Micol Advanced BASIC™

Structured Compiled Language System for the Apple IIe and Apple IIc

Version 4.0

Micol Advanced BASIC™

Structured Compiled Language System for the Apple IIe and Apple IIc

Version 4.0

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

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.

ProDOS 8 is a copyrighted program of Apple Computer, Inc. licensed to Micol Systems Inc. to distribute for use only in combination with Micol Advanced BASIC (e/c 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 (e/c 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 reverse side of the System 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 0-921270-10-0

Software and Documentation: Micol Systems Inc., Willowdale, Ontario

FIRST EDITION, July 1989.

SECOND EDITION, revised, corrected, and enlarged.
First printing, March, 1992
Fourth printing, November, 1992

Table of Contents

Introduction

Part One: Overview of the Language

Part Two: The Programming Environment

Introduction

Introduction

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 IIe/c 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 IIe, Apple IIc or Laser computer.

The BASIC program is created using the full-screen Editor. Communication with the ProDOS 8 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 will operate on any Apple IIe with 128K RAM memory and a

65C02 CPU and an 80 column card (almost every Apple IIe), any Apple IIc, any Apple IIGS and any Laser 128 computer. Yet, the entire operating system and language system can fit on a single 5.25 inch floppy diskette with some room to spare.

The executable files created by *Micol Advanced BASIC* are binary files, but because the Run Time Library is maintained separately, a special Micol loader is required. However, because all loading is done automatically, this will be transparent to the user.

Source code files created with the Apple IIGS version of *Micol Advanced BASIC* are highly compatible with those created with the e/c version; memory permitting, only a few changes are needed to use the full power of the IIe/c version.

Micol Advanced BASIC can use all the memory available to your Apple IIe/c 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 compiled 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 most any standard ASCII text file. The Editor has the following features:

- 80-column, full-screen editor
- Word-processor-like commands
- Fast-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 65C02 code
- 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 TurnKey system

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. Five totally distinct graphics modes are supported and it is easy to create interesting sound effects.

The Micol Advanced BASIC language systems offers the following features:

- Upward compatible with the Applesoft BASIC language
- Optional line numbers
- Dynamic character strings with ultra-fast garbage collection
- Simple variables and arrays of type boolean
- Ultra fast and sophisticated string manipulation
- True integer calculations (no conversion to real and back)
- 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, Double Low, High, Double High and Super Double High Resolution graphics

- Mixed text and graphics with Super Double High Resolution graphics
- Sound capabilities
- Complete and easy-to-use ProDOS 8 file handling
- Exclusive Controlled Uncertainty™

How this Manual is Organized

This manual is divided into seven 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 is a brief introduction to Desktop programming.
- Part Five 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 Disk

You have received with this product:

- The *Micol Advanced BASIC* e/c Reference Manual, Second Edition
- One 5.25 inch system disk labeled Micol Advanced BASIC
- A product registration card
- Information about the *Micol Advanced BASIC* Users Group (MABug)
- Other Product information

Part One: Overview of the Language

Side one of the System Disk contains the operating system, ProDOS 8 (file PRODOS), the *Micol Advanced BASIC* language system itself, an AutoExec file, and a UTILITY folder. Side two of the system disk contains some repeated system files, an information file, file INFO.DOC, a UTILITY folder and example programs in folder PRG.EXAMPLES. Be certain to look at the reverse side of the System Disk.

IMPORTANT

> Make backup copies of both sides of the System Disk 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 of the following files: MICOL.SYSTEM, SHELL, EDITOR, COMPILER, LINKER, LIBRARY, and the UTILITY/ folder. MICOL.SYSTEM is the system loader. SHELL is the system Command Shell. EDITOR is the source code Editor. COMPILER is the system Compiler. LINKER is the system Linker and LIBRARY is the run time Library.

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; it may be deleted.

File INFO.DOC on side two of the system disk contains the latest information which is not contained in this manual. If this file is absent, the manual is complete.

What You Need to Know

Before you continue reading this manual, you should know:

- How to set up and use your Apple IIe/c 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 ProDOS 8 to manipulate disk files (see the Apple IIe or Apple IIc System Software User's Guide)
- Some knowledge of Applesoft BASIC or any other dialect of BASIC

Hardware Requirements

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

- An Apple IIe with: a 65C02 CPU, an 80 column card and 128K RAM
- An Apple IIc
- A Laser 128
- An Apple IIGS

With:

- One 5.25 inch disk drive
- A monochrome monitor capable of displaying 80 columns
- ProDOS 8, the DOS required by *Micol Advanced BASIC*, is supplied on disk.

Suggested Additional Hardware

- A second 5.25 inch drive or one, or more, 3.5 inch drives
- A printer
- A hard disk drive
- A color monitor
- An Apple IIe/c Mouse

How *Micol Advanced BASIC* Loads

There is not enough memory in your computer to hold all the system software for *Micol Advanced BASIC*. For this reason, the system must go to disk whenever it requires a different function such as the Compiler or Editor.

When *Micol Advanced BASIC* boots, the system takes note of the directory containing the *Micol Advanced BASIC* files and set this directory as the System Directory. Anytime a system file is required, the System Directory is accessed for the file. This means this directory and its files must always be online during development.

For example, if you are in the Shell and wish to edit a program, the Editor is loaded from the System Directory and the Shell disappears from memory. When you wish to compile a program, the Compiler is loaded from the System Directory and the Editor disappears from memory. The same is true for the Linker, Shell and Run Time Library.

Using *Micol Advanced BASIC* on a Single Drive System

If you only have a single 5.25 inch drive, you may still develop programs under *Micol Advanced BASIC*, but there are a few things you must do first. Note that your programs cannot be as long as those developed under a two drive system, a system with a 3.5 inch drive, or a hard drive.

To create a single 5.25 inch drive system, do the following:

1. Using any suitable ProDOS copy utility, make two exact copies (on two diskettes) of side one of the *Micol Advanced BASIC* System Disk.
2. On the second disk, delete all the files except: SHELL, EDITOR, COMPILER, LINKER and LIBRARY. This means you will be deleting files: ProDOS, MICOL.SYSTEM, AutoExec and the UTILITY folder (and maybe others).
3. Mark the first diskette (Disk One) something like "Disk One" and the second diskette (Disk Two) something like "Disk Two" to distinquish them.

Part One: Overview of the Language

When you wish to boot *Micol Advanced BASIC*, use Disk One. Once *Micol Advanced BASIC* has booted, remove Disk One and insert Disk Two. Disk Two will be your work diskette. Unfortunately, there is not a great deal of room on Disk Two for program development. We strongly recommend you purchase a second drive, a 3.5 inch drive, or a hard drive.

Using Micol Advanced BASIC on a 3.5 inch Drive

One 3.5 inch diskette has about 800K of storage. This is enough room for the operating system, the language system, the UTILITY folder, and your program development. To transfer *Micol Advanced BASIC* to a 3.5 inch diskette, using any suitable ProDOS 8 copy utility, do the folllowing :

1. Insert a blank disk into the 3.5 inch drive. Format the diskette with any suitable volume name other than /Micol.Adv.BASIC (you cannot have two identical volume names in your system while copying).
2. Insert the *Micol Advanced BASIC* System Disk, label side up, into a 5.25 inch drive. Copy all files on this side of the System Disk to the 3.5 inch diskette.
3. Turn the System Diskette over and copy all the files, except SHELL, EDITOR and LIBRARY (if present) to the 3.25 inch drive. You may now rename the 3.5 inch diskette to /Micol.Adv.BASIC if you wish.
4. Label the 3.5 inch diskette. This is your new *Micol Advanced BASIC* System Disk.

Using *Micol Advanced BASIC* from a RAM Disk

You may copy the *Micol Advanced BASIC* system files to a RAM disk and use this RAM disk as the directory from which all system files are accessed. However, if you do not have an Apple manufactured RAM disk, then you may have trouble. This is due, we believe, to a memory conflict between the RAM disk driver and *Micol Advanced BASIC*. We attempted to work around this problem, but were unable to.

Please note that if your RAM disk is non-standard, and the system should malfunction during development, you will have to abandon the RAM disk for system usage. Other uses should still be okay, however.

To set up *Micol Advanced BASIC* on your RAM disk, using any suitable ProDOS 8 copy utility, do the following:

1. Insert the *Micol Advanced BASIC* System Disk into any suitable drive.
2. Copy the files: MICOL.SYSTEM, SHELL, EDITOR, COMPILER, LINKER and LIBRARY from the System Disk to the RAM disk. You may also copy PRODOS.
3. Quit the copy utility (this will probably be to a ProDOS Quit).
4. Set (or select) the default prefix to that of the RAM disk (this step is **not** optional).
5. As next application, stipulate (or select) MICOL.SYSTEM.

You are now in *Micol Advanced BASIC*. Anytime a system file is required, the RAM disk will be accessed.

Please note that you can set up an AutoExec file that will do most of this work just

described anytime you boot *Micol Advanced BASIC*. Please see Part Two, Chapter One, for a discussion on the AutoExec file.

Setting up *Micol Advanced BASIC* on a Hard Drive

1. Create a subdirectory called Micol.Adv.BASIC anywhere of your hard disk.
2. Copy the *Micol Advanced BASIC* system files: MICOL.SYSTEM, SHELL, EDITOR, COMPILER, LINKER, LIBRARY, and the UTILITY folder from side one of the System Disk to the subdirectory Micol.Adv.BASIC you just created on your hard disk. Lock these files.
3. Put the original *Micol Advanced BASIC* disk away in your archive box.

Now, if you set the default prefix to that of the folder to which you copied the system files, and execute the file MICOL.SYSTEM, you will start up *Micol Advanced BASIC*.

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.

Part One: Overview of the Language

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 Five 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 IIGS

Micol Advanced BASIC for the Apple IIGS source code files are highly compatible with *Micol Advanced BASIC* for the Apple IIe/c. You may use the same source files. Since you have less program and data space, you may have to reduce the program's abilities.

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

Earlier Versions of MAB for the Apple IIe/c

Programs developed with *Micol Advanced BASIC* e/c v2.0 to v3.1 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 IIe/c must be recompiled to execute under Version 4.0 of *Micol Advanced BASIC*.

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*. If you only have a single drive, create a single drive system as described in the previous chapter. If you have a two drive system, format a blank diskette and call it WORK.DISK Insert WORK.DISK into drive two of your system.

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 into drive one. Turn on the monitor and the computer.
   a) The ProDOS 8 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 the work disk named WORK.DISK into the second drive and go to step 4
   b) If you have a single drive, remove the *Micol Advanced BASIC* master disk and insert Disk Two (the System Disk with the deleted files) into the drive.
4. If you have a two drive system, enter **PREFIX** /WORK.DISK<CR>. **PREFIX** tells the Shell to use the work disk as the default disk. The Command Shell does not care where the disk is, as long as ProDOS 8 can find it; otherwise the message "Volume not found" will be displayed. Unless otherwise instructed, the system always uses the "prefixed" disk for saving and loading of program files. 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 cause the Source Code Editor to load and execute.
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 II, 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.

Part One: Overview of the Language

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.

Acknowledgments

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

- All our beta testers, especially Peter Cameron.
- Walter Torres-Hurt for his selfless dedication over the years. We also wish to thank him for the *Micol Advanced BASIC* Users Group (MABug).
- Michael Crawford for his generous support and assistance.
- 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.

<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.

COMPLINK File

There is a special type of batch file you can create that is especially designed for program development, i.e. designed for compiling and linking files.

A normal batch file will terminate if it is told to compile and link a program, at the end of the process. If you wish to compile several programs at one time from a batch file, then create a text file that contains all the compile commands you would enter from the keyboard, and save this file as COMPLINK. Then if you BATCH COMPLINK<CR>, all the COMPILE commands in the batch string will execute.

