MICROSOFT® APPLESOFT®

COMPILER

System

For Apple® II

MICROSOFT LICENSE AGREEMENT

CAREFULLY READ ALL THE TERMS AND CONDITIONS OF THIS AGREEMENT PRIOR TO BREAKING THE DISKETTE SEAL. BREAKING THE DISKETTE SEAL INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS.

If you do not agree to these terms and conditions, return the unopened diskette package and the other components of this product to the place of purchase and your money will be refunded. No refunds will be given for products which have opened diskette packages or missing components.

1. LICENSE: You have the non-exclusive right to use the enclosed program. This program can only be used on a single computer. You may physically transfer the program from one computer to another provided that the program is used on only one computer at a time. You may not electronically transfer the program from one computer to another over a network. You may not distribute copies of the program or documentation to others. You may not modify or translate the program or related documentation without the prior written consent of Microsoft.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM OR DOCUMENTATION, OR ANY COPY EXCEPT AS EXPRESSLY PROVIDED IN THIS AGREEMENT.

2. BACK-UP AND TRANSFER: You may make one (1) copy of the program solely for back-up purposes. You must reproduce and include the copyright notice on the back-up copy. You may transfer and license the product to another party if the other party agrees to the terms and conditions of this Agreement and completes and returns a Registration Card to Microsoft. If you transfer the program you must at the same time transfer the documentation and back-up copy or transfer the documentation and destroy the back-up copy.

3. COPYRIGHT: The program and its related documentation are copyrighted. You may not copy the program or its documentation except as for back-up purposes and to load the program into the computer as part of executing the program. All other copies of the program and its documentation are in violation of this Agreement.

4. TERM: This license is effective until terminated. You may terminate it by destroying the program and documentation and all copies thereof. This license will also terminate if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy all copies of the program and documentation.

5. HARDWARE COMPONENTS: Microsoft product hardware components only include circuit cards and the mechanical mouse.

6. LIMITED WARRANTY: THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT MICROSOFT OR ITS DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, MICROSOFT DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF, THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK.

Microsoft does warrant to the original licensee that the diskette(s) on which the program is recorded be free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of delivery as evidenced by a copy of your receipt. Microsoft warrants to the original licensee that the hardware components included in this package are free from defects in materials and workmanship for a period of one year from the date of delivery to you as evidenced by a copy of your receipt. Microsoft's entire liability and your exclusive remedy shall be replacement of the diskette or hardware component not meeting Microsoft's limited warranty and which is returned to Microsoft with a copy of your receipt. If failure of the diskette or hardware component has resulted from accident, abuse or misapplication of the product, then Microsoft shall have no responsibility to replace the diskette or hardware component under this Limited Warranty. In the event of replacement of the hardware component the replacement will be warranted for the remainder of the original one (1) year period or 30 days, whichever is longer.

THE ABOVE IS THE ONLY WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE THAT IS MADE BY MICROSOFT ON THIS MICROSOFT PRODUCT. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

NEITHER MICROSOFT NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THIS PROGRAM SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES ARISING OUT OF THE USE, THE RESULTS OF USE, OR INABILITY TO USE SUCH PRODUCT EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR CLAIM. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

7. UPDATE POLICY: In order to be able to obtain updates of the program, the licensee and persons to whom the program is transferred in accordance with this Agreement must complete and return the attached Registration Card to Microsoft. IF THIS REGISTRATION CARD HAS NOT BEEN RECEIVED BY MICROSOFT, MICROSOFT IS UNDER NO OBLIGATION TO MAKE AVAILABLE TO YOU ANY UPDATES EVEN THOUGH YOU HAVE MADE PAYMENT OF THE APPLICABLE UPDATE FEE.

8. MISC.: This license agreement shall be governed by the laws of the State of Washington and shall inure to the benefit of Microsoft Corporation, its successors, administrators, heirs and assigns.

9. ACKNOWLEDGEMENT: YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF AGREEMENT BETWEEN THE PARTIES AND SUPERCEDES ALL PROPOSALS OR PRIOR AGREEMENTS, VERBAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN THE PARTIES RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement, please contact in writing Microsoft, Customer Sales and Service, 10700 Northup Way, Bellevue, WA 98004.

Microsoft® Applesoft® Compiler

User's Manual

Microsoft Corporation

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

Part No: 22F22B
Doc. No: 8106-300-02

Contents

Appendices to Part 1

Part 2 64K Version

Appendices to Part 2

Part 3 128K Version

Introduction

Microsoft® Applesoft® Compiler is designed to complement the Applesoft BASIC interpreter, to extend the Applesoft language, and to enhance execution of Applesoft programs. The interpreter/compiler combination is the ideal Applesoft program development tool. Programs can be quickly entered and debugged with the interpreter, then optimized for speed with the compiler.

The compiler supports the Applesoft language with only a few modifications. Most programs already written in Applesoft can be compiled with little or no change.

System Requirements

Three versions of Microsoft Applesoft Compiler are included in this package. The 48K version is the standard compiler. The 64K version is an enhanced version that requires 64K of memory. The 128K version is a special compiler for the 128K Apple® IIe. It includes special features that use the additional 64K of available memory.

All three compilers require a minimum of 48K RAM and 1 disk drive. The MS-Applesoft Compiler is provided on disks in DOS 3.3 16-sector format. MS-Applesoft Compiler can be converted to DOS 3.2 13-sector format with the standard DOS 3.3 DEMUFFIN utility. System configuration requirements for the three compilers are listed in Table 1.

Table 1

System Configuration Requirements

System	Version		
	48K	64K	128K
48K Apple II			
Basic system	-	-	-
+ Applesoft BASIC ROM Card	*	-	-
+ RAMCard™	*	-	-
48K Apple II Plus			
Basic system	*	-	-
+ RAMCard	*	*	-
64K Apple IIe			
Basic system	*	*	-
+ ADDED 64K BANK (128K)	*	*	*

An asterisk (*) shows which versions will run on which systems.

A hyphen (-) indicates that the version will not run on that configuration.

A plus (+) means "in addition to the basic system."

16K memory cards similar to the Microsoft RAMCard are acceptable substitutes for RAMCards in the systems listed above.

All systems must have Applesoft available in ROM or loaded into a RAMCard.

> Apple II computers have Integer Basic in ROM, and therefore require either the Applesoft ROM card or Applesoft loaded into a RAMCard.
>
> Apple II Plus systems have Applesoft (in ROM) built in.
>
> Apple IIe has 64K of RAM built in, but 64K can be added with a plug-in "extended 80-column" card. This 64K + 64K = 128K configuration is the "ADDED 64K BANK (128K)" system listed above.

The Microsoft Applesoft Compiler provides the following benefits:

1. Increased execution speed

   Applesoft programs compiled with the compiler normally run from two to twenty times faster than the same programs run under the interpreter.

2. Compact object code

   MS-Applesoft Compiler is designed to produce compact compiled programs. Compiled programs are usually longer than their interpreted Applesoft equivalents, but the compiler uses special techniques to minimize expansion.

3. True integer arithmetic

   Unlike the Applesoft interpreter, the compiler can perform true integer arithmetic. Integer arithmetic can greatly increase execution speed.

4. Interprogram communication

   Programs can pass information to each other by using COMMON variables.

5. Disk-based compilation

   Instead of creating the machine language version of the program in memory, MS-Applesoft Compiler writes out the machine language program to disk as it compiles. This allows the compiler to compile programs of virtually any size.

6. Source code security

   MS-Applesoft Compiler creates machine language equivalents of Applesoft BASIC programs. This machine language file is all that need be distributed when a commercial application is sold. Therefore, the Applesoft program (called a "source" program) is protected from copying or plagiarism.

In addition, the 64K version of the compiler allows you to access the extra memory on RAMCard, and the 128K version is specially designed to take advantage of the extra memory available on a 128K Apple IIe. 128K MS-Applesoft Compiler generates special code to allow variables and strings to be stored in the extra 64K bank of memory. Since the extra 64K of memory is not directly accessible to Applesoft, compiling with MS-Applesoft Compiler has the added advantage of increasing the amount of memory available to Apple IIe users.

These benefits are important for speed-critical applications, and for applications where large program size is a problem.

MS-Applesoft Compiler is particularly outstanding for large, complex applications. Large programs can be separated into several small programs that communicate values with COMMON variables. The smaller programs reduce memory requirements and are usually easier to maintain. The compiler is an example of such a system, since it was separated into parts and used to compile itself. This illustrates the power of Microsoft Applesoft Compiler as a programming tool.

Microsoft Applesoft Compiler is also appropriate for commercial applications that require source code security.

Contents of the Microsoft Applesoft Compiler Package

The MS-Applesoft Compiler package includes this manual and three disks, labeled 48K Version, 64K Version, and 128K Version.

Important

Microsoft Applesoft Compiler is simple to use, but this manual is an important part of the compiler package. Read Part 1, the basic 48K section, and perform the demonstration run (Chapter 1) before attempting to compile programs with MS-Applesoft Compiler.

How to Use This Manual

This manual is divided into three parts, one for each of the three versions of the compiler. Part 1 describes the basic 48K MS-Applesoft Compiler, the standard version. This description is also the basis for the discussions of the 64K and 128K versions. Part 2 describes the 64K (extended memory) version and Part 3 covers the 128K (Apple IIe extended memory) version. 48K MS-Applesoft Compiler users need only read Part 1 (48K). Users with 64K systems should read Part 1, then continue on to the 64K section, Part 2. Apple IIe users should read all three parts.

Part 1
48K Version

Chapters cover compilation, debugging with the interpreter, executing a compiled program, compiler/interpreter language comparison, language enhancements, explanation of how the compiler works, a list of error messages and their solutions, and a demonstration run. Appendices discuss moving binary files with the ADR utility, creating a turnkey disk, little-known Applesoft features, memory usage, and zero page usage.

Part 2
64K Version

Chapters cover the 64K version of MS-Applesoft Compiler, the DOSMOVER program, compilation, and executing compiled programs. The appendices discuss RENUMBER.UPDATE and DOSMOVER technical notes.

Part 3 128K Version	Chapters cover the 128K version of MS-Applesoft Compiler, default compilation, executing 128K programs, specifying different memory configurations, and additional differences from the 48K version.

This manual assumes that the user has a working knowledge of the Applesoft language. For additional information on Applesoft programming, refer to "Resources for Learning Applesoft" in the Introduction.

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

< > Angle brackets indicate user-entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, the user must press the key named by the text; for example, <RETURN>.

CAPS Capital letters indicate portions of statements or commands that must be entered exactly as shown.

Boldface Boldface text represents prompts produced by the compiler. All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

Runtime License Requirements

Microsoft Applesoft Compiler is authorized for single computer use only, and only by the registered owner. Applications created with the compiler may be distributed with the runtime library. No royalties are required; however,

"PORTIONS COPYRIGHTED BY MICROSOFT CORP., 1981, I982, I983"

must appear on the media and documentation.

In addition, the DOSMOVER utility discussed in Chapter 11 can be distributed only with applications compiled with the MS-Applesoft Compiler and the copyright message.

Resources for Learning Applesoft

This manual provides complete instructions for using the compiler. However, it does not provide tutorial material for learning Applesoft. The following texts are good sources for this information:

1. Albrecht, Robert L., Leroy Finkel, and Jerry Brown. *BASIC*. 2nd ed. New York: Wiley Interscience, 1978.

2. Apple Computer Inc. *Applesoft II BASIC Programming Reference Manual*, 1978.

3. Apple Computer Inc. *The Applesoft Tutorial*. Cupertino, California: Apple Computer, 1979.

4. Coan, James S. *Basic BASIC*. Rochelle Park, N.J.: Hayden Book Co., 1978.

5. Dwyer, Thomas A. and Margot Critchfield. *BASIC and the Personal Computer*. Reading, Mass.: Addison-Wesley, 1978.

Chapter 1
Demonstration Run

This chapter takes you step by step through the compilation of a sample program. This demonstration uses the basic 48K compiler to introduce you to Microsoft Applesoft Compiler. The demonstration is simple, but it provides you with a basic understanding to build on later. If you enter commands exactly as described in this chapter, you should have a successful session with the compiler. We strongly recommend that you perform the demonstration run *before* compiling any other programs.

Note

> Before beginning this demonstration run, make a back-up copy of the 48K MS-Applesoft Compiler disk. Store the master disk in a safe place and work with the back-up copy.

MS-Applesoft Compiler is simple to use. First insert your 48K MS-Applesoft Compiler disk into your computer and then type

```
]CATALOG
```

to display the file directory. Note that the prompt is in boldface type.

The 48K MS-Applesoft Compiler disk contains the following files:

1. APCOM—48K version of the Microsoft Applesoft Compiler.

2. PASS0, PASS1, PASS2—Internal subprograms of 48K MS-Applesoft Compiler.

3. RUNTIME—Runtime library.

4. ADR—Utility for binary files.

5. CREATE ADR—Utility for creating ADR on other disks.

6. BALL—Demonstration program.

These files and their use are described more completely in the chapters that follow. For now, invoke the compiler by typing

```
]RUN APCOM
```

Next, a few simple questions must be answered to begin the compilation process. The first two prompts ask you for the names of the source and object files:

```
SOURCE FILE? BALL

OBJECT CODE FILE:
(DEFAULT BALL.OBJ)? <RETURN>
```

The source file is an Applesoft program named BALL that already exists on disk. The object file is the machine language binary file that is created by the compiler. The name of the object file defaults to the original filename with the extension .OBJ added, so the object file produced for the BALL program is BALL.OBJ. The default is specified by pressing the <RETURN> key.

The source file is assumed to be located on the same disk as the compiler, unless you specify otherwise. The object file defaults to the same disk that the source is on. Different slots or drives can be specified using the normal ,S<slotnumber> and ,D<drivenumber> syntax. Disk commands can be executed by typing <CTRL-D>, followed by the command and <RETURN>.

The next two prompts ask you whether you want default values for all other compilation options. Since most compilations are performed with the same set of options, you should press the <RETURN> key after each prompt to specify the default values:

```
MEMORY USAGE:
NORMAL CONFIGURATION:
(DEFAULT YES)? <RETURN>
COMPILATION OPTIONS:
NORMAL CONFIGURATION:
(DEFAULT YES)? <RETURN>
```

If you had refused the default configurations above, you would need to explicitly specify the values of several memory usage and compilation options. These options are explained in Chapter 4, "Compilation." The actual compilation process starts without further input, since you have specified the defaults above.

When compilation begins, the disk is accessed almost constantly to either read the source file or to write the object file. The compiler lists the source program as it is being compiled and generates appropriate messages if it encounters any errors. When the source stops listing, the first part of compilation is finished, and the compiler prints:

```
*****BEGINNING PASS2
```

The second part of compilation also uses the disk extensively. To indicate that it is still compiling, the compiler prints a percent sign (%) on the screen every few seconds. When it is finished, the compiler prints:

```
*****CODE GENERATION COMPLETE
```

At this point, the actual compilation process is complete.

The next prompt offers a listing of compilation information. Press <RETURN> to see the listing:

```
LIST COMPILATION INFORMATION:
(DEFAULT YES)? <RETURN>
```

This input also accepts <CTRL-D> disk commands. You can direct where you want to list the compilation information using the Apple DOS <CTRL-D> disk commands. For example, if you want to list the information on your printer, type in the following:

```
<CTRL-D> PR#<printerslot>
```

The last prompt offers a listing of the object code addresses for each program line. Type *Y* or *YES* to see what the listing looks like:

```
LIST LINE NUMBER ADDRESSES:
(DEFAULT NO)? Y
```

The address of the object code for each line is listed next to the line number. When the listing is finished, the compiler prints the following message and returns to the interpreter:

```
*****COMPILATION COMPLETE
```

The increase in the BALL program's execution speed is quite apparent when compared to the same program running under the interpreter. Compare speeds by first running the interpreted program:

```
]RUN BALL
```

Next, execute the compiled program by entering the following DOS commands:

```
]BLOAD RUNTIME
]BRUN BALL.OBJ
```

Note that the runtime library must be loaded into memory with the BLOAD command before BALL.OBJ can be run with BRUN.

You have now successfully completed the demonstration run. Be sure to read Chapter 6, "Compiler/Interpreter Language Comparison," before attempting to compile other Applesoft programs.

Chapter 2
Introduction to Compilation

This chapter introduces the vocabulary of compilation, compares compilation to interpretation, and describes the development process for compiled Applesoft programs. These three topics serve as an introduction to compilation.

Vocabulary

Although this manual attempts to keep technical language to a minimum, the following terms must be understood:

Source file. The Applesoft program is commonly called a source file because it is the source from which an equivalent machine language file is created. The source file is the input file to the compiler. CATALOG lists the names of Applesoft files, with the letter "A" preceding the size of each file.

Object file. MS-Applesoft Compiler translates source files into machine language object files. The object file is the output file created by the compiler. The object file is an executable binary file that is the machine language equivalent of the source. CATALOG lists the names of binary files, with the letter "B" preceding the size of the file.

Compiletime. The time during which the compiler is translating a source file into an object file.

Runtime. The time during which a compiled program is executing. By convention, runtime refers to the execution time of a compiled program, and not to the execution time of the compiler.

Runtime library. A collection of machine language routines that are used by compiled programs. These routines all reside in the file named.

RUNTIME. RUNTIME must be loaded into memory before an object file can be executed.

Compilation vs. Interpretation

Since the microprocessor in the Apple can execute only its own machine instructions, it does not execute Applesoft program statements directly. Instead, statements must be simulated by machine language routines that perform the operations specified by each BASIC statement.

Compilers and interpreters approach this translation problem differently. This difference is demonstrated in the following analogy.

Suppose you wish to build a stereo from a kit. Unfortunately, you find that the instructions are written in Japanese, a language you do not know.

One way to approach this problem is to translate each instruction with a Japanese-English dictionary and perform the instructions one by one. This process parallels the interpretation of Applesoft statements by the Applesoft interpreter. Your construction of a stereo kit is analogous to the Apple's execution of a program.

Inefficiencies can arise in this process, especially if you fail to write down the translated instructions as they are carried out. Suppose, for example, that halfway through the construction process you translate an instruction that says:

> Go back to Instruction 14. Then repeat the preceding steps for the second speaker.

One problem is immediately apparent: you can't even find Instruction 14 without scanning the instructions from the start for the Japanese characters for fourteen. You then face the time-consuming task of retranslating each instruction so the second speaker can be assembled precisely like the first. Likewise, the Applesoft interpreter must repeatedly translate each statement executed inside a FOR/NEXT loop.

An alternative approach to constructing the stereo is to sit down with pencil and paper and translate the entire instruction sheet into English. When you are finished, you have a new set of instructions written in English. These English instructions correspond to the object file created by the compiler.

The actual assembly of the kit can then proceed without any further translation, and even without any further need for the original Japanese text. When you finally sit down to build your speakers, you can construct them more quickly. Similarly, a compiled BASIC program does not require the original BASIC source and runs very quickly in comparison to the interpreted version of the same program.

This analogy gives you a feel for the difference between interpretation and compilation. The following paragraphs discuss interpretation and compilation more technically , but also more directly.

Interpretation

The interpreter translates Applesoft source statements line by line at runtime. Each time the interpreter executes an Applesoft statement, it must analyze the statement, check for errors, and call machine language routines that perform the desired function.

When statements must be executed repeatedly, as must those within a FOR/NEXT loop, the translation process must be repeated every time the statements inside the loop are executed.

In addition, BASIC line numbers are stored in a list. GOTOs and GOSUBs force the interpreter to search this list to find the desired line. When the line is near the end of a long program, this search is slow.

The interpreter keeps track of variables by using a similar list. When it encounters a reference to a variable, the interpreter searches from the beginning of the list to find the desired variable. If the variable is not present in the list, the interpreter creates a new entry for it. This procedure also slows interpreted programs.

Compilation

A compiler, on the other hand, takes a source program and translates it into a machine language object file. This object file consists of a large number of machine language CALLs to routines in the runtime library and to routines in the Applesoft interpreter. By calling routines in the Applesoft interpreter, the compiler assures good language compatibility with the interpreter.

