"
Menu$ (41) = "::About Example1\N257"
Menu$ (42) = "::Help...\*?/N258"
Menu$ (43) = "::-"
Menu$ (44) = "."
```

Example of a typical Menu definition:

```
ROUTINE DefineMenuBar
Array (0) = 1 {1 = Create Menus}
Menu$ (01) = ">> Water \N3"
Menu$ (02) = "::Salt\N301"
Menu$ (03) = "::Fresh\VN302" {A dividing line will appear
                between Fresh and Poisonous}
Menu$ (04) = "::Poisonous\N303"
Menu$ (05) = "."
Menu$ (06) = ">> Edit \DN2"
Menu$ (07) = "::Undo\*ZzN250"
```

```
Menu$ (08) = "::-\DN9999"
Menu$ (09) = "::Cut\*XxN251"
Menu$ (10) = "::Copy\*CcN252"
Menu$ (11) = "::Paste\*VvN253"
Menu$ (12) = "::-\DN9999"
Menu$ (13) = "::Clear\N254"
Menu$ (14) = "::-\DN9999"
Menu$ (15) = "::Show Clipboard\N260"
Menu$ (16) = "."
Menu$ (17) = ">> File \DN2"
Menu$ (18) = "::Open...\*OoNxxx"
Menu$ (19) = "::Close\V*WwNxxx"
Menu$ (20) = "::Quit\*QqNxxx"
Menu$ (21) = "."
Menu$ (22) = ">>@\XN1"
Menu$ (23) = "::About Example1\N257"
Menu$ (24) = "::Help...\V*?/N258"
Menu$ (25) = "."
Menu$ (26) = ""
{End of Menu Bar definition}
```

Defining the Menu

MENU (EventRecord% (,DesktopArray$ ( )

The **MENU** command handles the Menu definition. Its required parameters are an integer array and a string array. The left parentheses of the array designators may not be left off the **MENU** definition, otherwise the Compiler will issue an error. The string array was described above and must be set before this command is executed.

Element zero of the integer array must contain the Menu Control Number which instructs **MENU** the command usage. Other integer array element uses will be defined below.

Be certain the arrays are large enough to hold the entire Menu Definition; otherwise, an error will occur during execution.

Example:

```
{Define Menu Bar and its Menu Items}
EventRecord% (0) = 1 {Menu Control Number, Create}
{Menu defined in array DesktopArray$}
MENU (EventRecord% (, DesktopArray$( )
```

How to Use the Menu Control Numbers

The Menu Control Numbers determine which actions will be performed by the **MENU** command. A Menu Control Number of one is used to create the Menu. The appearance of this Menu may be changed using other Menu Control Numbers. Menu Lists and Items may be deactivated and reactivated or completely removed from the Menu Bar.

The actions are performed according to the following values passed to element 0 of the integer array passed to **MENU**:

Table 4.2.3 Menu Control Numbers

Code	Action
0	Remove a Menu List from the Menu bar
1	Create the Menu
2	Reserved for Future Expansion
3	Reserved for Future Expansion
4	Enable a disabled Menu Item or List
5	Disable a Menu Item
6	Remove a Menu Item from a Menu List
7	Add New Desk Accessories

Remove a Menu List (0)

Menu Control Number of zero closes the specified Menu List. The Menu List will disappear from the screen, and be removed from memory.

NOTE

> The entire Menu Bar must be recreated to display the removed Menu List again.

To remove a Menu List, assign a zero to element zero of the integer array passed to **MENU** and pass the Menu Identification Number (assigned by you) of the Menu List to be removed to element one of the integer array.

Example: <Program fragment>

```
EventRecord% (0) = 0
EventRecord% (1) = Menu_List_ID
MENU (EventRecord% (, DesktopArray$ ( )
```

Create the Menu (1)

This Menu Control Number sets up and displays the Menu as it was defined in the

string array passed to **MENU**.

To create the Menu Bar and its Menu Lists, assign a one to element zero of the integer array. The string array defining the entire Menu must be assigned as described in the previous section.

Example:

```
{DeskTopArray$ previously defined}
EventRecord% (0) = 1 {Create Menus}
MENU (EventRecord% (, DesktopArray$ ( )
```

Reserved for Future Expansion (2)

Reserved for Future Expansion (3)

Enable a Disabled Menu List or Item (4)

A Menu Control Number of four reactivates a disabled Menu List or Item that was disabled by Menu Control Number five.

To reactivate a Menu List/Item, assign a four to element zero of the integer array, and assign the number of the List/Item to be reactivated to element one of the integer array.

Example: <Program fragment>

```
EventRecord% (0) = 4
EventRecord% (1) = ItemNumber
MENU (EventRecord% (, DesktopArray$ ( )
```

Disable a Menu Title or Item (5)

A Menu List or Item is disabled by using a Menu Control Number of five. The attribute character D is used only to disable a Menu Item when it is first displayed.

To deactivate a Menu List or Item, assign a five to element zero of the integer array passed to **MENU**, and pass the List or Item ID to be disabled to element one of this integer array.

Example: <Program fragment>

```
EventRecord% (0) = 5
EventRecord% (1) = MenuItem
MENU (EventRecord% (, DesktopArray$ ( )
```

Remove a Menu Item from a Menu List (6)

Menu Control Number of six removes a Menu Item from the specified Menu List. The Menu Bar must be recreated to display that Menu Item again.

To remove a Menu Item, assign a six to element zero of the integer array passed to **MENU**, and assign the Item ID to be removed to element one of this integer array.

Example: <Program fragment>

```
EventRecord% (0) = 6
EventRecord% (1) = MenuItem
MENU (EventRecord% (, DesktopArray$ ( )
```

Add New Desk Accessories (7)

If you have any New Desk Accessories within the DESK.ACCS folder on the boot volume, the operating system will load these NDAs into memory at boot time. By using a Menu Control Number of seven, you may cause all NDAs to be contained within the Menu List.

The restrictions are that you must have an Apple Menu (the Menu defined by @) and this Menu List must have a Menu ID of one.

To have these NDAs appear in your Menu List, assign a seven to element zero of the integer array passed to **MENU** and invoke the **MENU** command. That's all there is to it.

Example: <Program fragment>

```
EventRecord% (0) = 7
{Add Desk Accessories}
MENU (EventRecord% (, DesktopArray$ ( )
```

Unhighlight a Menu Title

Each time a Menu Item has finished its task, the Menu List under which it appears may be returned to its original state to indicate that the operation is now finished.

To unhighlight the Menu List of the selected Item, use Toolbox call 44 to the Menu Manager. Place this line at the end of the Event Loop.

Example: <Program fragment

```
TOOLBOX (15,44: 0, MenuNumber) {Unhighlight it}
```

Monitoring the Menu

MOUSE (IntegerArray ( )

It is not enough just to define a Menu; you must also monitor the user's response and take actions based upon this response.

To monitor a Menu, you must make use of the **MOUSE** command. The only parameter to **MOUSE** is an integer array dimensioned to at least 20 elements. This integer array may contain any values when **MOUSE** is executed, but **MOUSE** will return the values with which you may analyze the user's response when control is returned to you.

You must monitor the user's response to the **MENU** command from within a loop

called the Event Loop. This Event Loop is best defined within a subroutine to facilitate changes and maintenance.

When monitoring the Menu bar for a selection using the **MOUSE** command, look for a seventeen (17) returned in element zero of the integer array passed to **MOUSE**. Element zero of the passed integer array contains the Event code, and 17 is the Menu code. Once a 17 is received, immediately access elements 9 and 10 of the integer array.

Elements 9 and 10 of the passed integer array will contain the Menu Item ID number, and the Menu List ID number respectively. These values are needed to determine which Menu Item has been chosen. Elements 1 through 10 of this integer array will contain the Task Record which contains additional information about the user's response (see table 4.2.4).

Once a response is received, the program must direct the execution to a module that will handle the action indicated by the Menu Item. For example, if Quit is selected, the program should exit the Event Loop and invoke the ShutDown routine.

Example:

```
{Menus previously defined}
Quit! = FALSE
WHILE NOT Quit! {Top of Event Loop}
    MenuID = 0
    MOUSE (Array% ( )
    TaskValue = Array% (0)
    IF TaskValue = 17 THEN BEGIN {a Menu was chosen}
      MenuID = Array% (9) {Menu Item}
      MenuTitle = Array% (10) {Menu Number}
      {Direct execution to proper module}
      CASE_OF MenuID
           DO 261 {New}
                GOSUB NewGame {Open a Window}
           ENDDO
           DO 262 {Quit}
                Quit! = TRUE {Quit program}
           ENDDO
      ELSE_DO
           BELL {Error condition}
           BELL
      ENDCASE {MenuID}
      TOOLBOX (15,44: 0 {False}, MenuList) {Unhighlight it}
    ENDIF
WEND {Event Loop}
{...ShutDown Code...}
```

```
<Program continues>
```

Table 4.2.4. Menu Codes Returned by MOUSE

Event Code Element Number	Description
0	Task value
	17 = Mouse on Menu Item
1	What field of Task Record
	1 = Mouse Down Event
2 & 3	Message field of Task Record
4 & 5	When field of Task Record
6 & 7	Where field of Task Record
	(Mouse location)
8	Modifiers field of Task record
9 & 10	Task Data

For more information on defining standard Menu applications, refer to Apple's Human Interface Guidelines: The Apple Desktop Interface.

The MENU program in the Desktop.Samples/ folder on the MAB.SUPPORT disk demonstrates how to use the Menu Control Numbers.

Chapter Three

Windows

Overview

This chapter shows how to do Windows and provides guidelines on how to create, manage and draw in Windows. To use special Windows, consult the Apple IIGS Toolbox Reference Manuals or some other comprehensive documentation of the Apple IIGS Toolbox.

Examples of the techniques described in this chapter are included in the program WINDOW in the Desktop.Samples folder of the MAB.SUPPORT Disk. Please refer to the WINDOW program as you study this chapter.

What are Windows

A Window is a structure in which information, such as a document or a picture, is presented to the user by the application program. Any text or graphics image that may be drawn with the Super High Resolution commands may be presented in a Window.

A Window consists of a frame that surrounds the image and a content area inside the frame in which the image is presented. Although a Window frame may be any size or shape, two standard styles of Window frames are supported: the document Window frame and the alert Window frame.

The document-style frame supports optional controls that may be used to change the size of the Window and the position of the document within the Window. The alert frame is mainly used to create alert dialogs. This style of frame, however, may also be used for a Window.

The controls in a document frame are optional, and may be used in any combination. They include the title bar, close box, zoom box, vertical scroll bar, horizontal scroll bar, size box and information bar.

Managing Windows

WINDOW (Integer_Array (, String_Array$ ( )

The **WINDOW** command supports the creation and closure of Windows, and provides a method to convert Window pointers to Window numbers and vice versa.

WINDOW requires two parameters: an integer array and a string array. These arrays will contain the information required to create and/or modify the Window. Be certain these arrays are dimensioned large enough to contain all the information required by this command.

Element 0 of the integer array holds the Window Control Number that specifies

which function is to be performed by the **WINDOW** command in accordance with the following table:

Table 4.3.1 Window Control Numbers

Code	Action
0	Close an application Window
1 - 10	Create the specified Window
11	Find the pointer of a specific Window
12	Find the Window number of a specific pointer.

Creating the Window

To create a Window, assign the values from the table below to the integer array passed to the **WINDOW** command.

Programmers

> The variable names from the Toolbox manual are included in the table below to aid in identifying the associated NewWindow parameters in the Toolbox manual.

Table 4.3.2 Create Window Parameters

Element	Variable Value
0	**Window Number to Create (1 - 10)**
1	**wFrameBits** Framebits. See Table 4.3.3.
2	**wTitle** String array element of Window title.
3	**wDataH** Height of Window data area. Used to compute the right scroll bar. Set to 0 if the Window has no right scroll bar.
4	**wDataW** Width of Window data area. Used to compute the bottom scroll bar. Set to 0 if the Window has no bottom scroll bar.
5	**wMaxH** Max content height allowed when using the size box. If set to 0, will

Element	Variable Value
	default to take up the height of the desktop. Set to 0 if the Window has no size box.
6	**wMaxW** Max content width allowed when using the size box. If set to 0, will default to take up the width of the Desktop. Set to zero if the Window has no size box.
7	**wScrollVer** Number of pixels to scroll the content region when the up or down arrows are selected in the right scroll bar. Set to 0 if the Window has no right scroll bar.
8	**wScrollHor** Number of pixels to scroll the content region when the right or left arrows are selected in the bottom scroll bar. Set to 0 if the Window has no bottom scroll bar.
9	**wPageVer** Number of pixels to scroll the content region when the up or down page regions are selected in the right scroll bar. Set to 0 if the Window has no right scroll bar.
10	**wPageHor** Number of pixels to scroll the content region when the right or left page regions are selected in the bottom scroll bar. Set to 0 if the Window has no bottom scroll bar.
11-14	**wPosition** Rectangle data structure that uses elements 11 through 14 of the array to define the initial position of the Window on the screen.
11	**Y minimum value** Top edge of Window.
12	**Y maximum value** Bottom edge of Window.
13	**X minimum value** Left edge of Window.
14	**X maximum value** Right edge of Window.

Programmers

All other NewWindow parameters are set to default values when **WINDOW** makes the NewWindow call to create the Window. This should be satisfactory for most Windows. If other values are needed, then you must either use Tool calls to change the desired values after the Window has been created, or you must call NewWindow directly. The latter requires much more manipulation of data structures and precludes using the *Micol Advanced BASIC* **WINDOW** command.

If the Window has a title bar, pass the Window title string in the appropriate element of the string array, as specified in element 2 of the integer array.

Creating The Window

Assign a Window number into element 1 of the integer array. The Window number is an arbitrary identifier for the window in the application program. Valid Window numbers are in the range 1 - 10. This means that a *Micol Advanced BASIC* Desktop program may have a maximum of 10 Windows open at one time. The Window number is used by the Close function of the **WINDOW** command to determine which Window to close.

Programmers

When creating a Window, the **WINDOW** command will return a pointer (address) to the Window's Grafport. This pointer is a four-byte long integer. The low word of the pointer will be returned in element 0, and the high word in element 1 of the integer array. This pointer may be needed to make calls to the Window Manager. The pointers may either be saved in temporary variables as each Window is created, or a Window Control Number may be used to look up the pointers whenever they are needed. The use of the **WINDOW** command to look up pointers to open Windows is discussed later in this chapter.

Setting Wframebits

This section is a description of wFrameBits (described in Table 4.3.2); the value assigned to element one of the integer array passed to the **WINDOW** command.

WFrameBits is a bitflag (a binary number that is treated by *Micol Advanced BASIC* as an integer value) that determines the type of Window frame and which optional controls will be available to the application Window. It also specifies other optional behaviors of the Window frame.

To calculate the integer value of wFrameBits, start with zero, then add the values from the table below to select the features that you want. If you specify a value of 0 for wFrameBits, the 0 will be replaced with a default value that will create a Window with all of the standard Window features.

Table 4.3.3 Window FrameBits

Bit	Label	Value	Feature On (1) / Off (0)
0	fHilited	DS	Highlighted / Not Highlited
1	fZoomed	2	Display Size: Zoomed / Not zoomed
2	fAllocated	DS	Window Record: Allocated / Not Allocated
3	fCtlTie	DS	Window Controls: Inactive / Active
4	fInfo	16	Info bar / No Info bar
5	fVis	32	Visible / Invisible
6	fQContent	DS	Mouse Activates Window
7	fMove	128	Movable / Non-movable
8	fZoom	256	Zoom Box / No Zoom box
9	fFlex	512	Maintain origin / Change origin
10	fGrow	1024	Grow Box / No Grow box
11	fBScroll	2048	Horizontal Scroll Bar / No bar
12	fRScroll	4096	Vertical Scroll Bar / No bar
13	fAlert	8192	Window Frame: Document / Alert
14	fClose	16384	Close box / No Close box
15	fTitle	32768	Title bar / No Title bar

DS = Default Setting. The values you set are ignored.

Bit 0 fHilited

Used internally. The value you set does not matter.

Bit 1 fZoomed (value = 2)

Zooms the Window to the size of the entire Desktop when this bit is set to 1.

Bit 2 fAllocated

Used internally to free the memory area used by a Window. The value you set does not matter.

Bit 3 fCtrlTie

Used internally. The value you set does not matter.

Bit 4 fInfo (WARNING: Set to zero)

This bit is used to add an information bar to the Window. Info bars may be supported only by an assembly language routine linked in. If this bit is set to 1, the program will crash.

Bit 5 fVis (value = 32)

This bit determines if the Window is visible when it is created. If this bit is cleared, the Window will be invisible when created. If this bit is set, the Window will be visible.

Programmers

> After creation, Windows may be made visible or invisible using the ShowWindow and HideWindow Tool calls. According to the Toolbox documentation, this bit is used internally by the Window Manager and the value you set does not matter. However, contrary to Apple's Toolbox documentation, for System v5.04 (GS/OS v3.3) and earlier, we have found this information to be incorrect.

Bit 6 fQContent (value = 64)

Used internally. The value you set does not matter.

Bit 7 fMove (value = 128)

Permits the Window to be dragged by the title bar when this bit is set to 1.

Bit 8 fZoom (value = 256)

The title bar will have a zoom box when this bit is set to 1.

Bit 9 fFlex (value = 512)

If this bit is set to 1, the data height and width are flexible.

Bit 10 fGrow (value = 1024)

The Window will have a grow box (size box) if this bit is set to 1.

Bit 11 fBScroll (value = 2048)

The Window will have a horizontal scroll bar if this bit is set to 1.

Bit 12 fRScroll (value = 4096)

The Window will have a vertical scroll bar if this bit is set to 1.

Bit 13 fAlert (value = 8192)

The Window will have an alert frame if this bit is set to 1. In this case, set the following bits to 0: 8, 9, 10, 11, 12, 14 and 15.

Bit 14 fClose (value = 16384)

The Window will have a close box if this bit is set to 1.

Bit 15 fTitle (value = 32768)

The Window will have a title bar if this bit is set to 1.

If the Window has either scroll bars, it should also have a grow box and a zoom box (bits 8, 10 with either 11 or 12).

Closing a Window

Call the **WINDOW** command with the following values in the integer array to close a Window:

Element	Value
0	0
1	Window Number 1 - 10

This command will remove the appropriate Window from the Desktop and will free all memory used for the Window's data structures. To reopen the Window, your application must again execute the code to create the Window.

Example: <Program fragment>