AutoExec File

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

The *Micol Advanced BASIC* System 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 summons the Compiler from the System Directory. 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 ".BIN" 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 load and place the cursor on the line and approximate character where the compilation error occurred. To return to the Shell, press "S". The Shell will load and the Shell prompt 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.

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/.

Part Two: The Programming Environment

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 from the System Directory. The stipulated file must be a text file 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 automatically load the file to let you continue the editing process. The cursor will appear on 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, the stipulated file will be loaded from disk into the Editor's workspace. The file's Pathname will appear on the Data Line.

Example:

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

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, and RAM drives connected to the computer. **ONLINE** displays the names of the volumes currently recognized by ProDOS.

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/.

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}
{Add System/Desk.Accs/ to the current prefix}
PREFIX System/Desk.Accs/
{Prefix will 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 is used to set the slot number the printer is connected to, as well as: whether a line feed will be issued before a carriage return, whether to issue a carriage return character (line overflow) when a line is sent to the printer, and if a RAM disk exists (any volumes named RAM1-RAM7). Use this RAM disk for Compiler scratch work (greatly speeds up compiling and linking).

Image writers will need to take advantage of the second two options. To the first prompt, any key will increment the slot number, a carriage return will accept. To the other prompts, only 'Y' and 'N' are accepted.

Example:

```
PRINTER
```

QUIT

Use **QUIT** to leave the *Micol Advanced BASIC* language system. You will be prompted: "Are you certain you want to quit (Y/N)?". If "N" is entered, this command will be ignored. If "Y" is pressed, control will be returned to the operating system. Once you have entered "Y", you will leave *Micol Advanced BASIC*.

Example:

```
QUIT
```

RENAME Pathname1 TO Pathname2

RENAME changes the name of a file or 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 ".BIN" extension is added by **RUN**).

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* System Directory, 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** under the System Directory. 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):

```
INDENTER MICOL.PROG
```

The optional parameter, a simple ASCII string ended by a carriage return, will be placed into a buffer at location $280 hexadecimal (640 decimal). To access this string, concatenate the values in this buffer using the **CHR$** function until a zero is detected.

Example:

```
PROGRAM My_Utility
Param$ = ""
Address% = $280
REPEAT
    Number% = PEEK (Address%)
    IF Number% <> 0 THEN BEGIN
      Param$ = 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.

Supplied Utilities

There are two supplied Utilities which come with the *Micol Advanced BASIC* System disk: **FONT** and **INDENTER**. **FONT** is used for creating graphics fonts on the Super Double High Resolution screen (see Part Three, chapter Ten), and **INDENTER** is designed to indent programs reflecting their flow and logic.

INDENTER is on side one of the System Disk, while **FONT** is on the reverse side of the System Disk, both in their respective **UTILITY** folder.

Both Utilities have built in documentation, so if you simply invoke the Utility with the Utility name, you can easily get information with a simple command. It is therefore unnecessary to discuss these details here.

Both of these Utilities source code are on side two of the *Micol Advanced BASIC* System Disk, in folder **PRG.EXAMPLES**.

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.

Entering and Quitting the Editor

Entering the Editor (EDIT [Pathname])

To summon the Editor from the System Directory, 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 you have made any changes since the file was last saved, you will be prompted to save the file. Press "Y" if you wish the file saved, otherwise press "N". The Command Shell will then load and wait for a command.

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**
  - — If there is a clock installed, 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 automatically 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. 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 one line down. 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.

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 lowercase "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.

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.

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 move 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 12K (about half of the normal text buffer).

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.

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 (high bit 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 any Editor buffer, so be certain to save your file before you call up the Compiler.

Save File as ASCII (high bit off) (<Apple>T)

<Apple>T saves the source code text file as an ASCII file. 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.

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:

```
ProDOS 8 Version                    1.8
Micol Advanced BASIC e/c version    4.0
Last Modification Date              07 Mar, 1992
Bytes free in editor                1453
Bytes available in copy buffer      10009
Lines available for editing         200
```

The Editors' maximum buffer size is 26K kilobytes: enough for about 800 lines of code. The copy buffer is about 12 thousand bytes.

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 from the System Directory 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 any 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.BIN**.

Example Two:

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

The file **DISK.UTIL** will be compiled as file **FILER.BIN** (a **.BIN** 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 binary file, <FileName>.BIN

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.

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 the 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 that 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 65C02 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 65C02 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. Most of the work performed by your programs must be performed by the Library routines as the 65C02 CPU is not very powerful. It cannot even multiply simple integer values.

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 from the System Directory 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.BIN in the appropriate folder.

How the Linker Works

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

Second, the Linker creates the binary load module FileName.BIN 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 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 load the Editor with your file waiting to be edited. If you enter "N", the Shell will be loaded.

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 and Single High Resolution graphics routines in ROM, but almost all other functions are performed by internal run time Library routines.

The run time Library is brought into memory when any compiled program is loaded. Initially, the Library is loaded to Main memory, but then transferred to Auxiliary memory, where it performs its tasks.

The Library consists of scores of run time routines and buffer memory. It comprises about 26 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 own 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 Program Launcher 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 System Loader, the Micol Program Launcher and the run time Library (files **MICOL.SYSTEM**, **MICOL.LAUNCHER** and **LIBRARY**) 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 the *Micol Advanced BASIC* program 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, 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 the *Micol Advanced BASIC* Run Time Library, the Micol Program Launcher 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* System Disk. This disk contain a special, fully networkable version 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, HI_BUF
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 send 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>
```

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 to look like true assembly language.

Example:

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

The Compiler produces a listing like this for this simple program:

```
3   [0]  2048     $0800       HOME
                               ORG $0800
                               JSR LIBRARY
                               BYT PREINIT
                               WOR 0900
                               WOR 4C00
```

```
                                JSR LIBRARY
                                BYT INIT
                                WOR 0000
                                WOR 0000
                                JSR LIBRARY
                                BYT HOME
4 [0 ] 2070        $0816        END
                                JSR LIBRARY
                                BYT LNOUT
                                WOR 0004
                                JSR 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 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 **RESUME** in Part Three, Chapter 14.

Example:

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

GRAPHIC

If your program will be using Double High Resolution or Super Double High Resoluton graphics, then you may wish to make use of the **GRAPHIC** compiler option.

GRAPHIC starts the program space at 16384 ($4000), above the High Resolution graphics page in Main memory. **GRAPHIC** will also force the Compiler to avoid using data space between $2000 and $3FFF (8192 through 16383), i.e. the High Resolution graphics memory in Auxiliary memory.

This compiler option requires significant memory. If you are not using either Double or Super Double High Resolution graphics, then do not use this compiler option.

HI_BUF

As will be descibed in the next chapter, strings are normally stored in Auxiliary memory, above the normal data storage unless memory is short. If memory is short, strings are stored in a seldom used area of Auxiliary memory between $D000 and $FFFF.

Sometimes strings cannot reside in normal Auxillary memory. For example, because strings build down to the normal data area, if Double High Resolution graphics are used, the strings may build down into this graphics memory.

HI_BUF avoids this by forcing the strings into the high Auxiliary memory between $D000 and $FFFF which is away from everything else.

However, there is one drawback if you using **HI_BUF**. Because of a delay caused by bank switching, as well as other reasons, strings will react somewhat slower. Only use **HI_BUF** if you have reason to believe there will be a memory conflict.

Example

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

IO_BUFS = <Value>

ProDOS 8 requires 1024 bytes for each open file. If only one file is open, referenced either by File Allocation Number one or File Allocation Number eight (see Part Three, chapter Seven), one buffer is allocated away from all program and data space and need not concern you. However, if more than one file must be open at a time, then additional buffers must be allocated just below the run time Library, and just above the end of string storage.

If you must have two or more files open at the same time, then you must have an **IO_BUFS** compiler option within your program. <Value> must be a digit between one and eight. You can determine the value of <Value> by subtracting one from the maximum number of files open at one time in your program. But don't forget, every value above one requires 1024 bytes of data memory. If you are short of memory, avoid this compiler option.

Example

```
PROGRAM Example
@ IO_BUFS = 3 {Allow four open files at the same time}
<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 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.

LODATA = <Value>

The default starting address for data is at location $900 (2304 decimal) in Auxiliary memory. Because 256 bytes starting at $800 (2048 decimal) are required for temporary storage, $900 is the lowest possible address at which to begin data storage.

However, if you are using Double High Resolution graphics, which makes use of memory between $2000 and $3FFF (8192 through 16383 decimal) in both Main and Auxiliary memory, you may wish to use a different starting address for your data storage. This is the compiler option **LODATA**. The syntax for **LODATA** is identical to **LOMEM**, and <Value> may be either a decimal or hexadecimal value.

Example

```
PROGRAM Example
@ LODATA = 16384 {Start data at location 16384}
<Program Code>
```

LOMEM = <Value>

Unless overridden, the Compiler assigns a starting address of $800 (2048 decimal) in Main memory for any *Micol Advanced BASIC* program. Under most circumstances, because this is the lowest possible address at which a program may reside, this is the starting address to use.

However, if you are using any of the High Resolution modes, you may wish to move your program higher in memory. This is the function of **LOMEM**.

All High Resolution graphics modes use Main memory between locations $2000 and $3FFF (8192 through 16383 decimal). If you are using High Resolution graphics, you may wish to start your program at 16384, above the High Resolution graphics (also see the **GRAPHIC** compiler option).

Example

```
PROGRAM Example
@ LOMEM = $4000 {Decimal 16384}
HGR
```

NOGOTO

This compiler option is intended for teachers who wish to restrict their students to structured programming without using **GOTO**s and **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.

Part Three: The Advanced BASIC Language

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 significant increase in execution speed. Use **OPTIMIZ** to speed up the program once it is completely debugged.

IMPORTANT

> Another very 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 by the Shell **PRINTER** command.

Example:

```
PROGRAM Example
@ PRINTER
```

SHARE

Micol Advanced BASIC allows about 42K exclusively for your programs (excluding data space). Because the Compiler is very efficient in its code generation, this allows you to create very large programs indeed.

However, the time may arise when you require even more space, and two or more *Micol Advanced BASIC* programs must share the same data space and values. This is the function of **SHARE**, to give you more program space.

When the Compiler sees the **SHARE** compiler option, it retains the symbol table used by the previous compilation, and generates special code hindering the run time initializion of the the data space.

This means that the programs being **SHARE**d must be compiled in the order in which they will be executed, with the **SHARE** compiler option only being in the second and subsequent programs.

DATA statements and array **DIM**ensions may not be contained in a program which contains this compiler option.

In order to execute the **SHARE**d programs, you must make use of the **RUN** command described later in this manual.

You may wish to make use of a batch stream to compile your **SHARE**d programs. See **COMPLINK** in Part Two, Chapter One. Also see Part Five, Chapter Three.

Example:

```
PROGRAM First
@ LIST
HOME
<Program>
RUN "Second.BIN"

PROGRAM Second
@ SHARE
<Program>
```

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 own 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 statement 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.

A compiled listing looks something like this:

```
PROGRAM Example
Compiled listing of Example
3   [0 ]   2048   $0800      HOME
4   [0 ]   2070   $0816      FOR Counter = 1 TO 10
5   [1 ]   2101   $0835            PRINT "Counter = ";Counter
6   [1 ]   2123   $084B      NEXT Counter
7   [0 ]   2136   $0858      END
```

No errors in compilation

SYMBOL TABLE DUMP

```
 1 R0905          10 R090A          Counter     R0900

26          bytes variable storage
102         bytes code generated
17126       bytes string buffer in low memory
```

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 address in decimal, followed by the address in hexadecimal, followed by the actual program text 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 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 three lines which give a bit of helpful information. These lines are:

```
26      bytes variable storage
102     bytes code generated
17126   bytes string buffer in low memory
```

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 bytes of program code were generated by the Compiler. The third line tells you how much memory is available for storing strings, and in which part of memory, either low or high, the strings will reside.