Unlike the interpreter, the compiler analyzes all statements *before* runtime. In addition, absolute memory addresses are provided for variables and program lines. These addresses eliminate the list searching that occurs while an interpreted program executes.

The compiler, unlike the interpreter, implements true integer arithmetic and integer loop variables in FOR/NEXT loops. In comparison, the Applesoft interpreter converts all integers to real numbers before operating on them. These conversions make interpreted integer arithmetic relatively inefficient. In addition, the interpreter does not support the use of integers as loop control variables in FOR/NEXT loops.

These factors combine to make compiled programs considerably faster. In most cases, execution of compiled programs is two to twenty times faster than execution of the same program under the interpreter.

The Program Development Process

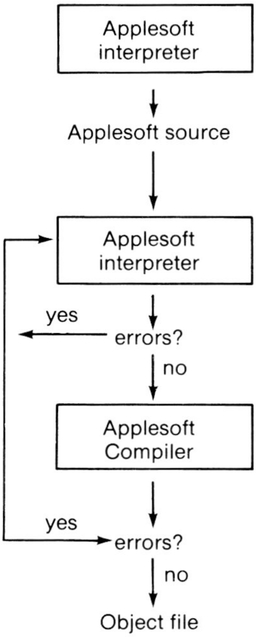
This discussion of the program development process is keyed to Figure 1. Refer to it when reading this text.

1. The best way to create and edit an Applesoft source program is to use the editing facilities of the Applesoft interpreter.

2. The source program should then be debugged using the Applesoft interpreter. Since the compiler and interpreter are very similar, running a program in the interactive environment of the interpreter provides a much quicker syntactic and semantic check than does compiling the same program.

3. Next, the source program is compiled. If compilation is successful, the compiler produces an object file.

4. Finally, the object file is executed as a machine language program. If the compiler detects errors, or if errors show up while executing the compiled program, the process starts over at step 2.



1. Create and edit Applesoft source.

2. RUN and debug source with the interpreter.

3. Compile source, creating a binary object file.

4. Execute compiled object file.

Figure 1. The Program Development Process

Chapter 3

Debugging With the Applesoft Interpreter

Debugging a program intended for compilation is a two-step process that involves

1. Creating the source program.

2. Running the program under the interpreter to check for errors.

These two steps are described in the following sections.

Creating a Source Program

Creating an Applesoft source program requires the use of the editor available within Applesoft. Programs are created by simply entering Applesoft statements from within Applesoft. Once a program has been created, it can be saved to disk with SAVE. The compiler can only compile Applesoft disk files. For more information on creating source files, see the *Applesoft II BASIC Programming Reference Manual* and the *DOS 3.3 Manual.*

Running a Program With Applesoft

Programs should be debugged before you attempt to compile them with the Applesoft interpreter. It may be necessary to debug the program with the compiler if the program to be compiled uses features that are not available in the interpreter. See "Notes on COMMON" in Chapter 7 for other debugging suggestions.

Microsoft Applesoft Compiler is highly compatible with the Applesoft interpreter. This compatibility allows the Applesoft interpreter to function as the primary debugging tool. The interpreter provides much better debugging facilities than a compiler, since it includes features for tracing execution of programs. With the interpreter, programs can also be halted and the values of variables examined, with immediate mode printing of their values. In addition, an interpreted program can be modified without having to repeat the more lengthy compilation process.

There are some drawbacks to debugging with the interpreter. Statements that are only executed under special circumstances might never be examined, whereas the compiler examines every statement in a program. The interpreter halts execution when it encounters the first error in a program; the compiler, on the other hand, can continue the compilation even if it encounters errors.

In general, compiling a program is an effective way to check for syntax errors; however, tracking program logic errors is easier with the interpreter.

Chapter 4
Compilation

This chapter discusses the more technical aspects of compilation. It addresses the following topics:

1. Options

2. Terminating Compilation

3. Compiling Large Programs

Note

If a compiled program does not run correctly, see Chapter 9, "Error Messages and Debugging," for possible solutions. Chapter 6, "Compiler/Interpreter Language Comparison," also provides information about possible problems and their solutions.

Options

The demonstration run showed only the most basic type of compilation. Microsoft Applesoft Compiler includes several options that can be used to control memory allocation and compilation. To explicitly specify the values for these options, simply answer *NO* when the compiler offers the default values, then enter a value.

Memory Usage

The memory used at runtime by the compiled code is divided into three areas:

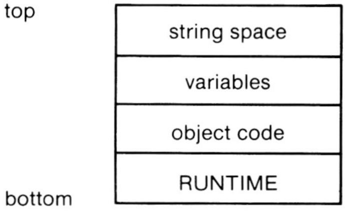
1. Runtime library

2. Object program

3. Variables

The compiler allows the location for each of these separate blocks to be specified individually. The memory allocation features can be used to protect machine language programs, HIRES shape tables, HIRES screens, or any other important part of memory.

The default allocation order of the blocks is: library, program, variables. The library is allocated lowest in memory, and the program and variables follow. The library begins at location 2051, or $803 (with the dollar sign ($) indicating a hexadecimal value). The default configuration for memory is:



Alternate addresses for the blocks are simple to specify. The new location for the library is entered as a number and defaults to $803. Addresses can be specified in either hexadecimal or decimal form. Hex addresses must be preceded by a dollar sign ($).

The library must be loaded, using the BLOAD command, before a compiled program can be run. By default, the library is loaded at $803. When a program is compiled to expect the library at a different address, the library must be loaded in at the correct address by using the "A" option with the BLOAD command. See Appendix A, Part 1, "Moving Binary Files With the ADR Utility," for more information about loading and saving the binary object and RUNTIME files.

The beginning address for the object code may be specified with

1. The word HGR1.

2. The word HGR2.

3. A decimal or hex number.

4. The <RETURN> key.

HGR1 and HGR2 simply set the beginning of the program above the appropriate HIRES screen. The 4K runtime library defaults to the space below the first HIRES screen. This default library location is suggested for programs that use HIRES (high resolution) graphics. An absolute decimal or hex address may also be specified, but care must be taken when doing so. Beware of overlapping memory allocations. Pressing the <RETURN> key causes the beginning of the object code to default to the end of the library.

Variable space may be specified explicitly or allowed to default. The beginning of variable space defaults to the end of the object code.

Compiled programs use the normal HIMEM pointer to determine the top of available string space. Strings grow downward from HIMEM. The bottom of available string space is set so that the block that is highest in memory is protected. Therefore, the normal default order (runtime library, object code, variables) sets the bottom of string storage to the end of variable space. Specifying another block to reside highest in memory sets the bottom of string storage to the end of that block. See Chapter 6, "Using HIMEM," for more information on HIMEM.

Compilation Options

There are five compilation options that can be specified before the compilation process begins. The options and their defaults are listed in Table 2.

Table 2

Compilation Options and Defaults

Compilation Option	Default
Compilation listing	YES
Pause on errors	YES
Integer arithmetic	YES
Integer constants	YES
RESUME/Debug code	NO

Answering *YES* or *NO* to the default option prompt provides a chance to turn each of these options on or off.

Compilation Listing Option

The compiler normally lists the source file. Turning the listing option off suppresses the listing. Errors, warnings, and special messages are printed as usual.

Pause on Errors Option

Errors normally halt compilation and allow the user to abort or continue compilation. Turning the pause option off suppresses the pause after any error messages are printed.

Integer Arithmetic Option

The compiler includes a full integer arithmetic package. True integer arithmetic allows operations on integers to be performed in about half the normal time. Including this option substantially increases the speed of programs that use integers, but there are some limitations. See "Integer Arithmetic" in Chapter 7 for more information on the integer arithmetic package.

Integer Constants

Constants in a compiled program can be treated as integers or floating-point numbers. Selecting the integer constants option allows constants that are used as integers to be stored in integer format. If a constant is also needed in floating-point form, the compiler includes both forms with the compiled code. Conversion of constants at runtime is totally eliminated. Including the integer constants option increases the speed of programs where constants are accessed as integers.

Integer constants take up two bytes in the object file; the floating-point representation requires five bytes. Including integer constants can slightly increase the size of the object code if both real and integer versions are needed, but it can also shorten the code if only the integer representation is required. The integer constants option should normally be left on. See "Compiling Large Programs" in this chapter for more information on handling any problems with longer programs.

RESUME/Debug Code Option

Turning on the RESUME/Debug code option causes code handling the RESUME statement to be included in the object program. The RESUME statement in Applesoft allows an error trapping routine to resume execution at the beginning of the statement that caused the error.

Including the RESUME/Debug code option requires the compiler to generate extra code at the beginning of each statement that may generate an error. Selecting the RESUME/Debug code option causes the object code to be larger and somewhat slower.

The RESUME/Debug code option *must* be turned on if RESUME statements are used in the program to be compiled. One advantage of using the RESUME/Debug code option is that any runtime error messages include the object code address. Normally, only some of the runtime errors generated by the runtime library include an object code address. The RESUME/-Debug code option can be useful for debugging with the compiler. However, including it decreases execution speed and increases the length of the compiled code. The RESUME/-Debug option should be disabled unless absolutely needed. If the option is disabled, the compiler ignores all RESUME statements.

Terminating Compilation

Ordinarily, <RESET> would be used to terminate compilation of a program. However, since the compiler often accesses the disk, using <RESET> is inadvisable. Instead, use <CTRL-C> to terminate compilation. This works because the compiler occasionally checks to see if a <CTRL-C> has been typed, and terminates compilation if it has. The compiler can also be halted by typing <CTRL-C> as the first character of an input response, but this does not correctly terminate compilation.

Since stopping compilation leaves the object file incomplete, the compiler deletes the object file if compilation is aborted. The compiler modifies DOS, so exiting by using <RESET> or typing <CTRL-C> in an input leaves DOS in its modified state. DOS must then be rebooted. Typing <CTRL-C> outside an input is the only way to correctly terminate compilation. Normal exiting of the compiler restores DOS to its normal state.

Compiling Large Programs

Features such as compact object code, disk-based design, and COMMON variables make Microsoft Applesoft Compiler an outstanding compiler for large programs. Because of the compiler's minimal object code expansion, most large programs can be compiled without modification. However, if a program is simply too large to be compiled or to fit into the available memory, special measures must be taken. The sections that follow describe possible solutions.

Reducing Symbol Table Space

The compiler's symbol table stores information about variables, functions, constants, line number references, and COMMON assignments. Long programs that exhaust symbol table space yield a "SYMBOL TABLE FULL" error during compilation. There are several ways to correct this problem.

The simplest way is to turn off the integer constants option. With the option on, constants are initially stored as integers. If the constant is later needed as a floating-point number, it is converted and entered into the symbol table as a floating-point constant as well. The initial integer entry takes five locations. If the additional floating-point entry is required, it takes up eight locations.

With the integer constants option disabled, constants are stored only in floating-point form. Therefore, turning the option off saves five locations for every constant that is referenced as a floating-point value. Although this savings is usually insignificant, it may be important for long programs with many constants.

Disabling the integer constants option also slows down the object code slightly. The option should be left on whenever possible. The symbol table is examined before each new entry is made. This examination prevents duplication of information. The first use of a variable requires the compiler to create a new entry, but later references do not require additional space. Similarly, multiple references to a single line number do not require multiple entries. In practice, it is difficult to reduce the number of variables and line number references in a program, but doing so will save some symbol table space. If the "SYMBOL TABLE FULL" error persists, a program can often be separated into two parts, as described below.

Separating a Program Into Parts

When the object code for a program does not fit in the available memory, or when a program requires too much symbol table space, the program can often be separated into two smaller programs. Programs that spend most of their time in one section of code, then move on to another section and do not return, are good candidates for this technique. A program of this type can be broken into two smaller programs quite easily. The first small program performs the first part of the process, then passes any needed values on to the second program.

Programs without a natural division present more of a problem. An artificial division can often be created with this type of program. In some cases, the separated programs may have to run each other alternately. It is preferable, however, to find a division where alternate execution is not required, since running programs alternately from disk is very slow.

Since most programs that are separated into parts need to pass values from one program to another, the usual procedure is to pass the needed values in a disk file. One program writes the information out to disk, and a second reads it in. This is a workable solution, but it is slow when a large amount of information needs to be passed.

Microsoft Applesoft Compiler is designed to simplify compiling large programs. The compiler allows you to pass the needed values in COMMON. The variables are simply declared in COMMON statements in both programs, and the compiler allocates storage so that saving the values on disk is unnecessary. This is the technique that the compiler uses to communicate between its three parts: PASS0, PASS1, and PASS2. See "Chain With COMMON" in Chapter 7 for more information about the compiler's powerful COMMON features.

Chapter 5

Executing a Compiled Program

Microsoft Applesoft Compiler is designed to implement the Applesoft language as closely as possible. Executing a compiled program performs the same functions as executing the source program using the interpreter. However, the process of loading and running a compiled program is different from running an interpreted program, because the object file is a machine language program stored as a binary file.

Differences between the compiler and the interpreter, along with other related topics, are discussed below:

1. Program disk storage

   Interpreted programs are stored as Applesoft files. Applesoft files are indicated by an "A" in the disk CATALOG. Interpreted programs are executed by typing

   ```
   RUN <filename>
   ```

   In contrast, compiled programs are machine language files stored in binary format. Binary files are indicated by a "B" in the CATALOG.

   Since compiled programs are not Applesoft files, attempting to RUN a compiled program generates a "FILE TYPE MISMATCH" error. Instead, compiled files are executed by typing

   ```
   BRUN <filename>
   ```

   The "B" prefix to the RUN command indicates a binary file. The runtime library must already be in memory when a program is BRUN. The library need not be reloaded if the runtime library is still in memory from a previous program run and is located where the next program expects it.

The normal sequence for executing a compiled program is:

```
BLOAD RUNTIME
BRUN <filename>
```

Compiled programs can only be executed when the Applesoft interpreter is in memory. Compiled programs will not work with Integer BASIC.

2. Use of Ampersand (&)

Once the compiled program has been loaded and executed, it can be reexecuted by typing an & followed by a <RETURN>. The compiled program sets the Ampersand vector to point to the beginning of the object code when it executes, so the Ampersand vector can be used as long as the program is the last program run. Using the Ampersand is much more convenient than using the CALL statement to call the code explicitly, since using CALL requires knowing the address of the beginning of the object code.

3. Halting execution of a compiled program

Typing <CTRL-C> during execution does not interrupt a compiled program. <CTRL-C> as a response to an INPUT statement does terminate execution, but <CTRL-C> at any other time is simply ignored. Unless the object program is explicitly looking for input, any characters typed are ignored.

Since <CTRL-C> is normally ignored, compiled programs must be interrupted by using <RESET>. Unfortunately, <RESET> stops the compiled program without reinitializing the interpreter. The NEW command must be used to insure that the interpreter is correctly initialized after using <RESET>.

4. The NEW command

NEW causes the interpreter to reset pointers, but does not clear the program space. Therefore, a compiled program can be safely reexecuted after a NEW command as long as no program lines have been typed in and stored into the program space.

5. Immediate commands

   The compiled code does not maintain a variable list, so the interpreter cannot find the values of variables used in a compiled program. Executing a compiled program that uses the variable A and then typing the immediate command

   ```
   PRINT A
   ```

   returns a value unrelated to the variable in the compiled program.

Chapter 6
Compiler/Interpreter Language Comparison

Microsoft Applesoft Compiler produces compact and efficient code that simulates the Applesoft interpreter in very accurate detail. This chapter describes any differences between the compiler and the interpreter that must be taken into account when compiling programs. If a compiled program does not run correctly, also see Chapter 9, "Error Messages and Debugging," for more information.

Statements Not Implemented

The very nature of compilation makes supporting some features of the interpreter impractical, since the source file is unavailable while the object code produced by the compiler is executing. Therefore, Applesoft statements that depend on the source file (such as LIST and DEL) are not available with MS-Applesoft Compiler. The cassette I/O features of Applesoft have also been removed. Most other features of Applesoft are implemented without change.

The following Applesoft statements are not included in the compiler:

CONT
DEL
LIST
LOAD
LOMEM:
NOTRACE
RECALL
SAVE
SHLOAD
STORE
TRACE
&

Some other Applesoft statements and features must be used differently in compiled programs. These differences are described in the following sections.

Features Supported With Limitations

The following Applesoft features are supported with some limitations:

DEF FN
DIM
<CTRL-C>

The differences between the compiled and interpreted versions of these statements are described below.

User-Defined Functions: DEF FN

Both Microsoft Applesoft Compiler and the Applesoft interpreter allow the definition of single-argument, real-valued, arithmetic functions with the DEF FN statement. In addition, the interpreter allows functions to be redefined with a later DEF FN statement using the same function name. The compiler *does not* support function redefinition.

In the interpreter, a DEF FN does not define a function until the DEF FN statement is actually executed at runtime. The compiler, on the other hand, scans all function definitions at compiletime. Therefore, function definitions can be located anywhere within the source file, and functions are defined from the beginning to the end of program execution. The source file cannot contain more than one definition for a given function, even if the definitions are identical.

The compiler's treatment of user-defined functions prevents "UNDEF'D FUNCTION" error messages at runtime. All user-defined function references are matched with the corresponding definition at compiletime. References to undefined functions are detected and flagged as errors during compilation.

The Dimension Statement: DIM

The interpreter provides three methods of dimensioning an array:

1. Explicit constant dimensioning

   Executing a DIM (dimension) statement in which the specified dimensions are constants sets aside the same amount of storage for the array each time the program is run.

2. Explicit dynamic dimensioning

   Excuting a DIM (dimension) statement in which the specified dimensions are arithmetic expressions sets aside space for the array depending on the computed value of the expressions.

3. Default dimensioning

   If an array reference is encountered before a DIM statement, the array is given the default maximum value of 10 for each dimension of the array. Accessing an element of a three-dimensional array before dimensioning the array produces the default dimensions (10,10,10). Applesoft allows the use of 0 as an array subscript, so an array dimensioned to 10 actually has 11 elements (0-10).

The compiler *does not* support dynamic dimensioning of arrays. Any DIM statements in the program must use *integer* constants, not floating-point values or arithmetic expressions. DIM statements can be located anywhere within the file, and they need not precede the first reference to the array. Arrays can be dimensioned more than once, but the dimensions specified must be identical. The compiler's added INTEGER and COMMON statements can also dimension arrays. See Chapter 7, "Language Enhancements," for more information on INTEGER and COMMON statements. An array that is referenced and not dimensioned receives a default dimension of 10 for each subscript.

References to an array within a DIM statement or arithmetic expression must be consistent with the number of dimensions used throughout the file. An error is generated if an array is referenced with a number of subscripts that differs from the number first used.

The lack of dynamic array dimensioning can be overcome by dimensioning arrays to the largest size likely to be needed. The array dimensions must be within the limits of the available memory.

Use of <CTRL-C> to Halt a Compiled Program

The Applesoft interpreter allows the user to interrupt execution of a program by typing a <CTRL-C>. In addition, typing <CTRL-C>followed by a <RETURN> during INPUT causes the program to halt.

Compiled code does not check for <CTRL-C> during execution. A compiled program can be halted only by using the <RESET> key. However, <RESET> interrupts cannot be trapped by compiled programs. As a result, <RESET> does not properly terminate compiled programs, and a NEW command must be executed to re-initialize the interpreter. See Chapter 5 under "Halting Execution of a Compiled Program" for more information on the use of <RESET>.

While the compiled code does not check for <CTRL-C> during execution, it *does* support the use of <CTRL-C> during an INPUT statement. Typing <CTRL-C> followed by a <RETURN> in response to an INPUT prompt causes program termination and a "BREAK IN ####" message. The compiled INPUT statement functions the same as the interpreted INPUT.

If necessary, a compiled program can simulate the interpreter's handling of <CTRL-C> during execution by periodically checking the keyboard strobe. See the *Applesoft Manual* for more information about the keyboard strobe.

Other Language Differences

A few Applesoft statements have been modified to return reasonable results under a wider variety of conditions. These are described below.