```
{Close a Window}
EventRecord% (0) = 0
EventRecord% (1) = WindowNumber%
WINDOW (EventRecord%(, DeskTopArray$( )
```

Using A Specific Window

Before anything may be done in a Window, either draw in a Window or call an update routine, one of two methods may be used to have access to a Window:

1. Use the Window Control Number (1-10) assigned by you to the Window during this creation
2. Use the pointer of the Window assigned by *Micol Advanced BASIC* to the Window during its creation. Most Window Manager Tool functions need the Window's pointer to do its task.

Obtaining the Pointer of a Window

The pointer is returned in elements 0 and 1 of the integer array when the Window is created by the **WINDOW** command. The pointer may be saved in variables as the Windows are created or the **WINDOW** command may be used to look them up whenever they are needed.

The **WINDOW** command will return the pointer to any of the ten Windows that may be created using the **WINDOW** command. If the pointer to a Window number that is not currently open is requested, zeros will be returned.

Element	Value
0	11
1	Window number (1 - 10)
2	pointer returned (low word)
3	pointer returned (high word)

Example: <Program fragment

```
{Get the Window's pointer}
EventRecord% (0) = 11
EventRecord% (1) = WindowNumber%
WINDOW (EventRecord% (, DeskTopArra $( )
{Low part of the pointer}
WinPtrL% = EventRecord% (2)
{High part of the pointer}
WinPtrH% = EventRecord% (3)
```

Obtaining the Number of a Window

The **WINDOW** command may also be used to determine which Window number a pointer belongs to. For example, if it is detected that the user clicked in a Window's close box, a pointer to the Window to be closed will be returned in the integer array used by the **MOUSE** command. The application may then use the **WINDOW** command to determine which Window number the pointer belongs to, then use the **WINDOW**

command again to close the Window with that Window number.

Element	Value
0	12
1	Window number (1 - 10 returned)
2	Pointer (low word)
3	Pointer (high word)

Example: <Program fragment>

```
{Get the Windows pointer}
EventRecord% (0) = 12
EventRecord% (2) = WinPtrL% {Low part of the pointer}
EventRecord% (3) = WinPtrH% {High part of the pointer}
WINDOW (EventRecord% (, DeskTopArray$( )
WindowNumber% = EventRecord% (1)
```

Programmers

> The **WINDOW** command is designed to be a general use command in creating the Apple IIGS Desktop. It is suitable for the vast majority of uses. However, Advanced programmers should know that internally, **WINDOW** uses the Window Manager Tool to create and manage Windows. If you wish to create more elaborate Windows, you may use the **TOOLBOX** command to the Window Manager to do this.

Monitoring Windows

MOUSE (Integer_Array ( )

After a Window has been created, it is necessary for the application program to monitor and respond to certain events that affect the Window, and to maintain the image in the Window's content area.

Monitoring events in Windows is handled by the **MOUSE** command.

The following actions are done automatically by the **MOUSE** command when a Window is being monitored:

- Activating an inactive Window to bring it to the top (assuming more that one Window is open) and making it active. A click of the Mouse on any region of a Window activates it.
- Dragging the active Window by holding the Mouse button down while the cursor is on the Title Bar.
- Changing the size of the active Window when the user clicks in the Zoom Box.

- Changing the size of the active Window to resize it by holding and dragging the Size Box.

The following actions are partially automated by the **MOUSE** command:

- Closing the active Window when the user clicks in its Close Box. The application program will be notified when a close box has been clicked, and will return a pointer to identify which Window's close box was clicked. The application program must actually close the Window.
- Redrawing the Window's contents when a hidden portion of the content region is exposed. Hidden portions are exposed when Windows are first opened, when the size or zoom boxes are used, or when Windows are dragged around the desktop exposing previously covered or off-screen areas of Windows.

Window Watching Information

Only one event at a time may be reported. As events occur, they are stored in chronological order in an Event Queue, and are reported and cleared from the queue, one at a time, each time **MOUSE** is called. The application program uses the **MOUSE** command inside a loop called an Event Loop. This loop removes events from the event queue one at a time, and calls the *Micol Advanced BASIC* routines to respond to each event. This is the heart of Desktop programming on the IIGS.

When an event is detected in a Window, element zero of the integer array referenced in the **MOUSE** command will return the Event number, which will tell you what action was taken by the user. Elements 1 through 10 will contain the Task Record which will provide additional information about the event.

The following values returned in element 0 indicate events that affect a Window, and its pointer. The values affecting the Window appear in elements 9 and 10 of the integer array passed to **MOUSE**:

Value	Description
0	No event to report
6	Update event. One of the Windows needs its contents redrawn.
1[illegible]	Mouse down in Content Region.
22	Mouse down in GoAway box (close box).
24	Mouse down in Info Bar. (not used by *Micol Advanced BASIC*).
27	Mouse down in Window frame (but not in scroll bar)

The following events are handled internally by the system. The application program need take no action in response to these events:

Value	Description
0	No event
8	Activate/Deactivate Window
19	Mouse down in Content Region. **MOUSE** will select (highlight) the Window.

Part Four: Creating the Apple IIGS Desktop

20 Mouse down in Title Bar. **MOUSE** will select the Window and handle dragging.

21 Mouse down in Size Box. **MOUSE** will select the Window and track the grow Window control if used.

23 Mouse down in Zoom Box. **MOUSE** will select the Window and track the zoom control.

Handling Window Updates

NOTE

> This section is intended mainly for advanced programmers; ignore the areas you cannot understand. Please note that Task Master is an internal routine within the Toolbox that automatically performs updates upon a Window whenever the user references this Window. Few of you will need to worry about this aspect of doing Windows.

Whenever a previously hidden portion of a Window's content area becomes exposed, the Window needs to be updated (needs to have its image redrawn). This occurs when a Window is first opened, when a Window is zoomed, when a Window's size box is used, when a portion of a Window is dragged from an off-screen to on-screen position, or when one Window is moved, exposing a portion of another Window that was underneath. Whenever a Window needs to be updated, an Update Event (6) is generated.

Two methods are provided for handling update events. If the application program has an assembly language routine that draws the Window's contents, the address of this routine may be passed in the field wContDefProc in the NewWindow tool call. In this case, TaskMaster will call the application's assembly language routine whenever it needs to update the Window. This method completely automates Window updates.

NOTE

> *Micol Advanced BASIC*'s run time routines cannot be called from inside a program, so this method is not possible without a linked-in assembly language routine. Such a routine is available commercially through the MaBug Users Group.

As an alternative method for handling updates, an update event is reported whenever a Window needs to be redrawn. This method is used whenever the application has not passed the address of a machine language update handler. Most *Micol Advanced BASIC* programs will use this method.

An update event is reported by returning a six (6) in element 0 of the integer array used by the **MOUSE** command. The pointer to the Window needing updating is

returned in Elements 9 and 10 of the integer array. Element 9 contains the low value of the pointer and element 10 contains the high value.

The application program must determine which Window needs to be updated, then call the appropriate DrawContent routine that is specific for that Window. (A DrawContent routine draws the contents of a specific Window.) The number of the Window that needs to be updated (with a 12 in element 0) to convert the Window pointer returned by the **MOUSE** command to the corresponding Window number.

The application should include a DrawContent Procedure for each Window. A DrawContent Procedure draws the entire current image of the Window's contents using Super High Resolution commands. The applications event loop should call the appropriate DrawContent Procedure in response to an update event for an application Window.

The DrawContent routine has a different structure for Windows with and without scroll bars. Both forms are illustrated in the WINDOW program on the MAB.SUPPORT disk.

For Windows without scroll bars, the DrawContent procedure should perform the following actions:

- Call GetPort (tool call $1C04) to save the current GrafPort into temporary variables.
- Call SetPort ($1B04) to make the Window's GrafPort the current GrafPort.
- Call BeginUpdate ($1E0E). This informs TaskMaster that you are handling the update event. TaskMaster will continue to report an update event until you inform it, through calls to BeginUpdate and EndUpdate, that the update event is being handled. Only then will TaskMaster report the next event in the Event Queue.
- Draw the image using Super High Resolution commands.
- Call EndUpdate ($1F0E) to inform TaskMaster that the update has been completed.
- Call SetPort ($1B04), passing the GrafPort pointer saved from the GetPort call above, to restore the current GrafPort.

For Windows with scroll bars, the procedure is slightly more complex. It should perform the following actions:

- Call GetPort ($1C04) to save the current GrafPort.
- Call StartDrawing ($4D0E). This makes the Window's GrafPort the current GrafPort and adjusts the Window's origin, and therefore the position in which the image will be drawn in the Window, to correspond to the current setting of the scroll bars.
- Call BeginUpdate ($1E0E).
- Draw the image using Super High Resolution commands.
- Call EndUpdate ($1F0E)
- Call SetOrigin ($2304) to restore the Window's origin.
- Call SetPort ($1B04) to restore the current GrafPort.

When a Window is created, an update event is generated immediately after the

Window frame is displayed. For this reason, an application does not need to provide code specifically to draw the initial image in the Window. An update event will occur when the Window is created. The applications event loop will then call the appropriate DrawContent routine to update the Window thus creating the initial image.

If you ever wish to change the image in a Window, the application may simply redraw the image. The application should call GetPort to save the current GrafPort, call SetPort to make the window's GrafPort the current GrafPort, draw in the Window, then call SetPort to restore the old GrafPort.

If a Window's image changes during the course of the program, that Window's DrawContent Routine must contain conditional logic to insure that the current image will always be drawn in response to update events. If a DrawContent Routine is not coded properly, the Window could be refreshed with an out-of-date image in response to an update event.

The sample program WINDOW includes a simple example of a Window whose image can change. Window number 3 contains either a happy face or a sad face depending upon which Menu Item has been selected by the user. The Menu handler sets the global boolean variable gHappy! to **TRUE** if the current image is the happy face or **FALSE** if the current image is the sad face. The DrawContent Routine checks this variable to determine which face to draw, happy or sad, in response to update events. In this way, the application always responds to update events in this Window by drawing the correct (current) image. Your applications must also guarantee that all DrawContent Routines always draw current images.

Drawing in a Window

Any valid Super High Resolution command may be used to draw inside a Window; this includes the **DRAWSTR**, **HPLOT**, and **HPLOT TO** commands, and Toolbox calls.

It is often necessary to know the exact size of a string, in pixels. Use the **LEN** (string length) function. **LEN** will store the size, in pixels, in **True_Value** (locations 202 and 203) when the Super High Resolution screen is active (**HGR** or **HGR2**).

To draw in a Window, the current GrafPort must be changd to the Window's GrafPort you want to draw into. A GrafPort is a data structure that completely describes a Super High Resolution drawing environment. Each time a Window is created using the **WINDOW** command, a GrafPort also is. The pointer returned by the **WINDOW** command is the pointer to the Window's GrafPort.

The correct procedure for changing GrafPorts is as follows:

- Call GetPort ($1C04) to get a pointer to the current GrafPort. The current GrafPort is the GrafPort that is currently open — the one in which the Super High Resolution is currently active.
- Save the pointer to the current GrafPort in temporary variables.
- Call SetPort ($1B04) to make the Window's GrafPort the current GrafPort.
- Execute the Super High Resolution commands to draw in the Window.
- Call SetPort to restore the previous GrafPort.

The reason the current GrafPort must be saved and restored is to assure that things are left as they were. If you fail to follow this practice, problems such as having Desk Accessories draw in your Windows because the Desk Accessories become confused about what is the current GrafPort will arise. Remember, leave things as you find them and you will have no problems.

Note to Advanced Programmers

The rectangle in elements 11 - 14 of the integer array referenced in the **WINDOW** command, when the Window is created, is passed to both wPosition and wZoom for the library's call to NewWindow. This design simplifies Windows for beginners, since these parameters need not be different for most Windows. Several other fields in the Window parameter list are also set to default values, for example, wRefCon and wContDefProc are set to zero. The default values may of course be changed using Window Manager Tool calls.

Advanced programmers may wish to call NewWindow directly. In this case, all of the parameters in the NewWindow parameter list may conveniently be passed in an integer array. When calling NewWindow directly, you may not use the **WINDOW** command to close the Window or to convert between pointers and Window numbers.

wInfoDefProc is also set to the default value zero by the **WINDOW** command when creating a Window. This is necessary because *Micol Advanced BASIC* cannot provide a machine language information bar drawing routine for TaskMaster to call. Accordingly, *Micol Advanced BASIC* does not support Window info bars unless the application provides a linked-in assembly language routine to draw the info bar.

Chapter Four

Dialog Boxes

Overview

This chapter shows how to create and monitor Dialog Boxes within your *Micol Advanced BASIC* programs.

Dialog Box Definition

A Dialog Box is like the front of a television or stereo set: it is a panel with control dials and buttons, etc. Just like a television, these dials and buttons are used for control.

A modeless Dialog Box is a panel that allows the user to do other things while the Dialog Box is still on the screen; for example, the tool of a paint/draw program or the Find/Replace box of a word processor or a data base program.

A modal Dialog Box is a rectangle that forces the user to act on it before doing anything else like an "About..." box. *Micol Advanced BASIC* may be used to create modal Dialog Boxes. If you wish to create modeless Dialog Boxes, you will have to make use of the **TOOLBOX** command.

Dialog Boxes, by convention, are centered on the display and do not cover the entire screen. They may contain graphics, descriptive or informative text, fill-in areas, and control-like buttons.

The point of origin for the Dialog Box is the upper left corner of the Desktop, position 0,0. Make sure not to exceed the maximum X coordinate on the horizontal axis depending on the graphics mode (320 or 640), otherwise the right part of the Dialog Box will be hidden from view.

To be meaningful, a Dialog Box must contain Parts to which the user may respond. Parts may be added, disabled, enabled, etc. as the need arises.

Do not forget to define the Super High Resolution graphics screen with either an **HGR** (320 mode) or **HGR2** (640 mode) before creating your Dialog Boxes.

Controls and Labels

The purpose of a Dialog Box is to give the user an easy, and straightforward way to communicate with the program. This is usually done with a symbol used in everyday life, such as a radio button, with a simple descriptor, that the user can easily relate to. This symbol, together with its descriptor, is what we will call a Dialog Box Part.

Each Part in a Dialog Box has 6 components:

1. **The Part ID Number**. Each Part must have a distinct ID number. The ID number may range from 1 to 255. An ID number of 0 is invalid and will cause a

run time error.

2. **The Part Location.** All Parts use coordinates relative to the upper left corner of the Dialog Box. These coordinates, expressed in pixels, are called local coordinates. The upper left corner of the Dialog Box is local co-ordinate 0,0. If the coordinates specified for the Part are greater than the maximum boundaries of the Dialog Box, the Part will be invisible. No error is generated.
3. **The Part Type.** Five different types of controls and one type of label may be displayed using *Micol Advanced BASIC*.
4. **The Characteristics of the Part.** The characteristic value (also called PartFlag) of a Part depends on the type of the Part used. Not all Part types use this component: see the specific Part for details.
5. **The Value of the Part.** This variable holds the value the Part has when it is first displayed. The Part Value depends on the type of the Part used. Not all Part types use this component: see the specific Part for details.
6. **The Part Descriptor.** Most Parts must be labeled with words describing, as closely as possible, the action to be performed. Not all Part types use this component. See the specific Part for details.

Table 4.4.1 Standard Dialog Part Types

Type Value	Part Type
10	Push Button
11	Check Box
12	Radio Button
13	Scroll Bar
15	Static Line
17	Edit Line

The Push Button

The push button produces an immediate or continuous action when it is pressed. It has round or square corners with a single or a bold outline. Its Part type is ten.

A push button with an ID of one will have a bold outline. All other push buttons with ID numbers from two to 255 will have a simple outline.

The button item with an ID of one is the default button. A default button should not be used to control a destructive command. If a default button is not needed on a Dialog Box, do not use ID number one for a push button.

The Return key is the keyboard equivalent of a button with ID one (usually "OK") and the Esc key is the keyboard equivalent of a button with ID two (usually "Cancel"). If a button does not have an ID of one or two, the keyboard equivalents will be non-functional.

The display rectangle of the button will be calculated automatically by supplying the upper-left coordinates (Min Y, Min X, relative to the Dialog Box) of the Part, and setting

the lower-right coordinates to 0,0.

The PartValue is unused (set to zero) with push buttons.

The itemFlag defines the shape of the button:

0	Round corners, single outline
256	Round corners, bold outline
512	Square corners, single outline
768	Square corners, single outline with shadow

The Check Box

A check box is a box that may be filled with an X (ON) or be left empty (OFF). It is often used to select options that will cause changes later on. The action of a check box is independent of other check boxes on the same Dialog Box. Its Part Type is 11.

The display rectangle of the check box will be calculated automatically by supplying the upper-left coordinates (Min Y, Min X, relative to the Dialog Box) of the Part, and setting the lower-right coordinates to 0,0.

The label is placed on the right side of the check box. Its description should be as long as necessary, but it may not exceed one line.

The PartValue has the initial setting of the button (0 = OFF, 1 = ON).

The PartFlag is not used (set to 0) with check boxes.

The Radio Button

A radio button simulates a button like the one in a car radio. The button is filled with a circle to indicate the ON setting or empty to indicate it is OFF.

A Radio Button is used to select only one option that will cause a change later on. The selected button will turn off the button previously selected on the Dialog Box. A Dialog Box should have at least two radio buttons, if they are used at all. The Part Type number is 12.

The display rectangle of the radio button will be calculated automatically by supplying the upper-left coordinates (Min Y, Min X, relative to the Dialog Box) of the Part, and setting the lower-right coordinates to 0,0.

The label is placed on the right side of the radio button. Its descriptor may be as long as necessary but may not exceed one line.

The PartValue has the initial setting of the button (0 = OFF, 1 = ON).

The PartFlag has a value from 0 to 32512. This value links the radio buttons together. Use a different value (in increments of 256) for each series of radio buttons.

The Scroll Bar

A scroll bar is a rectangle with an arrow at both ends. It causes an immediate result on the object it is controlling. The arrows are used to move the lever one notch at a time. A click of the Mouse on the grey area, above or below the thumb, will move the display

by the number of pixels represented by the thumb. The lever, also called a thumb, may be dragged with the Mouse to a precise position. Its Part type number is 13.

The display rectangle of the scroll bar sets its thickness, height or length depending on its orientation.

The scroll bar is not labeled.

The PartValue indicates the actual position of the thumb and ranges from 0 to 290.

The PartFlag indicates whether a horizontal or vertical scroll bar or indicator will be displayed. The proper values are:

0	Vertical indicator
3	Vertical bar
4096	Horizontal indicator
7168	Horizontal bar

The Static Line

A Dialog Box may be labeled using a static line of text. This static line of text may not be modified at a later time. The Dialog Box label uses the current font for the characters. The maximum length of a static line is set to 64 characters by *Micol Advanced BASIC*. Its Part type number is 15.

All static lines should be disabled by adding 32768 ($8000) to its ID number so they will not return Part numbers to the Event Loop.

The display rectangle of a Static Line must be large enough to show the entire text without overlapping other Dialog Box Parts.

To display a static text on one line, a rule of thumb is to allow at least 20 pixels wide by 10 pixels in height for a character. For example, a line of text with 8 characters will need at least 160 pixels in length.

To display a line of static text on multiple lines, concatenate a carriage return (ASCII 13) between each line. The next line of text will be placed 10 pixels below the preceding one.

The PartValue and the PartFlag are unused (set to zero) with a Static Line.

A Static Line should be the first Part displayed when a Dialog Box appears on the screen. To do this, assign the highest Part ID number to the static line. This line of text will help the user immediately understand the purpose of the Dialog Box.

The Edit Line

An Edit Line is a rectangular box containing information. It is often used to give a Pathname for operating system operations or to provide information in response to a query by the program. Its Part number is 17.

The display rectangle of an Edit Line must be at least 20 pixels wide by 12 pixels in height for each character. For example, a line of text with 8 characters will need at least 160 pixels in length. The next line of text will be placed 12 pixels below the preceding one.

The PartValue contains the maximum number of characters that may be entered.

The PartFlag is unused (set to 0).

A line of static text should appear above an Edit Line to indicate what is expected as input.

Defining the Dialog Box

DIALOG (IntegerArray (, StringArray ( )

A Dialog Box is defined by the **DIALOG** command. **DIALOG** requires two arrays: an integer array and a string array to hold the necessary values. The integer array holds the coordinates of the Dialog Box and those of the Parts within it. The string array holds the labels of the Dialog Box Parts.

Element zero of the passed integer array contains the Dialog Control Number. This Dialog Control Number instructs **DIALOG** as to what action to take, and controls which values are required in the passed arrays (see table 4.4.2).

Dialog Control Numbers

Element zero of the integer array passed to **DIALOG** must contain the Dialog Control Number. This Dialog Control Number determines the function of **DIALOG** and performs the following tasks:

Table 4.4.2 Dialog Control Numbers

Value	Task
0	Close the Dialog Box
1	Create the Dialog Box
2	Add a Part to the Dialog Box
3	Remove a Part from the Dialog Box
4	Enable a Part in the Dialog Box
5	Disable a Part in the Dialog Box

The Dialog Control Number assigned to element zero of the integer array passed to **DIALOG** will determine which further values are required in the integer array:

Close the Dialog Box (0)

If the Dialog Control Number specified is zero, the current Dialog Box will be closed and disappear from the screen. The memory used by the Dialog Box will be released.

Example:

```
Array% (0) = 0 {Close Dialog Box}
DIALOG (Array% (, Array$( )
```

Create the Dialog Box (1)

This is the most important Dialog Control Number. If the Dialog Control Number specified is one, the system will create (but not yet display) a Dialog Box with the controls and labels defined in the parameter arrays passed to **DIALOG**. Once the **MOUSE** command is executed subsequent to the Dialog Box creation, the Dialog Box will be displayed.

In order to create the Dialog Box, additional information is required and is stored in the parameter arrays. The integer array must contain the following information:

- Element one contains the minimum Y coordinate of the Dialog Box
- Element two contains the minimum X coordinate of the Dialog Box
- Element three contains the maximum Y coordinate of the Dialog Box
- Element four contains the maximum X coordinate of the Dialog Box
- Element 5 holds the total number of Parts in the Dialog Box. Any Parts added to the Dialog Box during the execution of the program are NOT counted here. The other array elements contain the descriptions of the Parts in the Dialog Box. A Dialog Box may have as many Parts as necessary.
- Element 6 is the identification(ID) number of the Part. This is the value passed back to the **MOUSE** command to inform you what the user response was. ID numbers must be unique for each Part and range from 1 to 255.
- Elements 7 through 10 contain the local (relative to the Dialog Box) coordinates of the Part. The coordinates are expressed in pixels. The coordinates are listed in this order: Y minimum coordinate, X minimum coordinate, Y maximum coordinate, X maximum coordinate.
- Element 11 stores the Part Type Number (see Table 4.4.1).
- Element 12 holds the Part Value (see the specific control).
- Element 13 contains the Part Flag (see the specific control).
- Element 14 holds the element number of the string array storing the label. Use zero if no label is required.

To define other Parts in the Dialog Box, repeat elements 6 through 14 in subsequent array elements for each Part until the number of Parts specified in element 5 are satisfied. All Dialog Box Parts will appear in the reverse order in which they were defined. Thus, the Part defined with the highest ID number will appear first and the one with the lowest ID number will appear last.

Next, the elements of the string array storing the labels must be defined. These array elements contain the strings referenced in the integer array as specified in element 14 (and subsequent integer array elements).

Example:

```
PROC DialogBox1 {Define Dialog Box}
   {0 = Close, 1 = Create Dialog Box}
   Array% (0) = 1          {Create the Dialog Box}
   Array% (1) = 99         {Y height of Dialog Box}
   Array% (2) = 319        {X width of Dialog Box}
```

```
Array% (3) = 50          {Y height of Dialog Box}
Array% (4) = 150         {X width of Dialog Box}
Array% (5) = 2           {Number of Parts on panel}
{Part1 definition}
Array% (6) = 1           {Part ID number}
Array% (7) = 27          {Y min start position}
Array% (8) = 2           {X min start position}
Array% (9) = 198         {Y max start position}
Array% (10) = 317        {X max start position}
Array% (11) = 15         {Part type: StatText}
Array% (12) = 0          {Part value: unused}
Array% (13) = 0          {Part flag: unused}
Array% (14) = 1          {String array # for "Dialog Box"}
{Part2 definition}
Array% (15) = 2          {Part ID number}
Array% (16) = 27         {Y min start position}
Array% (17) = 2          {X min start position}
Array% (16) = 0          {Y max start position}
Array% (19) = 0          {X max start position}
Array% (20) = 10         {Part type: Push Button}
Array% (21) = 0          {Part value: unused}
Array% (22) = 0          {Part flag: single/round}
Array% (23) = 2          {String array #2 "OK"}
{Part names}
Array$ (1) = "Dialog Box"
Array$ (2) = "OK"
{End of definition of the Dialog box}
DIALOG (Array% (, Array$ ( )
```

After the Dialog Box is created, you may wish to fetch the Dialog Port handle returned in integer array elements zero and one of the integer array passed to **DIALOG**. The low part of the handle is in array element zero; the high part of the handle is in array element one. Be certain to save this handle if you wish to make ToolBox calls later.

Example: <Program fragment>

```
Array% (0) = 1
{Create Dialog box}
DIALOG (Array%(, Array$( )
{Get the Dialog Box GrafPort handle}
DlgHdle_Low = Array% (0) {Low Handle}
```

```
DlgHdle_High = Array% (1) {High Handle}
```

IMPORTANT

> If you are creating a Dialog Box, the Dialog Box will not be displayed until the **MOUSE** command, monitoring the Dialog Box, is executed.

Add a Part to a Dialog Box (2)

A Dialog Control Number of two is designed to add another Part to a Dialog Box besides those that were included when the Dialog Box was created. The integer array passed to **DIALOG** must have the following entries:

Element	Part Type
1	Part ID Number
2	Minimum relative Y coordinate of Part
3	Minimum relative X coordinate of Part
4	Maximum relative Y coordinate of Part
5	Maximum relative X coordinate of Part
6	Part type (see Tables 4.4.1)
7	Part flag (see specific Part)
8	Part value (see specific Part)
9	Element number in string array containing text display.

Remove a Part from a Dialog Box (3)

If element zero of the integer array contains a three, the Part number assigned to element one of the integer array will be removed from the current Dialog Box.

Example:

```
Array% (0) = 3 {Remove Dialog Part}
Array% (1) = 3 {Part ID number to be removed}
DIALOG (Array% (, Array$( )
```

Enable a Part in a Dialog Box (4)

If element zero of the integer array passed to **DIALOG** contains a four, the Part with the number assigned to element one of the array will be enabled. Thereafter, the user will be able to respond to this Part.

Example:

```
Array% (0) = 4 {Enable Part in box}
Array% (1) = 3 {Part ID number to be enabled}
```

```
DIALOG (Array% (, Array$( )
```

Disable a Part in a Dialog Box (5)

If element zero of the integer array contains a five, the Part with the number assigned to element one of the integer array will be disabled. Thereafter, the user will not be able to access this Part.

Example:

```
Array% (0) = 5 {Disable Part in box}
Array% (1) = 3 {Part ID number to be disabled}
DIALOG (Array% (, Array$ (·)
```

Programmers

> *Micol Advanced BASIC* uses the standard Parts defined by the Control Manager, LineEdit, and QuickDraw II Tool sets. The Control Manager directs standard controls. The LineEdit Tool manages edit lines. QuickDraw II displays the StatText Parts.

Monitoring the Dialog Box

MOUSE (Integer_Array ( )

Once you have defined a Dialog Box, you must monitor it for a user response. Like monitoring the response to a Menu and Window, the **MOUSE** command is used to monitor Dialog Boxes. Unlike monitoring a Window or a Menu however, **MOUSE** is not placed within a loop to monitor a single response from the user; **MOUSE** will not return control to your program until the user has responded to the Dialog Box. Once the user has responded to the Dialog Box, the Part ID number (defined by you in the **DIALOG** command) will be returned in element zero of the integer array passed to **MOUSE**.

When monitoring the Dialog Box with **MOUSE**, the Part ID Number received in element zero of the integer array passed to **MOUSE** is used to direct the program flow to the program code handling the action for that control. The Dialog Box is displayed until the user selects the close button and the program closes the Dialog Box.

Example:

```
{The Dialog Box is previously defined}
DialogExit! = FALSE
REPEAT {Display until True}
   MOUSE (EventRecord% ( )
   ItemID% = EventRecord% (0)
   CASE_OF ItemID%
     DO 1 {OKAY Button}
```

```
            DialogExit! = TRUE
        ENDDO
        DO 3, 4, 5 {Check Boxes}
            {GetDitemValue}
            TOOLBOX (21,46:0,t1,t0,ItemID;ItemValue)
            {ItemValue = 0 if CheckBox is not checked,
                  ItemValue = 1 if CheckBox is checked.
                  If ItemValue = 0 then ItemValue = 1
                  If ItemValue = 1 then ItemValue = 0 }
            ItemValue = ABS (ItemValue - 1)
            {SetDitemValue}
            TOOLBOX(21,47:ItemValue,t1,t0,ItemID)
        ENDDO
    ENDCASE
UNTIL DialogExit! {Radio_Buttons}
```

See the demonstration program DIALOG to see how to use the Dialog Control Numbers and CONTROLS to see how to define the different types of controls. These example programs are on the MAB.SUPPORT disk in folder Desktop.Samples/.

Part Five: The Apple IIGS ToolBox

Chapter One

Direct Toolbox Access

Overview

This chapter demonstrates how to use *Micol Advanced BASIC*'s **TOOLBOX** command to get direct access to the Apple IIGS ToolBox.

Defining the ToolBox

What is the ToolBox? The ToolBox is a series of routines designed to perform specific tasks. Each particular task, like memory management or graphics, is divided into a specific Tool. Each Tool is given a unique ID number. Within each Tool are specific Functions which perform individual tasks. Each Function within a particular Tool set is also designated by a unique ID number.

For example, Tool number two is the Memory Manager and QuickDraw II, which does Super High Resolution graphics, is Tool 4. Function number two within each Tool starts up the Tool, while Function 15 in QuickDraw II returns the contents of a color table.

Books describing ToolBox functions will be necessary to write *Micol Advanced BASIC* programs that use the Apple IIGS Toolbox. The list of Tools appears in Table 5.2.1 and 5.2.2. (As this manual goes to press, thirty-three tool sets have been defined; of these, thirty-two sets were released in ROM and the IIGS System Disk v5.04.)

The Universal TOOLBOX Command

TOOLBOX (ToolNum, FuncNum [:Push List] [;Pull List])

The **TOOLBOX** command is designed to call virtually any Tool and consists essentially of four parts. The first part consists of the Tool Number. The second part is the Function number within the Tool. The third part, the Push List, consists of the parameters required by the Tool Function, and the forth part, the Pull List, is a set of integer variables which will contain the values returned by the Tool Function itself.

The first argument, the Tool Set Number, is an integer literal (expressed in decimal or hexadecimal) or integer variable.

The second argument, the Tool Function Number, is an integer literal (expressed in decimal or hexadecimal) or integer variable. Any value greater than 255 ($FF) generates an error.

An optional list of integer literals (in decimal or hexadecimal) or integer variables, separated by commas, follows the Function Number. This list is separated from the Function number by a colon (:) and is a set of values that will be pushed onto the ToolBox stack.

Last comes an optional list of integer variables, separated by commas, which will contain the values returned by the Tool Function. A semi-colon (;) precedes this list.

Most Tool Functions require values to be placed onto the ToolBox stack before the Function can be used. These values are often a four-byte memory location which may be represented within two two-byte integer literals or variables. Toolbox reference manuals list the values to be pushed or pulled from the stack in terms of words and long words. A word is equivalent to an integer, and a long word is equivalent to two integers. For the most part with the **TOOLBOX** command, long integers are treated like short integers.

Determining the Tool and Function Numbers

In any ToolBox reference manual, a Function is referenced in terms of a name and a call number. This call number is used by assembly language programmers to make the call to the Tool Function.

This call number is a hexadecimal number with the Function number and Tool number appended. You can easily unappend this number to determine the Tool number and Function number for use with the **TOOLBOX** command.

Let's say that the call number of a particular Tool Function is $2C03 (SysBeep). The dollar sign simply means that the number is hexadecimal. The next two characters in the number are the hexadecimal value of the Function Number, or $2C (44 decimal). The following two letters are the hexadecimal value of the Tool Number or $03 (3 decimal). Therefore, in this example, the Tool number is three, and the Function number is 44. Please note that **TOOLBOX** can accept either decimal or hexadecimal numbers, so there is no need to make any conversions.

Example:

```
TOOLBOX (03, 44) {SysBeep $2C03}
```

T' e Push List

The Push List of the **TOOLBOX** command is used to pass values to the particular Tool Function. Every description of a Tool Function will describe what values have to be pushed unto the stack. These are the values that go into the **TOOLBOX** Push List.

Values contained within the Push List will consist of either integer literals or integer variables. An integer is the same as a word that needs to be pushed onto the ToolBox stack. Two short integers must be used to equal a long word.

Be very careful when specifying values for the Push Stack. There is a one-to-one correspondence between values that are contained in the **TOOLBOX** command, and values that are pushed onto the ToolBox stack as defined in any ToolBox reference manual. **TOOLBOX** pushes values onto the stack in the order they are specified.

Example:

```
Colour = 15 {Clear to White}
TOOLBOX (04, 22: Colour) {ClearScreen $1504}
```

The Pull List

Many Tool Functions return values after their work is done. The values from the Tool Function will be returned in the integer variables in the Pull List in the order they were defined within the **TOOLBOX** command.

Example:

```
Rnd_Num% = 0
TOOLBOX (04,134: 0; Rnd_Num%) {Random $8604}
```

Example:

```
XCoord% = 240
YCoord% = 120
Pixel% = 0
{GetPixel $8804}
TOOLBOX (04,136: 0, XCoord%, YCoord%; Pixel%)
```

Error Checking

If the call to the Tool Function is successful, zero wil be returned in **True_Value** (locations 202 and 203). If the Tool Function should return an error, the error value will be returned in **True_Value** in LSB, MSB order. MSB (location 203) will be the Tool Number which returned the error, not always the Tool that was called.

PEEK location 202 to determine if the call was successful. If the value is zero, the call was successful, otherwise you should be able to determine the problem.

TOOLBOX and Long Integers

The **TOOLBOX** command will use long integers when the **LONGINT** compiler option is specified. Only two bytes (one word) from the long integer's value will be used, usually the two least significant. The Greater Than symbol (>) may be used to reference the most significant word in the long integer; if the long integer is storing an address, this is the bank number of the address.

Example: <Program fragment>

```
PROGRAM Show_LongInt
@ LONGINT
{Get the address of the pointer}
Address% = ADDR (Address% ( )
TOOLBOX (14, 19: >Address%, Address%) {ShowWindow $130E}
```

Long integers may be used in this manner only with the **TOOLBOX** command.

Future ToolBox Additions

The **TOOLBOX** command is a very versatile command. It was designed to let you take advantage of present and future Tool sets to come from Apple Computer Inc. or from third-party companies.

Allocating ToolBox Buffers

Many ToolBox calls need the use of a supplied buffer to accomplish their tasks. Essentially, there are two methods you may use to fetch memory for a ToolBox call.

The more difficult method to fetch this memory is to either use a **TOOLBOX** call to the Memory Manager, or the **GET_MEM** command. This method needs to be used if you are fetching Direct Page memory for the Tool, otherwise, use the easier method.

We will outline the easier method here. Allocate some memory within a dummy array with the **DIM** statement. Be certain this array is large enough for its work and do not use it for any other purpose, as internal values may become corrupted by the ToolBox call. The **ADDR** command may then be used to get the address of this dummy array, and this address may then be passed to the Tool call.

The following example program below should answer any questions you may still have about the **TOOLBOX** and **ADDR** commands. The example below uses Miscellaneous Tool (Set 03) to read the time and date as an ASCII representation, and then displays this time and date to the screen.

Example (line numbers are for reference only):

```
1.   PROGRAM Display_Time
2.   @ LIST
3.   DIM ToolBox_Buf% (10)
4.   Adr% = ADDR (ToolBox_Buf% ( )
5.   Bank_Number% = PEEK (202)
6.   TOOLBOX (3, 2) {Turn on Misc. Tools}
7.   {ReadASCIITime $0E03}
8.   TOOLBOX (3, $0E: Bank_Number%, Adr%)
9.   Adr& = ADDR (Toolbox_Buf% ( )
10.  HOME
11.  PRINT "The Current time is ";
12.  FOR Loop_Ctr = Adr& TO Adr& + 19
13.       Char = PEEK (Loop_Ctr)
14.       Char$ = CHR$ (Char)
15.       PRINT Char$;
16.  NEXT Loop_Ctr
```

```
17.   PRINT
18.   {Turn off Misc Tools}
19.   TOOLBOX (3, 3)
20.   END
```

How this program works...

1. At line 3, a 23 byte buffer is defined for the ToolBox output. This buffer should never be used as an array.
2. At line 4, the address of the ToolBox buffer is determined as an integer value.
3. At line 5, the bank number of the ToolBox buffer is obtained.
4. At line 6, the Miscellaneous Tool set is started.
5. At line 8, the ReadASCIITime function call from the Misc. Tools is done by pushing the bank number and the address of the buffer onto the ToolBox stack as required by this Tool Function.
6. At line 9, the full address of the ToolBox buffer is determined as a real value. We need the real address to be able to **PEEK** the Tool Function results from memory.
7. At lines 12 - 16, the time and date returned by the Tool are displayed. Note the use of **PEEK** within the loop. Because of the way *Micol Advanced Basic* parses a statement, we cannot use an integer variable at line 13, because, if we did, **PEEK** would try to take the integer value of **Loop_Ctr**, a value too great for an integer, and we would therefore receive an error.
8. At line 19, the Misc. Tools is shut down. It is wise to always wait until the end of your programs before performing any shutdowns as these Tools may also be used by the run time routines.

Chapter Two

Tool Set Tables

Some of the Tools listed below are used automatically by the run time routines: do not start them up or shut them down using the **TOOLBOX** command. Use the **TOOLBOX** command to start or shut down the tools that have no *Micol Advanced BASIC* equivalent or are not started by a run time routine (if the respective command is not active in your program).

Startup notes: You must observe the following:

- The following Tools are always necessary and should never be shut down by your program: Tool Locator, Misc. Tools, Memory Manager, Text Tools, and SANE.
- The Desktop Tools (Event Manager, Line Editor, Window Manager, Menu Manager, Dialog Manager, Scrap Manager, and Control Manager) are activated when one of these *Micol Advanced BASIC* commands is executed: **DIALOG**, **MENU**, or **WINDOW**.
- The Sound Tool Set is activivated by the **NOISE** command.
- The Sound Tool Set and the Note Synthesizer Tool are activated by the **MUSIC** command.
- QuickDraw II is started by the **HGR** or **HGR2** command.

Shutdown Notes: As a general rule, shutdown of the Tools is done in the reverse order as they were started up. You must observe the following:

- The Tools started by the Library routines are all deactivated and memory is deallocated by the **END**, **STOP** or **BYE** command.
- The Desktop and Graphics Tools are deactivated by **TEXT**.
- The Sound Tool Set and the Note Synthesizer Tool are both deactivated by the **SILENCE** command.
- The memory necessary for the Tool is deallocated automatically when the Tool is shut down.

The tables in the following pages enumerate the Tools, their startup, shut down order, and the memory needed by each set.

Legend:

- "AN" means "After N". N is a Tool Number that indicates that the tool with that number must be started before this one.
- (d) indicates a Desktop command (**MENU**, **WINDOW** or **DIALOG**)

Table 5.2.1 Startup List

Tool #	Tool Set Name	Started by	Order
1	Tool Locator	Library	1
2	Memory Manager	Library	2
3	Miscellaneous Tool	Library	3
4	QuickDraw II	HGR or HGR2	4
5	Desk Manager	(d)	18 (Last)
6	Event Manager	(d)	5
7	Scheduler	Library Routine	A3
8	Sound Tool Set	NOISE	A21
9	Apple Desktop Bus	Library Routine	A1
10	SANE	Library Routine	A2
11	Integer Math	Library Routine	A1
12	Text Tools	Library Routine	A1
13	Internal Use	Cannot Be Called	NA
14	Window Manager	(d)	6
15	Menu Manager	(d)	8
16	Control Manager	(d)	7
17	System Loader	Do Not Call	NA
18	QuickDraw II Aux.	TOOLBOX (18,02)	A4
19	Print Manager	TOOLBOX (19,02)	15
20	Line Edit	(d)	9
21	Dialog Manager	(d)	10
22	Scrap Manager	(d)	12
23	Stnd File Operation	TOOLBOX (23,02)	11
24	(Not Defined)		
25	Note Synthesizer	MUSIC	A21
26	Note Sequencer	TOOLBOX (26,02)	A21
27	Font Manager	TOOLBOX (27,02)	14
28	List Manager	TOOLBOX (28,02)	13
29	ACE Tool	TOOLBOX (29,02)	A2
30	Resource Manager	System Loader A1	
31	(Not Defined)		
32	MIDI Tool	TOOLBOX (32,02)	A21
33	Video Overlay Card	TOOLBOX (33,02)	A5
34	Text Edit Tool	TOOLBOX (34,02)	A6

Table 5.2.2. Shutdown List

Tool #	Tool Set Name	Shutdown by	Order
1	Tool Locator	Library Routine	29 (last)
2	Memory Manager	Library Routine	25
3	Miscellaneous Tool	Library Routine	21
4	QuickDraw II	TEXT	19
5	Desk Manager	TEXT (if started as (d)	First
6	Event Manager	TEXT if started as (d)	17
7	Scheduler	Library Routine	20
8	Sound Tool Set	SILENCE	11
9	Apple Desktop Bus	Library Routine	28
10	SANE	Library Routine	24
11	Integer Math	Library Routine	27
12	Text Tool set	Library Routine	20
13	Internal Use	Cannot Be Called	NA
14	Window Manager	TEXT if started by (d)	16
15	Menu Manager	TEXT if started by (d)	14
16	Control Manager	TEXT if started by (d)	15
17	System Loader	Do Not Call	NA
18	QuickDraw II Aux.	TOOLBOX(18,03)	18
19	Print Manager	TOOLBOX(19,03)	3
20	LineEdit Tool	TOOLBOX(20,03) (d)	13
21	Dialog Manager	TOOLBOX(21,03) (d)	12
22	Scrap Manager	TOOLBOX(22,03) (d)	6
23	Stnd File Operation	TOOLBOX(23,03)	7
24	(Not Defined)		
25	Note Synthesizer	SILENCE	10
26	Note Sequencer	TOOLBOX(26,03)	9
27	Font Manager	TOOLBOX(27,03)	4
28	List Manager	TOOLBOX(28,03)	5
29	ACE Tool	TOOLBOX(29,03)	23
30	Resource Manager	System Loader	22
31	(Not Defined)		
32	MIDI Tool	TOOLBOX(32,03)	8
33	Video Overlay Card	TOOLBOX(33,03)	A5
34	Text Edit Tool	TOOLBOX(34,03)	2

Table 5.2.3 Tool Sets Direct Page Memory Requirements

Tool #	Tool Name	Direct Page Allocation
04	QuickDraw II	Yes, 768 bytes
06	Event Manager	Yes, 256 bytes
08	Sound Tool Set	Yes, 256 bytes
10	SANE Tool Set	Yes, 512 bytes
14	Window Manager	No, Shares Tool 06 Direct Page
15	Menu Manager	Yes, 256 bytes
16	Control Manager	Yes, 256 bytes
18	QuickDraw II Auxiliary	No, Shares Tool 04 Direct Page
19	Print Manager	Yes, 512 bytes
20	LineEdit Tool	Yes, 256 bytes
21	Dialog Manager	No, Shares Tool 16 Direct Page
23	Standard File	Yes, 256 bytes
25	Note Synthesizer	No, Shares Tool 26 Direct Page
26	Note Sequencer	Yes, 768 bytes
27	Font Manager	Yes, 256 bytes
29	ACE Tool	Yes, 256 bytes + memory for buffers
31	(Not Defined)	
32	MIDI Tool	Yes, 768 bytes + memory for buffers
33	Video Overlay Card	Unknown
34	Text Edit Tool	Yes, 256 bytes

Part Six: Program Management

Chapter One

Program Debugging

Overview

This chaper is designed to help you debug your programs.

What Is Debugging? Debugging is the act of finding errors within a program.

In general, two classes of errors can occur in a program; syntax errors and logic errors.

Syntax errors occur when the syntax rules of the language are violated and are caused mainly by typing errors or by a misunderstanding of the rules of the language. These errors are almost always very easy to solve and will not concern us here.

Logic errors are much more difficult to determine than syntax errors and occur when a program is not properly designed to solve the problem in question. Logic errors cause the program to give different results and/or behave differently than what was expected.

No language system can find such logic errors because no language system can do what a human can do, think. The most a language system can do is to give the programmer some tools to help him/her find these logic errors. This is what *Micol Advanced BASIC* does and this is the subject of this chapter.

Debugging Statements

Often, a variable has a different value than is intended, or an area of code has executed when it should not have executed, or vise-versa.

Programs do exactly what you tell them to do; they do not do what you think you tell them to do. This is very often the cause of logic errors; the programmer has told the computer to do something other than had been intended. Do not assume that any code is automatically correct, this is a big mistake.

Another cause of logic errors is that the programmer has devised an incorrect solution to the problem. The program operates as intended, but incorrect results are coming out. This is a more serious problem, and more difficult to solve. Once the problem is located, the code must be rewritten.

The following statements are designed to help inform you where you are going wrong; they cannot find the problems themselves. Use these commands wisely, and your job will be a lot easier.

BELL

BELL can be a good tool to help you find your logic errors. Just place **BELL** in the section(s) of code where the program seems to be malfunctioning. If the speaker beeps when it should not or fails to beep when it should, a bug may have been found in the program. The beep gives you an aural message telling that something may be wrong.

Example:

```
IF PEEK (202) = 2 THEN BELL
```

PRINT

Insert a **PRINT** statement at strategic points in the program to determine what the contents of a particular variable are.

Example:

```
Alpha% = PEEK (True_Value)
PRINT "Alpha% = "; Alpha
```

STOP

STOP halts the program's execution, prints the line number where the program halted, and returns control to the Command Shell while using the programming environment.

Line number information can be valuable information in debugging as it is sometimes the case that a particular line should or should not be executing at a certain point in the program's execution. Then it's necessary to trace the logic in your program to determine why the program flow got to where it did.

This is what is known as setting a break point, and is the most frequently used debugging technique in assembly language programming. Break points may also be useful in high level debugging.

STOP may be placed anywhere in a program as it closes all text files currently open and sets the screen to text mode.

Example:

```
Variable = 3
IF Variable = 3 THEN STOP
```

TRACE

TRACE will print the sequential line numbers of the program as the line or structured loop statement is executing. Tracing a program's flow can be a great aid in determining the program's actual logic.

TRACE may be placed anywhere in a program and follows the flow of execution used

in the program.

To use **TRACE**, place it before the location from which you wish to begin the trace of your program. Any code executing before **TRACE** will not be displayed.

The tracing may be paused by pressing any non-Control character. Restart the tracing by pressing any non-Control character again.

WARNING

> Do not use the **OPTIMIZ** compiler option as it hinders the generation of line information required by **TRACE**.

Example:

```
PROGRAM Try_Trace
PRINT "This program will be traced"
HOME
TRACE
FOR Number% = 1 TO 4
    PRINT "Number% = ";Number%
NEXT Number%
NOTRACE {Turn off the TRACE}
END
```

STRACE

STRACE stands for SuperTRACE. **STRACE** will print the sequential line numbers of the program and the text of the line that is executing to the current output device as the line is executed.

To use the **STRACE** command, place it before the location at which you wish to begin the trace of your program. Any code executing before the **STRACE** will not be displayed. **STRACE** may be placed anywhere in a program

STRACE follows the flow of execution used in the program; so the lines will not be shown consecutively.

The tracing may be paused by pressing any non-Control character. Restart the tracing by pressing any non-Control character again.

WARNING

> **STRACE** takes the text it displays from the program currently in the Text Editor. This means that the program you wish traced must be in the Text Editor, which is normally the case. Do not use the **OPTIMIZ** compiler option.

Example:

```
PROGRAM Try_STRACE
STRACE {Turn on the STRACE}
HOME
FOR Number% = 1 TO 2
   PRINT Number%
NEXT Number%
NOTRACE
PRINT
PRINT "This program has been traced"
END
```

The above program produces something like this on the screen:

```
/<3>HOME\
/<4>FOR Number% = 1 TO 2\
/<5>  PRINT Number%\
1
/<6>NEXT Number%\
/<4>FOR Number% = 1 TO 2\
/<5>PRINT Number%\
2
/<6>NEXT Number%\
/<7>NOTRACE\
```

NOTRACE

NOTRACE turns off the effects of a **TRACE** or **STRACE**. The number of the line and its text of code will no longer appear after **NOTRACE** is executed.

Example: (see example under **TRACE** and **STRACE**.)

Chapter Two

Program Optimization

Overview

This chapter discusses some simple tricks to help you maximize the speed of your programs while at the same time minimizing the program size.

Saving Memory

Because of the large amount of memory available to the Apple IIGS, you may never have to worry about memory used for either programs or data. Under *Micol Advanced BASIC*, a program may have a maximum of over one million bytes and data space is limited only by the available memory in the machine.

However, if you have a system with only 768K, this section may be of use to you. Consider buying more memory as soon as your finances permit.

Generally, the tricks to help save memory are the same as in Applesoft BASIC.

Working within the Editor's Workspace

The text editor has enough work space for about 4000 lines of code. Use **INCLUDE** or **CHAIN** in the program if the program exceeds 3000 lines.

Saving Space in a Program

- Use the **OPTIMIZ** compiler option once your program is free of bugs; this can shrink your programs as much as one-third. If limited space is a problem during program development, you may use this compiler option to save memory, but determining where run time errors occur will become much more difficult.
- Avoid the use of the **ERROR** compiler option. The only function this compiler option has is in regards to the **RESUME** command, but **ERROR** causes a significant amount of code generation. You will have to handle your error recovery in a different fashion.
- Analyze your programs for repeated code. It may be possible to create one subroutine that will do the work of several portions of your program.
- Use arrays as rarely as possible. If you must use arrays, use integer arrays whenever possible. Do not make arrays any larger than you have to.
- Avoid **DATA** statements. **DATA** statements require significant memory. Data may just as well be stored on disk and recalled at run time.
- Do not use the **EXTEND** or the **LONGINT** compiler options if very large arrays

are used within your program.

- Avoid mixed arithmetic. Mixing reals and integers within a statement forces the Compiler to generate extra code, code that may possibly be avoided.
- As with any programming language, code efficiently.

Speeding Up Your Programs

Certain methods may be used to make a program execute more quickly. Some of the tips mentioned above apply here too.

- Make use of the **OPTIMIZ** compiler option as soon as your program is completely free of bugs. The code required for debugging purposes usually takes significant time to execute. Once your program is debugged, this code no longer has a useful purpose and may be eliminated.
- Do not mix your arithmetic. If calculating in real, be consistent with real; likewise for integers.
- Use integer variables whenever practical. *Micol Advanced BASIC* has its own built-in integer routines. The average increase in speed over real arithmetic may be as great as 300%.
- Use arrays wisely. Some time is needed at run time to calculate the address of the array element. However, if you have an algorithm which is faster than another and uses arrays, feel free to use them
- Avoid disk access as much as possible. If you have frequent disk access with the same file(s) being read again and again and you also have a lot of available memory, make use of a RAM disk together with the **COPY** command to transfer the files from a static disk to the RAM disk before your program reads these files.

Chapter Three

Program Segmentation

Overview

This section shows how to segment both source code and executable load modules under *Micol Advanced BASIC* and how to conceive large programs which would otherwise be very difficult to do.

Chaining Source Code Files

For very large programs, it may be necessary to segment your source code into two or more portions in order to manage the source code within the Text Editor. *Micol Advanced BASIC* has two methods to allow you to segment your program code: chaining text files, and creating a library of modules. Because the creation of library modules has been discussed in Part Three, Chapter 9 in this manual, it will be only briefly discussed here.

Segmenting the Source Code Files

In order to segment the source code file, you must first decide where you can logically break the program. You must make every attempt to keep subroutines intact.

Using the Text Editor, break this large program into several smaller source code files. To be safe, keep the original file safe just in case something goes wrong.

Then, simply terminate each source code segment, except the last, with a **CHAIN** statement, using the next source code filename as the **CHAIN** string parameter.

The second, and subsequent source code file(s) begin without a **PROGRAM** statement. The next file finishes with an **END** or with another **CHAIN** if another file is to be chained.

CHAIN String_Literal

The **CHAIN** statement must be the only statement on the line. It should be the last statement in the file: any subsequent line(s) of code following the **CHAIN** statement will be ignored by the Compiler.

String_Literal must be the Pathname of the source code file you wish to compile after the previous source code has finished compiling. The only accepted parameter to **CHAIN** is a string literal; a string variable will be rejected by the Compiler.

The file referenced must be online at the time of compilation, otherwise the appropriate operating system error will occur.

The Compiler displays the message "Chaining <Pathname>" before it starts reading

the file to be chained.

Example:

```
(Contents of file: Chain1)
PROGRAM Chain_Example
@ LIST
FOR Ctr% = 1 TO 10
   PRINT Ctr%
NEXT Ctr%
CHAIN "/RAM5/Chain2"

(Contents of file: /RAM5/Chain2)
FOR Ctr% = 11 TO 20
   PRINT Ctr%
NEXT Ctr% {End of chained program}
END
```

How to Debug a Chained Program

The Compiler does not number the lines of a segmented chained program the same way the Editor does; the Text Editor always begins numbering from the first line in the editor buffer.

During compilation, the chained file is treated as if it were a part of the previous file. This means that the sequential line numbers continue uninterrupted. If an error with a specific line number within a chained file occurs during execution, you will have to recalculate its editor line number to be able to correct the problem in the Editor. The same situation is true of syntax errors.

Consider using the **INCLUDE** statement as an alternative method of compiling large source code files. See Chapter Nine in Part Three for additional information.

Segmenting Executable Code Files

Micol Advanced BASIC programs may be a maximum of about one megabyte of memory. However, because of a limitation of the Apple IIGS microprocessor, a single program code segment may only occupy a maximum size of 64 kilobytes (one bank) of memory. The variable space is separate and is not affected by this limitation.

The vast majority of programs will not require a program space larger than 64 kilobytes, however, some will. *Micol Advanced BASIC* overcomes this 64K limitation with the creation of executable program segments.

A segment is a large, self-contained program section. The segment is used to break a program when it can no longer fit in one memory bank (64K).

A program using segments need not be broken up into different program files as could be expected, but, space permitting, may be stored in a large continuous file.

The Compiler resets its internal program counter to zero when it compiles a new segment and generates code that will cause the Loader to load this program segment as a separate entity and store its loading location in a suitable location.

How to Segment a Program

A *Micol Advanced BASIC* program may have up to sixteen segments sequentially numbered from 0 to 15. Each segment is given a sequential value by the Compiler; take note of this number as it will be needed to start the execution of the segment. The Compiler will display a message when it recognizes a new segment, and will display the number given to the segment.

Each segment shares the entire variable space with all other segments, so any change in global values in one segment will also be recognized in the others.

The executable code of segment zero (the starting segment) should not be larger than 50K to leave room for maintenance. All segments should leave room for enhancements of current features, and additions of new features. If the segment is too large for a single memory bank, the Linker will report that program space is exceeded.

The first segment containing the line **PROGRAM** Identifier is segment zero; execution always begins with segment zero. Compiler directives, declaration of literal data, type identifiers, and **DIM**ension statements may appear only in segment zero.

Because one segment cannot have direct access to the Functions, Procedures and Routines of another segment, all segments must be self-contained. Functions, Procedures and Routines necessary to one or more segments have to be duplicated in each of the segments needing them.

You may use the **CLEAR** statement only in segment zero; otherwise the program may crash.

SEGMENT [Identifier]

The **SEGMENT** statement forces the Compiler to segment a program. It must be the first and only statement on a line of code.

SEGMENT may have an optional segment identifier. The segment identifier should be a digit or word which describes the segment and is designed to help in documentation; it is ignored by the Compiler.

The keyword **SEGMENT** signals the end of the preceding segment and the start of the new one. When the Compiler encounters the reserved word **SEGMENT**, it generates code which will inform the Micol system loader to load that segment of code as a separate entity, and to set the program counter of the microprocessor accordingly.

Using a Segmented Program

Like a Function or Procedure, a segment will not execute by letting the program flow reach the **SEGMENT** statement; the segment must be called using **CALL**.

CALL Segment_Number

The **CALL** statement branches the program flow to the segment number indicated. Segment_Number must be a digit between 0 and 15. If the segment number is not of this range, an error will be signaled during compilation.

WARNING

> Do not use the **CALL** statement from a Function, Procedure, or Routine, as the maintenance done at the end of the modules has not been completed.

Example:

```
PROGRAM Segment_Ex1
{This is segment zero}
IF Counter% = 0 THEN HOME {Want only one HOME executed}
IF Counter% < 5 THEN BEGIN
   PRINT "Counter = "; Counter%
   PRINT "Start of segment zero"
   FOR Ctr% = 1 TO 100
     PRINT Ctr%
   NEXT Ctr%
   CALL 1 {Going to Segment One}
ENDIF
END
SEGMENT One
   PRINT "Start of segment one"
   FOR Ctr% = 101 TO 200
     PRINT Ctr%
   NEXT Ctr%
   CALL 2 {Going to Segment Two}
SEGMENT Two
   PRINT "Start of segment two"
   FOR Ctr% = 201 TO 300
     PRINT Ctr%
   NEXT Ctr%
   Counter% = Counter% + 1
   CALL 0 {Going to top of program}
END
```

This program simply prints the segment number followed by the sequential values

five times. Note that the counter is incremented in the final segment, and the test is done in the program segment.

If another segment has been called using the **CALL** statement, and the program must return to the statement following the original **CALL**, use **LRETURN**.

LRETURN

LRETURN (for Long **RETURN**), similar to **RETURN**, instructs the program to return to the statement following the **CALL** statement that called this segment. Only one **LRETURN** statement should appear in a segment.

WARNING

> **LRETURN** must never be used to end a Function, Procedure or Routine as unexpected results will occur.

Unlike **RETURN**, no automatic error handling is done with **LRETURN**, so be certain there is a segment to return to.

Example:

```
PROGRAM Seg_Example2
{This is segment zero}
IF Counter% = 0 THEN HOME
IF Counter% < 5 THEN BEGIN
   PRINT Counter% = ;Counter%
   PRINT "Start of segment zero"
   FOR Ctr% = 1 TO 100
     PRINT Ctr%
   NEXT Ctr%
   CALL 1
ENDIF
END   {Terminate each segment with END or LRETURN}
SEGMENT One
   PRINT "Start of segment one"
   FOR Ctr% = 101 TO 200
     PRINT Ctr%
   NEXT Ctr%
   Counter% = Counter% + 1
LRETURN {Go back to Segment Zero AND finish}
END {End of program}
```

Chapter Four

Linking Assembly Language Programs

Overview

Sometimes, a specific task cannot be performed by a higher level language or even greater speed is needed than is possible in a higher level language.

In these cases, a good solution is to integrate (or link) an assembly language module into your program. Under *Micol Advanced BASIC*, it is very easy to link in machine language programs you have developed.

Linking in the Assembly Language Program

LINK PathName

The **LINK** statement links in the assembly language program specified by PathName. PathName must be a string literal and is the complete Pathname of the assembly language file to be linked. The file must be online at compilation time. If it cannot be found, the Compiler will signal an error. The assembly language program must be already assembled and error free.

The Compiler will indicate, "Linking file" Pathname when it is linking a *Micol Macro* file into a *Micol Advanced BASIC* program.

Example:

```
LINK "/Assm.Prog.Util/ClrScreen.B"
```

IMPORTANT

> *Micol Advanced BASIC* for the GS uses assembly language files of type MCL ($F1) written with the *Micol Macro* GS assembler **only**. If you do not have this assembler, then you may purchase one directly from us.

How to write an assembly language program to be linked into a *Micol Advanced BASIC* program:

1. Write the assembly language program as required:
   a. Save all the registers at the start of the assembly language source code. Note that in *Micol Advanced BASIC*, the CPU is in 8 bit accumulator mode and 16 bit X and Y registers mode
   b. At the end of your assembly language code, restore all the registers to the values they had before the start of the assembly language routine

c. Do Not use an **RTL** ($6B) or **RTS** ($60) instruction to end the program; just let the assembly language code "fall" through. The *Micol Advanced BASIC* program will resume on its own

d. Thoroughly test this program for any errors.

2. Link the assembly language file into the *Micol Advanced BASIC* program:

a. Using the **LINK** statement, link this assembly language module into your *Micol Advanced BASIC* program where it is required. We recommend allocating a special Procedure for the assembly language module

b. Remember, it is the MCL file (type $F1) which gets linked in, not the assembly language source code text file.

Getting a Direct Page

An area of 256 ($100) bytes in memory bank zero is reserved for use as a Direct Page for your assembly language programs and is placed directly above the one used by *Micol Advanced BASIC.*

How to Use this Direct Page

Simply add $100 (256) to the current Direct Page Register upon entry of your machine language program and subtract $100 from the Direct Page Register upon exiting your assembly language program.

WARNING

> If you alter the value in the Direct Page register, be certain you reinstate it exactly as it was before or your *Micol Advanced BASIC* program is certain to malfunction.

On the system disk marked /MAB.SUPPORT, in folder Demo.Files/Prg.Examples is a file called LINKDEMO which demonstrates the use of assembly language routines with *Micol Advanced BASIC.* You may want to take a look at this file.

Chapter Five

Creating Independent Programs

Overview

This chapter tells how to take a compiled *Micol Advanced BASIC* program out of the programming environment - making it "stand alone", and execute it with a program launcher such as the Finder or as a turnkey system.

There are many ways to make *Micol Advanced BASIC* programs stand alone. This chapter will explore all the possibilities. Pay special attention if you intend to use your programs outside of the normal *Micol Advanced BASIC* programming environment.

Creating a Startup Disk for Launchable Programs

To create a system disk which you may use with the programs created with *Micol Advanced BASIC*, take the following steps (you may use any suitable GS/OS or ProDOS 8 copy utility (the Finder or Copy II Plus will do just fine):

1. Make an exact copy of the *Micol Advanced BASIC* System Disk (the disk labeled Master Disk). You may change the name of this copied disk, if you wish.
2. From this new copied disk, in folder **Micol.Adv.BASIC/**, remove the files **COMPILER.SHELL, EDITOR, AutoExec** and the **UTILITY** folder. This should only leave the files **Micol.Adv.BASIC** and **LIBRARY** remaining.
3. Move the Finder to this new disk:
   a) Delete the file named **START** under folder **SYSTEM/** of the new disk
   b) Copy the file **Finder** from the **MAB.SUPPORT** disk (under the **SYSTEM/** folder) to the **SYSTEM/** folder of the new disk
   c) On the new disk, rename file **Finder** to **Start.**
4. Lock the files and label your disk with an appropriate name. You should also include the version number of *Micol Advanced BASIC* you are using, as well as the version number of the operating system.

Now, if you boot this new system disk, you may directly launch your *Micol Advanced BASIC* programs by double clicking them with the Finder. Note, that such a disk cannot be booted for program development, as the Loader will be searching for the non-existant files **COMPILER.SHELL** and **EDITOR** in the **Micol.Adv.BASIC** folder.

Hard Disks and Launchable Programs

A normal *Micol Advanced BASIC* program may be easily launched directly from a hard disk using the Finder and need not be converted to a GS/OS application.

What is required to launch *Micol Advanced BASIC* programs from the Finder are:

1. A folder called **Micol.Adv.BASIC** directly under the boot volume. The boot volume is the volume that the operating system, GS/OS, was taken from when your computer was started up (this is probably your hard drive).
2. The files **Micol.Adv.BASIC** and **LIBRARY** taken from the **Micol.Adv.BASIC** folder of the disk labeled Master Disk. These files must reside in the folder **Micol.Adv.BASIC** described in 1.
3. The file **Micol.Icon** from the **Icons** folder from the disk labeled Master Disk. This file must reside in the **Icons** folder of the boot volume.

Now if, after booting, you were to double click a *Micol Advanced BASIC* program icon from the Finder, the Micol Loader will load and begin execution of the separate run time Library and BASIC program as if they were a merged application file.

IMPORTANT

> Utilizing the method described here as opposed to creating S16 files described below, has the major advantage of keeping your compiled programs small and can save a great amount of valuable disk space and speed loading time. The method described here is the recommended method for creating launchable programs.

Stand Alone *Micol Advanced BASIC* Programs

A *Micol Advanced BASIC* program which is completely self-sufficient (converted to a TurnKey system, an S16 application or a CDA) is written just like any other program. The one rule you must follow is that the *Micol Advanced BASIC* program must be thoroughly debugged before you make the conversion. This is because the *Micol Advanced BASIC* Programming Environment is designed for program debugging, and stand alone applications are not. If a bug should appear in a stand alone application, you will have to return to the programming environment to locate the problem.

How *Micol Advanced BASIC* Boots

In order for you to better understand what is required to create a stand alone application, we will describe what happens when you boot *Micol Advanced BASIC* from disk.

Contained under the **SYSTEM** folder on the booting volume is the *Micol Advanced BASIC* loader named **START**. After the operating system (GS/OS) has booted, **START** is the first file executed. During its execution, **START** searches first for the run time Library (file **LIBRARY**) located in a folder called **Micol.Adv.BASIC** under the boot volume. If this search is unsuccessful, it begins the same search, but this time on the same volume and directory as **START** is located. If this search fails, it searches outer directories until it locates **LIBRARY** in any folder, or the search fails.

Once **LIBRARY** is found, the folder in which **LIBRARY** resides is set as the system

folder from which all subsequent system access is done.

If a file called **MICOL.SYSTEM** is detected of type MAB ($F2), it is assumed the disk is a turnkey disk, and attempts to load the run time Library and load and execute the file **MICOL.SYSTEM.**

If the file **MICOL.SYSTEM** is not detected, it is assumed the system is intended for program development, and the files **COMPILER.SHELL, EDITOR** and **LIBRARY** are loaded, and you are placed into the programming environment.

If all of these tests fail, the file **START** issues an error message and gives up.

The file **Micol.Adv.BASIC** contained under the folder **Micol.Adv.BASIC/** on the Master Disk is identical to the file **START** just described. This file is intended for launching *Micol Advanced BASIC* programs directly from a program launcher. This process was described earlier in this chapter.

Creating a TurnKey System

A TurnKey system is simply a program that automatically executes when the disk on which it resides is booted. The normal GS/OS system disk is actually a TurnKey system for the Finder, as the Finder is automatically executed after GS/OS has booted. You will be creating a similar system, but for a *Micol Advanced BASIC* program.

To create a TurnKey system, take the following steps (you may use any suitable GS/OS or ProDOS 8 copy utility such as the Finder or Copy II Plus):

1. Make an exact copy of the *Micol Advanced BASIC* System Disk (the disk labeled Master Disk). You may change the name of this copied disk if you wish.
2. From this new copied disk, in folder **Micol.Adv.BASIC/**, remove the files **COMPILER.SHELL, EDITOR, Micol.Adv.BASIC, AutoExec** and the **UTILITY** folder. This should only leave the file **LIBRARY** remaining in this folder.
3. Copy the *Micol Advanced BASIC* program you wish to be automatically executed to the folder **Micol.Adv.BASIC/** on this new disk.
4. Rename this *Micol Advanced BASIC* program to **MICOL.SYSTEM.**
5. Copy all files required by your *Micol Advanced BASIC* program to this new disk.

Now, whenever this disk is booted, your *Micol Advanced BASIC* program will automatically load and execute.

Creating GS/OS Applications

There is a utility on the MAB.SUPPORT disk in folder **MAB.TO.S16/** which you may use to make the conversion from a normal *Micol Advanced BASIC* program to an S16 GS/OS application file. This utility is called **MAKE.SA.** Also in the same folder is a special version of the run time Library, file **LIBRARY.S.A** which will be merged with your program to make the S16 application.

NOTE

> The S16 files created with the method described here are much larger than the normal *Micol Advanced BASIC* compiled programs. This means they occupy much more memory on disk, and require more time to load. In earlier versions of *Micol Advanced BASIC*, it was not possible to launch programs directly from a program launcher. This is the main reason the method described here was devised. Unless you have some pressing reason for having a single, self-contained application on disk, we recommend you keep your *Micol Advanced BASIC* programs in the normal format and follow the rules on making your programs launchable described above.

Take the following steps to create your independent S16 application:

1. Start *Micol Advanced BASIC*.
2. Once the command shell prompt appears, insert the disk labeled MAB.SUPPORT into a suitable drive.
3. Using the **PREFIX** command, set the default prefix to **/MAB.SUPPORT/MAB.TO.S16**.
4. Enter **RUN MAKE.SA** followed by a carriage return (remember the system will append the **.LNK** extension automatically).
5. Follow the instruction which appear on the screen. You will receive detailed instructions what to do next. Briefly, two inputs are required: the Pathname of the file to be converted (complete with extension, if any) and the Pathname the converted file will have.

NOTE

> By specifying the full Pathname each time, you may convert any MAB file on the system, and have this file written to any volume anywhere within the system.

NOTE

> If you include files MAKE.SA and LIBRARY.S.A, just described, inside the UTILITY folder under the *Micol Advanced BASIC* system folder, you may create S16 applications by simply entering MAKE.SA<CR> from the Command Shell. This method is only suitable if you have a hard drive.

Creating Classic Desk Accessories

Micol Advanced BASIC supports two types of CDAs, Primary CDAs and Secondary CDAs.

Contained on the MAB.SUPPORT disk, under folder **MAB.TO.CDA**, is a utility for converting your MAB programs to CDAs. Also under this folder are two versions of the run time Library designed for creating CDAs: **LIBRARY.CDA** is intended for merging with your MAB programs to create Primary CDAs, and a dummy run time Library, **LIBRARY.SML**, intended for creating Secondary CDAs.

All systems that contain CDAs written under *Micol Advanced BASIC* must have one Primary CDA. This is because the Primary CDA contains the full run time Library which all CDAs, including Secondary CDAs, will use. For this reason, your first CDA must be a Primary CDA, and all other CDAs must be Secondary. Secondary CDAs are little more than a MAB file converted to CDA type files.

WARNING

> The Primary CDA must always be executed first. This is because there are pointers that must be set for the Secondary CDAs to use. If you attempt to access a Secondary CDA without first having executed a Primary CDA, the computer will probably crash.

In order to convert a MAB file to a CDA, take the following steps:

1. Boot *Micol Advanced BASIC*.
2. Insert the disk labeled MAB.SUPPORT into any suitable drive.
3. At the Shell monitor, enter **PREFIX /MAB.SUPPORT/MAB.TO.CDA/**<CR>.
4. Enter **RUN MAKE.CDA**<CR>.
5. Follow the detailed instructions that appear on the screen. This utility functions essentially the same as the **MAKE.SA** utility described above.

Once the conversion is completed, copy the CDAs to the folder **SYSTEM/DESK.ACCS/** on the boot volume. The next time this disk is booted, these CDAs will appear in the control panel of your Apple IIGS.

Chapter Six

Converting Applesoft Programs

Overview

Micol Advanced BASIC is a language system that is based on Applesoft BASIC. This means, that when *Micol Advanced BASIC* was first being developed, Applesoft was taken as the root language. Structured capabilities and the ability to access the power of the Apple IIGS were added to make what is now *Micol Advanced BASIC* for the Apple IIGS.

What this means to you is that, with a little work, you should be able to use your Applesoft programs under *Micol Advanced BASIC*.

It is the purpose of this chapter to explain most of the modifications you will have to perform in order to compile your Applesoft programs as *Micol Advanced BASIC* programs.

Source File Conversion

Applesoft files are essentially tokenized text files. Whenever you entered an Applesoft line of code and pressed Return, you probably noticed a slight delay before the cursor returned. This delay was caused by the Applesoft interpreter tokenizing this line of code. This means the Applesoft reserved words were converted into numeric equivalents, pointers to the next line were established and line numbers were converted into binary. This was done to speed the execution of the Applesoft program. If you think Applesoft is slow, think how slow it would be if these lines had not been tokenized.

The first task you will have to perform is to convert these Applesoft source files into text files which *Micol Advanced BASIC* can use. Don't worry, this task has been largely automated, so all you have to do is follow a few steps.

There is a file on the MAB.SUPPORT disk, under the volume directory, called CONVERT designed to convert your Applesoft programs to text files. Simply take the following steps:

1. Boot any ProDOS 8 System Disk. This will probably be the disk you used when you were originally developing your Applesoft programs.
2. Load the Applesoft program you want to convert into memory. This program must not have a line number less than twenty.
3. Insert the MAB.SUPPORT disk into any suitable drive.
4. Enter EXEC /MAB.SUPPORT/CONVERT<CR>.
5. Enter RUN<CR>. Your Applesoft program will be converted into a text file.
6. Enter SAVE <Pathname><CR> where Pathname will be the source filename of this text file.
7. Boot *Micol Advanced BASIC* and get into the Text Editor.

8. Load this converted text file into the *Micol Advanced BASIC* source code Editor.

You are now in a position to make the changes required to compile this file. Unfortunately, your first task will be to remove the leading spaces in each line generated by the file conversion.

General Conversion Rules

Following is a list of things to look out for when modifying a converted Applesoft program into a *Micol Advanced BASIC* program. Although this list is as complete as possible, we unfortunately cannot forsee every circumstance. Some problems probably will require a good knowledge of *Micol Advanced BASIC*.

DIM Statements

Applesoft allows **DIM** statements anywhere in a program and the dimensioning may be done with variables. *Micol Advanced BASIC* requires the **DIM** statements to be at the top of the program, and only integer literals are accepted as parameters.

DATA Statements

DATA statements must be at the top of the program, they cannot reside anywhere as in Applesoft. The following rules also apply with **DATA** statements:

1. Quotation marks must be around string literals, for example "This is a string".
2. Values read into real variables must be expressly specified as reals. For example, 22 must be written as 22.0.
3. No empty entries such as ,, are allowed.

Strings

If you are forcing a string garbage collection with a **PRINT CHR$(4); "FRE (0)"**, simply remove it. Our garbage collector is far faster anyway.

String functions such as **LEFT$** and **MID$** check for overflow errors which Applesoft does not do. You may have to check the string lengths before making these calls.

Slot Input/Output

Replace **IN#** and **PR#** with **INSLOT** and **OUTSLOT** respectively. Refer to the appropriate sections in this manual to understand the use of these commands.

Turning the Printer On and Off

Turn the printer on with a **PRTON** instead of **PR#1**. Your printer must be in slot one, however.

Turn the printer off and the screen on with a **TEXT**.

PRINTing

Unlike the **PRINT** statement in Applesoft, semi-colons are required and cannot be implied. The statement **PRINT "Your name is " N$** may be rewritten as **PRINT "Your name is "; Name$**

FLASH Command

FLASH is not supported. Replace **FLASH** with **INVERSE**.

Cursor Positioning

HTAB and **VTAB** require parentheses around the parameter. **SPC**, **TAB**, and **POS** must have a semi-colon following the parameter to hinder a carriage return.

Control of Flow

1. **IF <Real Variable> THEN** is not allowed. Only boolean variables may be so used. You may replace this statement with **IF <Real Variable> > 0 THEN.**
2. **IF** Relop **GOTO** is not allowed. An **IF** statement requires a **THEN**.
3. **NEXT** without its corresponding variable is not allowed. You will have to explicitly specify this variable.
4. Statements like **NEXT X, Y** are not allowed. These should be rewritten **NEXT X: NEXT Y**.
5. **FOR** loops behave a little differently than they do under Applesoft. If you are having trouble with your **FOR** loops, check the **FOR** loop rules described in this manual.

High Resolution Graphics

Under Applesoft, High Resolution graphics only has a resolution of 280 by 192. Under *Micol Advanced BASIC*, although the Super High Resolution graphics commands are the same as Applesoft's High Resolution graphics commands, the resolution is much higher.

Under *Micol Advanced BASIC*'s Super High Resolution graphics, the resolution is

either 320 by 200 (for **HGR**) or 640 by 200 (for **HGR2**). You will have to modify the High Resolution graphics coordinates in your Applesoft programs accordingly.

Also, the colors you set using the **COLOR** command are different; *Micol Advanced BASIC* has far superior colors to Applesoft. You will have to determine the best colors for your graphics and redefine them.

Shape Tables are not supported. You will have to create these shapes using the Functions described in Chapter Ten, Part Three of this manual.

PEEKs and POKEs

Some of the addresses you may have referenced in your Applesoft program with **PEEK**s and **POKE**s may be different under *Micol Advanced BASIC*. In particular, pay attention to addresses in zero page, that is, addresses betweeen 0 and 255.

Check Appendix A in this manual. Appendix A is the memory map for *Micol Advanced BASIC*. This should tell you which locations need to be modified. Note that some locations have no equivalent.

Functions

Any **DEF FN** lines may be converted to multi-line functions using the **FUNC..ENDFUNC** construct.

Disk Filing

Filing commands are the most complicated to modify. Unfortunately, these lines will have to be rewritten. Here are some thing to note:

- **PRINT CHR$ (4);** has no affect on the operating system under *Micol Advanced BASIC*
- Setting a new default prefix is **PREFIX "String"** or **PREFIX Svar**
- Getting the default prefix is **Volume_Name$ = PREFIX$**
- You will have to use **CAT$** to get a catalog

The following tables should help you make additional filing conversions:

Sequential Access Commands

Reading a File

Applesoft	Micol Advanced BASIC
"OPEN /VOL.NAME/FILE.NAME" "READ VOL.NAME/FILE.NAME"	ROPEN (1) "/VOL.NAME/FILE.NAME"

"INPUT L$"	INPUT (1) Line$
"CLOSE VOL.NAME/FILE.NAME"	CLOSE (1)
ONERR GOTO <Line Number>	IF EOF (1) THEN <Stms>

Writing a File

Applesoft	Micol Advanced BASIC
"OPEN /VOL.NAME/FILE.NAME"	
"DELETE VOL.NAME/FILE.NAME"	
"OPEN VOL.NAME/FILE.NAME"	WOPEN (1) "VOL.NAME/FILE.NAME"
"WRITE /VOL.NAME/FILE.NAME"	
"PRINT L$	PRINT (1) Line$
"CLOSE /VOL.NAME/FILE.NAME"	CLOSE (1)

If you are using random access files in your program, then you will have to learn the use of the **SEEK** command in *Micol Advanced BASIC*. Its usage is too complicated to explain here.

Note that the **PRINT CHR$ (4);** statement in the above tables has been removed from the Applesoft lines for reasons of space.

Go for It

Now that you have made the conversion, the fun can begin. Start using *Micol Advanced BASIC* as more than just an Applesoft compiler.

The first thing you will probably want to do is speed up your programs. If practical, convert the real variables into integers. You may want to use the Compiler Directive **INT (A-Z)** to force all reals to integers; then, in your program, you may selectivly convert some of these integers into reals with the "&" character.

Add structure to your programs. Make your arrays larger. Use extended arithmetic. Etc. Etc. Now, get your money's worth out of *Micol Advanced BASIC*.

Appendices

Appendix A

Memory Usage

Because all of *Micol Advanced BASIC*'s system files are relocatable, we cannot tell you where the Compiler, Editor and run time Library reside in memory. These locations will vary according to the conditions under which they were loaded. Besides, there is probably no time that you will need to know these locations as the files generated by the Linker are also relocatable load files and will load in the locations the system says are free.

Note the distinction between Direct Page locations for the run time Library given here, and Zero Page locations used by Applesoft. There is no relationship between the two. Because of system requirements, we could not make any locations the same, so all **PEEK**s and **POKE**s to Zero Page under Applesoft will have to be modified. Some locations in Applesoft will have no comparable locations in *Micol Advanced BASIC*.

Location	Usage	Comment
0	$5C, Absolute long jump constant	Don't Modify
1-2	COUT vector, absolute address	LSB, MSB order, restore if modified
3	COUT bank vector, part of bytes 1-2	3 bytes together for long address jump
4-5	Max. characters allowed for **INPUT**	Default of 255
6-15	Temporary storage	Don't modify
16-17	Temporary string usage	Frequent usage
18-19	Long Integer flag	Don't Modify
20-23	Temporary storage	ML usage okay
24-25	Library routine n· nber	Set by Linker, don't modify
26-27	Free locations	ML usage okay
28-31	Variable one relative location	Don't Modify
32-35	Variable two relative location	Don't Modify
36-39	Variable three relative location	Don't Modify
40-41	Integer random number	Don't Modify
42-43	**WAVE** and **INSTRUM** buffer counter	See **WAVE & INSTRUM**
44-45	Left border of text screen	Modify to shrink text screen
46-47	Right border of text screen	Modify to shrink text screen
48-55	Misc. usage	Don't modify
56-59	Pointer to **DATA** storage	Don't Modify
60-73	Misc. usage	Don't modify

Location	Usage	Comment
74	Back space flag	Delete mode flag for **INPUT**
76-95	Misc. usage	ML usage okay
96-99	Fast file buffer pointer	Don't Modify
100-101	Bottom of text screen	Modify to shrink screen
102-103	Top of text screen	Modify to shrink screen
104-115	Misc. usage	ML usage okay
116-117	Horizontal position of cursor	Best not to modify
118-119	Vertical position of cursor	Best not to modify
120-127	Screen output usage	Don't Modify
128-129	**PRINT USING** usage	Don't Modify
130-131	**PRINT USING** misc. usage	Don't Modify
132-135	Misc. usage	ML usage okay
136-143	Internal string usage	Don't Modify
144-145	Cursor character, default inverse space	Modify for another screen cursor
146-147	Output character mask (default 255)	Modify will change output char.
148-151	Video memory	Don't Modify
152-153	Internal stack counter	Don't Modify
154	Library error code storage	Needed in error trapping
155	GS/OS error code storage	Needed in error trapping
156-157	Extended real storage flag	Don't Modify
158-161	Actual address of first variable	Don't Modify
162-165	Actual address of second variable	Don't Modify
166-169	Actual address of third variable	Don't Modify
170-173	Start of **DATA** storage	Don't Modify
174-177	Last byte of **DATA** storage	Don't Modify
178-181	Start of string storage	Don't Modify
182-185	End of string storage	Don't Modify
186-187	**SPEED** setting	Set by **SPEED**, don't Modify
188-189	**TRACE** flag	Set by **TRACE**, don't Modify
190-193	**ONERR GOTO** address	Don't Modify
194-197	**RESUME** address	Zero if no **ERROR** comp. option
198	**PRINT USING** "," character	Modify as required
199	**PRINT USING** "$" character	Modify as required
200	**PRINT USING** "." character	Modify as required
202	**True Value**	Many uses
204-205	Line number where error occurred	Useful in error trapping
226-239	Misc. uses	Don't Modify

Location	Usage	Comment
240-255	System usage	Protected, **POKE**s cannot modify

Other Memory Usage

$E100EC-$E100EF	Secondary CDA jump location
$E100F0-$E100F3	Jump location to Run Time Library
$E100F4-$E100F7	Jump location to integrated Compiler/Shell
$E100F8-$E100FB	Jump location to Text Editor

Appendix B

Screen Output

Micol Advanced BASIC has its own super fast screen output routines, as you probably already have discovered. The screen output, however, functions very much as the screen output on older Apple IIs in that certain control codes perform certain actions, and certain memory locations control certain features. We will describe these briefly here.

Use **PRINT CHR$ (Value);** to perform the stated action:

Value	Action
8	Move cursor left one position
10	Move cursor down one line, scroll if necessary
11	Move cursor up one line, scroll if necessary
13	Carriage return
14	Set normal text mode
15	Set inverse text mode
21	Move cursor right one position
22	Scroll screen down one line
23	Scroll screen up one line
29	Clear to end of line
64-95	MouseText characters; issue **MS_TEXT** first

Important Direct Page Locations

Location	Function
44	Left border of text screen, default is 1.
46	Right border of text screen, default is 80
100	Top border of text screen, default is 1
102	Bottom border of text screen, default is 24
116	Current horizontal cursor position
118	Current vertical cursor position
144	Cursor character, default is 32, inverse space
146	AND mask for character output, default is 255

The above direct page locations may be modified (**POKE**d) to alter the text screen display. But be careful! Incorrect values may cause a system crash. For example, if you wish to create text windows, you may shrink the text screen by changing locations 44, 46, 100 and 102. Be certain the values are valid, and the cursor is within the new text screen before **PRINT**ing.

Appendix C

Run Time Error Codes

Whenever a run time error occurs, the error code is placed into one of two locations in the run time Library's Direct Page which may be accessed by a user's program.

If the error is generated within the run time Library itself, the error code is placed into location 154 and location 155 is zero. If the error was generated by the operating system, location 155 will contain the error code, and location 154 will be zero.

Each code is generated by a unique error situation which causes a unique message to be printed, if **ONERR GOTO** is not active. The run time Library's error codes are listed below, in this appendix, while GS/OS's error codes are listed in Appendix D.

You may disable an active **ONERR GOTO** by **POKE**ing zeroes into locations 191 and 192. Note that both locations must contain zeroes for **ONERR** to be disabled.

Code	Message output to screen	Comment
1	Error in exponentiation	Integer '^' range exceeded
2	**RETURN** without **GOSUB**	Perhaps **GOTO** instead of **GOSUB**
3	**RESUME** without **ERROR** option	Need **ERROR** compiler option
4	End of data	No more **DATA** to **READ**
5	Bad subscript error	Array limit exceeded
6	Illegal value in function	Probable bad math function parameter
7	Illegal **POKE** value	Value > 255 or bad address
8	Overflow in addition operation	Integer addition range exceeded
9	Return stack error	Too many **RETURN**s for **GOSUB**s
10	Comma tab error	Implied tabs overflowed before <CR>
11	**EXP** error	**EXP** function exceeded limits
12	Out of string data	**DATA** is not of string type
13	Overflow in **TAB**	**TAB** paramet. . is negative or > 80
14	Division by zero error	Integer division by zero
15	Overflow in subtraction	Integer subtraction result < -32767
16	String function overflow	Attempt to create string > 1023 chars.
17	Overflow in concatenation	Same as 16
18	Illegal string assignment attempted	General string error
19	String overflow error	
20	Applesoft graphics error	General graphics error message
21	Illegal real literal error	String cannot be converted to real
22	**FOR** variable overflow	Integer **FOR** counter out of range
23	Overflow in multiplication	Integer multiplication out of range

Code	Message output to screen	Comment
24	String overflow	Maximum of 1023 characters in string
25	**FOR** loop underflow	**FOR** loop stack problem
26	Negative square root attempted	
27	Illegal **PDL** value	Only 1, 2, 3 or 4 allowed
28	Illegal **SPEED** value	Attempt to set **SPEED** > 255
29	Insufficient string space reclaimed	Use **BANK_NO** compiler option
30	String access to unassigned variable	String var. in shaping function not set
31	File output error	Probable **SEEK** error. No data at point.
32	Attempt to read past EOF	Read past last write, or **SEEK** error
33	Invalid pathname in filename	Probable unassigned string variable
34	Dimension and array use mismatch	Number of dimensions mismatch
35	Mismatch in parameter type	Procedure and call parameters wrong
36	Assign in **ADDRESS** mismatch	Probable parameter corruption, rare
37	Specified value > 255 for **SPC**	Only 0 through 255 allowed in **SPC**
38	Maximum of 255 matches in **INDEX**	**INDEX** position parameter > 255
39	Time or date error	System error
40	**READ** data mismatch	**READ** attempt to other type
41	Invalid floating point operation	General floating point error
42	Floating point underflow	Try using **EXTEND** compiler option
43	Floating point overflow	Try using **EXTEND** compiler option
44	Floating point division by zero	
45	Floating point inexact conversion	Cannot convert real to string
46	Floating point error	General floating point error
47	Undefined segment call	No **SEGMENT** number for **CALL**
48	Stack underflow	Perhaps bad recursion attempted
49	No direct page memory for graphic	Bank zero memory used up
50	Quick Draw II startup error	Cannot start High Resolution Tool
51	Illegal tool number	**TOOLBOX** accepts only Tool # < 256
52	Illegal function number	**TOOLBOX** accepts only Func. # < 256
53	Unable to start up sound	Cannot startup Tool(s) for Noise/Music
54	Music or Noise startup error	Similiar to 53
55	Limit seven items off stack	Too many pull variables in **TOOLBOX**
56	No direct page memory for sound	Bank zero memory used up
57	Cannot start Noise Synthesizer tool	System error in **NOISE** command
58	Music or Noise error	Sound system error

Code	Message output to screen	Comment
59	Parameter stack overflow	Cannot store more function parameters
60	Function stack overflow	Too many unresolved **FN** calls
61	String buffer allocation	Probably **BANK_NO** set too large
62	File buffer allocation error	Not enough memory to read files
63	**INPUT** length out of range	**POKE** to location 5 > 3
64	**FUNC**tion return incompatable type	**FN** call to wrong function type
65	System memory error	Fatal error condition
66	Undefined library routine	Bad compiler code generation (call us)
67	Cannot **RUN** program or application	Bad **RUN** command issued
68	No **HGR/HGR2** issued for DESKTOP	Must set graphics mode for Desktop
69	Cannot start Dialog Tool	Fatal Desktop system error
70	Cannot start Event Manager	Fatal Desktop system error
71	No direct Page for Event Manager	Bank zero memory used up
72	Cannot start Window Manager	Fatal Desktop system error
73	Cannot draw Dialog Box	Desktop system error
74	No direct page for Control Manager	Bank zero memory used up
75	Maximum of 16 items for Dialog Box	Too many parts in **DIALOG**
76	Cannot start Control Manager	Fatal Desktop system error
77	Cannot start up Line Editor	Fatal Desktop system error
78	Item error	Part bad in **DIALOG**
79	Unable to create Window	Desktop system error
80	Unable to close Window	Desktop system error
81	Mouse Control error	Desktop system error
82	Unable to startup Menu	Error in **MENU**
83	Mouse control without Desktop	**MOUSE** needs Desktop command
84	Cannot start Scrap Manager	Desktop system error
85	Only class one calls supported	Probable GS/OS call number < $2000

Appendix D

GS/OS Error Codes

As mentioned in the previous Appendix, whenever GS/OS signals an error, that error is placed into location 155 and location 154 is zero. On some rare instances, the library routine may have trapped the error first.

Decimal Error Code	Message sent to screen
1	Invalid GS/OS call number
7	GS/OS is busy
16	Device not found
17	Invalid device request
32	Invalid request
33	Invalid control or status code
34	Bad call parameter
35	Character device not open
36	Character device already open
37	Interrupt vector table full
38	Resource not available
39	Input/output error
40	No device connected
41	Driver is busy
42	Error not defined
43	Write protected
44	Invalid byte count
45	Invalid block address
46	Disk switched
47	Device not online
64	Invalid pathname or device name syntax
67	Invalid file reference number
68	Subdirectory not found
69	Volume not found
70	File not found
71	Duplicate pathname
72	Volume full
73	Volume directory full

Decimal Error Code	Message sent to screen
74	Version error
75	Unsupported storage type
76	End of file encountered (out of data)
77	Position out of range
78	Access not allowed
79	Buffer too small
80	File is open
81	Directory structure damaged
82	Unsupported volume type
83	Parameter out of range
84	Out of memory
87	Duplicate volume name
88	Not a block device
89	Invalid level
90	Block number out of range
91	Illegal pathname change
92	Not an executable file
93	Operating system/file system not available
95	Return stack overflow
96	Data unavailable
97	End of directory
98	Invalid FST call
99	Missing Resource
100	Invalid FST ID
127	Illegal numeric value in file

All the error codes and messages but the last are standard GS/OS errors. The last is a special Micol error code.

Appendix E

Compiler Reserved Words

The following words have a special meaning and may not be used for any other purpose then they were intended. In particular, they may not be used as Program, variable, Function, Procedure or Routine names.

ABS, ADDR, ADDRESS, ALIAS, AND, APPEND, ASC, AT, ATN

BEGIN, BELIEVE, BELL, BKCOLOR, BLOAD, BSAVE, BYE

CALL, CASE_OF, CAT$, CHAIN, CHR$, CLEAR, CLOSE,
COPY, COLOR, COS, CREATE

DATA, DATE$, DELAY, DRAWSTR, DECLARE, DELETE,
DIALOG, DIM, DISPLAY, DO, DOUBT, DUNNO

ELSE, ELSE_DO, END, ENDCASE, ENDDO, ENDFUNC, ENDIF,
ENDPROC, EOF, EXP

FALSE, FILE, FLUSH, FOR, FORMAT, FN, FRE, FREEMEM, FUNC

GET, GET_MEM, GOSUB, GOTO, GR, GS_OS

HCOLOR, HGR, HGR2, HLIN, HPLOT, HOME, HTAB

IF, INCLUDE, INDEX, INKEY$, INPUT, INSERT$, INSLOT,
INSTRUM, INT, INVERSE

LEFT$, LEN, LET, LINK, LOCK, LOG, LOWER$, LRETURN

MENU, MID$, MOD, MOUSE, MOV_MEM, MS_TEXT, MUSIC

NEXT, NOISE, NORMAL, NOT, NOTRACE, NOTICE

OPEN, ON, ONERR, ONLINE$, OR, OUTSLOT

PDL, PEEK, PERFORM, PLOT, POKE, POP, POS, PREFIX, PREFIX$, PRINT, PRTON, PROC

QUIET

READ, REM, RENAME, REPEAT, RESUME, RESTORE, RETURN, RIGHT$, RND, ROPEN, ROUND, ROUTINE, RUN

SCRN, SEEK, SEGMENT, SGN, SILENCE, SIN, SQR, SPC, SPEED, STEP, STOP, STRACE, STR$

TAB, TAN, TEXT, THEN, TIME$, TO, TOOLBOX, TRACE, TRUE

UNTIL, UNLOCK, UPPER$, USING

VAL, VALUE, VLIN, VTAB

WARNING, WAVE, WEND, WHILE, WINDOW, WOPEN

Note: compiler options are not reserved words within a program.

Appendix F

ASCII Character Codes

The following is the table of the ASCII (American Standard Code for Information Interchange) codes supported by *Micol Advanced BASIC*. You may use the **ASC** and **CHR$** functions to go between the code and the character representation.

Value	Character	Value	Character
0	NUL	29	GS
1	SOH	30	RS
2	STX	31	US
3	ETX	32	(Space)
4	EOT	33	!
5	ENQ	34	"
6	ACK	35	#
7	BEL(Bell)	36	$
8	BS (Left Arrow)	37	%
9	HT (Tab)	38	&
10	LF (Line Feed)	39	'
11	VT (Up Arrow)	40	(
12	FF (Form Feed)	41	)
13	CR (Carriage Return)	42	*
14	SO	43	+
15	SI	44	,
16	DLE	45	-
17	CD1	46	.
18	DC2	47	/
19	DC3	48	0
20	DC4	49	1
21	NAK (Left Arrow)	50	2
22	SYN	51	3
23	ETB	52	4
24	CAN	53	5
25	EM	54	6
26	SUB	55	7
27	ESC (Escape)	56	8
28	FS	57	9

Value	Character
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\

Value	Character
93	]
94	^
95	_
96	‘
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	DEL (Delete)

Glossary

6502 addressing format Two byte addresses specified in least significant byte, most significant byte order.

6502 microprocessor CPU used in the Apple II+ and early models of the Apple IIe.

65C02 microprocessor CPU used in the enhanced Apple IIe and Apple IIc. Software written for the 6502 will run on it. This chip has 27 additional machine language instructions.

65816 microprocessor CPU used by the Apple IIGS and Apple IIe upgraded GS. Most software written for the 6502 and 65C02 will run on it. It is more than just a 16 bit version of the 6502 since it has many more instructions and can acess as many as 16 million bytes of memory.

Alphanumeric Usually used to describe characters which consist of letters of the alphabet and digits.

ASCII code The acronym of American Standard Code for Information Interchange. A standardized code used to represent letters, digits and punctuation symbols. The capital letter A is 65 (decimal) in ASCII code.

Assembler A program which can take as input an assembly language text file and translate it into the binary code the computer can execute.

Assembly code A formatted text file an assembler can translate into binary code.

Assembly language The lowest level of the programming languages, specific to a given microprocessor. AL uses short mnemonics corresponding directly to machine instructions and allows a programmer to use symbolic codes. At this level, the programmer is programming the CPU.

Batch processing Allows the system to take its commands from a file on disk rather then the keyboard. Under *Micol Advanced BASIC*, the **BATCH** command creates a batch process.

Binary code The same as machine code.

Binary files Machine language files saved to tape or disk.

BIT Acronym of BInary digiT. The smallest unit of information in a computer. Has a value of zero or one.

Byte A collection of bits wired together. In almost all cases, a byte consists of 8 bits. A byte can represent a character, a number between 0 and 255 or a machine instruction, among other things.

Chaining The process of joining separate text files by the compiler. The compiler can successfully compile separate text files, as though they were a whole program.

Compiler A program that converts a program, usually a text file written in a higher level language, into an intermediate code called an object module. A linker is then required to convert this object module into a machine usable file that can later be executed.

CPU Stands for Central Processing Unit, the "brain" of a computer. When writing in machine language, you are programming the CPU.

Cursor A special character, often blinking, used to show the user where on the screen he/she is entering characters.

Decimal A numbering system based on the number 10; the numbering system we use in every day life.

Direct Page A special 256 byte area in memory bank zero which can be treated as a zero page by a program. Unlike zero page, which begins at location zero in bank zero, direct page is referenced by a special register for this purpose and can begin at any location in bank zero. This distinction between direct page and zero page is important because **PEEK**s and **POKE**s referencing addresses less than 256 under *Micol Advanced BASIC* reference the run time library's direct page, and not zero page.

Editor Same as text editor. A program which allows the user to create, modify and save text files.

Error condition The state of a program after it has detected an error during its execution.

Executable module The binary code created by the linker, which is the actual code which will be executed.

Flag A boolean variable which can be set or unset, so that later a determination can be made based on its value.

File A collection of data stored in some memory device; this can be the computer's memory, disk or tape. On magnetic media, a file name is usually associated with the file.

Hexadecimal A number system based on the number 16 (base 16). Letters A through F are used to stand from 10 to 15.

Integer A variable type which has a limited range and no fractional part. *Micol Advanced BASIC* for the GS has two ranges of integers, short and long. Short integers have a range of ±32767, while long integers have a range of ±2,147,483,647.

Interpreter A program which reads program code written in a high-level language one statement at a time, executes it, then goes to read the next instruction until the program terminates. Traditional BASIC language systems are interpreted. Interpreters are remarkable for their convenience and lack of speed.

Library Contains the run time routines required by the executable

	module at execution time.
Linker	A program that converts the object module(s) created by the compiler into an executable load module.
Load	The act of bringing in information to the computer's memory from a long term storage device such as a disk drive.
Machine code	Almost synonymous with assembly code. Usually refers to the binary code which the computer directly executes.
Memory location	The same as a byte of memory. Can be thought of as an addressable little box in the computer containing a piece of information.
Micol Systems	A dynamic software house located in a suburb of Toronto, Canada. Dedicated to quality systems' software, MICOL is an acronym of MIcro COmputer Languages.
Mnemonic	A collection of characters which can help you remember something. "JMP", for example, can represent $4C in machine code and is a mnemonic for it.
Modularization	The act of breaking a program into small, easily maintainable parts. While little overhead is involved, it greatly minimizes program maintenance.
Octal	A number system based on the number 8 (base 8). Octal was once used more than today. A 10 in octal is decimal 8.
Program	A collection of instructions designed to perform (a) specific action(s).
Real number	The same as floating point number. A number which can contain a fractional part and has a large range. Under *Micol Advanced BASIC* there are two ranges of real numbers, normal and extended. Normal reals require two bytes of storage and have about seven digits of accuracy. Extended reals require 10 bytes of storage and have about 19 digits of accuracy.
Reserved word	A, usually English, word whic' has a special meaning to the compiler and cannot be used as a variable name. **GOSUB** is an example of a reserved word in BASIC.
Run time library	See Library
Save	The act of storing all or part of a computer's memory to some long term storage device such as a disk.
String	A collection of characters. The double quotation mark is used by the compiler to declare strings, e.g. "This is a string".
Structured design	A systematic approach to the creation of software by using a step-by-step procedure for solving the problem. It consists of a smooth program flow, modularization of code, meaningful identifiers, etc.

Two' complement value A number in which the negative value is achieved by adding one to the inverse bit pattern of the positive value. -1 is $FFFF in two's complement for short integers.

Zero Page The area in memory between locations 0 and 255 in bank zero. Do not confuse zero page with direct page which can be anywhere in bank zero.

Index

Compiler Directives

D

E

F

G

H

I

J

K

L

M

N

U

V

W

Printed in Canada ISBN 0-921270-04-6