NOTE

> The program will actually occupy a bit more memory than is specified by the second 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 can represent integer values in the range ±32767. Negative

values are represented as two's complement numbers. An integer occupies two bytes of memory.

Example:

```
Integer% = 32000
```

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 nine 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}
```

Reals can represent values in the range $\pm 3.4 \times 10^{\pm 38}$. Nine digits are significant in calculations. A real variable uses five bytes of storage.

Examples:

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

Scientific Notation

Large real values that are too large to be represented in decimal format (more than nine digits) 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.

Part Three: The Advanced BASIC Language

The length of a string is equal to the number of characters inside it. Each string variable occupies two bytes in data memory plus four (3) bytes of system information in addition to the characters in a separate string buffer. The maximum size a string can grow is 255 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 Auxiliary memory, normally just above the last declared variable. However, whenever the compiler option **HI_BUF** is used, or the Compiler notices that, because of a large amount of data space usage, probably caused by large arrays, it will assign High Auxiliary memory (between $D000 and $FFFF) as the string buffer area. Although this technique maximizes memory usage (giving you about 30K for data storage alone), extra time will be required to access these strings due to moving strings while bank switching.

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 error will be issued during compilation. **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.

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

A string array allocates one byte to hold the number of dimensions, two bytes per dimension size plus two times the number of elements plus two 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 operator 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
Long_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.

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
CLEAR
PRINT Variable {Value is 0}
```

Note that 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.

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.

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 (2.71828183) 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:

```
Logrithm = 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& = 3.1415927
   Radian& = Degree& * (Pi& / 180)
ENDFUNC [Radian&] {Return Radian as output}

{Take Radian as input}
FUNC RadianToDegree [Radian&]
   Pi& = 3.1415927
   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 255 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 255 characters. Any attempt to create a string greater than 255 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 48881 (**True_Value**) in zero 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.)

Normally, if the parameter passed to **CHR$** is less than 128, 128 is added to the passed value. This is necessary for internal purposes. However, for some uses (i.e. file output), adding 128 may not be desirable.

For this reason, a special location has been reserved in memory to override adding 128 and using the actual value specified in the parameter.

If you set memory location $BEDF (48863) to zero before performing a **CHR$**, the character representing the actual value will be sent by **CHR$**. $BEDF must be set to zero every time before **CHR$** is called as the default value is restored at the completion of the **CHR$** function.

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 can 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 the String$ from 1 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 as returned by the clock installed in your computer. If there is no clock installed, do not use this command.

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 (hours and minutes) from the clock installed in your computer. If there is no clock installed, do not use this command.

Example:

```
Clock$ = TIME$
```

```
PRINT Clock$
```

The time is displayed something like this: 10:24

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.

Example:

```
Free_Bytes% = FRE (0)
```

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

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 between zero and 255 (all values converted to modulo 256). 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)
```

Part Three: The Advanced BASIC Language

```
    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).

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 IIe/c. 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 Five, 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. The 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)
PRINT "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 48881) 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(48881) <> 0 THEN BEGIN
     PRINT String$
   ENDIF
UNTIL PEEK(48881) <> 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 48881) 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 (48881) = 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
```

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 or directory. 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.

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

PRODOS (Operation_Code, PathName$, Int_Array% ( )

The **PRODOS** command makes it possible to communicate directly with the operating system of the Apple IIe/c, ProDOS 8.

PRODOS is designed to call individual operations within the operating system. These calls can do a whole assortment of things such as get file information, etc; whatever ProDOS 8 is capable of. All of the disk access commands done by *Micol Advanced BASIC* are done by such calls to ProDOS 8.

To make use of this command, you will need a ProDOS reference manual. Your Apple dealer should have one available.

PRODOS requires three parameters: a ProDOS 8 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 ProDOS 8 call. The three parameters are:

1. The call number is the value required by ProDOS 8 to determine which operating system command is needed. This value is an integer literal (either decimal or hexadecimal).
2. A Pathname is not required by all ProDOS 8 calls, but PathName$ must appear in the **PRODOS** command. If an Int_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.
3. The list of parameters required by the call is provided to ProDOS 8 using Int_Array% starting with element zero. 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.
   a) The first parameter which goes into each parameter block, the number of parameters, must be multiplied by 256. If there are ten parameters required by the call, element zero of the integer array must contain a 2560. This is because the first byte of the array is ignored, allowing you easily to specify a string in element one.
   b) The integer array is passed to ProDOS 8 exactly as specified (except for element zero, as mentioned). Most ProDOS 8 calls are specified in bytes; don't forget that each integer array element is two bytes.
4. If an error occurs as a result of the call, the ProDOS 8 error value will be returned in **True _Value** (location 48881). 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 ProDOS 8 error codes.

Example:

```
PROGRAM OS_Example
@ LIST
INT (A-Z)
```

```
DIM Array%(20) {Minimum size array allowed, else error}
Array%(0) = 2560 {10 ProDOS parameters (10 times 256)}
Array%(1) = -1 {Use string in pathname}
PathName$ = "/MAB/File" {File we require information on}
PRODOS ($C4, PathName$, Array%( ) {$C4 is GetFileInfo}
IF PEEK (48881) = 0 THEN BEGIN {No error}
   Adr = ADDR (Array%() + 4
   FOR Ctr = Adr TO Adr + 20 {Display the result returned}
     PRINT PEEK (Ctr)
   NEXT Ctr
ELSE BEGIN
   PRINT "ProDOS 8 error" ;PEEK (48881)
ENDIF
END
```

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.

IMPORTANT

> Except for File Access Number eight, you must use consecutive digits. This is because file buffers are allocated according to the File Access Number. For example, if you have two files open at the same time, the second file must be opened with File Access Number two. You cannot use File Access Numbers one and eight at the same time. You may also need to specify the **IO_BUFS** compiler option in your program. Please see Part Three, Chapter One for information on **IO_BUFS**.

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. If you wish to close all currently open files, then specify a File Access Number of zero.

CLOSE does not generate an error, but stores the ProDOS 8 error number in **True_Value** (location 48881) if an error occurred. **PEEK** location 48881 after issuing a **CLOSE**. If the value is zero, no error occurred, else the returned value is the ProDOS 8 error number. Refer to Apendix D for a list of all ProDOS 8 error codes.

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** (48881) 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 48848 and 48849 in LSB, MSB order; in location 224 and location 225 is stored the Auxiliary file type.

Example:

```
File_Exists! = FILE (InputFile$)
IF File_Exists! THEN BEGIN
   FileType% = PEEK (48881)
   IF FileType% = 4 THEN BEGIN
     PRINT "The file "; InputFile$; " is of type TXT"
   ELSE BEGIN
     IF FileType% = 6 THEN BEGIN
          PRINT "The file " ; InputFile$; " is of type BIN"
     ENDIF
   ENDIF
ELSE BEGIN
   PRINT InputFile$;" 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 255 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

TABs 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 carriage 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.

NOTE

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, use the **RUN** command. Pathname must be the Pathname, including the ".BIN" 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.BIN"
Path_Name$ = "PROGRAM"
RUN Path_Name$
```

Flow Interruption

END

END terminates the program's execution, and invokes the Command Shell from the System Directory (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

> If the program was started as a TurnKey system, a ProDOS Quit 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 ProDOS 8 (even if the program was started from the Command Shell). **BYE** performs what is called a ProDOS QUIT.

Example:

```
PROGRAM Hello
HOME
PRINT "Hello"
BYE {Pass control to ProDOS 8}
```

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.

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 control 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 reused 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 than 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, which has the same syntax as any variable.

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 a 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.

Part Three: The Advanced BASIC Language

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 text file. 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 certainly has the greatest variety of graphics ever offered a programmer on an Apple IIe, an Apple IIc or Laser. In this regard, we have given you full access to the graphics capabilities of the hardware. In addition to the Low Resolution graphics (40 by 48) and High Resolution graphics (280 by 192) supported by Applesoft, *Micol Advanced BASIC* also supports Double Low Resolution graphics (80 by 48), Double High Resolution (color defined) graphics (140 by 192) and Super Double High Resolution (black & white) graphics (560 by 192). In addition, Super Double High Resolution graphics supports graphics text. Because all modes support text displays at the bottom of the screen, you have a total of 10 graphic modes available at your disposal.

Although there is a great deal of variety in graphics modes, the graphics under *Micol Advanced BASIC* are easy to learn, because this versatility has been achieved with just a few commands. Most of these graphics commands are already familiar to you.

If you wish your graphics to be as fast as possible, issue all of the commands described in this chapter using integer values. Real values will function just fine, but at about half the speed.

Low Resolution Graphics

There are two Low Resolution Graphics modes. You are probably already familiar with Low Resolution graphics so there is little to learn in this section. There also exists a Double Low Resolution graphics mode which is identical to Low Resolution graphics except Double Low Resolution mode has twice the resolution as Low Resolution graphics.

Color = Aexpr

Before you can use any of the Low Resolution or Double Low Resolution plotting routines, you must set the color by use of the **COLOR** command.

Aexpr may be any value between zero and 15 inclusive. Upon issuing the **GR**, **COLOR** is set to zero, or black. Colors under Low Resolution and Double Low Resolution graphics range from black at zero to white at 15. Colors in both Low Resolution graphics modes are identical.

Table 3.10.1 Low Resolution Colors

Value	Color
0	Black
1	Magenta
2	Dark Blue
3	Purple
4	Dark Green
5	Gray
6	Medium Blue
7	Light Blue
8	Brown
9	Orange
10	Grey
11	Pink
12	Green
13	Yellow
14	Agua
15	White

Example

```
PROGRAM Lo_Resolution
INT (A-Z) {Maximize speed}
GR2
FOR Loop1 = 0 TO 39
   FOR Loop2 = 0 TO 47
     COLOR = RND (14) + 1
     PLOT Loop1, Loop2
   NEXT Loop2
Loop1
DELAY = 500
TEXT
END
```

DGR

DGR is used to set the mixed text/graphics Double Low Resolution graphics mode. This will give you a graphics screen 80 blocks by 40 blocks. Each block will be half the width as under normal Low Resolution graphics, but there may be twice as many blocks.

IMPORTANT

> No Double Low Resolution command will execute without either a **DGR** or a **DGR2**, so be certain this command is issued before setting the Low Resolution color.

DGR will clear the Double Low Resolution screen and set the Double Low Resolution color to black (0).

The bottom four lines of the screen will be available for text, if you wish, and all text screen statements will function there.

DGR2

DGR2 is identical to **DGR** except you have a pure graphics screen instead of a mixed graphics/text screen. The Y coordinate may range between zero and 47.

GR

GR is used to set the Low Resolution mode. The screen will fit 40 blocks by 40 blocks (as in the example above).

You will have available for your use, at the bottom of the screen, four lines for text. All the usual screen output commands will still function there.

GR will also clear the Low Resolution screen so that you will have a clear slate with which to begin.

If you do not use the **GR** command before you use one of the Low Resolution graphics commands, the Low Resolution graphics commands will have no effect.

Under **GR**, the X and Y coordinates may range between zero and 39.

GR2

GR2 is identical to **GR** except you will have a pure graphics screen, 40 by 47, instead of the mixed text/graphics screen. Under **GR2**, the X co-ordinate may range between zero and 39 and the Y coordinate may range between zero and 47.

HLIN <X-Coord1>, <X_Coord2> AT <Y_Coord>

Each argument may be either an integer or real value.

HLIN will draw a Low or Double Low Resolution horizontal line in the most recently defined **COLOR** from point X_Coord1, Y_Coord to X_Coord2, Y_Coord.

UnderLow Resolution graphics, the X Coordinates may not be negative or greater than 39, and under Double Low Resolution graphics, the X coordinates may not be negative or greater than 79. The Y coordinate may not be negative or greater than 47 or

you will receive an error at run time.