IF/THEN Using Strings

The Applesoft interpreter was not designed to allow the IF/THEN statement to test a string expression. However, the interpreter does not ensure that the expression used in an IF/THEN statement is a number. A string expression can be used in an IF/THEN statement and the interpreter will not detect the error. However, using an IF/THEN statement with a string expression more than two or three times in a program causes a "?FORMULA TOO COMPLEX" error. In addition, the logical value returned for the string is not consistent.

A string expression is not simply an expression that contains string operands. String expressions are defined as expressions that evaluate to a string result. The following examples evaluate to a string result and are string expressions:

```
CHR$(3)
A$+B$
STR$(IxJ)
```

The following examples evaluate to a numeric result and are numeric expressions:

```
A$<B$+C$
A$<CHR$(2)
FLAG AND A$<C$
```

The compiler fully supports the use of IF/THEN in all its forms with a numeric argument, but an IF/THEN statement with a string expression is flagged as an error during compilation. IF/THEN is discussed more completely in the *Applesoft II Basic Programming Reference Manual.*

Numeric GET

The Applesoft interpreter's GET statement was designed for use only with strings. However, it is possible to use it with numeric variables. Unfortunately, the interpreted GET statement may be inconvenient to use with numeric variables. The biggest problem is that the interpreted numeric GET yields a syntax error and stops the program if the input response is nonnumeric. The compiled numeric GET eliminates this problem. Entering a nonnumeric input yields a zero as the entered value, and does not generate an error message.

Numeric READ

The Applesoft interpreter does not allow numeric strings to be read into numeric variables. For example, although the following DATA statements are treated identically when they are READ into a string variable, the second generates a "?SYNTAX error" when it is READ into a numeric variable:

```
10 DATA 1234
20 DATA "1234"
```

The compiled version of the READ statement accepts "1234" and 1234 as equivalent. The quotation marks enclosing the number simply insure that no leading or trailing spaces in the DATA statement will be treated as part of "1234" if it is read into a string variable. Quoted numbers as responses to INPUT requests still cause the "?REENTER" message.

Operational Differences

This section describes several additional differences between compiled and interpreted programs. The differences are usually insignificant, but they may need to be taken into account in some cases.

The Runtime Stack

Compiled and interpreted programs use part of memory as a stack to store GOSUB/RETURN and FOR/NEXT information. The routines that are used by a compiled program are more stack-efficient than those in the interpreter. Therefore, some programs that generate stack overflow errors when executed with the interpreter can be run without problems when compiled. Although compiled programs use less stack space than interpreted programs, compiled code does *not* check for stack overflow.

There are 254 bytes of free stack space. RETURN entries take 2 bytes, and FOR entries take 16. The stack overflows if subroutine calls are nested more than 127 levels deep or if FOR/-NEXTs are nested more than 15 levels deep. Part of the stack area is used as scratch space by the Applesoft PRINT and STR$ routines. Using these statements overwrites the last 16 bytes of available stack space. Overflowing the stack or overwriting the top of the stack when it contains information causes a compiled program to behave unpredictably. Since compiled programs are more stack-efficient than interpreted programs, observing the restrictions imposed by the interpreter should prevent stack overflow when running a compiled program.

ONERR GOTO and the Stack

The ONERR GOTO statement may cause difficulties with the stack in both compiled and interpreted code. Problems occur when an internal interpreter or runtime routine is exited with an error condition. Most internal routines use a small amount of stack space as temporary storage. Exiting with an error condition may leave some stored parameters on the stack. When ONERR is not in effect, the program stops execution, and the information on the stack does not cause any problems.

When ONERR GOTO is used to handle errors, the extra stack entries are left on the stack and control is transferred to the error handling routine. The stack is restored to its pre-error state and the extra bytes do not cause any problem, if the error-handler ends with RESUME. However, if the error-handler returns without executing a RESUME, the leftover information on the stack is never removed. Repeating this process too many times causes the stack to overflow.

The *Applesoft II BASIC Programming Reference Manual* includes a machine language program that restores the stack to its state before the last error occurred. Calling this routine at the end of an error-handler restores the stack and prevents overflow. This routine also resets the stack when used in a compiled program. However, the compiled code does not usually save the stack pointer before each statement. When the routine tries to restore the stack to its previous state, it will set the stack pointer incorrectly.

Compiled programs that use the stack clear routine *must* include the RESUME/Debug code to avoid this problem. The RESUME/Debug code saves the stack pointer before each statement, so including the RESUME/Debug code allows the stack clear routine to function correctly.

Failure to include the extra code will cause the compiled program to jump into the monitor, execute parts of the program twice, or encounter other problems. Programs that use the stack clear routine can usually be recognized by CALLs in their error handling routines. See the *Applesoft II BASIC Programming Reference Manual* under ONERR GOTO for information about the stack clear routine. See "Compilation Options" in Chapter 4 for information on invoking the RESUME option.

Note

> Do not use CALL-3288. This restores the stack pointer, but returns to the interpreter, not the compiler.

Special Machine Language Calls

A few Applesoft programs use special techniques for passing information to machine language routines. The most common method is to include extra text following the machine language call. For instance, the following statements might be used:

```
USR(0)"3,5,6"
CALL 520"PROGRAM 2"
```

These statements work with the interpreter only because the machine language program can change the interpreter's pointer into the current statement and prevent the added characters from being seen. Since the compiler scans all statements at compiletime, the extra text will be flagged as a syntax error. See "How COMMON Variables Work" in Chapter 7 for more information on passing parameters to machine language programs.

Using HIMEM

The Applesoft statement HIMEM sets the maximum memory address to be used by an Applesoft program. HIMEM affects string space, since strings are stored from the top of memory downward. Changing HIMEM moves the top of memory available for string space. If some strings have already been assigned values, changing HIMEM disrupts the string pointers and leaves the strings unprotected. The CLEAR statement should be used immediately after changing HIMEM to insure that string space is initialized correctly. The CLEAR statement should be used after changing HIMEM in both interpreted and compiled programs to insure that string space is reset correctly.

Compiled programs using COMMON string variables must include an additional CLEAR COMMON statement with the normal CLEAR statement. The compiled CLEAR affects only local (non-COMMON) variables, so the CLEAR COMMON statement must be used to insure that COMMON strings are initialized correctly as well. See "CLEAR CHAIN and CLEAR COMMON" in Chapter 7 for more information about the CLEAR COMMON statement.

Using MAXFILES From Within a Compiled Program

The DOS MAXFILES command sets the number of available file buffers. The number of available buffers determines how many files can be open simultaneously. Since the compiler fully supports all DOS commands, the MAXFILES command can be used from within a compiled program. However, DOS does not perform the additional operations needed to correctly execute MAXFILES in a compiled program. DOS changes the value stored in the HIMEM location, but DOS does not alter other pointers that must be changed so that a compiled program will conform to the new HIMEM value.

MAXFILES in a compiled program must be followed by a HIMEM statement to set all pointers correctly. The HIMEM statement should simply specify the new HIMEM value provided by the MAXFILES command. This can be accomplished with the following statements:

```
HIMEM: PEEK(115) + 256 x ABS(PEEK(116))
CLEAR
REM! CLEAR COMMON
```

The ABS() function is used to force the PEEK(116) to be treated as a real number instead of an integer. In turn, this prevents the multiplication from being treated as an integer multiply. Integer numbers are limited to -32768 and +32767, and the HIMEM address may be outside this range. The CLEAR and CLEAR COMMON statements are necessary to correctly initialize string space, as mentioned in this chapter under "Using HIMEM."

Using RUN With COMMON

The Applesoft RUN statement is normally used as an immediate command from the editor, but the interpreter also allows it to be included in a program. RUN in an interpreted program clears all variables and reexecutes the program. RUN with the optional line number specified also clears all variables, but begins execution at the specified line rather than the beginning of the program.

The compiler also includes the RUN command. RUN without the optional line number reexecutes the program by jumping to the Ampersand vector. The reexecution causes the same variable initialization that began the program. If the program has no COMMON variables, or specified them with USECOMMON, then only local variables are cleared. If the program specified DEFCOMMON, then both COMMON and local variables are cleared.

Use of the RUN <linenumber> form of the RUN command causes a CLEAR statement, followed by a GOTO to the specified line. The compiled CLEAR initializes only local (non-COMMON) variables, so RUN does not affect COMMON variables. See Chapter 7, "Language Enhancements," for more information on COMMON and CLEAR.

NEW, STOP, and END

The NEW statement in an interpreted program erases the current program before terminating execution. STOP prints the message "BREAK IN ####" before termination, and END simply terminates execution.

These statements function somewhat differently when compiled. STOP still prints the BREAK message, but the number specified is an object code address, not a line number. NEW is equivalent to END: a compiled program cannot be listed or edited, so there is no need for NEW to "erase" the compiled program. All three commands delete any interpreted program in memory, initialize all Applesoft pointers properly, then reenter the interpreter.

Abnormal Termination of a Compiled Program

Although the compiled and interpreted versions of a program produce the same output, the internal process is substantially different. Both the compiled code and the interpreter make extensive use of memory page zero, which resides at $00-FF (decimal 0-255). Many of the locations used by the interpreter are used for a different purpose when a compiled program is running. See Appendix E, Part 1, "Zero Page Usage," for a description of how the zero page is used by a compiled program.

Compiled programs begin with a call to an initialization routine that sets up the zero page for execution of the compiled program. When the compiled program executes an END, STOP, or NEW statement, or is interrupted by <CTRL-C> during an INPUT statement, it reinitializes page zero for the interpreter before terminating execution. This terminates the compiled program properly.

Using <RESET> to interrupt a compiled program stops the program, but does not reinitialize the zero page for the interpreter. Runtime errors also stop the program without reinitialization. Attempting to use the interpreter at this point is unwise. Even if statements appear to function normally, the interpreter may be destroying DOS or making other errors. A NEW command is necessary to properly reinitialize the interpreter. NEW sets the pointers stored on page zero to their correct values, allowing the interpreter to operate normally.

Applesoft Pointers Preserved by Compiled Code

There are only two pointers used by the interpreter that are preserved during execution of a compiled program. These two pointers are MEMSIZ and TXTTAB. MEMSIZ is the top-of-memory pointer affected by the HIMEM statement, and TXTTAB is the beginning-of-program pointer. They reside at $73-4 (decimal 115-6) and $67-8 (decimal 103-4), respectively.

Compiled programs use MEMSIZ for the same purpose the interpreter does. The compiled HIMEM statement changes the contents of MEMSIZ.

Compiled programs do not use TXTTAB, but the interpreter uses TXTTAB to point to the beginning of the current program in memory. The interpreter and DOS also use TXTTAB to decide where a loaded program should begin. Compiled programs preserve TXTTAB so that a compiled program can easily RUN an interpreted program using DOS.

The program storage format used by the Applesoft interpreter requires that the location just before the program area pointed to by TXTTAB contains a zero. A program can still be entered when the location is not zero, but attempting to RUN the program produces a "?SYNTAX ERROR." This restriction also holds when using DOS to LOAD or RUN interpreted programs. The location does not usually need to be explicitly set to zero, since TXTTAB normally points to its default location of $801. $800 is set to zero when Applesoft is initialized, so leaving TXTTAB at its normal value alleviates any problems.

To maintain this convenience, the compiler leaves location $800 protected in its default memory allocation mode. Preserving this location allows compiled programs to RUN interpreted programs, without first having to store a zero in $800. $801 and $802 are protected for similar reasons, so the default address for the library is $803 (decimal 2051).

Linking Between Compiled Programs

Interpreted programs are linked using the DOS RUN command. Attempting to RUN an object file produced by the compiler produces an error, since compiled programs are binary files, not Applesoft programs.

Compiled programs are linked using the BRUN command. To facilitate linking, the compiler allows COMMON variables, a powerful extension that is not available in the Applesoft interpreter. Programs executed in sequence can use COMMON variables to pass information. See Chapter 7, "Language Enhancements," for an explanation of this new feature.

String Operations

MS-Applesoft Compiler handles strings differently than the interpreter. The interpreter usually duplicates string values in an assignment such as A$=B$. If thirty strings have the same string value, the interpreter normally stores the same string thirty times in memory. Copying strings makes the interpreter especially slow in applications that move strings frequently.

The compiler eliminates string copying by allowing several strings to point to the same value in memory. This eliminates the need to duplicate strings and makes operations like sorting much faster. In exchange for being very fast in assignments, the compiler is slightly slower on operations that build and take apart strings. LEFT$, RIGHT$, and MID$ are less efficient. Since these functions are normally used less often than string assignments, the modified method used in the compiler is more efficient.

"Garbage collection" usually presents the biggest speed problem in string operations. Both the interpreter and the compiler must "houseclean" when the available string space is filled. Garbage collection compacts the strings that are still being used, and eliminates any string "garbage."

The frequency of garbage collection is determined by two factors: the amount of free space available and the rate of garbage production. The amount of free space affects how much garbage can accumulate before collection is necessary. In turn, the rate of garbage production determines how quickly the space is filled.

Garbage collection is normally a lengthy process. The time required depends on how many string variables are used in a program. Large string arrays substantially slow garbage collection, since the program must look through more variables to decide whether a string should be kept or discarded. Each time garbage collection is necessary, the program must suspend execution to houseclean.

The number of times garbage collection is necessary can be reduced by increasing the string space available or decreasing the rate of garbage production. More space can be obtained by reducing the size of arrays, shortening the program, or setting DOS MAXFILES to a smaller number. The rate of garbage production can be decreased by holding operations such as LEFT$, RIGHT$, MID$, and string concatenation (+) to a minimum. The number of assignments is not a factor, because the compiler does not duplicate strings for assignment.

The time spent during each garbage collection call can also be reduced by decreasing the number of string variables. Cutting the number of variables in half usually makes "garbage collection" about four times faster, because the time required is roughly proportional to the square of the number of string variables.

Using string operations efficiently can increase execution speed, and keeping the number of garbage collection calls to a minimum results in the best performance.

Chapter 7
Language Enhancements

Microsoft Applesoft Compiler provides two major enhancements to Applesoft that substantially increase speed and programming power:

1. True integer arithmetic, including integer FOR/NEXT loops
2. COMMON variables

The compiler makes these enhancements possible by including five new statements:

```
CLEAR CHAIN        INTEGER
CLEAR COMMON       USECOMMON
DEFCOMMON
```

These new statements and the language enhancements they allow are discussed in the sections that follow.

Integer Arithmetic

The compiler includes a full integer arithmetic package. The integer math routines allow very fast operations on integers. They are a major improvement over the methods used by the interpreter.

Integer Arithmetic in Interpreted Programs

The Applesoft interpreter includes the use of integer variables, but it does not actually perform integer operations. All integers are converted to floating-point numbers before being operated on. Since this conversion is necessary each time the variable is accessed, operations on integer variables are actually slower than operations on floating-point variables.

The only advantage to using integers in the interpreter is that elements of integer arrays occupy two bytes of memory rather than five, because Applesoft does not include integer operations. The interpreter also does not allow an integer variable as the index to a FOR loop. These problems combine to make the use of integer variables in interpreted programs of little value.

Integer Arithmetic in Compiled Programs

If real variables are changed to integers, programs compiled with MS-Applesoft Compiler can be made substantially faster. However, converting a program by adding a % sign to each variable name is time-consuming. To eliminate this problem, the compiler includes the declaration statement INTEGER. INTEGER allows variables to be defined as integers without the addition of the normal % sign. As an example, the real variable I can be declared as an integer by simply including the statement INTEGER I.

Integer variables offer faster arithmetic and more compact storage, but they also limit the range of values that can be stored. Integer values must be between -32768 and +32767, inclusive. Integer variables can only store whole numbers, they cannot store numbers with fractional parts. An assignment like I%=3.5 stores 3 in I%. Any fractional part is simply ignored. Since integers do not have the same amount of precision as real variables, changing real variables to integers in a compiled program may affect the computations in the program. Since only the integer portion of values assigned to the changed variables is considered, the results of computations that used the changed variables may be different. The needed precision must be considered carefully before changing real variables to integer variables.

The INTEGER Statement

Including the INTEGER statement as a normal statement causes a syntax error if the program is run under the interpreter, since the interpreter does not recognize the INTEGER statement. To avoid this problem, the INTEGER statement must be included in a special "active" REM statement. To allow the compiler to distinguish these special REM statements, "active" REMs are distinguished from normal "nonactive" REMs by using an exclamation point after the word REM. For example, the following statement declares I as an integer variable:

```
10 REM! INTEGER I
```

The other added statements described later must also be disguised in active REMs. The new statements are ignored when the program is run by the interpreter, but they are recognized by the compiler and treated as normal statements.

The INTEGER statement can declare either arrays or simple variables as integers. Arrays are declared by including the name and dimensions in the same form that they would be used in a DIM statement. The dimensions specified must be identical to the dimensions in any other INTEGER, DEFCOMMON, USECOMMON, or DIM statements. DEFCOMMON and USECOMMON are discussed later in this chapter.

Multiple variables can be declared as integers by separating the variable names with commas. Spaces are allowed between variable names and commas, but may not be embedded within variable names or array dimensions.

MS-Applesoft Compiler includes a wild card option for INTEGER, because many programs can declare all their variables as integers. Using an asterisk (*) as the first nonspace character following INTEGER causes all numeric variables to be treated as integers. The effect of including INTEGER * is the same as putting the name of every real variable and array in an INTEGER statement.

A variable may be declared as INTEGER several times, but only the first declaration has any effect. Multiple INTEGER statements are allowed. The following declarations are all acceptable:

```
10 REM! INTEGER I,J,K,L
20 REM ! INTEGER  I,J , AB(3,9), R(3)
30 REM! INTEGER *
```

The following declarations include spaces embedded within variable names, and are therefore unacceptable:

```
10 REM! INTEGER A  %  ,A 2, A 3
20 REM! INTEGER BA(  3,  7),  D %(3,7)
```

INTEGER statements can follow or be mixed with normal inactive REMs, but they must precede all other statements in the program. The compiler scans all REM statements, and ignores those whose messages do not begin with an exclamation mark. It also ignores REMs that do not contain one of the keywords INTEGER, COMMON, CLEAR COMMON, or CLEAR CHAIN. During compilation, the compiler notifies the user that it has processed an active REM by printing "RECOGNIZED" on the screen. This message is displayed even when the listing option is turned off. These messages must be monitored carefully, since an incorrect active REM that is not being RECOGNIZED is otherwise hard to detect.

Integer FOR/NEXT Loops

The interpreter does not allow integer variables denoted by a percent sign (%) to be used as loop variables in FOR/NEXT loops. The INTEGER statement allows the compiler's integer FOR/NEXT loops to maintain compatibility with the interpreter.

Instead of modifying Applesoft syntax and allowing normal integer variables as FOR loop index variables, Applesoft Compiler's integer FOR/NEXTs use integer variables declared with the INTEGER statement. For example:

```
10 REM! INTEGER I
20 FOR I = 1 TO 10 : PRINT I : NEXT I
```

The INTEGER statement solves the problem of interpreter compatibility, since the control variable is treated as a real variable by the interpreter. The compiler recognizes the loop variable as an integer and produces special code for the loop. Since the loop variable is an integer, the initial, final, and step values are considered integers, and must be in the range -32768 to +32767. Integer loops are usually about twice as fast as their real counterparts.

Integer Operations

Integer values in an expression cannot always be operated on in integer mode. If the operation also contains real variables, or if the integer is used as an argument to a function that expects a floating-point value, the integer will be converted to floating-point form.

The use of integer operations is controlled by the setting of the integer arithmetic option. Addition, subtraction, multiplication, and negation can generate overflows in the integer accumulator, so their use is controlled by the integer arithmetic option.

If the option is on, true integer operations are performed whenever both operands are integers. If the switch is off, integer operations are only performed for comparisons and logical tests. These operations cannot generate overflows, so their use is automatic and is not affected by the setting of the integer arithmetic option.