Example

```
PROGRAM Example
INT (A-Z)
DHGR2
FOR Loop1 = 0 TO 79
   FOR Loop2 = 0 TO 47
     Y_Coord = RND (47)
     X_Coord1 = RND (79)
     X_Coord2 = RND (79)
     COLOR = RND (14) + 1
     HLIN X_Coord1, X_Coord2 AT Y_Coord
   NEXT Loop2
NEXT Loop1
DELAY = 1000
TEXT
END
```

PLOT <X_Coord>, <Y_Coord>

PLOT places a Low Resolution or Double Low Resolution block at the location specified. The range of coordinates are the same as under **HLIN** described above,

The **COLOR** of the block must be specified before this command is executed, or the block will be black (invisible).

Example

```
PROGRAM Example
INT (A-Z)
DGR2
FOR X_Coord = 0 TO 79
   FOR Y_Coord = 0 TO 47
     COLOR = RND (15)
     PLOT X_Coord, Y_Coord
   NEXT Y_Coord
NEXT X_Coord
```

SCRN (X_Coord, Y_Coord)

SCRN returns the Low Resolution or Double Low Resolution **COLOR** code (0 to 15) of the block whose location is specified by the arguments passed.

X_Coord and Y_Coord must be within the range as specified under **HLIN**.

TEXT

TEXT will change the screen from Low or Double Low Resolution graphics to the normal text screen with its character screen we all know and love.

IMPORTANT

> The text screen will probably be filled with garbage after a **TEXT** is issued. Use **HOME** to clear the screen.

VLIN <Y_Coord1>, <Y_Coord2> AT <X_Coord>

VLIN functions exactly as **HLIN** except it draws a vertical line instead of a horizontal line. This also means that the X and Y coordinates in the statement are reversed.

Example

```
PROGRAM LOW_Resolution
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
TEXT
END
```

High Resolution Graphics

Micol Advanced BASIC supports three modes of High Resolution graphics: Single (Applesoft) High Resolution grahpics, color-defined Double High Resolution graphics, and black/white Super Double High Resolution graphics.

Single High Resolution graphics uses an 8K graphics screen in Main memory (between locations $2000 through $3FFF) and has a maximum resolution of 280 by 192. This graphics mode is what you are already familar with in Applesoft BASIC.

Color-defined Double High Resolution graphics uses a 16K graphics screen in both Main and Auxillary memory (between $2000 and $3FFF) and has a resolution of 140 by 192. Although this resolution is lower than Single High Resolution graphics, the color definition is far superior.

Super Double High Resolution graphics uses a 16K graphics screen in both Main and Auxillary memory (between $2000 and $3FFF), has a maximum resolution of 560 by 192, but supports no colors.

NOTE

> If you are using Super Double High Resolution graphics and are using a color monitor, you should turn off or lower the color ability on your monitor, as false colors will be created by your computer's hardware.

The graphics mode you are using is entirely determined by the command issued to start the graphics mode. These commands are: **HGR**, **HGR2**, **DHGR**, **DHGR2**, **SDHGR**, and **SDHGR2**. These commands are described below.

DHGR and DHGR2

DHGR and **DHGR2** are used to set the color-defined Double High Resolution graphics screen. This graphics mode differs from the Super Double High Resolution graphics mode in that its resolution is less and that it supports colors set by the **HCOLOR** command described later (see Table 3.10.3).

Like Super Double High Resolution graphics, Double High Resolution graphics has a dot resolution of 560 by 192. However, because 4 bits (half a byte) are required to define the colors, you effectively have a pixel resolution of only 140 by 192. As all examples we have seen by Apple using Double High Resolution graphics have to do with drawing horizontal lines, it is possible drawing horizontal lines is its primary function.

As is the case with other commands described in this section, **DHGR** sets a mixed text/graphics mode (as under Low Resolution graphics) and **DHGR2** sets a pure graphics mode.

DRAWSTR (Svar)

DRAWSTR is used for placing text directly on the Super Double High Resolution graphics screen. Svar may be either a string variable or string literal.

DRAWSTR will draw the characters contained within Svar determined by the current Super Double High Resolution plotting position. You may use **HPLOT**, describe later in this section, to move the plotting point to the necessary position.

Any visible character may be drawn, and placed in inverse, if an **INVERSE** command has been issued and is still active.

NOTE

> Characters are drawn on the Super Double High Resolution screen. If the colors of your monitor are too intense, the letters will be illegible, so be careful.

DRAWSTR is designed to be used with Super Double High Resolution graphics and assumes Super Double High Resolution co-ordinates. If you set the Double High Resolution screen with an **DHGR** or **DHGR2** command, you will not be able to move the X co-ordinate past position 139 in a Super Double High Resolution position. You may wish to pad your text with spaces to push the characters over.

Note that you may create your own characters to be used with **DRAWSTR**. Contained on side two of the *Micol Advanced BASIC* System Disk is a Utility called **FONT**. With the reverse side of the System Disk in a drive, from the Command Shell, simply enter FONT<CR>. The Utility should load and execute and you will be able to get instructions.

Example

```
SDHGR2
HPLOT 10, 100
DRAWSTR ("This is a Super Double High Resolution display")
INVERSE
String$ = "This string is in inverse"
HPLOT 10, 120
DRAWSTR (String$)
```

ERASE

This command may only be used after an **SDHGR** or **SDHGR2** command is active. **ERASE** is designed for erasing Super Double High Resolution dots or lines from the screen. Once this command is issued, **HPLOT** and **HPLOT TO**, rather than plotting a point or line, will erase any dot or line that exists within the co-ordinates specified.

In order to cause **HPLOT** or **HPLOT TO** to function normally again, simply issue this command a second time. This sequence may be repeated as often as desired.

HGR and HGR2

These commands set the Single High Resolution graphics modes that are 280 by 160 or 280 by 192 respectively as under Applesoft.

Unlike **HGR2** under Applesoft, **HGR2** under *Micol Advanced BASIC* makes use of a single graphics screen located between $2000 and $3FFF in Main memory.

HGR sets a mixed graphics/text screen while **HGR2** sets a pure graphics screen.

HCOLOR = <Color code>

As with Low Resolution graphics, before you can display any graphics, you must set the color which will be used. The High Resolution color set differs between Single High Resolution and Double High Resolution graphics.

NOTE

> **HCOLOR** has no effect on Super Double High Resolution graphics; the colors are always black and white.

The following color table is used if you have issued an **HGR** or **HGR2** command:

Table 3.10.2 Single High Resolution Colors

Value	Color
0	Black
1	Green
2	Violet
3	White
4	Black
5	Orange
6	Blue
7	White

The following table is used if a **DHGR** or **DHGR2** command was issued:

Table 3.10.3 Double High Resolution Colors

Value	Color
0	Black
1	Magneta
2	Brown
3	Orange
4	Dark Green
5	Gray
6	Green
7	Yellow
8	Dark Blue
9	Purple
10	Gray

Value	Color
11	Pink
12	Medium Blue
13	Light Blue
14	Aqua
15	White

Any of these values will set one of the above colors, modulo 16.

HPLOT <X_Coord>, <Y_Coord>

HPLOT places a dot on the screen at the coordinates specified in the mode specified by the initial graphics command.

If you issued an **HGR** or **HGR2**, X coordinates may range between 0 and 279 and Y co-ordinates may range between 0 and 191.

If you issued a **DHGR** or **DHGR2**, X coordinates may range between 0 and 139 and the Y coordinates may range between 0 and 191.

If you issued an **SDHGR** or **SDHGR2**, X coordinates may range between 0 and 559 and Y coordinates may range between 0 and 191.

If you are using Super Double High Resolution graphics and wish to use the **DRAWSTR** command, you will probably wish to issue an **HPLOT** to position the first character.

HPLOT TO <X_Coord>, <Y_coord>

HPLOT TO will plot a straight line from the last graphics point plotted to the position stipulated, using the latest **HCOLOR** (if not in Super Double High Resolution mode). The same ranges apply as under the **HPLOT** command.

SDHGR and SDHGR2

SDHGR or **SDHGR2** will set the Super Double High Resolution mode. This graphics mode gives you a graphics screen of 560 dots horizontally by 192 vertically.

When this mode is active, the position(s) stipulated is simply set to on (or off if **ERASE** is active).

Although **HCOLOR** has no effect on these commands, colors will be set on a color monitor according to the position of the dots in memory. This is a complicated subject, and you will have to experiment with this graphics mode to determine the graphics effects you wish.

Example

```
PROGRAM Draw_Box
```

```
SDHGR2 {Super Double High Resolution set}
HPLOT 10, 10
HPLOT TO 549, 10
HPLOT TO 549, 150
HPLOT TO 10, 150
HPLOT TO 10, 10
HPLOT TO 549, 150
HPLOT 549, 10
HPLOT TO 10, 150
HPLOT 250, 20
DRAWSTR ("The box is drawn")
DELAY = 1000
END
```

This example will plot a box crossed with an X and text in the display for about 10 seconds.

High Resolution Shapes

Under High Resolution graphics there are two types of shapes you can draw: Applesoft High Resolution shapes, and Double High Resolution shapes.

Single High Resolution Shapes

You may already be familar with this mode of shapes as these shapes are drawn with the shape tables as described in the Applesoft manual.

Shape tables is a complex topic. Because this topic is discussed in detail in the Applesoft manual, there is no need to repeat this discussion here. Suffice to say, the shape tables supported by *Micol Advanced BASIC* are identical to the shape tables supported by Applesoft BASIC.

Shape tables are supported in Single High Resolution graphics only. This means, you may only use shape tables if you have issued an **HGR** or **HGR2**.

There are two changes between Applesoft BASIC and *Micol Advanced BASIC* you must observe:

- **ROT** and **SCALE** are not directly implemented. The **ROT**ation factor must be **POKE**d into location **True_Value** (location 48881) and the **SCALE** factor must be **POKE**d into location $E7 (location 231) directly before the **DRAW** or **XDRAW** command is issued.
- The syntax to **DRAW** and **XDRAW** is slightly different than under Applesoft.

DRAW X_Coord, Y_Coord At Shape_Table_Number
XDRAW X_Coord, Y_Coord AT Shape_Table_Number

Notice that the order of the arguments is different than under Applesoft. Except for this factor, and the manner in which **ROT** and **SCALE** have been implemented, **DRAW** and **XDRAW** function exactly as under Applesoft BASIC.

Example:

```
HGR2
HCOLOR = 3 {Must be 3 or 7 for XDRAW to work}
POKE $E8, 00  {Shape table at location $1E00}
POKE $E9, $1E
FOR Ctr = 1 TO 50
   POKE $E7, Ctr {Scale factor; Applesoft was SCALE = Ctr}
   POKE 48881, Ctr {Rotation; Applesoft was ROT = Ctr}
   X_Coord% = 139
   Y_Coord% = 79
   Shape_Table% = 1
   DRAW X_Coord%, Y_Coord% AT Shape_Table%
   DELAY = 100
   XDRAW X_Coord%, Y_Coord% AT Shape_Table%
   DELAY = 100
NEXT Ctr
```

Double High Resolution Shapes

With the Double High Resolution commands described in this chapter, it is very easy to draw a large variety of shapes, from triangles to circles, in any size, shape, or rotation. All that is required is the inclusion of the following Procedure into your *Micol Advanced BASIC* program. This technique may be used with all High Resolution graphics modes.

```
PROC Draw_Shape [No_Of_Sides, Radius, X_Offset, \
                Y_Offset, Distortion, Rotation]
   Flag! = TRUE
   SDHGR2
   Step_Size = 6.28 / No_Of_Sides
   Limit = 6.28 + Step_Size
   FOR Degree_Step = 0 TO Limit STEP Step_Size
     X_Cord = Radius * COS (Degree_Step + Rotation)
     Y_Cord = Radius * SIN (Degree_Step + Rotation)
```

```
        X_Step = Distortion * X_Cord + X_Offset
        Y_Step = Y_Offset - Y_Cord
        IF Flag! THEN BEGIN
            HPLOT X_Step, Y_Step
            Flag = FALSE
        ELSE BEGIN
            HPLOT TO X_Step, Y_Step
        ENDIF
    NEXT Degree_Step
ENDPROC
```

- **No_Of_Sides** determines the figure. Three sides is a triangle, four a rectangle. More than 15 sides makes a circle.
- **X_Offset** is the value of X from the middle of the screen.
- **Y_Offset** is the value of Y from the middle of the screen.
- **Radius** is the radius if the shape is a circle (more than 15 sides).
- **Distortion** is a distortion factor, which is used to alter the shape.
- **Rotation** is a rotation factor in radians (about 57 degrees).

Note that these variables are all real. If you have overridden the default types with an **INT** or **STR** compiler directive, you will have to force real variables with an '&' after the variable name.

Example:

```
X_Offset = 260
Y_Offset = 96
Radius = 90
Distortion = 2
Rotation = 0 {Use default rotation}
FOR No_Of_Sides = 3 TO 30 {Draw different shapes}
    GOSUB Draw_Shape [No_Of_Sides, Radius, X_Offset, \
                      Y_Offset, Distortion, Rotation]
    DELAY = 500
    TEXT
NEXT No_Of_Sides
END
```

Note that this technique for drawing shapes was adapted from a program in the manual *Microcomputer Graphics* by Roy Myers. If you wish to develop sophisticated graphics on your Apple II or Laser computer using *Micol Advanced BASIC*, then you will probably find this book very useful. All of the examples in this text are in Applesoft and can therefore be easily adapted to *Micol Advanced BASIC*.

Chapter Eleven

The Sound of Music

Overview

Micol Advanced BASIC for the Apple IIGS has a great sound and music ability. Great sound was possible to implement because the Apple IIGS has its own built-in sound generator, which is reputed to be the best in the personal computer world.

Your Apple IIe, Apple IIc or Laser doesn't have the built-in sound hardware an Apple IIGS has, but that doesn't mean you can't create delightful sounds with your computer. Your Apple does have one sound making ability, it can make a click. While this doesn't sound like much, when you click your computer at machine language speeds, as *Micol Advanced BASIC* does, it is possible to get some great sound effects.

Under *Micol Advanced BASIC* you have two sound commands, one for simply getting attention, and the other for making music or game sounds.

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.

Example:

```
BELL: BELL {Ring bell twice}
```

Sound

In order to complement the graphics capabilities of *Micol Advanced BASIC*, we have implemented a command for playing music. With a little inventiveness, you should be able to get some very interesting effects.

MUSIC (Pitch, Duration)

If you wish to play a musical note under *Micol Advanced BASIC*, then make use of the **MUSIC** command. Only integer variables or integer literals are accepted as arguments for this call.

How high the note will be played is determined by the first argument. The length the note is played is determined by the second argument.

The duration of the note will deviate somewhat with pitch, so please take note of this fact. You will have to experiment to get the sounds you desire.

Example:

```
PROGRAM Sound
INT (A-Z)
FOR Pitch = 1 TO 20
    MUSIC (Pitch, 10)
NEXT Pitch
```

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. A zero argument returns the previous random number. A set negative argument returns a set result. Any other argument returns the random result.

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 to move 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.

NOTE

> All memory locations less than 256 (i.e. in zero page) are different in *Micol Advanced BASIC* than in Applesoft BASIC. If you are compiling an Applesoft BASIC program, you must note this. Refer to Appendix A for a memory map of *Micol Advanced BASIC*.

Example:

```
Integer% = PEEK (Location%)
Real& = PEEK (Location&)
PRINT PEEK (Location&)
```

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.

IMPORTANT

> As mentioned under **PEEK**, all addresses less than 256 are different under *Micol Advanced BASIC* and Applesoft BASIC. See Appendix A for the *Micol Advanced BASIC* memory map.

Example:

```
POKE Location%, Number%
POKE Location&, Number&
```

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 address used during execution of the program which is the same as the address displayed by the Symbol Table Dump at the end of compilation (if the **LIST** or **PRINTER** option is used).

NOTE

> If **ADDR** is assigned to an integer variable and the result returned is greater than +32767, this value will be represented as a negative number. Add 65535 to a real variable get the positive value.

Example

```
Simple_Address& = ADDR (Variable)
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 or a binary program into memory.

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 in Auxiliary memory. 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:

```
BSAVE "FILE", 8192, 8192
BLOAD "FILE", $2000, $2000
```

BSAVE Svar, Start_Address, Bytes_to_Save

BSAVE stands for **B**inary **SAVE**. Use **BSAVE** to save any information from Auxiliary 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.

You can **BSAVE** a Double High Resolution graphics image you create with a *Micol Advanced BASIC* program, but the image must be saved to two files, one for each memory bank, Main and Auxillary. You must first **BSAVE** the image in Auxiliary memory, then using the **MOV_MEM** command described later in this chapter, move the image in Main memory to Auxiliary and do another **BSAVE**. When you wish to **BLOAD** the memory image you saved, **BLOAD** the files in the reverse order in which you **BSAVE**d them.

Example: (See example at end of chapter)

Moving Memory

It sometimes becomes necessary to move a significant amount of memory from one location in memory to another. We have already mentioned one example with saving

graphics images. Another good example is in creating text windows on the screen that have to be quickly restored.

MOV_MEM Start_Addr, Num_of_Bytes AT Dest

To copy 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 direction of the move is determined by the value in **True_Value** (location 48881) according to the following rules:

- If there is a zero in **True_Value**, the copy will be made from Auxiliary memory to Main memory.
- If there is a 128 in **True_Value**, the copy will be made within Auxiliary Memory only.
- If there is a 254 in **True_Value**, the copy will be within Main Memory only.
- Any other value in **True_Value**, the copy will be made from Main memory to Auxiliary memory.

Example

```
PROGRAM Save_Pic
INT (A-Z)
SDHGR2 {Make the graphics screen}
FOR Ctr = 1 TO 30
   HPLOT TO RND (559), RND (191)
NEXT Ctr
BSAVE "SCREEN.AUX", 8192, 8192
POKE 48881, 1 {Copy from Main to Aux. memory}
MOV_MEM 8192, 8192 AT 8192 {Move half 2X Hi Res picture}
BSAVE "SCREEN.MAIN", 8192, 8192

PROGRAM Load_Pic
INT (A-Z)
SDHGR2
BLOAD "SCREEN.MAIN", 0, 0 {Load at default}
POKE 48881, 0 {Move from Aux. memory to Main}
MOV_MEM 8192, 8192, 8192
BLOAD "SCREEN.AUX", 0, 0
```

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 48856 or 48857).

Location 48856 holds the error number returned by the run time routine. Location 48857 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 a zero into location 48861 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 48882 and 48883 in LSB, MSB order. The following program line will determine at which sequential line the error occurred:

Line_Error& = PEEK (48882) + 256 * PEEK (48883)

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 48861,0 {Turn off future ONERR GOTOs}
IF PEEK (48856) > 0 THEN BEGIN
   PRINT "Language error # ";PEEK (48856);
ELSE BEGIN
   PRINT "ProDOS error # ";PEEK (48857);
ENDIF
PRINT " in line "; PEEK (48882) + 256 * PEEK (48883)
```

```
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.

WARNING

> 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: Humanizing the Interface

Chapter One

Desktop Description

Overview

Most of you have heard of the Apple Desktop used on the Macintosh and Apple IIGS computers. This is the interface designed to make computers available to people who know nothing about computers. This interface is so effective that even Microsoft has created a similar interface on the IBM which has precipitated some tensions in the microcomputer world.

This chapter explains the Desktop metaphor created by Apple and describes what is needed in a Desktop program written in *Micol Advanced BASIC* on the Apple IIe and Apple IIc.

Hardware Requirements

Part Four is the only part of this manual that may cost you additional money to implement. To have a proper Apple Desktop you will need an Apple II Mouse. If you have an Apple IIc, the firmware for the Mouse is built into your computer already and only requires purchasing an Apple Mouse that can plug directly into your computer.

The Apple IIe has absolutely nothing built into itself to support a Mouse. A card, together with the Mouse, must be purchased and this card must be installed into one of the Apple IIe slots. If you have an Apple IIe, and wish to have a Mouse controlled interface, you will have to purchase this card and Mouse, and we understand it still costs more than $100 dollars.

If you are using the Apple IIe/c version of *Micol Advanced BASIC* on an Apple IIGS, then you have nothing more to buy. You may use your Apple IIGS Mouse without difficulty.

The Desktop Environment

What is the the Apple 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 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 a little time to write into code.

Essentially, there are three types of displays on the Apple Desktop: Menus, Windows and Dialog Boxes. We will only discuss Menus and Windows in this section. Dialog Boxes can be simulated by means of Windows.

Unlike the Apple IIGS version of *Micol Advanced BASIC*, there are no commands built into the Apple IIe and Apple IIc version to create the Menu and Window displays. However, it is possible to create text-based Menus and Windows in *Micol Advanced BASIC*, and then monitor the user response to these displays with the built-in **MOUSE** command.

The Desktop Construction Set

Micol Systems has spent considerable time developing routines that can create a Desktop display on your Apple IIe, Apple IIc or Laser. We call this set of routines the *Desktop Construction Set*. If you do not wish to take the time and effort to write your own Menu and Window displays described below, you may purchase the *Desktop Construction Set* directly from Micol Systems. An order form was probably included with this package.

The *Desktop Construction Set* is a set of subroutines, written in *Micol Advanced BASIC*, you can easily integrate into your own programs that will automatically create Menus and Windows and allow a user response either from the Mouse or from the keyboard. You simply include the *Desktop Construction Set* routines within your program and make the appropriate calls.

The *Desktop Construction Set* comes with complete documentation on disk.

If you wish to have an Apple Desktop on your Apple II, then we recommend you purchase the *Desktop Construction Set*.

Menus

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 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 List of Items disappears from the screen when the List is released. The pull-down List is the entire collection of Menu Items for this Menu List.

A Menu Item is the actual command the computer will respond to when selected by

the user. The code to perform this command must be contained within the program containing the Menu.

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 text image that may be created with the normal text commands may be used within 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. A Window frame is rectangular. In order to allow scrolling within the content area of the Window, you will probably wish to change the screen borders to that of the content area. See Appendix B for further information.

The controls in a document frame are optional, and may be used in any combination. They include:

- The title bar at the top of the Window which describes the Window's function.
- A small close box, also at the top of the Window, which, when referenced by the user, causes the Window to disappear from the screen.
- A vertical scroll bar, on the right side of the Window, which caues the contents of the Window to scroll up and down when accessed by the user.
- A horizontal scroll bar, at the bottom of the Window, which causes the contents of the Window to move left and right when accessed by the user.

Saving and Restoring Windows and Menus

When a Menu list is displayed and then released, the screen background must be restored. Windows are usually designed so that multiple Windows may appear on the screen, with the last most referenced being the active Window. This action can be easily simulated in *Micol Advanced BASIC* with the use of **MOV_MEM**.

There is a one-to-two correspondence between the image that appears on the screen and memory in your computer. In Main memory, from locations $400 through $7FF are stored one set of characters, and in Auxillary memory, also between locations $400 and $7FF are stored the alternate characters. This means every other character is stored in one of the two memory banks.

To save the Window or Menu List, so it may later be restored, you may make use of the **MOV_MEM** command, and save the proper locations in both memory banks.

To save the entire contents of the screen requires 2048 bytes of memory. If you wish to support only five Windows at a time, this will require more than 10K of memory. If you are clever, you do not need to save the entire screen, but only the lines on the screen that are displayed. This is the technique used by the *Desktop Construction Set*.

Unfortunately, the lines of text as represented in memory are not consecutive. Table 4.1.1 is a memory map of the text display. Don't forget that these locations apply to both Main and Auxiliary memory, and one line consists of 40 characters from each bank. Every first character is stored in Auxiliay memory, and every second character is stored in Main memory.

Table 4.1.1 Text Screen Memory Map

Line Number	Memory Location
1	$400 (1024)
2	$480 (1152)
3	$500 (1280)
4	$580 (1408)
5	$600 (1536)
6	$680 (1664)
7	$700 (1792)
8	$780 (1920)
9	$428 (1064)
10	$4A8 (1192)
11	$528 (1320)
12	$5A8 (1448)
13	$628 (1576)
14	$6A8 (1704)
15	$728 (1832)
16	$7A8 (1960)
17	$450 (1104)
18	$4D0 (1232)
19	$550 (1360)
20	$5D0 (1488)
21	$650 (1616)
22	$6D0 (1744)
23	$750 (1872)
24	$7D0 (2000)

Monitoring the Desktop

One you have created the Menu and/or Window(s) on the screen, you must monitor the user's response to the display.

There is a *Micol Advanced BASIC* command, **MOUSE**, which is designed to monitor this user response. This command allows you to control the Mouse and passes back the following information:

- The current position of the Mouse. If you know the horizontal and vertical position of the Mouse then, because you have built the Menu or Window display, you also know what the Mouse is currently referencing.
- Whether the Mouse button is being pressed or not. By knowing the Mouse

location and knowing that the Mouse botton is down, you therefore know the selection the user is making.

- The character currently under the cursor. This can be an aid in restoring the screen display.

The **MOUSE** command is discussed in detail in the next chapter, so please read on. You will certainly still have questions and the following chapter will probably answer them.

Chapter Two

Monitoring the User Response

Overview

As mentioned in the previous chapter, any Apple IIe or Apple IIc may have a Mouse connected to it. If you wish to create a human engineered interface, similar to what's possible on the Macintosh or Apple IIGS, then there is a command in *Micol Advanced BASIC* that will be a great help. This command is the **MOUSE** command, and is the subject of this chapter.

Unfortunately, Apple IIe's and Apple IIc's do not come equipped with this piece of equipment, but it must be purchased extra. This chapter may give you some incentive to buy one.

The Mouse Command

MOUSE (Array ())

MOUSE is the command to control the Apple Mouse. Array is an integer array that must be **DIM**ed to at least seven elements. The Mouse command is controlled by the value placed into element zero (Array (0)) of the passed array, the Mouse Control Number.

MOUSE functions under the text screen, not the graphics screen, and maintains its own cursor. The following Mouse Control Numbers determine the **MOUSE** command's functionality:

Table 4.2.1 Mouse Control Numbers

Value	Function
1	Home the Mouse cursor
2	Position the Mouse to specified co-ordinates
3	Read the Mouse cursor position, button status, etc.
4	Set a new Mouse cursor
5	Turn Mouse cursor off
6	Turn Mouse cursor on (default)
7	Set fast Mouse (twice as fast)
8	Set slow Mouse (default)
9	Limit the horizontal co-ordinates of movement (1-80)
10	Limit the vertical co-ordinates of movement (1-24)

The screen positions recognized by **MOUSE** are the same as those specified in the **HTAB** and **VTAB** commands (1-80, 1-24) respectively. Please note that some of these commands alter the values within the passed array, so you should always set the array's values before invoking **MOUSE**. Also, because internal calculations are done using integer math, sometimes a value set by the user may be off by one. If this is important, be certain to test the setting.

In order to test whether a Mouse is available on the host computer, issue a read **MOUSE** command (Mouse Control Number 3) and test location **True_Value** (location 48881). If there is a Mouse card installed, this location will contain a zero, else a non-zero value will be in **True_Value**.

Homing the Mouse

If you set integer array element zero to one, you will home the Mouse cursor, i.e. set the Mouse's cursor to the upper left corner of the screen. Values in elements one and two of the integer array are ignored. This command will probably be the first command executed after the initial read to determine if a Mouse card is available.

Example:

```
Array (0) = 1
MOUSE (Array ()
```

Mouse cursor is at top left most position of screen.

Positioning the Mouse

Setting integer array element zero to two will position the Mouse. Position Mouse sets the Mouse's position according the the values in elements one (horizontal position, 1 - 80) and element two (vertical position, 1 - 24) of the passed array.

Example

```
Array (0) = 2
Array (1) = 40   {Row 40}
Array (2) = 10   {Line 10}
MOUSE (Array ()
```

Mouse cursor sits at row 40 and line 10.

Reading the Mouse

To read the Mouse, set integer array element zero to three.

Read Mouse is the work horse of the Mouse control. You will probably make more use of this command than all the other commands combined. Read Mouse fetches the values set by the firmware, and places these values into their respective array locations starting with array element one in the following order:

Part Four: Humanizing the Interface

Table 4.2.2 Read Mouse Return Values

Array Element #	Function
One	X co-ordinate (1 - 80) of Mouse
Two	Y co-ordinate (1 - 24) of Mouse
Three	1 = Mouse was moved; 0 = Mouse not moved
Four	1 = Button was down last; 0 = was not down last
Five	1 = Button is down; 0 = Button not down
Six	ASCII value of character under cursor

To be effective, Read Mouse must be placed into a loop which tests the condition of the Mouse Button by reading one of the elements in the passed integer array.

Example:

```
REPEAT
   Array (0) = 3
   MOUSE (Array ()
UNTIL Array (5) = 1
PRINT "Mouse clicked at X = ";Array (1)
PRINT "Mouse clicked at Y = ";Array (2)
PRINT "Character under cursor is ";CHR$ (Array (6))
```

Note that delays may be needed in your program as many lines of code can be executed in the time the botton is held down. This can be true even for a very fast click.

Altering the Mouse Cursor

To set the Mouse cursor to one of your own choosing, set integer array element zero to four. The ASCII value of the new cursor, usually a MouseText character, is placed in array element one. Initially, the Mouse cursor is a MouseText arrow (66). The new value should be between 64 and 95 (no check is made). The time delay cursor is 67.

Example:

```
Array (0) = 4
Array (1) = 67
Mouse (Array ()
```

Turning the Mouse Cursor Off

To turn the Mouse cursor off (and replace the cursor with the character under the cursor), set integer array element zero to 5. The position will remain unchanged.

This command is necessary if you are saving and restoring the screen to make Windows or Menus (by means of the **MOV_MEM** command). Without use of this command, you will also be saving and restoring the Mouse's cursor, and the character

under the cursor will be lost. Use the following described command to turn the cursor back on.

Example:

```
Array (0) = 5
MOUSE (Array ()
```

Turning the Mouse Cursor On

To turn the Mouse cursor back on, set integer array element zero to six. If the Mouse cursor has been turned off by the command described just above, the cursor will probably need to be restored. The Mouse cursor position will be that determined by the Mouse itself, which may not be the Mouse cursor's previous position.

Example:

```
Array (0) = 6
MOUSE (Array ()
```

Setting the Fast Mouse

To set the fast Mouse, set integer array element zero to seven. The Mouse will move four times as fast as the normal slow Mouse.

Because the Mouse's position may change with this command, you may wish to position the Mouse to the co-ordinates using Array (0) = 2 (position Mouse) and set the co-ordinates to that of the last position read.

Example:

```
Array (0) = 7
MOUSE (Array ()
```

Setting the Slow Mouse

To set the slow Mouse, set integer array element zero to eight. This is the default speed.

Example

```
Array (0) = 8
MOUSE (Array ()
```

Limiting the Mouse's Horizontal Movements

To set the left and right most position the Mouse may move horizontally, set integer array element zero to nine. The minimum X positon is in element one of the passed array, and the maximum X position is in element two of the passed array.

Example

```
Array (0) = 9
Array (1) = 10
Array (2) = 20
MOUSE (Array ()
```

Mouse may now move only between rows 10 and 20.

Limiting the Mouse's Vertical Movements

To set the minimum and maximum co-ordinates the Mouse may move vertically, set integer array element zero to ten. Integer array element one will contain the top most position to which the Mouse may move, and integer array element two will contain the bottom most position to which the Mouse may move.

Example

```
Array (0) = 10
Array (1) = 5
Array (2) = 10
MOUSE (Array ()
```

Mouse may now only move between lines 5 and 10.

Example Program

Included on the *Micol Advanced BASIC* System Disk, on the reverse side, in folder **PRG.EXAMPLES,** is an example program, file **MOUSE.ALIAS** that demonstates how to use the **MOUSE** command, as well as the power of Aliases discussed in Part Three, Chapter One. Be certain to give this file a look-see.

Part Five: 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 (48881) = 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 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
```

NOTRACE

NOTRACE turns off the effects of a **TRACE**. The number of the line will no longer appear after **NOTRACE** is executed.

Example: (see example under **TRACE**.)

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 limited amount of memory available to Apple IIe or Apple IIc, you may have need for this section. Under the Apple IIe and Apple IIc version of *Micol Advanced BASIC*, a program may have a maximum of 42K for program space and about 30K for string and data space.

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 800 or 900 lines of code. Use **INCLUDE** or **CHAIN** in the program if the program exceeds about 600 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.
- 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 400%.
- 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 have the use of a RAM disk, make use of this 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. Use file access number eight for sequential files.

Chapter Three

Managing Large Programs

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) are just a continuation of the program code (as if there were a single file). 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 mean 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.

Sharing Executable Code Files

Your Apple IIe, Apple IIc or Laser computer has 128K of memory in two separate 64K memory banks. This is a significant amount of memory. However, not all of this 128K is available for your *Micol Advanced BASIC* programs. There are three primary reasons why all this memory cannot be used.

Firstly, some of the memory is needed exclusively for the computer. The operating system, screen displays, etc., require a certain amount of memory. ProDOS 8 alone requires 16K. Secondly, the 128K memory is divided into two 64K memory banks, each of which is separate from the other (the 65C02 microprocessor, the CPU in the Apple IIe, and Apple IIc can only access 64K bytes at one time). Some duplication of memory usage is necessary to overcome this problem.

The third reason has nothing to do with your computer. You've probably noticed that

Micol Advanced BASIC is very powerful, giving you features unheard of in any Apple IIe and Apple IIc. This power requires memory usage in the form of the run time Library. The run time Library is about 26K bytes in length and resides mostly in Auxillary memory, but also has some shared locations in Main memory.

Micol Advanced BASIC programs may be a maximum of about 42K bytes of memory. This limitation may be overcome by the means of sharing.

Sharing refers to the ability of several programs to share the same data space. Normally when a program is executed, the data space is initialized. Sharing stops this initialization except in the starting program.

Another aspect of sharing is that all variables of the same name share the same memory locations. This means, in effect, the variables within shared programs are identical and may be treated as such.

How to Share Programs

In order to make programs share data space, there are four points you must observe:

1. Make use of the **SHARE** compiler option (only) in secondary programs.
2. Compile the programs in the order in which they will execute, with the first executing program not having the **SHARE** compiler option, and all other programs requiring it.
3. All **DIM** and **DATA** statements must appear in the first executing program.
4. Make use of the **RUN** command to cause execution of subsequent programs.

WARNING

> Any **SHARE**d program may **RUN** any other **SHARE**d program, but if you **RUN** the top program again, you will reinitialize the data space (i.e., a **CLEAR** will be performed).

NOTE

> You will probably wish to make use of a **COMPLINK** batch file to compile your shared programs. Once this batch file is created, the entire set of programs may be compiled automatically. Please see Part Two, Chapter One for further information.

Using Shared Programs

Once the programs are compiled, the rest of the work is easy. The final task is to cause the execution of the programs.

Shared programs are executed by means of the **RUN** command. At the suitable location in the program, simply **RUN** the next shared program.