Integer operations should be turned off only as a last resort. If including true integer arithmetic causes overflow in only a few cases, the expressions that produce overflows can be forced into real mode. For instance, if A can be temporarily assigned to a real variable before the addition and B can be added to the real variable instead, the integer mode addition of the two integers A and B produces an overflow. Since one of the addends is a real variable, B is converted to floating-point form, and the addition is forced to real mode. This forcing technique requires some small modifications to the program, but the added speed produced by using integer operations elsewhere in the program usually justifies these changes.

Many Applesoft statements and functions use integer parameters. Since the interpreter treats all values as floating-point numbers, it must compute the parameters as floating-point numbers, then convert them to integers. Similarly, a few functions return integer results and the interpreter must convert the results back to floating-point form. These continual conversions often slow interpreted programs.

Functions and statements that expect or return integer values can be made much faster by using them with integer expressions, since compiled code can evaluate expressions in integer mode. Graphics statements, string functions, and game controls all expect integer parameters. Using integer expressions as the arguments to these operations results in a significant increase in speed. Eliminating the conversions that are normally necessary often doubles the speed of the operation. Games and graphics programs usually benefit substantially from this technique. The compiled code always converts parameters if necessary, but the conversion makes the operation slower.

The following operations expect integer values:

CHR$
COLOR=
DRAW
FOR (integer)
HCOLOR=
HLIN
HPLOT
HTAB
IN#
LEFT$
LET (integer)
MID$
ON GOSUB/TO
PDL
PLOT
POKE (2)
PR#
RIGHT$
ROT=
SCALE=
SCRN
SPC
SPEED=
subscripts
TAB
VLIN
VTAB
WAIT (2 & 3)
XDRAW

Numbers in parentheses indicate which parameters are treated as integers when the operation has parameters of mixed types.

The integer arithmetic package always performs integer comparisons when both operands are integers. Since division usually produces fractional results, division is always performed in real mode. The following operations can be performed in integer mode when all operands are integers and the integer math package is turned on:

addition
negation
multiplication
subtraction

The following operations can take either floating-point or integer values without forcing conversion:

AND
FRE
IF/THEN
NOT
OR
POS

All other operations expect floating-point values. Variables should be typed as floating-point or integer depending on which way they are used most often in the program. Failing to convert variables that can normally be treated as integers will unnecessarily slow the compiled code. On the other hand, if a variable is normally needed in floating-point form, it should be left as a real variable. Matching the types of the parameters supplied to the types expected will result in appreciable speed increases.

The following operations return integer values:

ASC
LEN
PEEK
PDL
POS
SCRN

The values returned by these operations should be treated as integers whenever possible to increase speed and prevent conversions.

Using the integer features provided in the compiler will significantly improve the execution speed of compiled programs. *The importance of effectively using integers cannot be overemphasized.*

Chain With COMMON

MS-Applesoft Compiler provides the ability to pass COMMON variables between chained programs, which is another important extension to Applesoft. The addition of COMMON allows several programs to pass values without having to alternately store and recall the values to and from disk. The implementation of COMMON in the compiler is normally referred to as "blank" COMMON. Blank COMMON uses a single COMMON block. COMMON is implemented by using a static block of memory for the COMMON variables to occupy. This block of memory is left protected as the chained programs are BRUN in succession. Each program refers to the same block of values as it executes. Variables that are declared as COMMON in a program are allocated in the block in a specific order, so that variables in different programs can correspond.

The actual names used to refer to the variables do not matter. The common memory allocation is responsible for the correspondence. For instance, assume PROGRAM1 declares variable A1 as its first COMMON variable, and PROGRAM2 declares variable A2 as its first COMMON variable. Since the location of the two variables in the COMMON block is the same, the value left in variable A1 at the end of PROGRAM1 is the initial value of variable A2 at the beginning of PROGRAM2. "Creating a System of Programs," in this chapter, describes a typical use of COMMON variables.

USECOMMON and DEFCOMMON

The DEFCOMMON and USECOMMON statements are used to declare COMMON variables. Both forms of the COMMON statement are used in an active REM as follows:

```
10 REM! DEFCOMMON I,J,K
```

Multiple COMMON statements are allowed. Any INTEGER declarations must precede COMMON statements, and both must precede any other statements (except normal inactive REMs). Incorrect ordering produces the "DECLARATION" error during compilation. COMMON may include any type of variable in either simple or array form, including variables declared as integers with INTEGER.

Specifying a variable more than once in a COMMON statement generates the "DECLARATION" error during compilation. Arrays are specified in the same way as they are with INTEGER, and the same rules hold; any DIM statements must specify the same dimensions for the array.

DEFCOMMON sets up a COMMON block and initializes the variables in it. USECOMMON accepts a COMMON block already set up by another program without clearing the block.

DEFCOMMON presents a problem when several programs chain back to a main menu. Using DEFCOMMON in the menu would cause information returned from the subprograms to be erased, but USECOMMON would fail to initialize the COMMON block at all.

The solution is to add another program containing DEFCOMMON that chains to the menu program, and then use USECOMMON in the menu, as Figure 2 illustrates:

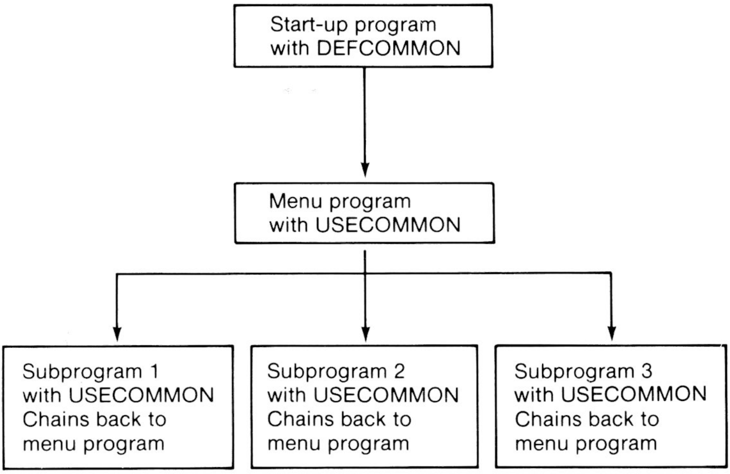


Figure 2. Using a Start-Up Program With a Menu

The start-up program is initially executed to set up the COMMON block. It then runs the menu program and is never *returned to.* This arrangement prevents the DEFCOMMON from erasing the COMMON variables each time you chain back to the menu.

CLEAR CHAIN and CLEAR COMMON

The compiled version of the Applesoft CLEAR statement does not clear COMMON variables, since COMMON variables are normally intended to be passed to a later program. The compiler provides an additional statement, CLEAR COMMON, that clears COMMON variables. CLEAR COMMON does not affect local variables. The CLEAR COMMON statement must be used in an active REM.

Programs chained with COMMON are linked as usual by using the BRUN command. However, an additional CLEAR CHAIN statement must be included immediately before the BRUN statement. The CLEAR CHAIN statement sets up string storage so that strings are properly preserved across the chain. The CLEAR CHAIN statement reinitializes all local variables and forces "garbage collection." Failure to execute a CLEAR CHAIN statement causes some string values to be lost or modified. The CLEAR CHAIN statement should be included in an active REM on the line before the BRUN command.

How COMMON Variables Work

Space in the COMMON block is allocated in the same order that the COMMON variables are declared. Integers occupy two bytes. Real numbers occupy five. String variables require two bytes each; but since they are allocated in a separate portion of the block, there is no possibility of mixing them with numeric variables. The compiler's string format is different from the interpreter's. The length byte for a string is stored at the beginning of the string value, rather than with the pointer to the string.

Arrays are allocated with the rightmost subscript varying most quickly, i.e, A(1,1) A(1,2) A(2,1) A(2,2). Each element occupies two or five bytes, according to its type.

Space in the COMMON block is divided into two subblocks: numeric variables and strings. Within each subblock, the order of variable space allocation is determined by the order in which the variables are declared in COMMON statements. When the compiled code is executed, it checks to make sure that the size of the numeric and string COMMON blocks it expects is identical to the size of the blocks actually passed to it by the previous program.

This is the extent of the type-checking performed. The compiler does not protect or prevent the user from accessing the ten bytes declared as two five-byte real numeric variables in one program as five two-byte integer variables in another program. This type of error can be easily avoided by making the COMMON declarations in the two programs identical. Mismatched COMMON block sizes produce the "?TYPE MISMATCH" error at runtime.

COMMON can also be used to pass parameters to machine language routines. The beginning of the block and the positions of the variables within the block are fixed, so machine language programs can reference static locations for the variables. The compiled program can also POKE the variable values into a predetermined location before calling the machine language routine. Machine language routines designed for interpreted programs often locate variables by looking them up in the interpreter's variable list. These routines will not work with a compiled program, since the variable list is eliminated. These programs must be modified to use COMMON or POKE values explicitly.

Notes On COMMON

Space for the COMMON block is allocated at the beginning of the space declared for program storage. Programs that share a COMMON block must all be compiled with the same starting address for the program space. As the compiler scans the COMMON statements, it increments the starting memory address of the compiled program to leave room below it for the COMMON variables. Including a GOTO to one of the REMs that declare the COMMON block causes the compiled program to jump into the COMMON block, producing undefined results.

Using the interpreter to debug programs that use the compiled COMMON statement is difficult, since COMMON is not included in the Applesoft language. However, DOS provides a machine language program that can simulate most of the facilities of the compiled chain with COMMON. See the Apple *DOS Manual* under CHAIN for more information.

An alternative debugging method is to simulate the COMMON block by using disk files. Variable values can be written to a text file at the end of one program and read in at the beginning of the next. Using the PRINT and INPUT statements to write out and read in the COMMON variables in the same order that they occur in the COMMON block will help to detect any ordering errors. This method requires modifying the programs, but it also provides a reasonably good simulation of the compiled COMMON.

Creating a System of Programs

The DEFCOMMON and USECOMMON statements are designed for creating large systems of programs that communicate with one another. A sample situation is described here to show possible interactions in a large system of programs.

Consider the following integrated accounting system containing three packages for general ledger (GL), accounts payable (AP), and accounts receivable (AR). Entry into each package is controlled by a main menu program. The system structure is shown below:

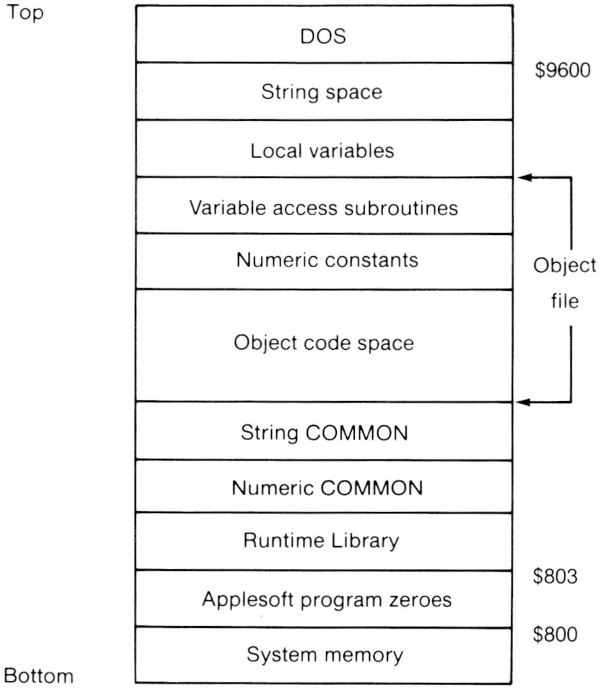


Figure 3. Typical COMMON Application

In order to use COMMON effectively, it is important to structure the system and the COMMON information logically. In the system shown in Figure 3, the subprograms within each of the three packages must pass information to one another.

There may also be COMMON information between the MENU and each of the packages. However, because COMMON blocks must be the same size in programs that communicate with one another, any information passed in COMMON from the MENU also has to be included in the COMMON blocks of the other programs.

There are two possible solutions to this problem of communicating between programs:

1. Use the same COMMON declarations in all programs so that all COMMON information can be shared.

2. Use the same set of COMMON declarations within each of the three packages, with no information shared via COMMON with the other packages or with the main MENU program. In this case, there are three sets of COMMON declarations, one for each package.

For a large, integrated set of programs, the second method allows more flexibility, since program control is switched from package to package through the main MENU. Any input information that would ordinarily be passed from the MENU would instead be obtained in the main program for each of the three packages.

The following program fragments demonstrate how the programs in the figure might be linked. The short fragments also provide examples of when to use a simple BRUN and when to use BRUN with COMMON.

MENU

```
10      D$ = CHR$(4)
        .
        .
1000    INPUT “WHICH PACKAGE? ”; N
1010    IF N = 1 THEN PRINT D$ “BRUN GL”
1020    IF N = 2 THEN PRINT D$ “BRUN AP”
1030    IF N = 3 THEN PRINT D$ “BRUN AR”
```

GL

```
10      REM! DEFCOMMON A, B(3,4), C$, D$
20      D$ = CHR$(4)
        .
        .
1000    REM! CLEAR CHAIN
1010    PRINT D$ “BRUN GL1”
```

GL1

```
10      REM! USECOMMON A1, B1(3,4), C1$, D$
        .
        .
1000    REM! CLEAR CHAIN
1010    PRINT D$ “BRUN GL2”
```

GL2

```
10      REM! USECOMMON A2, B2(3,4), C2$, D$
        .
        .
1000    REM! CLEAR CHAIN
1010    PRINT D$ “BRUN GL3”
```

GL3

```
10      REM! USECOMMON A3, B3(3,4), C3$, D$
        .
        .
1000    PRINT D$ “BRUN MENU”
```

The examples are shown for the GL package. The other two packages would be similar. Notice that the MENU does not have a COMMON declaration, since it does not pass or receive information. GL has a DEFCOMMON declaration because it must pass information on to GL1, but it does not receive information from the MENU. Declaring DEFCOMMON sets up the COMMON block and initializes it.

GL uses a CLEAR CHAIN statement before the BRUN, since GL must pass information on to GL1. GL1 declares USECOMMON, since GL1 must receive and not initialize the COMMON block passed by GL.

GL1 includes a CLEAR CHAIN statement in order to pass information on to GL2. GL2 is similar to GL1, and passes the COMMON information on to GL3. GL3 also declares USECOMMON so that it can accept the values passed to it by GL2. GL3 does not need a CLEAR CHAIN statement before the BRUN, since no information is passed from GL3 back to the menu.

Chapter 8
How the Compiler Works

This chapter explains how Microsoft Applesoft Compiler works, and attempts to make the internal operation of the compiler less mysterious. The primary function of the compiler is to translate the source program into machine language. This process is divided into two basic steps:

1. Syntax analysis

   The recognition of Applesoft statements that the compiler takes as input.

2. Code generation

   The production of machine language equivalents for the Applesoft statements.

Discussion of these two major steps is preceded by a description of the programs that perform them.

PASS0, PASS1, and PASS2

Microsoft Applesoft Compiler is a "two-pass" compiler, since it compiles in two major steps. PASS0 simply picks up user input and sets up compilation parameters, so it is not really part of the actual compilation process. The Applesoft program APCOM runs PASS0.

PASS0 and PASS1 chain to PASS1 and PASS2, respectively. All three passes were written largely in Applesoft, and MS-Applesoft Compiler was used to compile itself.

PASS1 is the first pass. PASS1 performs syntax analysis and generates most of the code. As it examines the program, PASS1 also collects information about variables and line numbers and stores the information in a symbol table.

The symbol table is used to store all the information about the program. "Compiling" all this information and inserting it where it is needed eliminates the runtime searching that occurs during interpretation.

PASS1 cannot allocate variable storage space or know the addresses of all the line numbers until it has processed the entire program. Therefore, it cannot insert actual addresses for variable or line number references into the code it generates. Instead, it must leave a trail so the addresses can be patched in later. PASS2 performs this patching after PASS1 has finished processing the source file.

PASS2 uses the information provided by PASS1 to allocate variable storage space. PASS1 keeps a record of variable usage in the symbol table. PASS2 uses the symbol table to allocate storage space. PASS2 uses the stored variable types to decide how much storage space to allocate. PASS2 then saves the address of the storage space allocated for each variable with the other information about the variable.

Line numbers are handled differently. Keeping all line numbers and addresses in memory requires too much memory, so PASS1 stores the addresses of the line numbers in the disk file "CLINENUM" as it generates the code. PASS2 uses this file to match line number references to their actual addresses.

When the actual addresses are determined, PASS2 must find all the references left undefined in PASS1. Rather than keeping a huge list of all the locations in the program where each variable is used, PASS1 links all the references for each variable or line number into a chain. Each reference contains the address of the previous reference, rather than the unknown address.

Because the file is processed forward, these chains lead backward from the current position toward the beginning of the file. PASS1 keeps the most recent reference in the symbol table. The most recent reference points backward to the next most recent, which points to the one before it, and so on.

This linking leaves PASS2 with the end of the chain for each set of references, and PASS2 uses the chains to trace backwards and patch the undefined references. This process is described as "backpatching," and it is the sole purpose of PASS2. Backpatching disk files is necessarily a slow process, but building the files on disk eliminates the program length limitations imposed by RAM-based compilation.

Syntax Analysis

Syntax analysis is the first step in the compilation process, since the compiler must be able to understand the source program before it can generate code. The syntax analyzer must examine the current line of input and decide what statement it represents before it can tell the code generator what kind of code to produce.

The process of recognizing statements can be broken into two parts: lexical analysis and parsing. These two steps are demonstrated by reading a sentence. Grouping letters into words is lexical analysis; grouping words into sentences is parsing.

Lexical Analysis

The lexical analyzer looks at the characters in the input line and separates out the words that are part of Applesoft statements. These words, like PRINT, FOR, etc., are called "keywords." A lexical analyzer might take "FORI=ABTOC" and produce "FOR I = AB TO C". The lexical analyzer substitutes a numeric "token" for each keyword it finds, making the keywords easy to recognize later. Tokenizing "FOR", "=", and "TO" in the previous example might produce "<129> I <208> AB <193> C".

The interpreter lexically analyzes each program line as it is typed in, so Applesoft programs are stored in "tokenized" format. When a program is listed, the interpreter substitutes the appropriate word for each token, and the program comes out in readable form. Programs stored on disk are also in tokenized format.

Parsing

Since the Applesoft interpreter tokenizes programs, the compiler does not have to perform lexical analysis. However, the compiler must still parse the input. Parsing is normally accomplished by one of two approaches: top-down or bottom-up. Both methods produce the same result: they take input that has been lexically analyzed and then identify logical groups of information. Top-down parsing first assumes that the input is a certain statement, then attempts to find the parts that should be there. Bottom-up parsing first examines the parts, then deduces the type of statement they represent. Most Applesoft statements can be identified by looking at the first character, so the compiler parses statements from the top down.

Arithmetic and string expressions must also be parsed.

Although they are not composed of letters and words, expressions still include symbols that the compiler must be able to organize and understand. The parser must examine the expression and decide how it should be evaluated. The parser uses operator precedence rules and any parentheses present to decide what operations must be performed first.

The code generator does not consider precedence, so the parser has to reorder the operations so they can be output sequentially during code generation. Expressions are parsed bottom-up, and reordering is accomplished with a stack.

The reordered expressions are passed to the code generator in the form of "triples." A triple is a unit consisting of an operator and one or two operands. Operations or functions that have only one operand leave the space for the other unused. The operand of one triple is often the result of a previous triple. The following set of triples represents the expression A + B x C:

Triple #	Operator	Left Operand	Right Operand
1	load	B	
2	load	C	
3	multiply	triple 1	triple 2
4	load	A	
5	add	triple 4	triple 3

Triples #1 and #2 simply indicate that the variables B and C must be loaded. Triple #3 instructs the code generator to generate a multiplication of B and C. Triple #4 indicates that the value for A must be loaded. Triple #5 specifies that A should be added to the result produced by the multiplication in triple #3.

Code Generation

Code generation takes place in two different ways. Code for expressions is generated by an explicit call to the code generator with a set of triples. Code for statements is generated during syntax analysis. For instance, the call to the CLEAR routine is generated as soon as the syntax analyzer recognizes the CLEAR statement. This method is called syntax-directed code generation. Most statements such as END, GR, and TEXT, etc., are represented by a single call to a library or Applesoft routine.

Statements that involve expressions are normally single calls also. The syntax analyzer recognizes the statement, then parses the expression. A call to the code generator produces the instructions to evaluate the expression. The syntax analyzer then finishes the code for the statement with a machine language call to the routine that uses the value.

The interpreter must search lists of variables and line numbers to find a variable or line number. The compiler, on the other hand, generates absolute addresses for variables and line numbers. The compiler is faster than the interpreter because it uses these absolute addresses instead of searching long lists.

The compiler collects all the information about the program at compiletime, rather than searching at runtime. Instead of searching for a line number when it encounters a GOTO 80 statement, the compiled code simply jumps to the address already provided by the compiler. Compilation allows program execution without searching through lists. However, the compiler must perform all searching and organization at compiletime.

Special Techniques

MS-Applesoft Compiler was designed with minimal object code expansion as one of the main objectives. Execution speed of the object code was also considered, but a slight decrease in speed is occasionally traded for a worthwhile reduction in object code size. The compiler is designed for all types of programs, but it is intended to be especially beneficial for large programs and systems. Most compilers produce object code that is significantly larger than the original source. This may create difficulties when the program to be compiled is already large. Speed is important, but a fast program that won't fit in the available memory is worthless. Therefore, the compiler uses several special techniques to generate compact code.

Variable Accessing

Variable accessing is a typical cause of code expansion. Compiled and interpreted code must both load and store values to variables. Variable accessing is required almost constantly, so most compilers use general purpose load and store subroutines. Using subroutines saves quite a bit of space. The routine used to transfer a value to and from memory need be included in the object code only once, rather than being duplicated each time it is needed.

Since these subroutines handle the load and store operations for all the variables in the program, they must be given information about which particular variable to load or store. The information passed to the subroutine is normally the memory address of the variable. The address of the variable to be operated on must be passed to the subroutine each time the routine is called. The address is normally passed in two of the microprocessor registers. Every time a variable is accessed, the same process must be repeated: load the registers, call the routine, load the registers, call the routine.

Setting up the registers to pass the address more than doubles the amount of code required for each variable access. There is a more space-efficient way to perform the same function. Instead of passing an address to a general purpose routine, it is more effective to have a special routine *for each variable* that performs the load and call.

The special subroutine for the variable takes the same amount of space as one load and call to the general purpose routine, but now each access for that variable can simply call the specialized subroutine. The subroutine does not need to be passed an address to indicate which variable it should operate on, since it is dedicated to only one variable. With the new specialized routines for each variable, the address no longer needs to be loaded before each call.

This technique takes more space if the variable is only referenced once, but saves space if the variable is referenced more than once. For example, referencing a variable twenty times takes only 67 bytes instead of the normal 140. Unfortunately, this technique works well only when the variable is referenced in one way. Since a variable can be accessed in several ways, such as loading, storing, and adding, there really should be a specialized routine for each variable *for each operation.* This would substantially reduce the space savings, since several routines would be required for each variable.

MS-Applesoft Compiler uses a more sophisticated method to overcome this problem. Rather than generate several routines for each variable, the compiler generates only one routine for each variable. This single routine has a different entry point for each type of operation on the variable.

Each different entry point loads a different number to indicate what operation is needed. All then use the same code to load the variable address. However, because the subroutine is now used to perform several different operations, it can no longer jump off to a single operation at the end. Instead, the subroutine jumps to a special library routine, and the library routine uses the operation number to dispatch and perform the correct operation.

Specialized variable accessing routines are a very effective way of solving the code expansion problem. This special accessing scheme was developed in answer to the need for compact object code and is one of the major innovations introduced by MS-Applesoft Compiler.

The Runtime Library

The compiler uses other techniques to reduce the size of the object code. Most operations have been moved out of the object code and into subroutines in the library. This increases the size of the library, but substantially reduces the size of the object code. This also allows extensive use of subroutine calls in the compiled code, and produces unbeatably compact object code with a minimal reduction in speed.

The runtime library included with the compiler has a second advantage. It reduces the amount of disk storage space required. The compiler produces object code that uses a runtime library organized in a standard format. This allows all programs to use a common library, and saves each object code file from having to include some of the same routines. With many programs on one disk, this produces a substantial space savings.

One disadvantage is that the whole library is used for every program. For programs that don't use any strings, for instance, the string routines are still included. However, large programs typically use most of the routines in the library, and small programs have enough memory left over so that including the extra routines does not present a problem. The library also fits well below the first HIRES screen, providing a way to effectively use space that might otherwise be wasted.

Chapter 9

Error Messages and Debugging

This chapter explains the error messages that can occur either at compiletime or runtime.

Compiletime Error Messages

Microsoft Applesoft Compiler produces two types of error messages: warnings and fatal error messages. Warnings simply indicate the use of statements that are ignored by the compiler, such as unsupported Applesoft features or unexecutable code. Fatal errors indicate problems that prevent successful compilation.

Warnings that indicate unexecutable code are usually caused by statements following a GOTO or RETURN on the same program line. Warnings do not prevent an object file from being created, but no code is generated for the flagged statements. Unsupported statements are simply ignored, and no code is generated for them. RESUME is ignored unless the RESUME compilation option is enabled.

Fatal errors yield both a message and an error token. The error pointer, !ERR!, appears in the incorrect statement at the point where the error was recognized. Fatal errors cause the incomplete object file to be deleted. Compilation continues only so that any other errors can be detected; code generation does not continue. The following list describes the fatal errors and their causes:

DECLARATION

INTEGER or COMMON declarations out of sequence or not at beginning of program.
USECOMMON and DEFCOMMON both declared in a single program.
Variable declared as COMMON more than once.

INCOMPLETE

Incomplete expression.
Missing right parenthesis in expression.

OBJECT CODE TOO LONG

Object code or variables for compiled program extend past 48K. See "Compiling Large Programs" in Chapter 4.

OPERAND

Illegal operand in expression.
Arithmetic constant too large.

REDEFINED

Function defined more than once.
Specified array dimensions different from the first dimensions specified.

SUBSCRIPT

First subscript missing.
Dimension not an integer constant.
Dimension negative or greater than 32767.
More than 88 subscripts.
Different number of subscripts than in first usage.

SYMBOL TABLE FULL

Compiler out of symbol table space. See "Compiling Large Programs" in Chapter 4.

SYNTAX

Missing or added character or item.
Line number greater than 65534.

TOO COMPLEX

Expression too complex.
IF/THEN nesting too deep.

TOO LONG

Input line longer than 240 characters.
Array larger than 48K declared in COMMON statement.

TYPE MISMATCH

Numeric expression where string was expected, or vice versa.
String expression in IF/THEN.

Undefined line numbers or functions produce fatal errors at the beginning of PASS2. When possible, the compiler gives the line number of the reference to the undefined item. The line number given in the error message is the *last* reference to the function or line. Any other references must be corrected as well.

Disk errors encountered during compilation cause compilation to be aborted. The DOS error code for the error encountered is printed, and compilation is terminated. Refer to the *DOS 3.3 Manual* for an explanation of the error codes.

The compiler expects a source file that has been tokenized and has had spaces removed by the interpreter; the compiler does not allow extra spaces. Additional spaces will cause syntax errors during compilation. If a file does have added spaces in it, the spaces can be removed by using the LIST command to list the program into a text file, then using the EXEC command to restore the text file back into memory. See the Apple *DOS Manual* under EXEC for more information.

Runtime Error Messages

The error messages produced by a compiled program are nearly identical to those produced by the interpreter. Two error messages are different with compiled programs. Compiled programs use the "?TYPE MISMATCH" error for a different purpose and also use the new "?MISMATCH" error.

The "?TYPE MISMATCH" error in its normal sense never occurs at runtime, since type checking is performed during compilation. The "?TYPE MISMATCH" error is used instead to indicate mismatched COMMON blocks. See "CHAIN with COMMON" in Chapter 7 for more information.

The new "?MISMATCH" error indicates that the program cannot run with the current library. The "?MISMATCH" error occurs if a program compiled with 48K or 64K is run using the library from 128K MS-Applesoft Compiler. The 128K library can be used only with 128K object code.

Since the compiled code is a machine language program, it does not print line numbers in error messages. However, whenever it can, it provides an object code address. This object code address can be matched with the corresponding line in the Applesoft source program by using the line-number-to-object-code-address table offered at the end of PASS2. If a program has errors, it should be debugged again with the interpreter.

Some errors that occur at runtime cannot include an object code address. These errors are printed with an address of 0. Using the ONERR/Debug option makes it possible for the compiler to always include an object code address. However, including this option produces a longer and slower object file. For this reason, program development and debugging should be done with the interpreter whenever possible.

Runtime errors cause a compiled program to halt without correctly reinitializing the interpreter. Typing NEW after a runtime error ensures that the pointers and locations used by the interpreter are set up correctly.

Sources of Common Problems

The compiler is exceptionally compatible with the Applesoft interpreter, but there are still some problems that may occur with compiled programs. Chapter 6, "Compiler/Interpreter Language Comparison," explains most of the language differences between compiled and interpreted programs. This section is a troubleshooting guide that discusses other problems that might arise.

The Applesoft Interpreter

Compiled programs can only be executed when the Applesoft interpreter is available at its normal location in memory. Attempting to execute a compiled program without the Applesoft interpreter prevents the compiled code from running. Attempting to execute a compiled program from Integer BASIC rather than Applesoft produces the same problems. The Applesoft interpreter may be present in any form: ROM, RAMCard, etc. Compiled code will *not* run with the old version of Applesoft that loads in as an Integer program.

HIRES Graphics

The compiler fully supports all HIRES graphics without modification, but the HIRES screens and shape tables often cause memory conflicts. Applesoft uses two areas of memory that correspond to the two HIRES "screens," HGR and HGR2. HGR resides in the area of memory from 8192 to 16383 (hex $2000-$3FFF), and the area from 16384 to 24575 (hex $4000-$5FFF) is used by HGR2. These areas of memory are mapped to the screen to display the HIRES graphics.

The HGR and HGR2 commands write zeroes through the appropriate area of memory. This zeroing destroys any part of a compiled program, library, or variables that extends into this area. If part of one of these blocks is destroyed, the compiled code behaves unpredictably. This problem is usually easy to recognize and correct. The problem occurs only with programs that use HIRES graphics, and usually occurs shortly after these programs attempt an HGR or HGR2 statement.

Problems with the HIRES screens can be prevented by examining the statistics provided at the end of compilation. The addresses included indicate where the program, library, and variables reside. If the program uses HIRES graphics, check the addresses to ensure that they do not conflict with the addresses given above for the appropriate HIRES pages. If there is a conflict the program must be recompiled, using the alternate memory allocation methods discussed in Chapter 4, "Compilation."

Shape tables used by HIRES graphics often cause more subtle problems. Shape tables must be loaded with the BLOAD command or poked into memory with the POKE statement just like machine language programs, and often present the same memory conflict problems. The shape table may reside in the same place that the compiled code is expected to occupy.

In fact, shape tables for interpreted programs tend to be stored exactly where programs compiled to fit around the HIRES pages reside. The interpreted program normally fits below the HIRES page, so the shape table is often kept just above the page. However, this is exactly where the program resides if it is compiled using the HGR1 and HGR2 memory allocation options. When shape tables are used, their location can usually be determined by looking at the addresses specified in the POKEs that put them in memory. If there is a conflict, it is usually simpler to relocate the shape table. However, the program can also be compiled at a different address by using the information in Chapter 4, "Compilation."

Normal LORES graphics do not present a problem. The memory mapped to the screen by LORES graphics is the same as the memory mapped in normal text mode. This area is out of the way of both compiled and interpreted programs.

Machine Language Programs

Machine language programs used with Applesoft often depend on variables or program lines residing in a specific place. Since compilation totally changes the internal representation of the program, some machine language programs simply do not work with compiled code.

When a machine language program does not work because it depends on characteristics of the interpreted program, the machine language routine must usually be rewritten. However, some machine language programs create problems only because they present memory allocation conflicts.

Machine language programs are usually written and assembled to reside at a certain address. The Applesoft program normally puts the machine language program in memory using BLOAD or POKE. CALLs, POKEs, PEEKs, and BLOADs often indicate the use of a machine language program.

Some machine language programs are also tacked onto the end of Applesoft programs, so that the machine language routine is loaded in with the Applesoft file. These machine language programs are harder to detect, and make successful compilation difficult. “PENNY ARCADE” on the Apple demonstration disk is an example of this type of program. The original author is usually the only one who can easily straighten out these programs.

Machine language programs are normally called with the CALL statement from the Applesoft program. The addresses specified in the CALLs or POKEs are usually a good indication of whether or not the machine language programs will cause problems. Most routines are written to reside in page 3, which is unused by both interpreted and compiled programs. Parts of page 3 are used by DOS, but the majority of it is free for use. Page 3 occupies locations 768-1023 (hex $300-3FF).

Routines located in page 3 usually do not cause problems. Page 3 is not used by the compiled code, so the machine language program should not present a memory conflict. However, some routines present other compatibility problems.

One common page 3 routine is the stack clear routine used with ONERR GOTO. CALLs into page 3 (decimal 768-1023, hex $300-$3FF) from ONERR GOTO handlers are usually calling this routine. See “ONERR GOTO and the Stack” in Chapter 6 for more information about the ONERR machine language routine.

Self-Modifying Programs

Some interpreted programs achieve special effects by modifying themselves as they run. PHONE LIST on the Apple demonstration disk is an example of this unusual technique. The interpreted version of PHONE LIST uses the POKE statement to modify DATA statements inside itself, saving the data by modifying itself as it runs. Other programs “hide” parts of themselves to speed execution. These programs do not work properly when compiled. These programs must be rewritten to remove the self-modification before they will run successfully after compilation.

Appendix A

Moving Binary Files With the ADR Utility

Binary Files

Microsoft Applesoft Compiler compiles Applesoft files to produce machine language files. Since the machine language files are not Applesoft programs, their format on disk is different from that of Applesoft programs. This appendix describes how to load and save the machine language files.

DOS currently uses three main file types: Applesoft, Text, and Binary. The type of a file is indicated by a single letter in the CATALOG ("A," "T," or "B"), which represent Applesoft, Text, and Binary files. The input files to the compiler are Applesoft programs, so they are indicated by an "A" in the catalog. The machine language files produced are stored in binary format, and they are indicated by a "B" in the catalog.

Saving and Loading Binary Files

The usual LOAD <filename>, SAVE <filename>, and RUN <filename> DOS commands work with Applesoft files, but they cannot be used with binary files. DOS provides a corresponding set of commands for binary files—BLOAD, BSAVE, and BRUN. The "B" prefix to the commands denotes a binary file. The commands function much the same as the corresponding commands for Applesoft files, but there are some differences.

The DOS SAVE command stores Applesoft programs on disk by using information from the Applesoft interpreter. The interpreter provides the information about the beginning and length of the program to be saved, and DOS writes the program to disk. DOS saves the length information with the file so the program can be loaded back into memory later. The user does not have to worry about telling DOS where the program starts or how long the program is: the Applesoft interpreter provides DOS with all the necessary information.

The Applesoft interpreter cannot provide the information necessary to load and save machine language programs. The interpreter keeps track of where the current Applesoft program resides, but it does not retain information about machine language programs. Therefore, the user must specify additional information to BLOAD and BSAVE machine language programs.

The BSAVE command is similar to the SAVE command for Applesoft programs. BSAVE saves a portion of memory to disk in binary format. Compiled programs are treated like any other binary data in memory. Since the Applesoft interpreter can no longer provide the beginning and length information necessary, the user must specify the starting address and length of the memory to be saved by the BSAVE command. The normal syntax for the BSAVE command is:

```
BSAVE <filename>, A <address>, L <length>
```

The two letters "A" and "L" precede the numbers that indicate the starting address and length of the memory to be saved. The numbers can be specified in either hexadecimal or decimal format. Hexadecimal numbers are base sixteen, and decimal numbers are the normal base ten. If hexadecimal numbers are used, they must be preceded by a dollar sign "$". Decimal numbers are used without any special characters. The usual slot, drive, and volume parameters can also be used with BSAVE, BLOAD and BRUN.

BLOAD and BRUN are similar to the Applesoft file commands LOAD and RUN. LOAD for Applesoft programs simply moves the program into memory; RUN loads the program, then runs it. Similarly, BLOAD loads a section of memory from a binary disk file; BRUN loads the section, then executes it. Since the binary files stored by the compiler are machine language programs, BLOAD will load a compiled program; BRUN loads the program and runs it. As explained in Chapter 5, "Executing a Compiled Program," the runtime library must be in memory before a compiled program can be BRUN.

BLOAD normally loads the contents of the disk file back into the same area of memory from which the file was saved with BSAVE. For instance, loading a machine-language program with the BLOAD command that was saved with the BSAVE command starting from location 4000 loads the program back in at location 4000. However, it is also possible to BLOAD a disk file to a different location in memory. The "A" parameter can be used to specify a different starting address for the file. Typing "BLOAD <filename>, A 6000" always loads the binary file into memory starting at location 6000.

The "A" parameter cannot be used to load and run an object file at an address different from the one at which it was compiled. Compiled programs will not run correctly if they are run at a location different from where they normally load.

The "A" parameter with BLOAD is normally used only to BLOAD the runtime library at an address different from its default. The default address for the library is 2051 (hex $803), and because the library was saved with BSAVE to disk from that address, it normally loads at that address when the BLOAD command is used. Specifying the "A" parameter allows the library to be loaded at an address different from its default. The "A" parameter is also accepted in the BRUN command, but as just mentioned, compiled programs cannot be run at an address different from their normal location. Neither BLOAD nor BRUN can include the "L" parameter, since the whole disk file is always loaded. See the *DOS 3.3 Manual* under "Binary Files" for more information about binary file commands.

The ADR Utility

Moving compiled programs from disk to disk is not as simple as moving Applesoft programs, since the address and length of the memory range must be specified in the BSAVE command. The compiled program can still be loaded with BLOAD without specifying any extra information, but it cannot be saved with BSAVE without knowing its beginning address and length. The ADR utility is included in the MS-Applesoft Compiler disk to make finding the required information simple.

DOS saves the address and length of the binary file most recently loaded. The ADR utility uses the saved information to print the beginning and length of the most recently loaded file. ADR is a text file, so it should be executed by typing "EXEC ADR". ADR does not affect any program or file in memory; it simply executes a PRINT statement to provide the needed information. Since the PRINT statement is only understood by the Applesoft interpreter, ADR will not work from the monitor.

ADR prints out the decimal beginning address and length of the most recently loaded file. The numbers printed should be used with the "A" and "L" parameters to BSAVE the machine language program to disk. The normal sequence for moving a program from one disk to another is:

```
BLOAD <filename>
EXEC ADR
BSAVE <filename>,A<address>,L<length>
```

The address and length for the BSAVE command are found by executing the ADR file with EXEC and are simply included in the BSAVE command, preceded by "A" and "L". The same procedure can be used to move the binary files on the compiler disk—RUNTIME, PASS0, PASS1, and PASS2.

The program CREATE ADR is included so that the ADR file can be used on other disks. ADR is a text file and cannot be LOADed and SAVEd or BLOADed and BSAVEd to transfer it to another disk. Instead, the CREATE ADR program must be used to write a copy of the text file onto the new disk.

The procedure for transferring the ADR file to a new disk is simple:

1. Load the CREATE ADR program from the MS-Applesoft Compiler disk.

2. Remove the compiler disk.

3. Insert the disk that the new copy should be created on.

4. Type RUN.

The CREATE ADR program opens up a text file called "ADR" on the disk, writes the PRINT commands into it, then closes the file and stops. The new ADR file will be identical to the ADR file on the compiler disk. See the *DOS 3.3 Manual* under "EXEC" for more information on EXEC text files.

Appendix B
Creating a Turnkey Disk

It is quite simple to make a turnkey disk that runs a compiled program when the disk is booted. The Applesoft program given in this section loads the runtime library and runs the desired file. Typing this program in as the HELLO program on a disk allows the compiled object code to be executed when the disk is booted. See the *DOS 3.3 Manual* for more information on creating turnkey disks.