Example

```
PROGRAM Main
INT (A-Z)
DATA 1, 2, 3, 4, 5
DATA 6, 7, 8, 9, 10
DIM Array (10)
HOME
PRINT "In starting program"
FOR Loop_Ctr = 1 TO 10
   READ Array (Loop_Ctr)
NEXT Loop_Ctr
RUN "SUBSEQUENT.BIN"
END

PROGRAM Subsequent
@ SHARE
INT (A-Z)
PRINT "In subsequent program"
FOR Ctr = 1 TO 10
   PRINT Array (Ctr)
NEXT Ctr
```

When **Main** is compiled, the Compiler's Symbol Table (the list of all variables and addresses which the Compiler maintains) is set to empty. This Symbol Table memory space, located between $D000 and $FFFF in Auxillary memory, is used by *Micol Advanced BASIC* only for Compiler symbol table storage, a possible string buffer during program execution, and a copy buffer in the Text Editor, and is therefore safe as long as programs are being compiled without interruption.

When **Subsequent**, above, is compiled, the **SHARE** compiler option tells the Compiler not to reinitialize the Symbol Table, but rather reuse the previous variables and their respective addressed in the current compilation. The Compiler also generates code which will hinder reinitialization of the data space. Because the Symbol Table from **Main** is still available, all variables (and their addresses) referenced in **Main** will also be available in **Subsequent.**

Chapter Four

Assembly Language Integration

Overview

Sometimes, a specific task cannot be done by a higher level language or even greater speed is needed than is possible in this higher level language.

In these cases, a good solution is to integrate (or link) an assembly language module into the program. Under *Micol Advanced BASIC*, it is very easy to link in machine language programs you have developed.

Bringing 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 in the binary file. You may only link in a file of type BIN or you will receive an error at compile time.

Example:

```
LINK "/System.M2000/ClrScreen.B"
```

IMPORTANT

> Not any machine language file can be linked in by the **LINK** command. During linking, the binary file is brought into a *Micol Advanced BASIC* program and the binary program is relocated to the location at which the **LINK** statement resides in the program. For the relocation to be successful, there are certain rules, described below, you must follow.

NOTE

> Any suitable assembler may be used to develop your machine language programs as long as the assembler can generate a binary image. You may purchase, at a special price, a very fine assembler package from us called *System M2000*. Order form included with this package.

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. The first part of the machine language program must contain only program code, no data at all (if you are using *System M2000*, **RES** statements are allowed).
   b. Following the program section, you may have a data section, but it must be preceded by two $FFs (in *System M2000* **WOR $FFFF**).
   c. You must not attempt to pass an address in immediate addressing mode or in a pseudo operation. Such statements as **LDA #>LAB**, **LDX #<ADDRESS**, or address labels within a **WOR** or **BYT** pseudo op will not be relocated by the Linker. In the demo program below, the label **Str_Addr** demonstrates how to define (and reference) addresses which must be relocated to a new address by the Linker.
   d. By default, an executing *Micol Advanced BASIC* program reads from Main memory, and writes to Auxiliary memory. However, your machine language program will probably need to write to Main memory. A **STA $C004** at the beginning of your program will accomplish this. Be certain to restore the default mode of read from Main memory and write to Auxiliary memory with a STA **$C005** before exiting your program.
   e. Remember that the page $BE vectors (see Appendix A) are in Auxiliary memory which can be **PEEK**ed from a *Micol Advanced BASIC* program, but not **LDA**ed directly from your machine language program, which will reside in Main memory.
   f. Do <u>not</u> use an **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
   g. Thoroughly test this program for any errors before linking it.
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 binary file which gets linked in, not the assembly language source code text file.

Example (assembly language):

```
;Example of an assembly language program that can
; be linked directly into Micol Advanced BASIC.
;
;Done using System M2000 syntax, your assembler may
;be different
;
ZPG_Tmp   EQU $A0                This is a safe temp location
```

Part Five: Program Management

```
COUT      EQU $FDED              Apple's character output
HOME      EQU $FC58              Apple's Clear Screen
          ORG $1000              Any address will do
          STA $C004              Read/Write Main memory
          JSR HOME
          LDA STR_ADDR+1         Fetch only the address part
          STA ZPG_TMP
          LDA STR_ADDR+2
          STA ZPG_TMP+1
TOP       LDY #0
LOOP      LDA (ZPG_TMP),y
          BEQ LOOP_END
          JSR COUT
          INY
          BNE LOOP
LOOP_END  INC Ctr                Print line 256 times
          BNE TOP
          BRA FINISH             Skip data section to follow
; The following statement is to force a relative address that
; can be used by the program. The LDA is necessary to force
; the linker to relocate the address of "String". This line
; will never be executed by the program.
Str_Addr  LDA String
          WOR $FFFF              Terminate the program, start data
String    STR 'This is the output string'
          BYT $8D,0
Ctr       BYT 0
FINISH    STA $C005              Write to Auxiliary memory again
          EQU *                  Fall down to BASIC program
```

Example (*Micol Advanced BASIC* program using above program)

```
PROGRAM Example
@ LIST
HOME
PRINT "Linking in a machine language program"
LINK "/System.M2000/EXAMPLE.B"
END
```

Chapter Five

Creating Independent Programs

Overview

Once you have developed your *Micol Advanced BASIC* programs, you will probably wish to have a method to execute these programs without going into the programming environment every time.

There are two methods that allow you to execute your *Micol Advanced BASIC* programs outside of the normal programming environment, the first is from a TurnKey system, and the second is by means of the Micol Program launcher. We will discuss both methods 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 ProDOS 8 system disk is actually a TurnKey system for the BASIC.SYSTEM file, as BASIC.SYSTEM is automatically executed after ProDOS 8 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 ProDOS 8 copy utility such as Copy II Plus or System Utilities that came with the system disk):

1. Format a 3.5 inch or 5.25 inch diskette as a ProDOS disk. You may give the disk any suitable volume name.
2. From the *Micol Advanced BASIC* system disk, side one, copy the operating system, file **PRODOS**, the *Micol Advanced BASIC* loader, file **MICOL.SYSTEM** and the run time Library, file **LIBRARY** to the volume directory of the diskette formatted in step one.
3. Copy, to the volume directory of this new diskette, the *Micol Advanced BASIC* program you wish automatically executed.
4. Rename this *Micol Advanced BASIC* program to **Micol.Adv.BASIC**.
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.

If you have a hard disk, you may create launchable *Micol Advanced BASIC* programs by creating a separate folder for each program you wish to launch and perform steps two through five of the above procedure (do not copy file **PRODOS**). The *Micol Advanced BASIC* program may then be launched from a ProDOS Quit by setting the default prefix to that of the specific folder and specifying **MICOL.SYSTEM** as the next application, or by double clicking **MICOL.SYSTEM** from the respective folder from the GS/OS Finder.

The Micol Program Launcher

There is a program on the back side of the *Micol Advanced BASIC* System Disk called **MICOL.LAUNCHER** which is designed to allow easy access to your *Micol Advanced BASIC* programs independent of the normal programming environment. **MICOL.LAUNCHER** is the Micol Program Launcher.

When the Micol Program Launcher is executed, it will display all of the *Micol Advanced BASIC* programs, in alphabetical order, contained under the default directory. The program being selected is displayed in inverse, and using any arrow key will select another file. Once you have selected the file you wish, press the Return key, and the selected file will be launched.

In order to make use of the Micol Program Launcher, you must do the following:

1. Do one of the following to create a suitable environment for the Launcher
   a) Either format any suitable 3.5 or 5.25 inch diskette. You may give the diskette any suitable name. You may create a special folder on this diskette for launching your programs, if you so wish
   b) On your hard disk, create a folder with any suitable name.
2. From side one of the *Micol Advanced BASIC* System disk, copy the following files to the volume or directory stipulated in step one:
   a) **MICOL.SYSTEM**
   b) **LIBRARY**
3. Turn the *Micol Advanced BASIC* System Disk over, and to the same directory as these other two files were copied, copy file **MICOL.LAUNCHER**. Do not give this file another name, or the Micol Loader will not be able to find it.
4. Copy, to a maximum of about 80 files, the *Micol Advanced BASIC* programs you will wish to launch. Do not copy any other binary type files to this folder as these files are unsuitable for this environment. No program should have the name **Micol.Adv.BASIC** or **Micol.Launcher**.

In order to make use of the system just created, the default prefix must be set to that of the volume or folder in which these files reside, and the Micol Loader, file **MICOL.SYSTEM**, must be the first file executed.

From a ProDOS Quit, select the folder containing your *Micol Advanced BASIC* programs, then cause **MICOL.SYSTEM** to execute. This will cause the Micol Program Launcher to automatically execute.

Once the launched program has finished execution, a ProDOS Quit will be performed. If you launch **MICOL.SYSTEM** again from this folder, you will be able to select and launch another program.

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 full power of the Apple IIe, Apple IIc or Laser computer were added to make what is now *Micol Advanced BASIC* for the Apple IIe and Apple IIc.

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.

On the back side of the System Disk, under the volume directory, is a text file called **CONVERT** designed to convert your Applesoft programs to text files. Simply take the following steps (you will have to modify this procedure somewhat if you only have a single 5.25 inch floppy disk drive):

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 wish to convert into memory. This program must not have a line number less than twenty.
3. Insert the *Micol Advanced BASIC* system disk, reverse side up, into any suitable drive.
4. Enter EXEC /Micol.Adv.BASIC/CONVERT<CR>.
5. Enter RUN<CR>.
6. Enter SAVE <Pathname><CR> where Pathname will be the source filename of this

text file. Your Applesoft program will be converted into a text file and saved with the stipulated filename.

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.

Program Conversion Rules

Following is a list of things to look out for when you are modifying the 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 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 the 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 Shape Tables

Although Applesoft Shape Tables are supported, the syntax to the respective commands will have to be modified. Please see the section on Shape Tables in Part Three, Chapter 10 for more information.

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"	ROPEN (1) "/VOL.NAME/FILE.NAME"
"READ 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. Change Applesoft High Resolution graphics to Super Double High Resolution graphics. Etc. Etc. Now, get your money's worth out of *Micol Advanced BASIC*.

Appendices

Appendix A

Memory Usage

Overview

As was noted earlier, the Apple IIe and Apple IIc have 128K of RAM built in. However, the 65C02, the CPU inside these computers, has only a 16 bit program counter. This means this chip can access a maximum of only 64K of memory at one time. Apple has gotten around this limitation by means of a technique called bank switching.

Bank switching is the ability to make the CPU look at different segments of memory at different times. For example, at one time, one block of 64K can be active (the program bank, for example), and at another time, another bank can be active at another time (the data bank, for example). While this is an oversimplification of what actually happens to give you access to more than 64K of memory in your programs, this should give you an idea of what is happening.

Micol Advanced BASIC uses each 64K bank of memory for different purposes. The tables below will detail this memory usage.

Main Memory Usage

Location	Usage
$0000 - $00FF	Zero page (primary)
$0100 - $01FF	Run Time Stack (Primary)
$0400 - $07FF	Main text and Low Res graphics screen
$0800 - $BFFF	Compiler, Editor, Linker code & buffers
$2000 - $3FFF	High & Main Double High Res graphics screen
$0800 - $B3FF	Program space
$B400 - $B8FF	Shared Run Time Library Routines
$B900 - $BDFF	Reserved for future usage
$BE00 - $BEFF	MAB System locations (see below)
$BF00 - $BFFF	ProDOS 8 Usage (If Library is inactive)
$D000 - $FFFF	ProDOS 8 (Applesoft and Monitor in ROM)

Auxiliary Memory Usage

Location	Usage
$0000 - $00FF	Alternate Zero Page (if high strings active)
$0100 - $01FF	Alternate run time stack (if high strings active)
$0400 - $07FF	Secondary text & Double Low Res graphics screen
$0800 - $08FF	Run time workspace
$0900 - $4BFF	Data space & primary string storage
$2000 - $3FFF	Alternate Double Hi-res graphics screen
$4C00 - $B5FF	Run Time Library and buffers
$B400 - $B8FF	Shared Run Time Library routines
$B900 - $BCFF	First disk file & Fast disk file buffer
$BD00 - $BDFF	**FOR/NEXT** buffer, etc. usage
$BE00 - $BEFF	*Micol Advanced BASIC* Library usage (see below)
$BF00 - $BFFF	ProDOS 8 usage (only if Library is active)
$D000 - $FFFF	High string buffer, Compiler Symbol Table, and Editor Copy buffer

Run Time Library System Locations

Note the distinction between zero page locations for the Run Time Library given below, 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, either under *Micol Advanced BASIC* for the Apple IIGS or under Applesoft BASIC will have to be modified. Some Applesoft locations will have no comparable locations under *Micol Advanced BASIC* because compiled and interpreted languages behave differently.

We have avoided using Zero Page as much as possible in order to avoid conflicts with Apple software and in order to allow you as much Zero Page usage as possible.

Please note that *Micol Advanced BASIC* uses the Monitor ROM only if output is directed through a printer or another slot, and makes use of Applesoft routines only for floating point calculations, and for single high resolution graphics. All other functions are performed within the Run Time Library.

Note: If the comment column in the tables below has a "DM" for "don't modify", then any **POKE**s to these locations may cause the system to malfunction.

Zero Page Usage

Location (in decimal)	Usage	Comment
6 - 9	Misc. usage	DM
11 - 12	Garbage collection	Temporary usage only
24 - 25	Garbage collection	Temporary usage only
26 - 29	Temporary usage	Okay to modify
72 - 73	Temporary screen usage	Okay to modify
78 - 79	Integer random # seed	Modified by **GET**, **INPUT** , **INKEY**
105	Library routine number	DM
106 - 107	Address of first variable	DM
108 - 109	Address of second variable	DM
109 - 110	Address of third variable	DM
115 - 116	Temporary usage	Okay to modify
121 - 122	Temporary usage	Okay to modify
125 - 126	Address of current **DATA**	DM
129 - 130	Temporary usage	Okay to modify
157 - 163	1st floating point accum.	Okay to modify
165 - 172	2nd floating point accum.	Okay to modify
203 - 204	**PRINT USING** usage	DM
206	File usage	DM
207	**RESUME** stack pointer	DM
214	Internal stack counter	DM
215	**FOR** stack counter	DM
224 - 228	Double Hi-Res usage	DM
231	Shape Table **SCALE** value	Set by user
232	Shape Table address	Set by user
235	Horizontal cursor position	Values between 1 and 80
236	Vertical cursor position	Values between 1 and 24
242 - 243	File buffer pointer	DM
254 - 255	Misc addresses	Okay to modify

The following address may be accessed by **PEEK**s, modify at your own risk.

Useful Page $BE Library Usage (Auxiliary memory)

Location (in decimal)	Usage	Comment
48848	File block size	Used only by **FILE**
48850	Character ouput mask	Default 255
48852 - 48853	Start of string storage	
48854 - 48855	Top of string storage	
48856	Run time error code	> 0 if run time error
48857	ProDOS 8 error code	> 0 if ProDOS 8 error
48858 - 48859	PC at program line start	
48860 - 48861	**ONERR GOTO** address	
48862	Used by **INDEX**	
48863	Used by **CHR$**	See **CHR$** in manual
48865 - 48866	End of **DATA** storage	
48871	**SPEED** value	Values are 0 - 255
48872	**TRACE** flag	0 is **NOTRACE**
48874 - 48875	Misc **PRINT USING** usage	
48876	**PRINT USING** comma	Modify if "," not desired
48877	**PRINT USING** "$"	Modify if "$" not desired
48878	**PRINT USING** period	Modify if "." not desired
48879 - 48880	Start of **DATA** values	Used in **RESTORE**
48881	**True_Value**	Misc. usage, very important
48882 - 48883	Line where error occurred	Use for error trapping
48884 - 48885	Current line executing	Only if no **OPTIMIZ** used
48886 - 48887	**RESUME** address	
48892 - 48893	Micol output vector	Alter for own char. output
48894 - 48894	Micol input vector	Alter for own char. input

Some Useful Page $BE System Usage (Main Memory)

Location (in decimal)	Usage	Comment
$48640 - 48701	System prefix	Length followed by ASCII string
48720 - 48783	Current text file edited	Length followed by ASCII string
48710	Delete/Backspace flag	0 = delete, 1 = BS (Apple normal)
48784 - 48887	Editor tab values	Initially set at boot up

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 Memory Locations

Location	Function
235	Current horizontal cursor position
236	Current vertical cursor position250
248	Top border of text screen, default is 1
249	Bottom border of text screen, default is 24
250	Left border of text screen, default is 1
251	Right border of text screen, default is 80
252	Cursor character. Default is 32, inverse space
48850	AND mask for character output, default is 255

The above memory 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 248, 249, 250 and 251. 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 $BE 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 48856 and location 48857 is zero. If the error was generated by the operating system, location 48857 will contain the error code, and location 48856 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 ProDOS 8's error codes are listed in Appendix D.

You may disable an active **ONERR GOTO** by **POKE**ing a zero into location 48861. This is the high order **ONERR GOTO** address byte.

Code	Message output to screen	Comment
1	Exponent error	Integer '^' range exceeded
2	**RETURN** without **GOSUB**	Perhaps a **GOTO** to a routine
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 **LOG** value	Error in using **LOG** command
7	Illegal **POKE** value	Value > 255 or bad address
8	Add overflow	Integer addition range exceeded
9	Return stack underflow	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	**TAB** overflow	**TAB** parameter is negative or > 80
14	Division by zero error	Integer division by zero
15	Subtraction overflow	Integer subtraction result < -32767
16	String function overflow	Attempt to create string > 1023 chars.
17	Concatenation overflow	Same as 16
18	Illegal string assignment	General string error
19	String overflow error	Too many characters in string
20	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

Code	Message output to screen	Comment
23	Multiplication overflow	Integer multiplication out of range
24	String overflow	Maximum of 255 characters in string
25	**FOR** underflow	**FOR** loop stack problem
26	Negative **SQR**	Only positive values for **SQR**
27	Illegal **PDL** number	Only 1, 2, 3 or 4 allowed
28	Illegal **SPEED** value	Attempt to set **SPEED** > 255
29	Out of string space	No more memory for string storage
30	Unassigned string	String var. in shaping function not set
31	File input error	Probable **SEEK** error. No data at point.
32	Past EOF	Read past last write, or **SEEK** error
33	Invalid pathname	Probable unassigned string variable
34	Dim and array mismatch	Number of dimensions mismatch
35	Mismatched parameters	Procedure and call parameters wrong
36	Assign in **ADDRESS** mismatch	Probable parameter corruption, rare
37	**SPC** overflow	Only 0 through 255 allowed in **SPC**
38	More than 255 matches in **INDEX**	**INDEX** position parameter > 255
39	Time or date error	Can't read clock
40	**READ/DATA** mismatch	**READ** attempt to other data type
41	Invalid floating point operation	General floating point error
42	Floating point underflow	FP value less than 10^{-38}
43	Floating point overflow	FP value greater than 10^{38}
44	Floating point division by zero	Bad denominator in division
48	Stack underflow	Perhaps bad recursion attempted
59	Parameter stack overflow	Maximum of 16 parms stored during recursion
60	FUNCtion stack overflow	Too many unresolved Function calls
64	FUNCtion returns incompatible type	FN variable and FUNC are of different types
65	Where is the System Shell?	Probable insertion of system disk requested
66	Undefined library procedure	Possible Compiler bug, please call us

Appendix D

ProDOS Error Codes

As mentioned in the previous Appendix, whenever ProDOS signals an error, that error number is placed into location 48857 and location 48856 is zero. On some rare instances, the Library routine may have trapped the error first.

Decimal Error Code	Message sent to screen
1	Invalid call number
4	Invalid parameter
37	Interrupt vector table full
39	Input/output error
40	No device connected
43	Write protected
46	Disk switched
64	Invalid pathname
66	FCB table full
67	Invalid file reference number
68	Path not found
69	Volume not found
70	File not found
71	Duplicate pathname
72	Volume full
73	Volume directory full
74	Version error
75	Unsupported storage type
76	End of file encountered (out of data)
77	Position out of range
78	No Access allowed
79	Buffer too small
80	File is open
81	Directory structure damaged
82	Unsupported volume type
83	Parameter out of range
85	VCB table full
86	Invalid I/O buffer

Decimal Error Code	Message sent to screen
87	Duplicate volume name
90	Block number out of range
127	Illegal numeric value in file

All the error codes and messages but the last are standard ProDOS errors. The last is a special Micol error code.

In future versions of ProDOS 8, it may be possible for other errors to happen. If an error number is returned that is not in this list, you will have to check the latest ProDOS manual for its meaning.

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 names, variable names, or Function, Procedure or Routine names.

ABS, ADDR, ADDRESS, ALIAS, AND, APPEND, ASC, AT, ATN

BEGIN, BELIEVE, BELL, BLOAD, BSAVE, BYE

CALL, CASE_OF, CAT$, CHAIN, CHR$, CLEAR, CLOSE,
COPY, COLOR, COS, CREATE

DATA, DATE$, DELAY, DGR, DGR2, DHGR, DHGR2, DRAW,
DRAWSTR, DECLARE, DELETE, DIM, DO, DOUBT, DUNNO

ELSE, ELSE_DO, END, ENDCASE, ENDDO, ENDFUNC, ENDIF,
ENDPROC, EOF, ERASE, EXP

FALSE, FILE, FLUSH, FOR, FN, FRE, FUNC

GET, GOSUB, GOTO, GR, GR2

HCOLOR, HGR, HGR2, HLIN, HPLOT, HOME, HTAB

IF, INCLUDE, INDEX, INKEY$, INPUT, INSERT$, INSLOT,
INT, INVERSE

LEFT$, LEN, LET, LINK, LOCK, LOG, LOWER$

MID$, MOD, MOUSE, MOV_MEM, MS_TEXT, MUSIC

NEXT, NORMAL, NOT, NOTRACE, NOTICE

OPEN, ON, ONERR, ONLINE$, OR, OUTSLOT

PDL, PEEK, PERFORM, PLOT, POKE, POP, POS, PREFIX, PREFIX$, PRINT, PRODOS, PRTON, PROC

READ, REM, RENAME, REPEAT, RESUME, RESTORE, RETURN, RIGHT$, RND, ROPEN, ROUND, ROUTINE, RUN

SCRN, SDHGR, SDHGR2, SEEK, SGN, SIN, SQR, SPC, SPEED, STEP, STOP, STR$

TAB, TAN, TEXT, THEN, TIME$, TO, TRACE, TRUE

UNTIL, UNLOCK, UPPER$, USING

VAL, VALUE, VLIN, VTAB

WARNING, WEND, WHILE, WOPEN

XDRAW

Note: compiler options are not reserved words within a program.

Appendix F

ASCII Character Set

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	Value	Character
58	:	93	]
59	;	94	^
60	<	95	_
61	=	96	‘
62	>	97	a
63	?	98	b
64	@	99	c
65	A	100	d
66	B	101	e
67	C	102	f
68	D	103	g
69	E	104	h
70	F	105	i
71	G	106	j
72	H	107	k
73	I	108	l
74	J	109	m
75	K	110	n
76	L	111	o
77	M	112	p
78	N	113	q
79	O	114	r
80	P	115	s
81	Q	116	t
82	R	117	u
83	S	118	v
84	T	119	w
85	U	120	x
86	V	121	y
87	W	122	z
88	X	123	{
89	Y	124	|
90	Z	125	}
91	[	126	~
92	\	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 **BI**nary digi**T**. 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 begin 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 some 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 which 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

D

E

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Printed in Canada ISBN 0-921270-10-0