The following program loads and executes the binary file PROGRAM.OBJ. To use it, change PROGRAM.OBJ to the name of the actual program.

```
10  PRINT CHR$(4) + "BLOAD RUNTIME" + CHR$(13) +
          CHR$(4) + "BRUN PROGRAM.OBJ" + CHR$(13)
```

CHR$(4) is the disk character <CTRL-D>. CHR$(13) yields a <RETURN>. The PRINT statement executes two disk commands—a BLOAD and a BRUN. Both <RETURN>s are critical: the first separates the two commands, the second insures execution of the BRUN command.

The Applesoft program itself is usually overwritten by the loaded runtime library, so both the BLOAD and the BRUN must be in one PRINT statement. If they were in separate PRINTs, the second PRINT would be destoyed by the BLOAD before it could be executed. This is also why concatenation (+) is used to join the strings. If the strings are not actually concatenated, each substring is output separately, and the same problem occurs.

Appendix C
Notes on Applesoft

Microsoft Applesoft Compiler is designed to implement the features of Applesoft as closely as possible, so there are very few differences between the interpreter and the compiler. The differences that do exist are explained in Chapter 6, "Compiler/Interpreter Language Comparison." This appendix further explains some of the features and peculiarities of Applesoft. It contains some information about Applesoft statements that is not included in the *Applesoft II BASIC Programming Reference Manual.*

Information is included on the following:

1. TAB and SPC in PRINT statements
2. Reentry of parameters after an error in INPUT
3. Screen wrap-around with DRAW and XDRAW
4. The ONERR GOTO statement

TAB and SPC Functions

The operation of TAB and SPC is different than might be expected. When used as the last item in a PRINT statement, TAB and SPC act as if they are followed by a semicolon, and suppress the printing of a carriage return.

For example, the following program lines are equivalent:

```
PRINT "SAME" TAB(10) : PRINT "LINE"
PRINT "SAME" TAB(10); : PRINT "LINE"
PRINT "SAME" TAB(10) "LINE"
```

All three forms PRINT:

```
SAME     LINE
```

The INPUT Statement

The description of the INPUT statement in the Applesoft manual explains the "?REENTER" message in detail. However, when the "?REENTER" message is displayed, the *entire* INPUT statement is reexecuted. Consider the following response to the statement "INPUT A,B,C":

```
2,3,"NOT A NUMBER"
```

The "?REENTER" message is printed because the string "NOT A NUMBER" cannot be assigned to the numeric variable C. However, Applesoft is requesting the reentry of all three inputs, starting with the value for A. It is not just expecting another single input for C.

DRAW and XDRAW Commands

DRAW and XDRAW are HIRES commands for plotting shapes. DRAW and XDRAW require legal X, Y coordinates to specify the starting point for plotting the shape. However, the shape is not necessarily a single point, so it can occupy space on either side of the specified point. As long as the starting point is on the screen, plotting a shape near the edge of the screen does not yield an "?ILLEGAL QUANTITY ERROR."

If the shape extends past the edge of the screen, the Applesoft DRAW routines wrap the off-screen portion around to the other side. This is not usually a problem, but it can produce abnormal displays.

ONERR GOTO Statement

The interpreted ONERR GOTO has a problem that causes the interpreter to ignore any statements following an ONERR GOTO statement in a given program line.

The statement PRINT "AND AFTER" in line 10 of the following program is never executed:

```
10 PRINT "BEFORE" : ONERR GOTO 30 : PRINT "AND AFTER"
20 STOP
30 PRINT "ERROR"
```

Placing statements after an ONERR GOTO statement is reasonable, since the ONERR GOTO statement does not necessarily transfer control when it is executed. Unfortunately, however, the interpreter *always* ignores them. To avoid any confusion, the compiler handles ONERR GOTO in the same way as the interpreter.

Appendix D
Runtime Memory Map

Figure 4 shows the default memory configuration for a compiled program. The default can be modified by explicitly specifying the memory addresses at the beginning of compilation. See Chapter 4, “Compilation,” for more information.

String variables point into a string area that contains the actual string values. The string values grow downward from the top of memory toward the bottom of the available space. When the strings fill the available memory, the compiler forces garbage collection and frees any unused space. If not enough space is available after garbage collection to store the next string, the compiled code yields an “?OUT OF MEMORY” error.

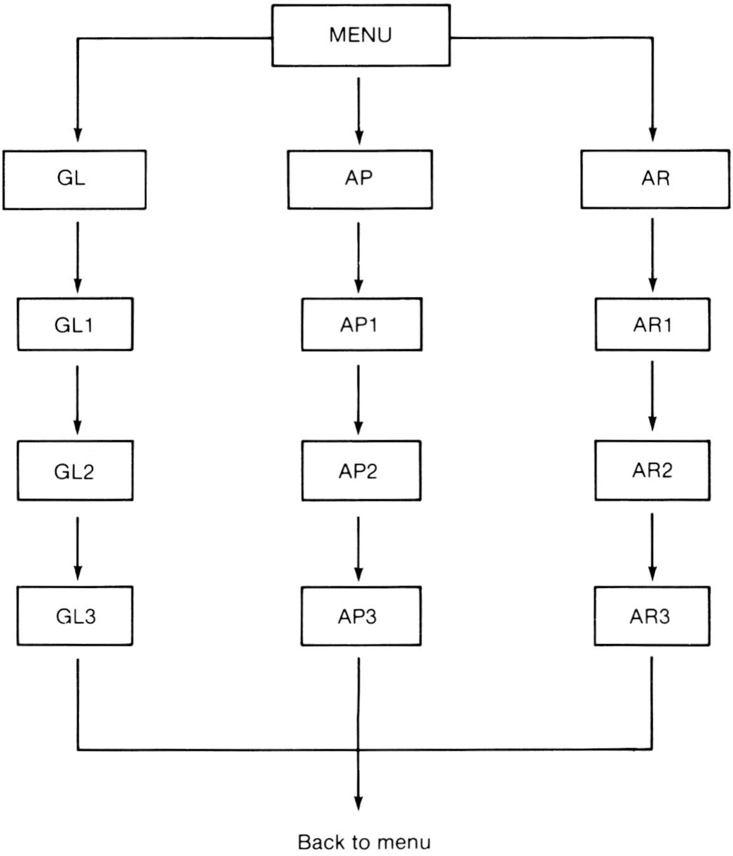


Figure 4. Runtime Memory Map

Appendix E
Zero Page Usage

Compiled programs use routines from the Applesoft interpreter and the runtime library. Both the interpreter routines and the runtime library make extensive use of a portion of memory called the "zero page." The zero page resides from $00 to $FF (decimal 0-255).

Both page zero and page 3 ($300-$3FF, or decimal 768-1023) are also often used by short machine language programs. Compiled code changes the Ampersand vector at $3F5 to allow reexecution of the compiled program, but does not otherwise modify page 3.

Zero page usage by compiled programs is largely confined to the locations that are already used by the Applesoft interpreter. The only zero-page locations used by compiled programs that are *not* normally used by the Applesoft interpreter are locations $17-$19 (decimal 23-25). The map below shows the zero-page locations used differently by compiled programs.

Zero Page Usage by Compiled Programs

Hex address	Usage
OD-OE	Temporaries
10	Temporary
12-13	Address of first DATA string
14-15	Highest location used by library, program, or variables
16	Temporary
17-19	JMP to library AYINT routine
50-51	Temporary
55-56	Temporary
58-5A	JMP to library routine that floats integer accumulator
5B-5D	JMP to code currently handling ONERR
62-66	Temporaries
69-6A	Beginning of COMMON block - 1
6B-6C	End of numeric common
6D-6E	End of string common
6F-70	Bottom of string storage - 1
71	Input buffer pointer
75-76	Address of object code being executed
77-7A	Temporaries
7B-7C	Beginning of local numeric variables - 1
7D-7E	End of local numeric variables
7F-80	End of local string variables
81-84	Temporaries
8A-8E	Temporaries
93-96	Temporary
9E-A3	Integer accumulators
AD-AE	Temporary
AF-B0	Address of library routine that pops FOR entries off the stack
B1-C8	Applesoft CHRGET, modified, but still functions normally
D9	Temporary
DC-DD	Temporary
F6-F7	Address of current DATA string

Chapter 10
64K Microsoft Applesoft Compiler

The 64K extended memory version of the Microsoft Applesoft Compiler disk provides two features not available on the standard 48K disk. First, the DOSMOVER program included on the disk relocates DOS from the main 48K of memory, freeing an additional 10.5K of memory for running compiled programs.

Second, the 64K version of MS-Applesoft Compiler uses the memory freed with DOSMOVER to speed compilation, producing the same object code as the 48K compiler in about half the time. These added features are important for developing large programs since they increase the amount of runtime memory and shorten compilation time.

The 64K version enhancements use the extra 16K bank of memory available in a 64K machine. The extra 16K allows DOSMOVER to relocate Apple DOS out of the main 48K and into the extra 16K bank. This relocation is not possible on Apple computers with less than 64K of memory. Apple computers that have 64K of memory and can therefore take advantage of these features include the 48K Apple II or II Plus machine with unused 16K memory cards, and Apple IIe, which comes standard with 64K of RAM.

The 64K version compiler disk contains the following files:

1. APCOM—64K version of Microsoft Applesoft Compiler.

2. PASS0, PASS1, PASS2—Internal subprograms of 64K Applesoft Compiler.

3. RUNTIME—Runtime library, same as 48K APCOM.

4. DOSMOVER—DOS relocation program.

5. RENUMBER.UPDATE—Updated version of Apple's Applesoft RENUMBER program.

6. BALL—Demonstration program.

These files and their uses are described in Part 2.

Chapter 11
The DOSMOVER Program

Apple DOS normally loads into the top of the main 48K of memory. This leaves the last 16K of memory on 64K Apple computers free. DOS normally occupies 10.5K of the main 48K, so moving DOS into the extra 16K frees up 10.5K of main memory. The main memory freed allows faster compilation and the execution of larger compiled programs. This relocation is the function of the "DOSMOVER" program provided on the 64K compiler disk. The DOSMOVER program moves DOS into the extra 16K, freeing the memory normally occupied by DOS. In addition, the DOSMOVER program also modifies DOS, as described in the next two sections.

Speed Improvements

Normal Apple DOS is slow in loading and saving large files, so DOSMOVER modifies DOS to increase its speed. The modifications affect the LOAD, SAVE, RUN, BLOAD, BSAVE, and BRUN commands, increasing their speed by several times without otherwise changing their operation. The modifications do not affect text file I/O.

Using DOSMOVER

DOSMOVER is simple to use. Running DOSMOVER with DOS in its normal position moves DOS into the extra 16K of RAM. Executing DOSMOVER with the BRUN command when DOS is not in its normal location prints "DOS NOT AT NORMAL ADDRESS" and leaves DOS unmodified. In either case, DOSMOVER overwrites any program in memory.

Relocating DOS frees up the top of main memory, so DOSMOVER sets HIMEM to 49152 (hex $C000). This allows the newly available memory to be used by both interpreted and compiled programs. The main memory from 0-2047 ($0-$800 hex) is used by the system and text screen, so this 2K is always occupied. This leaves 46K of the main 48K available. DOS normally occupies 10.5K of memory. When DOS has not been relocated, there is only 35.5K (46K - 10.5K) of usable memory. When DOS has been relocated, the full 46K is available.

The 10.5K of main memory gained by relocating DOS can be important for several reasons. Long compiled programs may need the extra space for object code or variables. The extra space is also useful as string space. Some programs "garbage collect" often because of frequent string operations or small string space. Increasing the size of the string space decreases the frequency of garbage collection.

When DOS has been relocated with DOSMOVER, it is accessed like normal DOS in most instances. Relocated DOS is compatible with compiled or interpreted programs that access DOS through the normal command interface. The normal interface from an Applesoft program is

```
PRINT <CTRL-D> <command>
```

Most programs use this interface.

Disk utility programs often do not use the normal interface. Utilities like the Apple COPY, FID, and MUFFIN programs access DOS at a low level. Instead of using PRINT <CTRL-D> statements, these programs use machine language routines that call entry points in DOS. When DOS is relocated, these entry points are moved.

Utilities that depend on these entry points do not run correctly when DOS is relocated. Usually such programs will hang, but they might also execute incorrectly. If DOSMOVER has been used to relocate DOS, DOS must be rebooted before running programs that access DOS at a low level. Since rebooting is inconvenient, grouping the use of these programs saves time.

The DOS INIT command is absent in the relocated DOS to provide space for the added features. The relocated DOS cannot initialize disks, so normal DOS must be rebooted if the INIT command is needed. Initialization should be grouped together with copying, etc., to reduce the need for rebooting.

The relocated DOS requires additional routines to switch in the extra 16K of memory whenever DOS is needed. These routines occupy part of page 3 in main memory. This is the only main memory space taken up by the relocated DOS (see Appendix B, Part 2, "DOSMOVER Technical Notes," for more information).

The maximum number of files that can be open simultaneously (MAXFILES) is restricted to five with relocated DOS. Requesting a MAXFILES allocation of more than five generates a range error. Few programs access more than five files simultaneously. Those programs that do should be modified or executed with unrelocated DOS.

The ?ADR Command

The locations for the beginning and length of BLOADed files are no longer easily accessible when DOS is relocated. Using the PEEK statement or using the ADR utility from the 48K MS-Applesoft Compiler disk no longer gives the desired information. To overcome this problem, DOSMOVER modifies the relocated DOS to include a binary address print command, ?ADR.

The ?ADR command prints out the beginning address and length of the last binary file loaded. The ?ADR command gives the same information as executing the ADR file with EXEC with unrelocated DOS, and is used in the same fashion. After loading or running a file with BLOAD or BRUN, just type ?ADR instead of EXEC ADR. The name ?ADR is identical to the short form of the BASIC statement PRINT ADR, and is therefore easy to remember.

The address and length reported by ?ADR are printed in hexadecimal, as indicated by the dollar sign preceding the number. The addresses are printed as *A$<hex address>, L$<hex length>*, instead of *A<dec address>, L<dec length>*, as with the ADR exec file. DOS accepts hexadecimal addresses, so no conversion to decimal is necessary—just use the dollar sign and hex numbers in the same way as the decimal numbers printed by executing ADR with EXEC.

Chapter 12

Compiling With the 64K Version of Microsoft Applesoft Compiler

Compiling with the 64K extended memory version of Microsoft Applesoft Compiler is very similar to compiling with the 48K version. Both compilers look the same, but the internal process used in the 64K version takes advantage of the memory freed by relocating DOS. The 64K version of MS-Applesoft Compiler uses the additional memory for disk buffering, allowing faster compilation than with 48K MS-Applesoft Compiler. However, although the compilation *process* used in the 64K extended memory version is different from the *process* used in the 48K standard version, the object code files that *result* from the two compilers are exactly the same.

Since the object code produced by the 48K version and the 64K version is the same, the runtime libraries on the disks of 48K and 64K versions of MS-Applesoft Compiler are identical. The runtime library is included on both disks because it is used by the compilers during compilation.

Using the 64K extended memory version of the compiler is different from using the standard 48K version in one very important way. Since the 64K version needs the memory used by DOS, DOSMOVER must be used to relocate DOS before using 64K MS-Applesoft Compiler. The 64K version cannot compile if DOS has not been relocated. Running the 64K compiler when DOS has not been relocated causes "DOS NOT RELOCATED—CAN'T COMPILE" to be displayed as soon as the compiler begins to run. When this occurs, either use 48K MS-Applesoft Compiler without relocating DOS, or relocate DOS with DOSMOVER and restart the 64K version of the compiler.

DOSMOVER must be used to relocate DOS before you use the 64K version to compile a program. No other DOS relocation program is recognized by 64K MS-Applesoft Compiler. Running the 64K version when DOS has been relocated by some method other than running DOSMOVER with the BRUN command generates the "DOS NOT RELOCATED—CAN'T COMPILE" error message.

The comments in this chapter about using DOSMOVER with the 64K extended memory version of the compiler concern *compilation only*. DOS must *always* be relocated to *compile* with 64K MS-Applesoft Compiler. Only long compiled programs that would overwrite DOS require *execution* with relocated DOS. The next section discusses *executing* compiled programs with relocated DOS.

Chapter 13
Executing Compiled Programs With Relocated DOS

This section discusses executing programs with relocated DOS. It does not discuss differences between executing programs compiled with the 64K version and the 48K version of Microsoft Applesoft Compiler. The object code produced by the 64K extended memory version is identical to the object code produced by the standard version. Executing programs compiled with the 64K version is no different from executing programs compiled with the 48K version.

There are two likely sources of problems in executing compiled programs when DOS has been relocated. First, only programs that access DOS through the normal command interface (PRINT <CTRL-D> <command>) execute correctly when DOS has been relocated. This limitation is caused by the low-level differences between normal and relocated DOS. This restriction applies to both interpreted and compiled programs. Compiling a program does not introduce any additional restrictions.

The restriction on the type of DOS access permitted is the most likely source of problems when DOS is relocated. If a compiled program does not run with relocated DOS, try running the program with normal DOS. If the program runs correctly with normal DOS but does not run with relocated DOS, the program must access DOS at a level lower than the command interface. The program must be modified or run with unrelocated DOS.

The second common problem is forgetting to relocate DOS before running a program that uses the memory normally occupied by DOS.

Warning

When a compiled program or its variables extend into the area normally occupied by DOS, running the program without relocating DOS destroys DOS. If the program object code loads in over DOS, then DOS usually fails while executing the BRUN or BLOAD commands. If only the variables overwrite DOS, the problem usually occurs after the program starts running. The initial zeroing of variables may destroy DOS, or DOS may destroy variables during program execution.

Conflicts with DOS are particularly dangerous, since destroying DOS usually hangs the computer and may destroy a disk. Check the memory usage information provided at the end of compilation to see if the program or variables extend into the space normally occupied by DOS (memory locations 38400-49151, $9600-$BFFF hex). Programs that extend into this space must be run with DOS relocated. Adding a suffix like .LONG to the object filename (specifying PROGRAM.OBJ.LONG) helps keep track of programs that must have DOS relocated.

Appendix A
RENUMBER.UPDATE

RENUMBER.UPDATE is a slightly modified version of the Applesoft RENUMBER program supplied on the Apple DOS 3.3 System Master disk. The original Applesoft RENUMBER program works only when HIMEM is set at 38400 (hex $9600), the value for a 48K Apple with DOS. The value of HIMEM changes when DOS is relocated, and the uncorrected Applesoft RENUMBER program does not function correctly.

RENUMBER.UPDATE works with the HIMEM values for both normal and relocated DOS. RENUMBER.UPDATE also works with any other HIMEM values that are multiples of 256 (hex $100). Be sure to use only the corrected copy, since the uncorrected RENUMBER program does not work properly when DOS is relocated.

Using the uncorrected RENUMBER program by accident is inconvenient but not usually fatal. The uncorrected RENUMBER program jumps into the monitor with a beep instead of returning to Applesoft after renumbering. To recover, type <CTRL-C> <RETURN>, then type <CLEAR>. List the program to make sure it is intact. If the program is destroyed, it must be reentered or reloaded from disk.

RENUMBER.UPDATE prints “RELOCATION UPDATED” just below the normal “APPLESOFT RENUMBER” line in the header displayed when running the RENUMBER program. Checking the header is an easy way to determine whether a particular copy of the RENUMBER program is an original or corrected version. Name the original and corrected versions of the RENUMBER program differently to prevent mistakes.

Appendix B
DOSMOVER Technical Notes

This appendix provides detailed technical information about the relocation methods used in DOSMOVER. An understanding of the information in this appendix is not required to use DOSMOVER.

Relocation Information

DOSMOVER accomplishes two tasks. First, DOSMOVER modifies DOS to increase the speed of the load and save routines. These modifications are discussed as "speed changes." Second, DOSMOVER modifies and moves DOS into the top 16K of bank-switched memory. The modifications necessary for relocation are discussed as "banking changes."

DOSMOVER performs the relocation in the following order:

1. Insures DOS is in normal position by checking that I/O hooks in page zero point to DOS routines at $9E81 and $9EBD. If not, generates the following error message:

   ```
   DOS NOT AT NORMAL ADDRESS
   ```

2. Disconnects DOS.

3. Installs patches in DOS, page 3, and $9000-$912B ($9000 maps to $D000, the speed changes).

4. Changes addresses in DOS code segments to relocated addresses.

5. Changes addresses in DOS data segments to relocated addresses.

6. Moves DOS image into final position in bank-switched memory.

7. Coldstarts new DOS.

Patches are installed before DOS is relocated, so the addresses in the patches are for unrelocated DOS. The address relocation includes both the unmodified parts of DOS and the new patches.

DOS is relocated to reside in the bank-switched memory that lies in the address space $D000-$FFFF. Bank 2 is selected in the $D000-$DFFF address space; bank 1 is left unused. The main DOS routines normally residing at $9600-$BFFF are moved to $D600-$FFFF. The space from $D15A-$D5FF is left for two additional file buffers, allowing a maximum of five. The space from $D000-$D12B is occupied by code for the speed changes. $D12C-$D159 is left unused.

Most patching is accomplished in-line in DOS. Short patches simply replace the old code they modify. Patches that will not fit are put in one of two large "patch areas," and are jumped to from the code they modify. The first large patch area is in the space from the file manager INIT handler ($AE8E-$AF07). The second large patch area is in the space from the routine to write DOS to tracks zero through two ($B700-$B78C).

Additional space is used in page 3 for the banking patches. The normally free space from $396-$3CF is occupied by patch code, and the usual DOS entry code at $3D0-$3EE is heavily modified. These patches in page 3 are vital and must not be overwritten. Only three of the normal page 3 entry calls are preserved—coldstart ($3D0), warmstart ($3D3), and reconnect ($3EA).

The file manager and RWTS calling points in page 3 are disabled with BRK instructions. The file manager and RWTS still exist in the relocated DOS, at the corresponding addresses (add $D600-$9600=$4000 to get the relocated addresses). Modifying the parameter lists is no longer simply a matter of storing the desired values. Instead, modification requires switching in the relocated DOS, modifying the lists, and switching the ROM back in. Similarly, calls to the relocated DOS must switch in DOS, call, then restore the ROM. The BRK instructions are substituted to force programs that do not take the relocation into account to break.

Speed Modifications

The speed increase in the LOAD, SAVE, and RUN commands results from improvements to the file manager. The commands use the file manager "read/write a range of bytes" routines, so these routines are modified. The single-byte read/write routines are not changed.

The unmodified file manager performs the I/O *n* byte commands by doing *n* single-byte I/O operations. This introduces a substantial amount of overhead. In the relocated DOS, the I/O *n* byte requests are vectored to the special routine at $D000-$D12B. Complete sectors are read or written directly between memory and disk without intermediate buffering. Partial beginning or ending sectors are handled with normal DOS buffering. This speeds up the I/O *n* byte routines considerably.

Normal DOS 3.3 verifies after writing to insure that the data was written correctly. This verification slows down writing to disk somewhat, but also makes it more reliable. The modified DOS leaves this verification intact. Note that the normal routine in the file manager to write *n* bytes, actually writes *n+1* bytes. The modified DOS also writes *n+1* bytes to insure compatibility.

Chapter 14

128K Microsoft Applesoft Compiler

128K MS-Applesoft Compiler provides several new features that take advantage of the additional memory available on the 128K Apple IIe. First, the 128K compiler uses the extra memory to compile faster and provide more symbol table space. The larger symbol table allows compilation of even longer programs. Second, the 128K compiler produces special object code that allows variables and strings to be stored in the extra memory.

The 128K version disk is intended for use on an Apple IIe with 128K RAM. The Apple IIe comes standard with 64K RAM. In order to have 128K, the basic Apple IIe must be equipped with an *extended* 80-column card containing an additional 64K bank of memory. Note that a plain 80-column card does *not* provide an additional 64K bank of memory; only the *extended* card includes the auxiliary 64K bank.

128K MS-Applesoft Compiler generates special object code to allow compiled programs to access the auxiliary memory. The object code produced allows variables and string space to reside in the auxiliary bank of memory. Moving variables to the extra bank frees up main memory for use by the object code. In addition, DOSMOVER can free even more main memory by relocating DOS.

To appreciate the increase in program size that the 128K version of the compiler allows, consider the space available for object code on 48K, 64K, and 128K systems. With 48K of memory, object code, variables, and DOS must all fit into the 48K of available main memory. With 64K of memory, DOSMOVER can remove DOS into the top 16K so that only object code and variables must fit into the main 48K of memory.

With 128K of memory, DOSMOVER and 128K MS-Applesoft Compiler can be used together to free the entire main 48K for use by object code. The 128K Apple IIe extended memory version frees part of main memory by allowing variables and string space to reside in the auxiliary memory. DOSMOVER frees the rest of main memory by relocating DOS into the top 16K. 128K MS-Applesoft Compiler and DOSMOVER together allow the main 48K to be used entirely for object code, making it possible to compile extremely long programs.

128K MS-Applesoft Compiler is even more useful because the Applesoft interpreter does not use the auxiliary bank of memory. Interpreted programs can use only the main bank of memory. Programs compiled with the 128K version of the compiler not only execute more quickly, but also allow the use of the otherwise unavailable auxiliary memory.

The 128K MS-Applesoft Compiler disk contains the following files:

1. APCOM—128K version of the Microsoft Applesoft Compiler.

2. PASS0, PASS1, PASS2—Internal subprograms of 128K Applesoft Compiler.

3. RUNTIME—Runtime library, *different from 48K RUNTIME.*

4. DOSMOVER—DOS relocation program.

5. RENUMBER.UPDATE—Updated version of Apple's Applesoft RENUMBER program.

6. BALL—Demonstration program.

These files and their use are described in Part 3.

Chapter 15
Default Compilation With 128K MS-Applesoft Compiler

128K MS-Applesoft Compiler is quite easy to use. Defaults allow basic compilation without requiring knowledge of the memory configuration of the Apple IIe. Just as with the 48K version, only the source filename must be provided to compile in the default mode. In addition, of course, there are options to specify different memory configuration and compilation modes. This chapter provides the information necessary to compile in default mode with 128K MS-Applesoft Compiler.

Using the 128K Applė IIe extended memory version of the Applesoft Compiler is basically the same as using the 48K version. Remember, though, that the 128K version requires relocated DOS. If DOS is not already relocated, relocate DOS with DOSMOVER before running the 128K version. See "Using DOSMOVER" in Chapter 11 for details on using the DOSMOVER program. Once DOS is relocated, everything is set for compilation. The 128K compiler is invoked with RUN APCOM. After the compiler finishes loading, a header appears on the screen and the compiler requests the source filename.

Two possible mistakes may cause error messages before the compiler requests the source filename. "MISMATCH" indicates that MS-Applesoft Compiler is not being run on a 128K Apple IIe. 128K MS-Applesoft Compiler can be used *only* on the 128K Apple IIe. Second, "DOS NOT RELOCATED—CAN'T COMPILE" indicates that DOS has not been relocated with DOSMOVER.

From this point on, using the 128K compiler with defaults is the same as using the 48K compiler with defaults. To appreciate the similarity to 48K Applesoft Compiler, try actually compiling BALL, the demonstration program, with the 128K version. When the compiler prompts for the source filename, follow the instructions in Chapter 1, "Demonstration Run," for instructions on running BALL.

Follow the instructions that begin after the direction to type "RUN APCOM". Note that the compilation information printed at the end of PASS2 now includes information for both the main and auxiliary banks.

Try running the compiled version of BALL. Remember that some of BALL's variables reside in the auxiliary memory. BLOAD the runtime library from the 128K disk, and BRUN the new BALL.OBJ program. There is little visible indication that the object code accesses the auxiliary bank. This is the power of the Microsoft Applesoft Compiler; Applesoft programs the extra memory without the problems of getting over to the auxiliary bank.

Despite the added power of the 128K Apple IIe extended memory version, compiling with the 128K compiler is much the same as compiling with the 48K compiler. On the outside, the 128K compiler looks a lot like the 48K compiler. The object files produced are quite different, though, since the object code from the 128K compiler handles the auxiliary bank of memory. This chapter has shown how simple it is to use 128K MS-Applesoft Compiler in default mode. The following chapters give additional information about the 128K version.

Chapter 16
Executing 128K Programs

Executing programs compiled with 128K Microsoft Applesoft Compiler is similar to executing programs compiled with the 48K compiler, but there are a few differences. This section discusses those differences. Refer to Chapter 5, "Executing a Compiled Program," for general information about executing compiled programs.

The object code produced by the 128K compiler is substantially different from the object code produced by the 48K compiler. The runtime libraries used by 128K and 48K object code are also different. The differences in the 128K object code and runtime library enable 128K object code to access variables and strings in the auxiliary bank.

128K object code can only be executed with the 128K RUNTIME library file, and 48K object code can only be executed with the 48K RUNTIME file. Mismatching object code and library generates an error message as soon as the object code begins executing. Running object code compiled with one version using the runtime library from the other version generates the "?MISMATCH" error. Load the correct RUNTIME file, and reload the object code. The runtime libraries for the two versions are different lengths, so part of the object code may be destroyed in a mismatch.

128K object code *requires* the presence of the auxiliary bank. Even if memory allocation has been reassigned so that all variables reside in the main bank, string space still resides in the auxiliary bank. In addition, the RUNTIME file copies itself into the auxiliary bank at the beginning of execution. Attempting to run object code compiled by the 128K compiler on a machine that is not a 128K Apple IIe also generates the "?MISMATCH" error. The object code produced by the 128K compiler can only be run on a 128K machine. If the object code must be run on some machine other than a 128K Apple IIe, compile with 48K or 64K MS-Applesoft Compiler.

The default memory allocation of the 128K compiler assumes that the whole main bank is free for use by the compiled code. This assumes that DOS has been relocated. The 128K compiler in default mode may produce object code that extends into DOS space. If the compiled code occupies part of the space occupied by DOS and DOS has not been relocated, then either the compiled program or DOS is destroyed.

There are several ways to prevent this. The most simple way is to use relocated DOS. Relocated DOS can be used whenever disk utility programs do not require the normal DOS. When DOS has been relocated, there is no chance that compiled programs will conflict with DOS. Another approach is to check the compilation information provided at the end of PASS2 each time a program is compiled. If the program or its variables extend into the area normally occupied by DOS (locations 38400-49151, $9600-$BFFF hex), then specially name the program as suggested in Chapter 13, "Executing Compiled Programs With Relocated DOS."

Still another option is to explicitly reserve the space occupied by DOS. The memory allocation options in the 128K Apple IIe extended memory version can be used to set the highest memory location in the main bank used by the compiled code. Setting the highest available location below DOS insures that the compiled code will not overwrite DOS. Chapter 20, "Specifying Different Memory Configurations," explains how to limit the space used by compiled code.

Chapter 17
Banked Memory

128K Microsoft Applesoft Compiler is designed specifically for the Apple IIe. MS-Applesoft Compiler is designed so that compilation requires a minimum of knowledge about the memory layout of the Apple computer. Still, specifying nondefault memory allocation requires some understanding of the memory layout of the IIe. This chapter introduces banked memory, one of the concepts important in understanding the Apple IIe. This chapter combined with the next explain how the memory in the Apple IIe is organized.

The Apple IIe

The Apple IIe comes standard with 64K of RAM. The Apple IIe can be extended by adding 80-column capability and extra memory. There are two types of 80-column cards available: those that offer only 80-column capability, and those that offer 80-column capability plus an additional 64K of RAM. The extra 64K of RAM available on these cards extends the total amount of RAM to 128K. 128K of memory is *required* to use 128K MS-Applesoft Compiler.

The memory layout in the Apple IIe is not fully described by saying that the Apple IIe has "128K" of memory. There are several important aspects of computer memory. The aspect most commonly discussed is simply the *amount* of memory available. Most computers are identified as having "16K," "48K," "256K," etc., of memory.

The *amount* of memory is only part of the story. The *configuration* of the memory is also very important. The *amount* of RAM available in a 128K Apple IIe is 128K, but the RAM is configured as "bank-switched" memory. "Bank-switched" memory must be accessed differently than "plain" memory. Bank-switched memory and its use on the Apple IIe are described in the following sections.

Bank-Switched Memory

The basic idea of "memory" is easy to understand. Memory is a way to save things: things can be stored in memory and recalled from memory. Most memories can store more than one item, so memories are usually divided into a number of "one-item memory locations." Each memory location stores exactly one item, and "accessing" (storing to or recalling from) the location changes or recalls the single item stored there.

To make storing in or recalling from a particular location easier, memory locations are usually assigned names or numbers to help distinguish between them. As an example, suppose four items are to be stored in memory. Since each item requires a separate location, there must be four locations. To make storing and recalling easier, each location needs a name. Suppose the names are the four letters 'A', 'B', 'C', and 'D'.

In this four-location memory, each location has a different name. 'A' corresponds to the first location, 'B' to the second, 'C' to the third, and 'D' to the fourth. The names 'A', 'B', 'C', and 'D' serve as the "addresses" of the locations. The item stored in the first location can be accessed by specifying address 'A', the item stored in the second location can be accessed by specifying address 'B', and so on. Each address specifies a location, and each location contains one item.

If each location is to have a unique address, there must be at least as many possible addresses as there are locations. This is not always the case. Suppose that addresses must be single capital letters: 'A', 'B', 'C', . . . 'Z'. Since there are 26 letters in the alphabet, this provides 26 possible addresses. As long as there are 26 locations or less, each location can have a different address.

Suppose there are 52 locations and still only 26 addresses. Since there are more than 26 locations, there are not enough addresses for each location to have a different address. Suppose each address is paired with two locations. Imagine the first 26 locations have addresses ‘A’ through ‘Z’, and the next 26 locations also have addresses ‘A’ through ‘Z’. Now each address is shared by two locations.

With this identification scheme, each location has an address, but each location no longer has a *unique* address. The address ‘A’ is paired with both the first location and also the twenty-seventh location. The address ‘B’ is paired with both the second and twenty-eighth locations, and so on. Specifying an address alone no longer singles out a certain location.

This addressing scheme is difficult to use. With only 26 addresses and 52 locations, it is no longer possible to specify an address and be sure which location is being accessed. There clearly needs to be some additional distinction between the first 26 locations and the second 26 locations.

Suppose that the first 26 locations are considered to be “bank one” and the next 26 locations are considered to be “bank two.” Bank one therefore includes locations 1 through 26, and bank two includes locations 27 through 52.

Dividing the memory into two banks provides a way to distinguish between the two different locations that are paired with the address ‘A’. Locations 1 and 27 lie in different banks. Saying, “address ‘A’ in bank one” specifies location 1, and saying, “address ‘A’ in bank two” specifies location 27. Locations 1 and 27 still share the same letter address ‘A’, but the two locations can now be distinguished by the bank they are in.

The 52-location memory is now divided into two banks. Bank one includes locations 1 through 26, and bank two includes locations 27 through 52. Within each bank, locations are distinguished by their address; ‘A’, ‘B’, ‘C’, etc. Each of the 52 locations can be accessed individually by first specifying the bank the location is in, then the address of the location:

	A	B	C	D . . .	X	Y	Z
bank 1:	loc1	loc2	loc3	loc4 . . .	loc24	loc25	loc26
bank 2:	loc27	loc28	loc29	loc30 . . .	.loc50	loc51	loc52

Separating memory into banks allows a limited number of addresses to be used with a larger number of locations. Each address represents several locations, but the locations are in different banks. This type of memory configuration is often called "banked," or "bank-switched," since the memory is divided into several different "banks." Chapter 18, "Apple IIe Memory," discusses how memory banking is used on the Apple IIe.

Chapter 18
Apple IIe Memory

Understanding bank-switched memory is important in understanding how the Apple IIe can provide 128K of memory. The Apple IIe uses bank-switched memory to get around some of its built-in limitations.

Computers generally access their memory by using numbers as the addresses of memory locations, and the Apple is no exception. For instance, a 48K Apple has 49152 (48 times 1024) locations of RAM. These 49152 locations are assigned addresses 0 through 49151, and each location has a unique address. Since there are as many addresses as memory locations, there is no need to use a banked memory configuration on a 48K Apple.

The heart of the Apple is its 6502 microprocessor, a complex chip on the main circuit board. The 6502 performs most of the operations needed to run the computer, and one of its common jobs is accessing the memory. The 6502 limits memory addresses to the range 0 through 65535. This means that the 6502 can address 65536 locations (64K=65536 divided by 1024). This figure is normally expressed by saying that the 6502 has 64K of "address space."

Since the 6502 in the Apple can address only 64K of memory, there is clearly a problem in trying to address the 128K of memory in the 128K Apple IIe. This problem is solved by dividing the 128K RAM into two 64K banks. The 128K is divided in the same way that the 52-location memory in the last chapter was divided into two 26-location banks. The Apple IIe contains 128K of memory, but the memory must be "banked."

A 128K Apple IIe uses two 64K banks. A 64K Apple IIe can get by with the 64K of address space provided by the 6502, so a 64K Apple IIe just uses one bank of 64K. Adding an 80-column card with the additional 64K of memory simply adds a second 64K bank to the first 64K bank. Together, the two banks provide 128K of memory.

The extra bank of memory in the Apple IIe increases the total amount of memory in the computer, but, in practice, the extra bank is difficult to access. Only one bank can be accessed at a time. Accessing the extra bank requires first switching out the main bank. However, the program accessing the extra bank normally lies in the main bank. So, accessing the extra bank requires switching out the main bank, but switching out the main bank also switches out the program. With the program no longer available, the computer hangs.

There are several ways to get around this problem, but none of them is directly available to the Applesoft user. Without MS-Applesoft Compiler, only specially written applications programs can take advantage of the extra bank of memory. With Applesoft Compiler, however, the problem of accessing the extra bank is taken care of by the compiler. 128K MS-Applesoft Compiler generates special code that takes care of all the bank-switching overhead, allowing the normally unavailable extra bank to be used by compiled Applesoft programs.

The 64K Apple IIe Memory Configuration

The exact memory configuration of the 128K Apple IIe is easier to explain after first examining the 64K Apple IIe. The memory in a 64K Apple IIe includes more than just the 64K of user RAM. In addition to the 64K of RAM, there is also 12K of dedicated ROM and 4K of I/O space. The ROM contains the Applesoft language and the system monitor, and the I/O space includes special locations such as the keyboard strobe and game controller locations.

To simplify the discussion of memory, the ROM and I/O space will simply be grouped together as 16K of "ROM." An additional 2K of space at the bottom of RAM is reserved for the system and screen memory. The overall picture for the 64K Apple IIe is illustrated in Table 3.

Table 3

Memory Configuration for 64K Apple IIe

		RAM	HARDWARE
$C000-$FFFF	(16K)	B/S RAM	"ROM"
$800-$BFFF	(46K)	RAM	none
$0-$7FF	(2K)	SYS RAM	none

(B/S: Bank-switched SYS: system)

The "ROM" is required for the internal operation of the Apple and is not directly available to the user. Similarly, although the bottom 2K of memory is RAM, it is used by the system for Applesoft, the monitor, and the text screen. Only a few parts of this memory are free for use, mainly the lower part of page 3 (decimal 768-1023, hex $300-$3FF). Most of the bottom 2K is off limits to the user.

In the diagram above, the top 16K of space is occupied by both RAM and the "ROM." The RAM and "ROM" in this space are bank-switched; either the RAM or the "ROM" can be accessed at any given time, but not both. Usually the "ROM" is switched in so that Applesoft and the system monitor are available. The 16K of RAM in this area is awkward to use, since accessing the RAM requires switching the "ROM" in and out.

The 16K of RAM in this area corresponds to the RAM added to a 48K Apple II by using a RAMCard. RAMCard memory is not directly accessible to Applesoft, and so it is usually used for special purposes. Applications or machine language programs sometimes use the RAMCard, and languages that are not in ROM are also often loaded into the RAMCard memory. The top 16K of RAM in a 64K Apple IIe acts just like a RAMCard in a 48K Apple II or II Plus; it is switched in and out in the same fashion, and is available for the same uses.

While the top 16K of address space can contain either the ROM or the RAMCard RAM, the address space for the first 48K is occupied only by RAM. This 48K of RAM is directly available since there is no bank-switching necessary to access it. This is the memory normally used for Applesoft programs and variables.

In summary, the memory configuration of a 64K Apple IIe is the same as the memory configuration of a 48K Apple II with a RAMCard. The first 2K of the main 48K of RAM is reserved for system functions, and the next 46K is directly available to the user. The last 16K of address space is occupied by the "ROM", but 16K of bank-switched RAM can also be switched in to occupy this space.

To avoid confusion, this 16K of bank-switched RAM will be called "RAMCard" memory. This name is logical, since the 16K of "RAMCard" memory in a 64K Apple IIe *acts* like the 16K of RAM that a RAMCard adds to a 48K Apple II. Remember, though, that the "RAMCard" memory in the Apple IIe is really just a part of the built-in 64K.

The 128K Apple IIe Memory Configuration

The 128K Apple IIe memory configuration is fairly easy to understand once you understand the configuration of the 64K Apple IIe. The 128K IIe has an additional 64K of RAM on the extended 80-column card, but this additional 64K of RAM is set up as an exact image of the first 64K. In a 128K Apple IIe, there are just two copies of the 64K of RAM, as shown in Table 4.

Table 4

Memory Configuration for 128K Apple IIe

		COPY 1	COPY 2	HARDWARE
$C000-$FFFF	(16K)	B/S RAM	B/S RAM	"ROM"
$800-$BFFF	(46K)	RAM	RAM	none
$0-$7FF	(2K)	SYS RAM	SYS RAM	none

(B/S: Bank-switched SYS: system)

Notice how the two copies of RAM are set up identically. They both have the first 2K of the main 48K as system RAM, and the next 46K as free RAM. Both have 16K of "RAMCard" bank-switched memory in the same address space as the "ROM." Thus, each 64K copy of RAM has 48K of "main" memory, and

16K of "RAMCard" memory. Since the two 64K copies cannot both be used at the same time, the two 64K copies are actually "bank-switched." The first copy (main bank) and the second copy (auxiliary bank) cannot be accessed at the same time; accessing one requires switching the other one out.

The reason for specially naming the 16K of bank-switched memory at the top of each bank as "RAMCard" memory should now be clear. The main and auxiliary 64K banks are themselves bank-switched, but each also contains 16K of "RAMCard" memory that is bank-switched in another sense. The special name "RAMCard" is used to avoid confusion between the two 64K banks of memory and the 16K of separately bank-switched memory within each bank.

There are five areas of memory that are important for compilation. To avoid phrases like "the 'RAMCard' memory in the auxiliary bank," or "the 46K of user memory in the main bank," each important area will be referred to by a fixed name. The names are shown in Table 5.

Table 5

Memory Area Names for 128K Apple IIe

		COPY 1	COPY 2	HARDWARE
$C000-$FFFF	(16K)	MAIN RAMCard	AUX RAMCard	"ROM"
$800-$BFFF	(46K)	MAIN BANK	AUX BANK	none
$0-$7FF	(2K)	unnamed	unnamed	none

Now, for instance, "'RAMCard' memory in the auxiliary bank" becomes the "auxiliary RAMCard," and "the 46K of user memory in the main bank" is just the "main bank." If at some point these shortened terms do not provide enough information, refer back to this explanation to recall what they represent.

Chapter 19
Default Memory Allocation

The default memory allocation provided by the 128K version is intended to be suitable for most uses. Still, there are situations where a particular area of memory needs to be protected from use by the compiled code. Memory might be reserved to allow use of the HIRES screens or machine language programs. Like the 48K version, the 128K version provides several features that allow special memory configurations.

Memory usage on the Apple IIe is much easier to describe with the terms explained in Chapter 18, "Apple IIe Memory." DOS-MOVER relocates DOS out of the "main bank" and into the "main RAMCard." Compiled programs use the "main bank" and the "auxiliary bank" of memory. The "auxiliary RAM-Card" is the only section of memory left unused. Machine language programs or data might be located in the auxiliary RAMCard, but the auxiliary RAMCard is difficult to access. It is much easier to simply set aside space in the main bank. Chapter 20, "Specifying Alternate Memory Configurations," discusses how to reserve main memory space.

The default memory allocation provided by the 128K compiler uses both the main and auxiliary banks. Object code and local (not in COMMON) numeric variables are assigned to the main bank. String space and local string variables are assigned to the auxiliary bank. The COMMON block defaults to the auxiliary bank. The runtime library must be available regardless of which bank is switched in, so the RUNTIME file is kept in both banks.

The runtime library in the 128K version is assigned to a fixed address. The runtime library resides at the bottom of memory, at 2051 (hex $803). The copies of RUNTIME in the main and auxiliary bank must be at the same address. RUNTIME is assigned to the bottom of memory in the auxiliary bank to prevent conflict with any variables in the auxiliary bank. This forces the runtime library in the main bank to also reside at the beginning of the bank. The runtime library will not function properly if used at a different address. The runtime library copies itself into the auxiliary bank at the beginning of execution.

Chapter 20
Specifying Different Memory Configurations

The memory allocation features in the 128K Apple IIe version allow a great deal of flexibility. Only a few general requirements must be satisfied. This chapter discusses those requirements and how to specify alternate memory configurations.

The general requirements are fairly easy to meet. First, the runtime library must reside at the bottom of both the main and auxiliary banks. The runtime library will not function from any other location. Object code must reside in the main bank. String space and local string variables must reside in the auxiliary bank. The auxiliary bank must be totally free for use by the compiled code. Space in the auxiliary bank cannot be explicitly reserved.

This is the full extent of the general requirements. The flexibility remaining is more than sufficient in most cases. A program can be compiled to start at any address in the main bank. This allows the program to start after either of the HIRES screens in case HIRES graphics are used.

If the program uses a COMMON block, the block can reside in either the main or auxiliary bank, and can start at any address. This allows the COMMON block to be moved to the auxiliary bank if it is too large to fit in the main bank with the object code. Note that the COMMON block includes both COMMON numeric and string variables. "Local" (non-COMMON) string variables must reside in the auxiliary bank, but there is no such restriction on the COMMON string variables in the COMMON block.

The top of usable memory in the main bank can be set to reserve space—for machine language programs, for example. The top of main memory available to the compiled code defaults to the beginning of ROM. This default allows the compiled program to use all of the main bank. The default does not reserve any space for DOS or other uses. Space can be reserved by explicitly setting the top of usable memory below the needed space.

Local numeric variables are allocated automatically by the compiler. This allocation is controlled by the address specified as the top of usable memory in the main bank. The compiler allocates local numeric variables from the beginning of local numeric variable space (usually the end of the object code) up to the specified end of usable memory. If still more space is needed, the compiler allocates the remaining local numeric variables in the auxiliary bank.

This ability to allocate local numeric variables in both banks makes the most use of the available memory. If the object code "almost" fills the main bank, the leftover space is not wasted. The compiler automatically puts as many of the local numerics as possible in the main bank, and assigns the rest to the auxiliary bank.

The flexibility of 128K MS-Applesoft Compiler becomes important for long programs. Long programs typically produce even longer object code, so object code space is usually the biggest limitation. The requirements for memory use are set up so that the object code can occupy the entire main bank if need be.

Local string variables and string space are always stored in the auxiliary bank. Local numeric variables are automatically assigned to the auxiliary bank if the main bank is full. Finally, even the COMMON block can be pushed off into the auxiliary bank. Only the runtime library and the object code must fit into the main bank. The runtime library is only about 4K long. Since the main bank minus the 2K of system memory is 46K long, this allows 42K of object code space.

Long programs often require a great deal of variable space. Large arrays, big common blocks, or numerous strings all require substantial variable and string space. The 128K compiler is able to provide the needed space by allowing variables and string space to reside in the auxiliary bank. Again, there is 42K (46K minus 4K) of space for variables and strings. Even if the main bank is totally filled by 42K of object code, there is still an additional 42K available for variables and strings. If the main bank is not full, the available variable space is even larger.

Needless to say, it is fairly difficult to write intelligible Applesoft programs that exhaust all the memory made available by MS-Applesoft Compiler. Considering the additional chain with COMMON features, there are very few programs that cannot be compiled with the 128K Apple IIe extended memory version.

The memory allocation options in the 128K version are powerful and convenient to use. To specify a memory configuration different than the default, answer *N* or *NO* when the compiler prompts:

```
MEMORY USAGE:
NORMAL CONFIGURATION:
(DEFAULT YES)?
```

When the default is not used, the next few prompts request information about the desired configuration. The first two prompts request the bank and address for the COMMON block. These prompts must be answered even if the program about to be compiled does not use COMMON variables. If the program does not have a COMMON block, the responses will not be used, so the default COMMON bank and address are sufficient.

The bank for the COMMON block defaults to the auxiliary bank, but either bank may be used. If the COMMON block is in the auxiliary bank, its location defaults to the end of the library. This location is also the default if the COMMON block is in the main bank, but alternate addresses may be specified if the COMMON block is in the main bank. Possible responses to the COMMON block address prompt are pressing <RETURN> to accept the default, entering a numeric address, or typing

```
HGR1
```

or

```
HGR2
```

"HGR1" and "HGR2" set the beginning of the COMMON block immediately above the corresponding HIRES screen.

The next prompt requests the starting address for the object code. The object code always resides in the main bank, so only the address for the object code must be specified. The possible responses are the same as they were for the COMMON block address; press <RETURN>, enter a number, or type

```
HGR1
```

or

```
HGR2
```

The compiler next requests the address for the beginning of variables in the main bank. Remember that the variables concerned are local numerics *only*. Local string variables are always assigned to the auxiliary bank, and the location of COMMON variables is determined by the COMMON block address. Local numerics are allocated from the specified address up to the top of usable main memory. Any variables that do not fit are assigned to the auxiliary bank.

The last prompt requests the bottom of reserved main memory. The address specified should be the first location that is *not* available to the compiled code. The default is the beginning of ROM (decimal 49152, hex $C000). The default assumes that no space needs to be reserved, and that DOS will be relocated at runtime.

After the prompts have all been answered, the compiler lists the specified addresses and requests confirmation. Examine the addresses carefully, since this is the last chance to discover an error until the final addresses are listed at the end of PASS2. Note that the "END OF MAIN MEMORY" is determined from the response to the "BOTTOM OF RESERVED MAIN MEMORY" prompt. The end of usable main memory is considered to be one less than the bottom of reserved main memory.

Several error messages may occur during compilation if the specified memory configuration cannot be satisfied. "TOO LONG" is generated if an array larger than 48K is declared in COMMON. "OBJECT CODE TOO LONG" is generated if the object code extends past the declared top of usable main memory. "OUT OF VARIABLE SPACE" is generated when all the available variable space in both banks has been exhausted. The only one of these errors that is likely to occur is the "OBJECT CODE TOO LONG" error. This error may occur if the address for the beginning of the object code is set abnormally high. If the object code address is left at its default, the maximum of 42K for object code space is available.

When compilation is complete, PASS2 prints the resulting memory addresses for object code, variables, and string space. Check to make sure that the addresses are as intended. If DOS will not be relocated when the object code is run, and the space occupied by DOS was not explicitly reserved, check to make sure that the compiled code does not extend into the area occupied by DOS.

If a nondefault address for the beginning of variables in the main bank was specified, check to insure that the address specified does not force an overlap between the variables and object code. Finally, remember that if the program uses HIRES graphics, the object code addresses must have been set so that the memory mapped to the appropriate screen is free. If no conflicts are apparent in any of the memory information, then the memory usage by the compiled program should not cause any problems. If problems are apparent, recompile using the correct addresses.

Chapter 21

Additional Differences From 48K Microsoft Applesoft Compiler

The basic information needed to use the 128K version is discussed in the previous chapters. The previous chapters outline the basic differences between the standard version and the 128K version, but assume a working knowledge of the 48K and 64K versions. Part 1, the 48K section, covers basic compilation issues, and Part 2, the 64K section, discusses the use of DOS-MOVER and RENUMBER.UPDATE.

This chapter lists additional minor differences between 128K and 48K MS-Applesoft Compiler. Each of the following sections discusses the changes that affect a particular chapter in Part 1, the 48K section.

Compilation (Chapter 4)

This section discusses additional changes affecting Chapter 4, "Compilation."

Memory Usage

Programs compiled with 128K MS-Applesoft Compiler no longer use the HIMEM pointer. Strings are kept in the auxiliary bank, so string space always starts at the top of the auxiliary bank rather than at HIMEM. The HIMEM statement is ignored by the 128K compiler.

Compiling Large Programs

Long compiled programs are less likely to present problems with the 128K Apple IIe extended memory version. The 128K compiler has about twice as much symbol table space as the 48K compiler, so "SYMBOL TABLE FULL" errors are rare. Programs are less likely to need separation into chained parts, since there is substantially more space available at runtime. If a long program does present problems, refer to "Compiling Large Programs'" in Chapter 4. Also consider trying a different memory allocation scheme, as discussed in Chapter 20, "Specifying Different Memory Configurations."

Compiler/Interpreter Language Comparison (Chapter 6)

This section discusses additional changes affecting Chapter 6, "Compiler/Interpreter Language Comparison."

Statements Not Implemented

As discussed in the previous section, HIMEM is no longer needed when string space is in the auxiliary bank. HIMEM statements are simply ignored.

Using MAXFILES From Within a Compiled Program

MAXFILES no longer requires an additional HIMEM statement to function correctly from a compiled program. MAXFILES still resets certain pointers, but these pointers are no longer used by the compiled code. However, since MAXFILES changes the amount of space occupied by DOS, changing MAXFILES may cause memory conflicts between DOS and the compiled code.

MAXFILES determines the number of file buffers allocated by DOS. With DOS relocated, the space used by the file buffers is in the main RAMCard. Since the compiled code does not use the main RAMCard, there is no possibility of conflict when DOS is relocated.

When DOS is not relocated, both DOS and its file buffers occupy the upper part of the main bank. The normal DOS beginning of 38400 (hex $9600) assumes three file buffers are allocated. Additional buffers require another 595 (hex $253) bytes each. With four buffers (one additional), the bottom of DOS is at 37805 (hex $93AD). With five buffers (2 additional), the bottom of DOS is at 37210 (hex $915A), etc.

If DOS is not relocated at runtime, both DOS and the compiled code share the main bank. If DOS is not relocated, the appropriate amount of space for DOS must be protected from use by the compiled code. Since increasing MAXFILES increases the amount of space used by DOS, the space reserved must be adequate for the maximum value of MAXFILES used by the program. See Chapter 20, "Specifying Different Memory Configurations," for information on how to set the top of main memory used by compiled programs.

Applesoft Pointers Preserved by Compiled Code

The HIMEM pointer is no longer directly affected by compiled code. The HIMEM statement is ignored in the 128K version, so the HIMEM pointer is left unused. However, remember that the DOS MAXFILES command *does* change HIMEM.

String Operations

The 128K version assigns string space to the auxiliary bank. Any space in the auxiliary bank that is not used by variables is allocated as string space. With the 2K of system memory, and 4K for the runtime library, there is about 42K of memory available in the auxiliary bank. Unless variables occupy an unusually large amount of space, the remaining string space should be quite large. A large string space reduces the frequency of garbage collection. Garbage collection should occur far less frequently with 128K object code than with 48K object code.

Reducing DOS MAXFILES no longer provides more string space, since DOS does not use space in the auxiliary bank. The amount of string space available in the auxiliary bank can only be increased by decreasing the amount of auxiliary bank space occupied by variables. Variable space can be reduced by moving the COMMON block to the main bank, cutting down the size of arrays, or reducing the number of variables. Increasing the available string space does not affect the time required for an *individual* garbage collection, it only reduces the frequency of garbage collection.

PEEK and POKE

The PEEK and POKE statements access only main bank memory. The auxiliary bank is not directly accessible to Applesoft programs. Compiled programs can use the auxiliary bank for variables and string space, but the PEEK and POKE statements do not affect the auxiliary bank.

Language Enhancements (Chapter 7)

This section discusses additional changes affecting Chapter 7, “Language Enhancements.”

How COMMON Variables Work

The COMMON block is no longer allocated at the beginning of the compiled program. In the 48K version, space for the COMMON block is set aside starting at the address specified for the beginning of object code. The beginning of the object code is pushed up in memory to provide the needed space. In the 128K version, the COMMON block location is independent of program location, though the default is still at the beginning of the program.

Remember that programs that pass variables in COMMON must use the same COMMON block address and configuration. The COMMON blocks must be in the same bank and at the same address. Mismatching COMMON blocks generates the “?TYPE MISMATCH” error at runtime, as with the 48K version.

Error Messages and Debugging (Chapter 9)

This section discusses additional changes affecting Chapter 9, "Error Messages and Debugging."

Compiletime Error Messages

The "OBJECT CODE TOO LONG" error message in the 48K version indicates that the combination of object code and variable space is too long. In the 128K version, the object code can overflow the main bank while there is still variable space left in the auxiliary bank, and vice versa. A new error message helps to distinguish between the two overflow conditions. The new "OUT OF VARIABLE SPACE" error is generated if the auxiliary bank is filled with variables. The old "OBJECT CODE TOO LONG" is generated only if the object code itself overflows the main bank.

Runtime Error Messages

The 128K object code requires the presence of the extra 64K bank. The "?MISMATCH" error is generated at the beginning of execution if 128K object code is executed on a machine without the extra bank.

128K object code must be executed with the 128K runtime library. Mismatching the object code and library also generates the "?MISMATCH" error at the beginning of execution. As mentioned in Chapter 16, "Executing 128K Programs," be sure to reload both the runtime library and the object code to insure that both are undamaged.

Zero Page Usage (Appendix E, Part 1)

The zero page usage by 128K object code differs from the 48K zero page usage. The map below shows zero page use by 128K object code. Asterisks mark the differences from 48K usage.

Table 5

Zero Page Usage by Compiled Programs

Hex Address		Usage
OD-OE		Temporaries
10		Temporary
12-13	*	Beginning of local string variables—1
14-15	*	End of local string variables
16		Temporary
17-18	*	Bottom of string storage—1
19	*	Unused
50-51		Temporary
55-56		Temporary
58-5A	*	JSR to library routine to set main bank of memory
5B-5D		JMP to code currently handling ONERR
62-66		Temporaries
69-6A		Beginning of COMMON block—1
6B-6C		End of numeric common
6D-6E		End of string common
6F-70	*	Unused
71		Input buffer pointer
72	*	Memory bank for COMMON block
73-74	*	Applesoft HIMEM (not used)
75-76		Address of object code being executed
77-7A		Temporaries
7B-7C		Beginning of local numeric variables—1
7D-7E	*	End of local numeric variables in main bank
7F-80	*	End of local numeric variables in aux. bank
81-84		Temporaries
8A-8E		Temporaries
93-94		Temporary
9E-A3		Integer accumulators
AD-AE		Temporary
AF-B0	*	Address of first DATA string
B1-C8		Applesoft CHRGET, modified, but still functions normally
D9		Temporary
DC-DD		Temporary
F6-F7		Address of current DATA string

Index

MICROSOFT™
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name ____________________

Street ____________________

City ____________ State ________ Zip ________

Phone ____________ Date ________

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

______ Software Problem

______ Software Enhancement

______ Documentation Problem
(Document #__________)

______ Other

Software Description

Microsoft Product ____________________

Rev. ________ Registration # ____________

Operating System ____________________

Rev. ________ Supplier ____________

Other Software Used ____________________

Rev. ________ Supplier ____________

Hardware Description

Manufacturer ____________ CPU ________ Memory ______ KB

Disk Size ______″ Density: Sides:

Single ______ Single ______

Double ______ Double ______

Peripherals ____________________

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support ____________ Date Received __________

Routing Code ____________ Date Resolved __________

Report Number ___________

Action Taken: