NOTICE

Apple Computer Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

APPLE COMPUTER INC. AND SILICON VALLEY SOFTWARE INC. MAKE NO WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL OR WITH RESPECT TO THE SOFTWARE DESCRIBED IN THIS MANUAL, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. APPLE COMPUTER INC. AND SILICON VALLEY SOFTWARE INC. SOFTWARE IS SOLD OR LICENSED "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAMS PROVE DEFECTIVE FOLLOWING THEIR PURCHASE, THE BUYER (AND NOT APPLE COMPUTER INC., SILICON VALLEY SOFTWARE INC., ITS DISTRIBUTOR, OR ITS RETAILER) ASSUMES THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION AND ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL APPLE COMPUTER INC. OR SILICON VALLEY SOFTWARE INC. BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE, EVEN IF APPLE COMPUTER INC. OR SILICON VALLEY SOFTWARE, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

This manual is copyrighted. All rights are reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer Inc.

©1980 by APPLE COMPUTER INC.
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

©1980 by SILICON VALLEY SOFTWARE INC.
1531 Sandpiper Drive
Sunnyvale, California 94087

The word APPLE and the Apple logo are registered trademarks of APPLE COMPUTER INC.

APPLE Product #A2D0032
(030-0118-00)

Apple II

Apple FORTRAN

Language Reference Manual

ACKNOWLEDGEMENTS

The Apple® Pascal System incorporates UCSD Pascal™ and Apple extensions for graphics and other functions. UCSD Pascal was developed largely by the Institute for Information Science at the University of California at San Diego under the direction of Kenneth L. Bowles.

"UCSD Pascal" is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only and is an indication that the associated product or service has met quality assurance standards prescribed by the University. Any unauthorized use thereof is contrary to the laws of the State of California.

TABLE OF CONTENTS

PREFACE x

OVERVIEW 1

2 Introduction
3 Using This Manual
4 What Is Apple FORTRAN?
4 Apple vs. ANSI 77 Subset FORTRAN
6 ANSI 77 vs. Full Language
6 ANSI 77 vs. ANSI 66

FORTRAN READER'S GUIDE 7

8 Getting Oriented
9 Guide to Pascal Documentation
9 The COMMAND Level
9 The Filer
9 The Editor
9 6502 Assembler
10 The Linker
10 Utility Programs

PROGRAMS IN PIECES 11

12 Introduction
12 Partial Compilation
13 Source Code in Pieces
13 Object Code in Pieces
14 Units, Segments, and Libraries

THE COMPILER 17

18 Introduction
18 Files Needed
19 Using the Compiler
21 Form of Input Programs
  21 Lower and Upper Case
  22 Line Length and Positioning
23 Compiler Directives
24 Compiler Listing

THE LINKER 27

28 Introduction
28 Diskfiles Needed
29 Using the Linker

PROGRAM STRUCTURE 33

34 Introduction
34 Character Set
35 Lines
35 Columns
36 Blanks
36 Comment Lines
36 Statements, Labels, and Lines
37 Statement Ordering
38 The END Statement

DATA TYPES 39

40 Introduction
40 The Integer Type
41 The Real Type
41 The Logical Type
41 The Character Type

FORTRAN STATEMENTS 43

44 Introduction
44 FORTRAN Names
  44 Scope of FORTRAN Names
  45 Undeclared Names
45 Specification Statements
  46 IMPLICIT Statement
  47 DIMENSION Statement
  48 Type Statement
  49 COMMON Statement
  50 EXTERNAL Statement
  51 INTRINSIC Statement
  51 SAVE Statement
  51 EQUIVALENCE Statement
52 DATA Statements
53 Assignment Statements
  54 Computational Assignment Statement
  54 Label Assignment Statement

EXPRESSIONS 57

58 Introduction
58 Arithmetic Expressions
59 Integer Division
59 Type Conversions and Result Types
60 Character Expressions
60 Relational Expressions
61 Logical Expressions
62 Operator Precedence

CONTROL STATEMENTS 63

64 Introduction
64 Unconditional GOTO
64 Computed GOTO
65 Assigned GOTO
65 Arithmetic IF
66 Logical IF
66 Block IF...THEN...ELSE
68 Block IF
69 ELSEIF
69 ELSE
69 ENDIF
70 DO
71 CONTINUE
72 STOP
72 PAUSE
72 END

INPUT/OUTPUT OPERATIONS 73

74 I/O Overview
74 Records
75 Files
75 Formatted vs. Unformatted Files
75 Sequential vs. Direct Access
76 Internal Files
76 Units
77 Choosing a File Structure
79 I/O Limitations
79 I/O Statements
81 OPEN
83 CLOSE
83 READ
84 WRITE
85 BACKSPACE
85 ENDFILE
85 REWIND
85 Notes on I/O Operations

FORMATTED I/O 89

90 Introduction
90 Formatting I/O
91 Formatting and the I/O List
92 Nonrepeatable Edit Descriptors
92 Apostrophe Editing
93 H Hollerith Editing
93 X Positional Editing
93 / Slash Editing
93 $ Dollar Sign Editing
94 P Scale Factor Editing
94 BN/BZ Blank Interpretation
94 Repeatable Edit Descriptors
95 I Integer Editing
95 F Real Editing
95 E Real Editing
96 L Logical Editing
96 A Character Editing

PROGRAM UNITS 97

98 Introduction
98 Main Programs
98 Subroutines
  99 SUBROUTINE Statement
  99 CALL Statement
100 Functions
  100 External Functions
  101 Intrinsic Functions
  102 Table of Intrinsic Functions
  105 Statement Functions
  106 The RETURN Statement
106 Parameters

COMPILATION UNITS 109

110 Introduction
110 Units, Segments, Partial Compilation
111 Linking
112 $USES Compiler Directive
113 Separate Compilation
113 FORTRAN Overlays

BI-LINGUAL PROGRAMS 115

116 Introduction
116 Pascal in FORTRAN Main Programs
118 FORTRAN in Pascal Main Programs
119 I/O from Bilingual Programs
120 Calling Machine Code Routines

SPECIAL UNITS 123

124 The Turtle Graphics Unit
  124 The Apple Screen
  124 The INITTU Subroutine
  125 The GRAFMO Subroutine
  125 The TEXTMO Subroutine
  125 The VIEWPO Subroutine
  126 Subroutines for Using Color
  127 Cartesian Graphics
  127 Turtle Graphic Subroutines
  128 Turtle Graphic Functions
  129 Sending an Array to the Screen
  130 Text on the Graphic Screen
131 The Applestuff Unit
  132 RANDOM Function/RANDOI Subroutine
  132 Using the Game Controls
  134 Making Music: the NOTE Subroutine
  134 The KEYPRE Function

APPENDICES 134

135 Appendix A - Part One: Single-Drive Operation
153 Appendix A - Part Two: Multi-Drive Operation
171 Appendix B: FORTRAN Error Messages
179 Appendix C: Tables
187 Appendix D: FORTRAN Syntax Diagrams
211 Appendix E: FORTRAN Statement Summary
215 Appendix F: ANSI Standard 66 vs. 77 FORTRAN
219 Appendix G: Apple FORTRAN vs. ANSI 77

BIBLIOGRAPHY 223

INDEX 225

PREFACE

This manual describes the Apple FORTRAN programming language for the Apple II and Apple II-plus computers. Apple FORTRAN conforms to the American National Standard FORTRAN subset, also known as ANSI subset FORTRAN 77.

Apple FORTRAN contains features which are extensions to the ANSI standard subset. For instance, it incorporates a number of features of the full language not included in the standard subset. Apple FORTRAN also has features that are the result of the unique operating environment of the Apple. The ANSI standard subset itself includes most of the important revisions made to the full language over the previous standard, ANSI FORTRAN 66.

The purposes of this manual are:

* Acquaint you with Apple FORTRAN's differences and extensions to standard FORTRAN 77.

* Acquaint you with the Apple FORTRAN operating environment on the Apple II and Apple II-plus. FORTRAN uses the Apple Pascal Operating System.

* Introduce you to the principal differences between ANSI FORTRAN 77 and ANSI FORTRAN 66, if you are not familiar with this more recent version of FORTRAN.

* Provide you with the complete language specification of Apple FORTRAN.

The complete Apple FORTRAN documentation includes one other manual:

* Apple Language System Installation and Operation Manual

To familiarize you with the Pascal Operating System, this FORTRAN manual refers you to "Pascal documentation." These manuals are the Pascal manuals included with your Apple Language System.

SPECIAL NOTE: You must configure your FORTRAN System in order to run it. Your FORTRAN System was presented to you on two diskettes, FORT1: and FORT2:. FORT2: contains the SYSTEM.COMPILER and the SYSTEM.LIBRARY. FORT1: contains FORTLIB.CODE. The instructions for configuring your system are included in Appendix A, Part One for single-drive users, and Part Two for multi-drive users. The configuration suggested in Appendix A assumes that FORT2: will be your boot diskette, and this manual is written from that perspective. You may choose to configure your FORTRAN System differently. In that case, you must be aware of the references in the manual to the boot diskette as being FORT2:.

configuration suggested in Appendix A assumes that FORT2: will be your boot diskette, and this manual is written from that perspective. You may choose to configure your FORTRAN System differently. In that case, you must be aware of the references in the manual to the boot diskette as being FORT2:.

Also note that if you have not used the Apple Pascal Operating System before, Appendix A contains a tutorial on the operating system, as well as start up procedures for Apple FORTRAN. Part One is for single-drive users, and Part Two is for multi-drive users.

The published Standard on which Apple FORTRAN is based is:
ANSI X3.9-1978, American National Standard Programming Language FORTRAN which is available from: American National Standards Institute, Inc., 1430 Broadway, New York, New York 10018.

CHAPTER 1

OVERVIEW

2 Introduction
3 Using This Manual
4 What Is Apple FORTRAN?
  4 Apple vs. ANSI 77 Subset FORTRAN
  6 ANSI 77 vs. Full Language
  6 ANSI 77 vs. ANSI 66

INTRODUCTION

Apple FORTRAN is a programming language that runs on the Apple Pascal Operating System. This powerful operating system is general enough to be able to support languages other than just Pascal. By putting the FORTRAN language on the Apple Pascal Operating System, you get several distinct programming advantages. For example:

* The complete FORTRAN program development facility (including a text editor, file handler, code library and library handler, assembly language compiler, plus various and sundry utility programs) is identical to that supplied for Pascal.

* It is easy to make a turnkey FORTRAN system that immediately begins running a given program when the computer is turned on.

* Both the FORTRAN and Pascal languages operate on the same Apple.

* Only one operating system needs to be learned to run either FORTRAN or Pascal.

* Pascal subroutines can be linked to FORTRAN programs, and vice versa.

* Assembly language subroutines can be linked to either Pascal or FORTRAN programs or both.

There are other advantages. The Pascal Operating System supports a number of desirable features that have been added to Apple FORTRAN. These extensions make it easier to program, and easier to use FORTRAN programs interactively. Putting FORTRAN on the Pascal Operating System has caused two minor language restrictions in FORTRAN which are discussed below under Apple vs. ANSI 77 Subset FORTRAN.

The essential difference between the Apple Pascal and Apple FORTRAN packages is in the Compiler program. There are some other minor differences that will be discussed later in this manual. The output code generated by both FORTRAN and Pascal compilers is the same, and both Pascal and FORTRAN created code files can be handled interchangeably by the single operating system.

The program development system consists of the Editor, Linker, and Filer and some other utility and library programs. The sequence of program development is:

* Use the Editor to write FORTRAN programs.

* Use the Filer to fetch and store files on diskettes.

* Use the FORTRAN compiler to translate text files into code files.

* Use the Linker to combine code files into one executable code file.

* Execute the program.

Additional steps are required to link Pascal and FORTRAN programs together, or to link FORTRAN and Assembly language programs.

USING THIS MANUAL

The next chapter discusses using FORTRAN in the Pascal Operating System environment. It tells you where to look to find the most important features in the documentation for the Pascal Operating System.

The remainder of the first section of this manual, Chapters 3 through 5, discusses how to organize FORTRAN programs efficiently, the input requirements of the FORTRAN compiler, and the use of the Linker. The next major section of this manual, Chapters 6 through 14, contains the language specifications and description of Apple FORTRAN. It covers details of data types, expressions, statements, I/O considerations, and the format of programs acceptable to the FORTRAN compiler.

The remaining chapters, Chapters 15 and 16, discuss linking FORTRAN to Pascal programs, CALLing assembly language routines from FORTRAN, color graphics techniques, and other special Apple features.

The appendices summarize information given elsewhere in the manual and provide other useful tidbits. New Apple Pascal/FORTRAN users should be sure to read Appendix A before attempting to use the system.

All those parts of the operating system which are common to both Pascal and FORTRAN have been put into the documentation for the Pascal Operating System. In order to learn how to create, edit and run FORTRAN programs, you must have a copy of either the Apple Pascal Reference Manual or the Apple Pascal Operating System Reference Manual. Note that the latter manual is a replacement for the Pascal Operating System portion of the former manual.

Appendix A contains a tutorial on how to get started using the program development system, the Editor, Linker and so on. It provides you with all the information you need to run either a single- or a multi-drive system.

The Pascal Operating System documentation describes two things: the program development system and the operating system proper. The examples in the Pascal documentation are Pascal programs. This will not pose any problem in learning about the Editor, Filer and Linker because these tools are independent of the programming language used.

Other parts of the Pascal documentation are specifically directed to the Pascal language user. Some of those sections have been rewritten in this manual. In certain other cases, you will be instructed to read a particular section of the Pascal documentation with "FORTRAN colored glasses," substituting the name of one diskette for another and the word FORTRAN for Pascal, and the like.

WHAT IS APPLE FORTRAN?

FORTRAN has been around longer than almost any other high level programming language. As such, it has been through various stages of development. In 1966 the American National Standards Institute (ANSI) issued a Standard for FORTRAN that helped a great deal to clarify the language. This is sometimes referred to as ANSI FORTRAN 66. After that, development of the language continued, and enough of the additions were of sufficient interest and generality that in 1977 ANSI produced another Standard called ANSI FORTRAN 77 to incorporate these developments. This newer Standard is just coming into wide acceptance now. It is upon the official ANSI subset of the ANSI FORTRAN 77 full language that Apple FORTRAN is based. FORTRAN continues to grow and to find new environments, so that almost every implementation of FORTRAN has some features which are unique to the particular processor being used. Apple FORTRAN is no exception.

For this reason, it is important for users familiar with other versions of FORTRAN to get a clear view of how Apple FORTRAN compares with other varieties. There are three questions that need to be answered:

* How is Apple FORTRAN different from ANSI Standard subset FORTRAN 77?

* How is the ANSI subset different from the full language?

* How is ANSI 77 different from ANSI 66?

We will now treat each of these in turn.

Apple vs. ANSI 77 Subset FORTRAN

While Apple FORTRAN conforms largely to the ANSI Standard subset, there are some small differences. It does not support some features, takes some others from the full language specification, and in some cases goes beyond the Standard.

In two instances Apple FORTRAN does not conform to ANSI subset FORTRAN:

* INTEGER and REAL data types do not use the same amount of memory. ANSI says they must be the same. REAL data types are given 4 bytes of storage whereas INTEGER and LOGICAL data types are given 2 bytes. This means for INTEGER data, the range of numbers representable is from -32768 to +32767. The magnitude of nonzero REAL constants must fall within the range of approximately 5.8E-39 and approximately 1.7E+38.

* Subprogram names cannot be passed to other subprograms as formal parameters.

There are some capabilities which Apple FORTRAN allows that are in the full ANSI FORTRAN language specification, but not in the subset. These are:

* Subscript Expressions - Apple FORTRAN and the full ANSI 77 language allow function calls and array references in subscript expressions.

* DO Variable Expressions - The subset restricts expressions that define the limits of a DO statement, but the full language does not. Apple FORTRAN also allows full integer expressions in DO statement limit computations. Similarly, arbitrary integer expressions in implied DO loops associated with READ and WRITE statements are allowed.

* Unit I/O Number - Apple FORTRAN allows an I/O unit to be specified by an expression.

* Expressions in I/O list - Apple FORTRAN allows expressions in the I/O list of a WRITE statement, provided that they do not begin with a left parenthesis. Note that expressions such as: (A+B)*(C+D) can be specified in an output list as +(A+B)*(C+D) to circumvent this problem. Incidently, this does not generate any code at run time to evaluate the leading plus sign.

* Expressions in computed GOTO - Apple FORTRAN allows an expression for the value of a computed GOTO.

* Generalized I/O - Apple FORTRAN allows both sequential and direct access files to be either formatted or unformatted. The subset language requires direct access files to be unformatted, and requires sequential files to be formatted. The OPEN statement has been augmented to accept additional parameters from the full language that are not included in the subset. The CLOSE statement, which is not included in the subset, is provided. I/O is described in more detail in Chapter 11.

* CHAR intrinsic function - Apple FORTRAN includes the CHAR intrinsic function.

In some cases Apple FORTRAN has features that are not anywhere in the ANSI Standard, subset or full langugage. These extensions are the Compiler Directives which have been added to allow you to transmit certain information to the FORTRAN compiler. An additional kind of line, called a Compiler Directive Line, is recognized by the compiler to enable it to receive this information. See Chapter 4 for a description of these statements. Also, Apple FORTRAN includes the EOF intrinsic function.

ANSI 77 vs. Full Language

To help make clear what features are available in the ANSI Standard subset, two appendices that summarize the subset have been included in this manual. Appendix D shows the syntax diagrams for the complete subset, along with those things which are specific to Apple FORTRAN. Appendix E gives a list of all statements in the subset and their syntax.

ANSI 77 vs. ANSI 66

The differences between ANSI FORTRAN 77 and ANSI FORTRAN 66, such as the fact that ANSI 77 deleted the Hollerith data type, are discussed in Appendix F. Additional capabilities were added to ANSI 77, and undefined areas in ANSI 66 were clarified.

CHAPTER 2

FORTRAN READER'S GUIDE

8 Getting Oriented
9 Guide to Pascal Documention
  9 The COMMAND Level
  9 The Filer
  9 The Editor
  9 6502 Assembler
  10 The Linker
  10 Utility Programs

GETTING ORIENTED

As we mentioned previously, this manual should be used with whichever Pascal manual you have to get the complete picture of how to use Apple FORTRAN. The two purposes of this chapter are to give you a list of things to read in those manuals, and to help you interpret those manuals in terms of FORTRAN.

The Pascal documentation gives a complete description of the Editor, Filer, Linker, and numerous other aspects of the operating system. The documentation necessarily gives program examples and diskette names for operating with Pascal. For FORTRAN, the only interpretation required in most cases will be the substitution of diskette names, and the word FORTRAN for Pascal. There are some instances when this will not suffice. All those cases are discussed in this chapter.

Here are some observations about the relationship between the Pascal and FORTRAN languages on the Pascal Operating System:

* Where the Pascal documentation refers to a particular file that is part of the Pascal operating system, such as SYSTEM.APPLE, its function will be the same in Apple FORTRAN as in Apple Pascal. The two principal exceptions to this are SYSTEM.COMPILER, which is the FORTRAN compiler, and SYSTEM.LIBRARY which contains the special unit, RTUNIT.CODE for use only with FORTRAN. Also, there is one file in the Pascal language system called SYSTEM.SYNTAX which has no FORTRAN equivalent.

* The Pascal documentation makes references to Pascal Pseudo-code P-code. Both the FORTRAN and Pascal compilers generate P-code, they don't generate the native 6502 machine code of the Apple. The P-code produced by the compilers is executed by a P-code interpreter, which translates P-code instructions into the native machine code of the Apple. This allows both FORTRAN and Pascal to run on the Pascal Operating System.

* There are two fundamental kinds of diskette files that the operating system uses: TEXT and CODE. TEXT files are in human-readable format, CODE files are machine-readable. TEXT files are just streams of characters, whether they are English, FORTRAN or whatever. CODE files are what the FORTRAN and Pascal compilers generate from reading FORTRAN or Pascal language TEXT files. Files that have names ending with the suffixes .TEXT and .CODE are treated specially by the operating system. For instance, the Editor will only allow you to edit a file that has a .TEXT suffix. The Linker will only link a file that has a .CODE suffix or a .LIBRARY suffix. Files with a .CODE suffix are assumed to have a particular inner organization that allows system programs to manipulate them in appropriate ways. While the system generally takes care of the suffixes itself, it is possible for you to change any file to have any suffix. Some care should be taken to make sure that files with text have a .TEXT suffix, and files with code have a .CODE suffix.

GUIDE TO PASCAL DOCUMENTATION

What follows is a reading guide to the Pascal Operating System documentation, suggesting what you should read for various FORTRAN applications, and making specific substitutions and rewordings. Read the introductory sections of whichever manual you have to get an overview of the Pascal Operating System.

The COMMAND Level

Read the entire chapter of whichever Pascal manual you have for all applications. The section in the Command Level Chapter entitled Making a Turnkey System is not in both of the Pascal manuals. Because that information is important to the FORTRAN user, it is included here.

The Apple Pascal system allows you to set up a turnkey system that will automatically begin running a particular program when the Apple is turned on. To set up your Apple as a turnkey system, first make a copy of all the files on diskette FORT2:, except the SYSTEM.COMPILER which is not needed for a turnkey system. Use the C(hange command in the Filer to change the name of the diskette. For example, you might want to name the copy TURNKEY:. Then T(ransfer a copy of your program codefile onto the turnkey diskette. You must give the new copy of your program the filename SYSTEM.STARTUP.

The Filer

Read the chapter on the Filer for all applications. Remember that the Filer is on diskette FORT1:, so this diskette must be in one of your drives before the Filer can be called.

The Editor

Read the chapter on the Editor for all applications. Remember that the Editor lives on diskette FORT1:.

In the section Text Changing Commands, under I(nsert - Inserting with A(uto-indent TRUE, F(illing FALSE, note that this is the normal setting for writing FORTRAN programs as well as Pascal programs.

6502 Assembler

It is possible to have a FORTRAN program call assembly language subroutines. The chapter on the 6502 Assembler, of whichever Pascal manual you have, discusses how to write and link such subroutines. However, the example host program given at the end of the chapter is in Pascal. The assembly language program ASMDEMO requires changes to a few lines of code to work with FORTRAN. Program ASMDEMO with the changes required by FORTRAN is shown in Chapter 15 of this manual.

The Linker

The section in the Pascal documentation on the Linker is superseded by this manual's Chapter 5.

The Linker combines separately compiled CODE files together into one executable CODE file. The same Linker is supplied with the Pascal and FORTRAN systems. The CODE files that are generated by the Pascal and FORTRAN compilers have the same structure, so they can be correctly linked. However, there are differences between Pascal and FORTRAN in the way that they use the Linker. See Chapter 5 in this manual for more details.

Utility Programs

The section called Formatting New Diskettes should be read for all applications. Both Pascal and FORTRAN systems use the same diskette format and the same Formatter program. Note that the FORMATTER code file is located on diskette APPLE3:, which must be in one of the drives before formatting can be accomplished. This diskette is furnished with the Pascal package. (See also Appendix A).

The section on the System Librarian should be read by those who will be creating, augmenting or altering library files. There is a main library, called SYSTEM.LIBRARY, which is provided on diskette FORT2:. Both Pascal and FORTRAN use the same Librarian program to manipulate libraries; however, that Librarian program must be FORTLIB.CODE found on FORT1:. LIBRARY.CODE on APPLE3: can only be used with Pascal. The structure of libraries is the same, but the content of the SYSTEM.LIBRARY is slightly different in FORTRAN. For one thing, the utility RTUNIT is included with FORTRAN but not Pascal.

Those using the Librarian should also read the section Library Mapping which describes a program that allows you to view the contents of any library.

If you will be using a non-standard console, you will need to reconfigure a part of the operating system so that it uses that console instead of the Apple's. You can reconfigure the Pascal operating system as discussed in the sections System Reconfiguration and Changing GOTOXY Communication.

CHAPTER 3

PROGRAMS IN PIECES

12 Introduction
12 Partial Compilation
13 Source Code in Pieces
13 Object Code in Pieces
14 Units, Segments, and Libraries

INTRODUCTION

The FORTRAN system has a variety of ways that allow you to optimize the stages of writing and organizing programs. One of the most important is that for large programming tasks you can remove groups of subprograms from the main program and put them into individual modules. These modules can then be developed by themselves, and perhaps even be used in more than one program. The ability to treat blocks of code as modules has several advantages over writing one large program:

* Large tasks may be broken down conceptually into sections which can be developed independently and later combined, thereby speeding their development by focusing attention on one problem at a time.

* Subprograms that can be used in many programs need only be written once, then stored in a library for later use.

* Many useful subprograms have been written for you and included in the system library. They may be used directly in your programs. Remember that this library is on diskette FORT2:.

* Procedures written in Pascal and Assembly language can be linked to FORTRAN language main programs.

* Several very efficient methods of handling subprograms are available in the operating system. These methods help make the size of compiled programs smaller, so that the storage capacity of either the disk or the computer's memory is not exceeded by a large program. These methods are discussed more fully later on in this chapter.

Sectioning programs is what makes it possible to treat blocks of code as modules. While this is an extremely useful capability, it introduces some complexities. The purpose of this chapter is to explain the use of the features of the Apple Operating System.

PARTIAL COMPILATION

A FORTRAN program consists of one main program and its appropriate subprograms. In the simplest case, the main program and any subprograms appear in the same TEXT file, called the source code, and are compiled at one time. The result is a CODE file, called the object code, which contains the entire program and is stored on the diskette. When the program is to be executed, it is all loaded in memory at once and started. A program loaded in memory is called a code image.

There are three levels where programs can be broken into modules: the source code level, the object code level, and the code image level while the program is running. Each level has distinct purposes. These purposes and the advantages of using each level will be discussed next.

Source Code in Pieces

It is possible for any part of a program to be split up and placed in different TEXT files. In place of the part of the program removed, special statements are entered in the program which cause the FORTRAN compiler to include the separate files in the compilation at the correct point.

The only restrictions are that you must break the program at line boundaries (you can't split up statements) and that the whole collection of files, when reassemled by the compiler, must still make up a complete FORTRAN program.

Dividing a program into modules can serve two purposes. First, you need not edit an entire program at once. Secondly, it means that different tasks of a program can appear in separate TEXT files. This helps keep program structure and organization clear. For more information on dividing programs into modules, see the discussion of the $INCLUDE statement in Chapter 4.

Object Code in Pieces

If the source code in a TEXT file is a complete subprogram, it can be compiled separately instead of being included in the compilation of the whole program to which it belongs. The technique of compiling subprograms independently from the main program is called separate compilation.

A complete subprogram can still be broken down into different TEXT files, if desired. The point is that no matter how the subprogram is spread across different TEXT files, all the pieces must be supplied to the compiler so as to make at least one complete subprogram.

In the simplest case, separate compilation is done in two steps as follows. First, each of the subprograms is compiled. A single TEXT file may actually contain more than one complete subprogram, if desired. When the compiler begins, it sees that there is no main program included, because the first statement is a SUBROUTINE or FUNCTION statement. The compiler then proceeds to compile the subprograms. When it has successfully compiled the subprograms, the compiler will write out a CODE file containing both the code of the compiled program, as well as a packet of information that will describe just what is in the CODE file. This linker packet will also include details about the subprograms in the compilaton unit such as their type, number of arguments and so on.

The second step in the separate compilation process is the compilation of the main program that uses the subprograms. A PROGRAM statement informs the compiler that it is compiling a main program. A $USES statement must precede the PROGRAM statement before any executable statement. The $USES statement tells the compiler that subprograms in the CODE file named in the $USES statement are required in the main program. The compiler then looks up the named file and reads in the names of all subprograms and their descriptions. The compiler is then able to compile any references to these subprograms that the main program may make. The $USES statement is described in more detail in Chapters 4 and 14.

If no $USES or $EXT statement is encountered by the FORTRAN compiler then the compiler will expect to find the entire program in the current TEXT files being compiled.

Note that the FORTRAN compiler does not incorporate the actual code of the separately compiled subprogram into the main program CODE file it is currently generating. Instead, it generates a packet of information that will tell the Linker how to couple the subprograms into the main program during the linking process.

O.K., that's the simple case. It is possible to do a variety of things with this mechanism, including having separately compiled subprograms called from other separately compiled subprograms.

When the main program and separately compiled subprograms are to be linked into one executable CODE file, you must tell the Linker all the files that make up the one executable program.

There is a companion facility to the FORTRAN $USES statement in the Pascal language compiler called the USES statement that enables FORTRAN programs and Pascal programs to be linked. The actual use of the Linker is discussed in Chapter 5. More complex uses of the $USES statement are discussed in Chapter 14.

UNITS, SEGMENTS, AND LIBRARIES

A group of one or more subprograms, when compiled together, form what is called a compilation unit. When a subprogram is separately compiled, the CODE file which results contains a compilation unit. The term compilation unit is not to be confused with the FORTRAN concept of an I/O unit. An I/O unit refers to a particular input/output device; a compilation unit is a part of a CODE file.

There are two principal ways of using compilation units, as regular units or as OVERLAY units. When regular units are linked into a main program, the subprograms in the unit are actually copied into the resulting CODE file which then contains the code for the main program as well as the code for the subprograms.

The purpose of OVERLAY units is to be able to split up programs that would otherwise be too big for memory. An OVERLAY unit is normally not resident in memory. When the program is loaded into memory prior to execution, the OVERLAY units are not loaded, but remain behind in the CODE file or in SYSTEM.LIBRARY. They remain out of memory until the main program executes a CALL statement for a subroutine or executes a function call in an expression that refers to a subprogram in an OVERLAY unit. Then the whole unit is brought into memory and the appropriate subprogram is executed. When execution of the subprogram is terminated, the space used for the unit is returned to the pool of available memory.

An OVERLAY unit must have been separately compiled in advance of use. It is compiled in the manner described above for separately compiled subprograms. It becomes an OVERLAY unit when it is named in a $USES statement that includes the word OVERLAY. See Chapter 14 for the details of this usage.

The ability to separately compile subprograms makes it possible for more than one main program to use the same subprogram. Usually, these subprograms are placed in a common CODE file, using the library utility program, FORTLIB.CODE on FORT1:. Such a CODE file is then termed a CODE library. This is described in Chapter 14. There is a CODE library that comes with your system named SYSTEM.LIBRARY. It contains a variety of compilation units such as a unit of run-time routines required by most running FORTRAN programs, called RTUNIT. In addition, it contains routines to do color graphics, and a variety of other things. You may put your own "homebrewed" compilation units into this library, too!

CHAPTER 4

THE COMPILER

18 Introduction
18 Files Needed
19 Using the Compiler
21 Form of Input Programs
21   Lower and Upper Case
22   Line Length and Positioning
22 Compiler Directives
24 Compiler Listing

INTRODUCTION

This chapter details the input requirements of the FORTRAN compiler and describes its operation.

All input source files read by FORTRAN must be .TEXT files. This allows the compiler to read large blocks of text from a disk file in a single operation, increasing the compile speed significantly. The simplest way to prepare .TEXT files is to use the screen oriented editor. This means, however, that one cannot type programs directly into the compiler from the keyboard. For a more precise description of the fields in a FORTRAN source statement, see Chapter 6 which explains the basic structure of a FORTRAN program.

The basic purpose of the compiler is to read TEXT files which contain FORTRAN source programs and convert them into P-code. The output file containing the P-code must always have the filename suffix .CODE. Such files are called CODE files.

When you instruct the computer to eXecute your compiled program, the Pascal operating system passes the P-code in the file to a P-code interpreter. After a successful booting operation, the interpreter is resident in the Apple's memory. It is the role of this interpreter to take the P-code instructions and use them to drive the Apple's central processing unit which uses low-level (6502) machine language instructions. The point of a P-code interpreter is that it allows your program to run on virtually any computer that operates Pascal without recompiling. For more information on the pseudo-machine code, see the Pascal documentation.

FILES NEEDED

These diskette files are needed to run the FORTRAN compiler:

Textfile to be compiled	Any diskette, any drive; default is boot diskette's text workfile, SYSTEM.WRK.TEXT, in any drive. For automatic compilation, SYSTEM.WRK.TEXT must be in the boot drive.
SYSTEM.COMPILER	FORT2:, any drive; required. The boot diskette is recommended.

Single-Drive Note: The files SYSTEM.COMPILER and SYSTEM.EDITOR are on diskettes FORT2: and FORT1:, respectively. If you have been using the U(pdate command in the Editor, then your file has been saved in the SYSTEM.WRK.TEXT file on diskette FORT1:. If you wish to C(ompile a file not on FORT2:, you should use the Filer's T(ransfer command to transfer that TEXT file to diskette FORT2:. Note that due to space considerations on FORT2:, large compilations cannot be handled. After being compiled, your program will need to be linked using the Linker. See this manual's Chapter 5 on that subject for help.

Multi-drive note: The file SYSTEM.COMPILER is on diskette FORT2:, which is normally kept in disk drive volume #4: in a multi-drive system. The Editor is in file SYSTEM.EDITOR and is found on diskette FORT1:, which is the normal drive volume #5 diskette. With these two diskettes, it is possible to edit, link, and run FORTRAN programs.

USING THE COMPILER

There are three steps to taking a FORTRAN program from a TEXT file to a running program. First you compile the source (the TEXT file), then you link the object code (the CODE file), then you execute the object code, which loads the code into memory and begins running it. There are two ways to do all these steps, either manually, or automatically. The Command level has a R(un command which automatically compiles, links, loads and executes a program. If you worked through the tutorial in Appendix A, you are familiar with the effects of the R(un command. Below are the details of the individual steps that operate the Compiler as a separate entity.

The Compiler is invoked by typing C from the Command level of the Pascal operating system. The screen immediately shows the message

```
COMPILING...
```

The Compiler automatically compiles the boot diskette's workfile SYSTEM.WRK.TEXT, if that file exists, or another workfile designated by the Filer's G(et command. If compilation is successful, the compiler saves the resulting code as SYSTEM.WRK.CODE. If there is a workfile, but you do not wish to compile that file, use the Filer's N(ew command to clear away the workfile before compiling. If no workfile is available, you are prompted by the Compiler for a source filename:

```
COMPILE WHAT TEXT?
```

You should respond by typing the name of the text file that you wish to have compiled.

If no .TEXT suffix is specified, the system will add one automatically. If you wish to defeat this suffix-adding feature and compile a textfile whose filename does not end in .TEXT, type a period after the last character of your filename.

Next you will be asked for the name of the file where you wish to save the compiled version of your program:

```
TO WHAT CODEFILE?
```

If you simply press the RETURN key the compiled version of your program will be saved on the boot diskette's workfile SYSTEM.WRK.CODE.

If you want the compiled version of your program to have the same name as the text version of your program, just type a dollar sign ($) and press the RETURN key. Of course, the suffix will be .CODE instead of .TEXT. This is a handy feature, since you will usually want to remember only one name for both versions of your program. The dollar sign repeats your entire source file specification, including the volume identifier, so do not specify the diskette before typing the dollar sign. Note that this use is different from the use of the dollar sign in the Filer.

If you want your program stored under another filename, type the desired filename. If no .CODE suffix is specified, the system will add one automatically. If you wish to defeat this feature, in order to specify an output filename that does not have a .CODE suffix, type a period after the last character of your output filename.

The Compiler then begins compiling the specified file. While the compiler is running, messages on the screen show the progress of the compilation. Below is an example of the messages which appear on the screen:

```
FORTRAN COMPILER II.1 [1.0]

<   0>...
ARR        [3849 WORDS]
<   3>...........
SUBARR     [3855 WORDS]
<  13>.
14 LINES.     0 ERRORS.
SMALLEST AVAILABLE SPACE = 3849 WORDS.
```

The identifiers appearing on the screen, in this case SUBARR and ARR, are the program name and the names of the subprograms that are included. The identifier for a subprogram is displayed at the moment when compilation of the body of the subprogram is started. The numbers within [ ] indicate the number of 16 bit words available for symbol table storage at that point in the compilation. The numbers enclosed within < > are the current line numbers. Each dot on the screen represents one source line compiled.

If the compilation is successful (that is, no programming errors are detected), the Compiler saves the compiled code under the filename SYSTEM.WRK.CODE on the boot diskette, or under another filename that you specified earlier.

If the compiler reports an error, it will tell you which line number was being compiled when the error was detected and will give a code for the kind of error. The following example shows an error message during compilation:

```
FORTRAN COMPILER II.1  [1.0]

<   0>...
RTEST    [3577 WORDS]
<   3>.........
***** ERROR NUMBER:  159 IN LINE:  12
<SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

Compilation stops when an error is detected. You are prompted on the screen to type E to go back to the Editor to fix the problem, press the spacebar to continue the compilation, or press the escape key to return to the Command level. If you choose to return to the Editor, the Editor will be started up automatically. The cursor will be positioned at the beginning of the line mentioned in the error message or at a line or two past the error line. You may then correct the problem and recompile. The list of compiler error messages with their corresponding error numbers appears in Appendix B of this manual.

The code workfile, SYSTEM.WRK.CODE, is automatically erased when any text workfile is U(pdated from the Editor. So, if you have compiled anything into that name, you may want to rename it using the Filer if you don't want to lose it.

FORM OF INPUT PROGRAMS

FORTRAN source programs will usually be prepared using the Editor. The FORTRAN system will process them regardless of their means of preparation, however, providing they are in valid TEXT format. For instance, a FORTRAN program from another processor can be transferred to the Apple over the REMIN: remote input port. The Filer's T(ransfer command can be used to copy text arriving via REMIN: into a diskette file. See the information on the Filer in the Pascal documentation. Source programs must be TEXT files and must be passed to the Compiler from a blocked file device (i.e., a diskette).

Lower and Upper Case

Apple FORTRAN accepts both upper and lower case input, or any mixture of upper and lower case with the following convention:

* No distinction is made between upper and lower case to represent FORTRAN keywords or defined symbols in a program. Example: oPeN, OPEN, and open all stand for the same FORTRAN keyword.

* In character constants, the exact letters provided by the user are passed directly through the system.

* Options to I/O statements which have the syntax of character constants may be specified in either upper or lower case. Example: OPEN(1,FILE='*Bats').

Line Length and Positioning

Apple FORTRAN allows lines of text to be up to 72 columns wide. Shorter lines are not padded out to 72 columns with blanks. Text beyond column 72 is ignored. FORTRAN reserves the first 6 columns to represent whether the line is an initial or continuation line. Columns 1 through 5 are for a label, and a nonblank character in column 6 denotes a continuation line.

The fact that shorter lines are not padded out with blanks means that character constants that are split across lines will not end up with a lot of blanks in them. See Chapter 7 for a discussion of the treatment of character constants.

Lines longer than 72 columns are truncated to 72 columns. No error message is generated by characters appearing beyond column 72 unless this truncation introduces some syntax error. It may later generate a run-time error, however, or simply cause some strange behavior in your program.

The FORTRAN compiler can produce a listing of the program source that it compiled. The listing reflects the columns read by the FORTRAN compiler and can be checked for unintentional line truncations. But it's far better to avoid the truncations in the first place.

Source lines that are empty or completely filled with blanks are treated as comment lines by FORTRAN 77. Neither ANSI FORTRAN 77 nor Apple FORTRAN allow comments following the final END statement of a program. This means that extra blank lines at the end of a source program file will cause a compile-time error to be generated stating: Missing END Statement.

To further complicate matters, it is a Pascal operating system convention that TEXT files must contain a blank line at the end of the file. To overcome this convention, the Apple FORTRAN compiler has been adapted to accept exactly one blank line following the final END statement. Files prepared with the Pascal operating system text editor automatically acquire this final blank line which the editor makes invisible to the user. Normally prepared source files which terminate with an END statement followed by a single RETURN character and nothing else will be acceptable to Apple FORTRAN. Two RETURN characters following an END statement will cause a confusing error message stating: Missing END Statement.

COMPILER DIRECTIVES

Compiler directives provide you with a way of communicating certain information to the compiler via the text of the file being compiled. To this end, Apple FORTRAN recognizes another kind of line, besides comment lines, initial and continuation lines, called a compiler directive line. A dollar sign ($) appears in column 1 of such compiler directive lines.

Some of these directives are restricted to certain locations in the text. Specifically, the $INCLUDE statement may occur anywhere a comment line may appear. The other directives must appear before any specification or executable statement, but otherwise have no restrictions on placement or order. These directives are:

$INCLUDE filename

To facilitate the manipulation of large programs, the Apple compiler has extended the FORTRAN 77 standard with an $INCLUDE compiler directive. The directive must have the $ appearing in column 1. The meaning is to compile the contents of the file 'filename' before continuing with the current file. The included file may contain additional $INCLUDE directives, up to a maximum of five levels of files (four levels of $INCLUDE directives). It is often useful to have the description of a COMMON block kept in a single file and to include it in each subroutine that references that COMMON area, rather than making and maintaining many copies of the same source, one in each subroutine. There is no limit to the number of $INCLUDE directives that can appear in a source file. An $INCLUDE can appear anywhere a comment line is legal.

$USES ident [IN filename] [OVERLAY]

The $USES statement has the effect of making separately compiled subprograms known to the FORTRAN compiler. This allows the program being compiled to refer to that separately compiled code. The named file must be a CODE file. The separately compiled FORTRAN subroutines or Pascal procedures contained in the named file, or in the file SYSTEM.LIBRARY if no file name is present, become defined and available to the currently compiling program. This directive must appear before the first non-comment input line. The optional OVERLAY statement is used if the named unit is to be brought into memory during execution only while being referenced from the main program, instead of having the code added to the CODE file of the host program. See Chapter 14 for more information on $USES.

$XREF

Produces a cross-reference listing of the compilation.

$EXT SUBROUTINE name #params
$EXT [type] FUNCTION name #params

The subroutine or function specified by 'name' is an assembly language routine. The routine has exactly '#params' reference parameters. $EXT for a given name should occur only once in a compilation. If a program and a unit called by that program both use a given assembly language routine, the $EXT should only occur in the program or the unit, but not both.

COMPILER LISTING

The compiler listing, if requested, contains various bits of information that may be useful to the FORTRAN programmer. The listing consists of the user's source code as read, along with line numbers, symbol tables, and error messages. Also, cross reference information is listed if the $XREF compiler directive is specified. A sample compiler listing follows. Note that the compiler listing does not stop printing when an error message is being listed. Note also that if you send a listing file, that file must be on a different volume from that of the output code file.

SAMPLE COMPILER LISTING:

```
FORTRAN Compiler II.1  [1.0]
    0.      0 C
    1.      0 C --- Example Program #1234
    2.      0 C
    3.      0
    4.      0 $XREF
    5.      0
    6.      0         PROGRAM EX1234
    7.      0
    8.      0         INTEGER A(10,10)
    9.      0         CHARACTER*4 C
   10.      0
   11.      0         CALL INIT(A,C)
   12.      6         I = 1
   13.      9   200 A(I) = I
***** Error number: 57 in line: 13
   14.     20         I = I + 1
   15.     26         IF (IABS(10-I) .NE. 0) GOTO 200
   16.     37
   17.     37         END

A      INTEGER       3                    8  11  13
C      CHAR*  4    103                    9  11
EX1234 PROGRAM                            6
I      INTEGER     105                   12  13  13  14  14  15
IABS   INTRINSIC                         15
INIT   SUBROUTINE               2,FWD    11


   18.      0         SUBROUTINE INIT(B,D)
   19.      0         INTEGER B(10,10)
   20.      0         CHARACTER*4 D
   21.      0
   22.      0         RETURN
   23.      2         END

B      INTEGER       2*                  18  19
D      CHAR*  4      1*                  18  20
INIT   SUBROUTINE               2        18


EX1234 PROGRAM
INIT   SUBROUTINE               2,7

24 lines.   1 errors.
Smallest available space = 3966 words.
```

The first line indicates which version of the compiler was used for this compilation. In the example it is version 1.0 for the operating system version II.1. The leftmost column of numbers is the source line number. The next column indicates the procedure relative instruction counter that the corresponding line of source code occupies as object code. It is only meaningful for executable statements and data statements. To the right of the instruction counter is the source statement.

Errors are indicated by a row of asterisks followed by the error number and line number, as appears in the example between lines 13 and 14. In this case it is error number 57, "Too few subscripts", indicating that there are not enough subscripts in the array reference A(I).

At the end of each program unit (function, subroutine, or main program), a local symbol table is printed. This table lists all identifiers that were referenced in that program unit, along with their definition. If the $XREF compiler directive has been issued, a table of all lines containing an instance of that identifier in the current program unit is also printed.

If the identifier is a variable, it is accompanied by its type and location. If the variable is a parameter, its location is followed by an asterisk, such as the variables B and D in the SUBROUTINE INIT. If the variable is in a common block, then the name of the block follows between two slashes.

If the identifier is not a variable, it is described appropriately. For subroutines and functions, the unit relative procedure number is given. If it resides in a different segment, then the segment number follows. If the compiler assumes that it will reside in the same segment, but has not appeared yet, it is listed as a forward program unit by the notation FWD.

At the end of the compilation the global symbol table is printed. It contains all global FORTRAN symbols referenced in the compilation. No cross reference is given. The number of source lines compiled and the number of errors encountered follows. If there were any errors, then no object file is produced.

The last line shows the maximum amount of RAM used, recorded at the first executable statement of each program unit. This can be used as an indication of the amount of memory that remains available for additional symbols. When the available memory space becomes less than 700 to 1000 words, the symbol table is nearly full, since some of the memory pool is used for other purposes during the compilation of executable statements.

CHAPTER 5

THE LINKER

28 Introduction
28 Diskfiles Needed
29 Using the Linker

INTRODUCTION

The Pascal operating system Linker is used to merge separately compiled CODE files. For a discussion of this strategy in organizing program development see Chapter 3. While the same Linker is provided for both Pascal and FORTRAN, its uses are slightly different. So ignore the information on the Linker in the Pascal Operating System documentation: read this chapter instead.

DISKFILES NEEDED

The following files allow you to use the Linker:

SYSTEM.LINKER	FORT1:, any drive; required
Host codefile	Any diskette, any drive. Default is to boot diskette's code workfile SYSTEM.WRK.CODE
SYSTEM.LIBRARY	FORT2:, boot drive; required by every FORTRAN program.
Library codefiles	Any diskettes, any drives; default is to the library file SYSTEM.LIBRARY, supplied on diskette FORT2:, any drive.
SYSTEM.CHARSET	Either FORTRAN diskette, any drive; required if the program uses WCHAR from TURTLEGRAPHICS.

Multi-drive note: On multiple-drive systems, diskette FORT2: is normally your boot diskette. If FORT2: is in the boot drive (volume #4, remember?), and FORT1: is in any other drive, your system will have available all the diskfiles it needs to E(dit, C(ompile, L(ink, X(ecute and R(un. But to FORMAT, you'll need APPLE3:.

Two-drive note: To L(ink when the host and library files are not already on FORT2:, you can use the Filer to T(ransfer the needed files onto one of the FORTRAN diskettes before linking.

Single-drive note: You can link without the linker file in the drive. First put FORT1: in the drive, and from Command level, type L. This loads the Linker routine into memory. Now you can put FORT2: back in the drive and continue the linking operation. See Appendix A for a complete description of this procedure.

USING THE LINKER

Even if a compiled FORTRAN program is contained in a single CODE file, the Linker is still required to link in code from the FORTRAN SYSTEM.LIBRARY. The SYSTEM.LIBRARY is necessary to execute all FORTRAN programs. So running the Linker will always be a step in creating a running program. With the R(un command, this step is done automatically. The following describes how to run the Linker manually.

Start the Linker by typing L for L(ink from the COMMAND level prompt of the FORTRAN system and receive the prompt:

```
LINKING...

LINKER II.1 [A4]
HOST FILE?
```

The hostfile is the main program CODE file into which subprograms are to be linked. If you just press the RETURN key in response to the prompt, the Linker uses the boot diskette's workfile SYSTEM.WRK.CODE as the hostfile. If your main program is not in the workfile, enter the filename containing it. If the Linker cannot find a file with the exact filename you typed, it adds the suffix .CODE to the filename, if the suffix was not specified originally, and tries again. For this reason, if you respond by typing the non-existent filename MYDISK:MYFILE the Linker returns the message

```
NO FILE MYDISK:MYFILE.CODE
```

You'll also notice that when the Linker reports information about a file it always displays the full name, including the diskette name.

After successfully finding a host file, the Linker then asks for the name of the first CODE file containing separately compiled subprograms that are to be linked to the main program.

```
LIB FILE?
```

There are two kinds of files that can be supplied here, either a library file or a single compilation unit. The Linker treats them in exactly the same way. Typing * and then pressing the RETURN key in response to a request for a library file name will cause the Linker to reference SYSTEM.LIBRARY on the boot diskette. This must always be done for FORTRAN main programs, because a special unit, called RTUNIT which contains all the required FORTRAN runtime routines is in that library.

Example:

```
        LIB FILE? *
        OPENING SYSTEM.LIBRARY
```

Once it has referenced SYSTEM.LIBRARY, it again prompts you with

```
        LIB FILE?
```

Now, if appropriate, is the time to enter the names of any CODE files besides SYSTEM.LIBRARY that your program requires. Again, the Linker looks first for the exact filename that you type, and if the search was unsuccessful, adds the suffix .CODE and looks again. In any case, it always displays the name of the file actually opened. The Linker will continue to prompt you for files. Up to eight library files may be included in one linking operation.

The term "library" is a little confusing, because in fact any CODE file can be treated as a library. A CODE file can be thought of as a library with only one compilation unit in it. Using the library utility program, FORTLIB.CODE, supplied on diskette FORT1:, it is possible to make CODE files contain more than one entry. You may also remove compilation units from libraries with the library utility program. The file SYSTEM.LIBRARY on diskette FORT2: was created with this program. For information on LIBRARIES and the LIBRARIAN see the documentation for the Pascal Operating System in the Utility Programs Section.

If you specify a library file that does not contain the proper information, you may get one of these messages:

```
        BAD SEG KIND    (Is this really a Pascal or FORTRAN program?)
        BAD SEG NAME    (Is this a text file ?)
```

If the file SYSTEM.LIBRARY is not available on the diskette in volume #4, this somewhat odd message appears:

```
        NO FILE *SYSTEM.LIBRARY
        TYPE <SP>(CONTINUE), <ESC>(TERMINATE)
```

This is that same interesting feature of the Linker that causes it to append .CODE to the end of a file not found and look again. In this case, its own internal specification told it to look for *SYSTEM.LIBRARY. Failing to find that, it then looked again, this time for *SYSTEM.LIBRARY.CODE which also failed because it was still looking on the boot diskette. After giving this message, the Linker does not allow you to specify a different library file so there is little point in continuing. Press the ESC key to go back to Command level and try again after relocating the SYSTEM.LIBRARY back on the boot diskette.

When all relevant library file names have been entered, answer the next LIB FILE? prompt by just pressing the RETURN key to proceed. The Linker will now prompt with:

```
        MAP NAME?
```

If you respond by typing a file name, the Linker writes a mapfile which contains a text version of what the Linker did to link the CODE files. Note that the suffix .TEXT is appended to the specified filename unless a period is the last letter of the filename. Normally you will simply press the RETURN key. This causes no mapfile to be written. The mapfile is a diagnostic and system programming tool, and is not required for most uses of the Linker.

The Linker now reads all the CODE files presented and begins the linking process. If all the right subprograms are not present, the Linker will respond with an appropriate message:

```
        UNIT,
        PROC,
        FUNC,
        GLOBAL,
        PUBLIC <identifier> UNDEFINED
        TYPE <SP>(CONTINUE), <ESC>(TERMINATE)
```

You can press the spacebar, and the Linker will proceed, trying to link whatever routines or UNITs are available in SYSTEM.LIBRARY. Later, you can use the Linker explicitly to link in the remaining subprograms.

When the Linker is ready to write an output CODE file, you are prompted to type a filename for it to use:

```
        OUTPUT FILE?
```

You will often want the same filename as that of the host file, but you may not use the $ same-name option offered by the Compiler and Filer. The Linker may not add any suffix to the output filename you specify; if you want to insure a normal, executable code file, you should explicitly include the .CODE suffix in the filename when you type it. After this output file specification has been typed, press the RETURN key and linking will commence. Responding with no filename by pressing only the RETURN key causes the linked output to be saved in the boot diskette's workfile, SYSTEM.WRK.CODE.

During the linking process, the Linker will report on all subprograms being copied into the output CODE file. The linking process will be stopped if any required routines are missing or undefined. You will be told what was missing and allowed to terminate or continue the linking process.

Here is a sample session with the Linker for linking a main program called MYPRG to a separately compiled code file named X.CODE. The main program has a $USES statement in it that references a subprogram in X.CODE.

```
Linker II.1 [A4]
Host file? MYPRG
Opening MYPRG.CODE
Lib file? *
Opening SYSTEM.LIBRARY
Lib file? X
Opening X.CODE
Lib file?
Map name?
Reading MAINSEGX
Reading MYPRG
Reading RTUNIT
Reading X
Output file? OUTPRG.CODE
Linking MYPRG    # 7
Linking RTUNIT   # 8
Linking X        # 9
Linking MAINSEGX #1
```

You could then eX(ecute the code file OUTPRG.CODE. If you don't specify the output file name, boot diskette is used.

CHAPTER 6

PROGRAM STRUCTURE

34 Introduction
34 Character Set
35 Lines
35 Columns
36 Blanks
36 Comment Lines
36 Statements, Labels, and Lines
37 Statement Ordering
38 The END Statement

INTRODUCTION

A FORTRAN program is a sequence of characters that are interpreted by the compiler in various contexts as characters, identifiers, labels, constants, lines, statements or other syntactic groupings. The rules the compiler uses to group the character stream into substructures, as well as various constraints on how these substructures may be related to each other in the source program, are the topic of this chapter.

First, however, it is important to note the notation conventions used in this manual:

* Upper case and special characters are to be written as shown in programs.

* Lower case letters and words indicate entities for which there is a substitution in actual statements as described in the text. The reader may assume that once a lower case entity is defined, it retains it meaning for the entire context of discussion.

> Example: The format which describes editing of integers is denoted 'Iw', where w is a nonzero, unsigned integer constant. Thus, in an actual statement, a program might contain I3 or I44. The format which describes editing of reals is 'Fw.d', where d is an unsigned integer constant. In an actual statement, F7.4 or F22.0 are valid. Notice that the period, as a special character, is taken literally.

* Brackets indicate optional items.

> Example: 'A[w]' indicates that either A or A12 are valid as a means of specifying a character format.

* Three dots (...) are used to indicate ellipsis. That is, the optional item preceding the three dots may appear one or more times.

> Example: The computed GOTO statement is described by 'GOTO (s [, s] ...) [,] i' indicating that the syntactic item denoted by s may be repeated any number of times with commas separating the items.

* Blanks normally have no significance in the description of FORTRAN statements. The general rules for blanks, covered in this chapter, govern the interpretation of blanks in all contexts.

CHARACTER SET

A FORTRAN source program consists of a stream of characters, originating in a .TEXT file, consisting of:

* Fifty-two upper and lower case letters A through Z and a through z

* Digits from 0 to 9

* Special characters consisting of the remaining printable characters of the ASCII character set

The letters and digits, treated as a single group, are called the alphanumeric characters. FORTRAN interprets lower case letters as if they were upper case letters in all contexts except in character constants and hollerith fields. Thus, the following user defined names are all the same to the FORTRAN system:

ABCDE abcde AbCdE aBcDe

In addition, actual source programs submitted to the FORTRAN compiler contain certain hidden or nonprintable control characters inserted by the text editor which are invisible to the user. FORTRAN interprets these control characters in exactly the same way that the text editor does and transforms them, using the rules of Apple Pascal .TEXT files, into the FORTRAN character set.

The collating sequence for the FORTRAN character set is the same as the ASCII sequence. Refer to Table 5 in Appendix C.

LINES

A FORTRAN source program may also be thought of as a sequence of lines, corresponding to the normal notion of lines in the text editor. Only the first 72 characters in a line are treated as significant by the compiler, with any trailing characters in a line ignored. Note that lines with fewer than 72 characters are possible and, if shorter than 72 columns, the compiler does treat as significant the length of a line. See Chapter 7 which describes character constants for an illustration of this. This 72 column format is a throwback to the days of punched cards, when each statement or line required its own card.

COLUMNS

The characters in a given line fall into columns that are numbered from left to right, beginning with column 1. The column in which a character resides is significant in FORTRAN. Columns 1 through 5 are reserved for statement labels, column 6 is used to indicate a continuation line, and executable statements start in column 7.

BLANKS

The blank character, with the exceptions noted below, has no significance in a FORTRAN source program and may be used for the purpose of improving the readability of FORTRAN programs. The exceptions are:

* Blanks within string constants are significant.

* Blanks within Hollerith fields are significant.

* Blanks on compiler directive lines are significant.

* A blank in column 6 is used in distinguishing initial lines from continuation lines.

* Blanks are included in the total count of characters that the compiler must process per line and per statement.

COMMENT LINES

A line is treated as a comment if any one of the following conditions is met:

* A C or c character in column 1.

* Asterisk (*) in column 1.

* Line all blanks.

Comment lines do not effect the execution of the FORTRAN program in any way. Comment lines must be followed immediately by an initial line or another comment line. They must not be followed by a continuation line. Note that extra blank lines at the end of a FORTRAN program result in a compile time error since the system interprets them as comment lines but they are not followed by an initial line.

STATEMENTS, LABELS, AND LINES

We will define a FORTRAN statement in terms of the input character stream. The compiler recognizes certain groups of input characters as complete statements according to the rules specified here. Specific statements and their properties will be covered individually. When it is necessary to refer to specific kinds of statements here, they are simply referred to by name.

A statement label is a sequence of from one to five digits. At least one digit must be nonzero. A label may be placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant. It is traditional to left-justify statement labels. A statement label on a nonexecutable statement is ignored.

An initial line is any line that is not a comment line or a compiler directive line and contains a blank or a 0 in column 6. The first five columns of the line must either be all blank or contain a label. With the exception of the statement following a logical IF, FORTRAN statements begin with an initial line.

A continuation line is any line which is not a comment line or a compiler directive line and contains any character in column 6 other than a blank or a 0. The first five columns of a continuation line must be blanks. A continuation line is used to increase the amount of room to write a given statement. If a statement will not fit on a single initial line, it may be extended to include up to 9 continuation lines.

A FORTRAN statement consists of an initial line, which may be followed by up to 9 continuation lines. The characters of the statement are the total number of characters, up to 660, found in columns 7 through 72 of these lines.

STATEMENT ORDERING

The FORTRAN language enforces a certain ordering among statements and lines which make up a FORTRAN compilation. In general, a compilation consists of from zero to some number of subprograms and at most one main program. Refer to Chapter 13 for more information on programs, subroutines, and functions, as well as the FORTRAN statements mentioned in this section. The rules for ordering statements appear below.

A subprogram begins with either a SUBROUTINE or a FUNCTION statement and ends with an END statement. A main program begins with a PROGRAM statement, or any other than a SUBROUTINE or FUNCTION statement, and ends with an END statement. A subprogram or a main program is often called a program unit.

Within a program unit, whether a main program or a subprogram, statements must appear in an order consistent with the following rules:

* A SUBROUTINE or FUNCTION statement, or PROGRAM statement if present, must appear as the first statement of the program unit.

* FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION statement, or PROGRAM statement if present.

* Specification statements must precede all DATA statements, statement function statements, and executeable statements.

* DATA statements must appear after the specification statements and precede all statement function statements and executable statements.

* Statement function statements must precede all executable statements.

* Within the specification statements, the IMPLICIT statement must precede all other specification statements.

These rules are summarized in the program rules chart that follows.

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="2">PROGRAM, FUNCTION, or SUBROUTINE Statement</td></tr>
<tr><td rowspan="5">FORMAT Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td>DATA Statements</td></tr>
<tr><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Guidelines for interpreting the Program Rules Chart:

* Classes of lines or statements above or below other classes must appear in the designated order.

* Classes of lines or statements may be interspersed with other classes which appear across from one another.

THE END STATEMENT

When creating FORTRAN programs with the Apple editor, the final END statement must be entered as an initial line. That is, there must be a RETURN character following the statement. Otherwise, the compiler will not find the END statement and will issue an error message. In addition, that RETURN character must be the final character in the program source file. Any further characters, even blanks, may be considered part of a subsequent subprogram by the compiler. The END statement is a source of many compilation errors. When you are ready to put the END statement in your program, type it carefully with the E in column 7. Type the three letters, followed immediately by one and only one RETURN.

CHAPTER 7

DATA TYPES

40 Introduction
40 The Integer Type
40 The Real Type
41 The Logical Type
41 The Character Type

INTRODUCTION

There are four basic data types in Apple FORTRAN: Integer, real, logical, and character. This chapter describes the properties of each type, the range of values for each type, and the form of constants for each type.

THE INTEGER TYPE

The integer data type consists of a subset of the integers. An integer value is an exact representation of the corresponding integer. An integer variable occupies one word, two bytes, of storage and can contain any value in the range -32768 to 32767. Integer constants consist of a sequence of one or more decimal digits preceded by an optional arithmetic sign, + or -, and must be in the allowable value range. A decimal point is not allowed in an integer constant. The following are examples of integer constants:

```
123     +123     -123     0      00000123     32767     -32768
```

THE REAL TYPE

The real data type consists of a subset of the real numbers. A real value is normally an approximation of the real number desired. A real variable occupies two consecutive words, four bytes, of storage. The range of real values within a power of 10 is approximately:

```
-1.7E+38  ...  -5.8E-39  0.0  5.8E-39  ...  1.7E+38
```

A basic real constant consists of an optional sign followed by an integer part, a decimal point, and a fraction part. The integer and fraction parts consist of one or more decimal digits. Either the integer part or the fraction part may be omitted, but not both. Some examples of real constants follow:

```
-123.456                 +123.456               123.456
-123.                    +123.                  123.
  -.456                    +.456                 .456
```

An exponent part consists of the letter 'E' followed by an optionally signed integer constant. An exponent indicates that the value preceding it is to be multiplied by 10 to the value of the exponent part's integer. Some sample exponent parts are:

```
E12             E-12            E+12            E0
```

A real constant is either a basic real constant, a basic real constant followed by an exponent part, or an integer constant followed by an exponent part. For example:

```
+1.000E-2               1.E-2                   1E-2
+0.01                   100.0E-4                .0001E+2
```

all represent the same real number, 1/100.

THE LOGICAL TYPE

The logical data type consists of the two logical values true and false. A logical variable occupies one word, two bytes, of storage. There are only two logical constants, .TRUE. and .FALSE., representing the two corresponding logical values. The internal representation of .FALSE. is a word of all zeros, and the representation of .TRUE. is a word of all zeros except a one in the least significant bit. If a logical variable contains any other bit values, its logical meaning is undefined.

THE CHARACTER TYPE

The character data type consists of a sequence of ASCII characters. The length of a character value is equal to the number of characters in the sequence. The length of a particular constant or variable is fixed, and must be between 1 and 255 characters. A character variable occupies one word of storage for each two characters in the sequence, plus one word if the length is odd. Character variables are always aligned on word boundaries. The blank character is allowed in a character value and is significant.

A character constant consists of a sequence of one or more characters enclosed by a pair of apostrophes. Blank characters are allowed in character constants, and count as one character each. An apostrophe (or single quote) within a character constant is represented by two consecutive apostrophes with no blanks in between. The length of a character constant is equal to the number of characters between the apostrophes, with doubled apostrophes counting as a single apostrophe character. Some sample character constants are:

```
'A'     ' '     'Help!'     'A very long CHARACTER constant'     ''''
```

Note the last example, '''', that represents a single apostrophe, '.

FORTRAN allows source lines with up to 72 columns. Shorter lines are not padded out to 72 columns, but left as input. When a character constant extends across a line boundary, its value is as if the portion of the continuation line beginning with column 7 is juxtaposed immediately after the last character on the initial line. Thus, the FORTRAN source:

```
     200   CH = 'ABC<cr>
          X DEF'
```

where <cr> indicates a carriage return, or the end of the source line is equivalent to:

```
     200   CH = 'ABC DEF'
```

with the single space between the C and D being the equivalent to the space in column 7 of the continuation line. Very long character constants can be represented in this manner.

CHAPTER 8

FORTRAN STATEMENTS

44 Introduction
44 FORTRAN Names
  44 Scope of FORTRAN Names
  45 Undeclared Names
45 Specification Statements
  46 IMPLICIT Statement
  47 DIMENSION Statement
  48 Type Statement
  49 COMMON Statement
  50 EXTERNAL Statement
  51 INTRINSIC Statement
  51 SAVE Statement
  51 EQUIVALENCE Statement
52 DATA Statements
53 Assignment Statements
  54 Computational Assignment Statement
  54 Label Assignment Statement

INTRODUCTION

This chapter describes specification statements, DATA statements, and assignment statements in Apple FORTRAN. The rules for forming FORTRAN names and the scope of names are included in this chapter also.

FORTRAN NAMES

A FORTRAN name, or identifier, consists of an initial alphabetic character followed by a sequence of 0 through 5 alphanumeric characters. Blanks may appear within a FORTRAN name, but have no significance. A name is used to denote a user or system defined variable, array, function, subroutine, and so forth. Any valid sequence of characters may be used for any FORTRAN name. There are no reserved names as in other languages. Sequences of alphabetic characters used as keywords are not to be confused with FORTRAN names. The compiler recognizes keywords by their context and in no way restricts the use of user chosen names. Thus, a program can have, for example, an array named IF, READ, or GOTO, with no error indicated by the compiler, as long as it conforms to the rules that all arrays must obey.

Scope of FORTRAN Names

The scope of a name is the range of statements in which that name is known or can be referenced within a FORTRAN program. In general, the scope of a name is either GLOBAL or LOCAL, although there are several exceptions. A name can only be used in accordance with a single definition within its scope. The same name, however, can have different definitions in distinct scopes.

A name with global scope may be used in more than one program unit, a subroutine, function, or the main program, and still refer to the same entity. In fact, names with global scope can only be used in a single, consistent manner within a program. All subroutine, function subprogram, and common names, as well as the program name, have global scope. Therefore, there cannot be a function subprogram that has the same name as a subroutine subprogram or a common data area. Similarly, no two function subprograms in the same program can have the same name.

A name with local scope is only defined within a single program unit. A name with a local scope can be used in another program unit with a different or similar meaning, but is in no way required to have a similar meaning in a different scope. The names of variables, arrays, parameters, and statement functions all have local scope. A name with a local scope can be used in the same compilation as another item with the same name but a global scope as long as the global name is not referenced within the program unit containing the local name. Thus, a function can be named FOO, and a local variable in another program unit can be named FOO without error, as long as the program unit containing the variable FOO does not call the function FOO. The compiler detects all scope errors, and issues an error message should they occur, so the user need not worry about undetected scope errors causing bugs in programs.

One exception to the scoping rules is the name given to common data blocks. It is possible to refer to a globally scoped common name in the same program unit that an identical locally scoped name appears. This is allowed because common names are always enclosed in slashes, such as /NAME/, and are therefore always distinguishable from ordinary names by the compiler.

Another exception to the scoping rules is made for parameters to statement functions. The scope of statement function parameters is limited to the single statement forming that statement function. Any other use of those names within that statement function is not allowed, and any other use outside that statement function is allowed.

Undeclared Names

When a user name that has not appeared before is encountered in an executable statement, the compiler infers from the context of its use how to classify that name. If the name is used in the context of a variable, the compiler creates an entry into the symbol table for a variable of that name. Its type is inferred from the first letter of its name. Normally, variables beginning with the letters I, J, K, L, M, or N are considered integers, while all others are considered reals, although these defaults can be overridden by an IMPLICIT statement. If an undeclared name is used in the context of a function call, a symbol table entry is created for a function of that name, with its type being inferred in the same manner as that of a variable. Similarly, a subroutine entry is created for a newly encountered name that is used as the target of a CALL statement. If an entry for such a subroutine or function name exists in the global symbol table, its attributes are coordinated with those of the newly created symbol table entry. If any inconsistencies are detected, such as a previously defined subroutine name being used as a function name, an error message is issued.

In general, one is encouraged to declare all names used within a program unit, since it helps to assure that the compiler associates the proper definition with that name. Allowing the compiler to use a default meaning can sometimes result in logical errors that are quite difficult to locate.

SPECIFICATION STATEMENTS

This section describes the specification statements in Apple FORTRAN. Specification statements are non-executable. They are used to define the attributes of user defined variable, array, and function names. There are eight kinds of specification statements: These are the

IMPLICIT, DIMENSION, type, COMMON, EXTERNAL, INTRINSIC, SAVE, and EQUIVALENCE statements.

Specification statements must precede all executable statements in a program unit. If present, any IMPLICIT statements must precede all other specification statements in a program unit as well. The specification statements may appear in any order within their own group.

IMPLICIT Statement

An IMPLICIT statement is used to define the default type for user declared names. The form of an IMPLICIT statement is:

```
     IMPLICIT type (a [,a]...) [,type (a [,a]...)]...
```

where: type is one of INTEGER, LOGICAL, REAL, or CHARACTER[*nnn]

a	is either a single letter or a range of letters. A range of letters is indicated by the first and last letters in the range separated by a minus sign. For a range, the letters must be in alphabetic order.
nnn	is the size of the character type that is to be associated with that letter or letters. It must be an unsigned integer in the range 1 to 255. If *nnn is not specified, it is assumed to be *1.

The following are examples of IMPLICIT statements:

```
        IMPLICIT INTEGER (I-N)

        IMPLICIT INTEGER (I-Z),REAL(A-G)

        IMPLICIT CHARACTER*100(H)
```

An IMPLICIT statement defines the type and size for all user defined names that begin with the letter or letters that appear in the specification. An IMPLICIT statement applies only to the program unit in which it appears. IMPLICIT statements do not change the type of any intrinsic functions.

Implicit types can be overridden or confirmed for any specific user name by the appearance of that name in a subsequent type statement. An explicit type in a FUNCTION statement also takes priority over an IMPLICIT statement. If the type in question is a character type, the user name′s length is also overridden by a later type definition.

The program unit can have more than one IMPLICIT statement, but all implicit statements must precede all other specification statements in that program unit. The same letter cannot be defined more than once in an IMPLICIT statement in the same program unit.

DIMENSION Statement

A DIMENSION statement is used to specify that a user name is an array. The form of a DIMENSION statement is:

```
     DIMENSION var(dim) [,var(dim)]...
```

where: var(dim) is an array declarator of the form:

var	is the user defined name of the array.
dim	is a dimension declarator.

The following are examples of the DIMENSION statement:

```
        DIMENSION A(100,2),B3(10,4)

        DIMENSION ARRAY(10)

        DIMENSION MATRIX(16,10)

        DIMENSION MAXDIM(4,4,5)
```

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is three. A dimension declarator can be one of three forms:

* An unsigned integer constant.

* A user name corresponding to a non array integer formal argument.

* An asterisk.

A dimension declarator specifies the upper limit of the dimension. The lower limit is always one. If a dimension declarator is an integer constant, then the array has the corresponding number of elements in that dimension. An array has a constant size if all of its dimensions are specified by integer constants. If a dimension declarator is an integer argument, then that dimension is defined to be of a size equal to the initial value of the integer argument upon entry to the subprogram unit at execution time. In such a case the array is called an adjustable sized array.

If the dimension declarator is an asterisk, the array is an assumed sized array and the upper bound of that dimension is not specified. The following program is an example of an asterisk array dimension:

```
      PROGRAM ARR
      DIMENSION RLARR1(10),RLARR2(20)
C TWO ARRAYS OF DIFFERENT SIZE TO PASS TO SUBARR BELOW.
      RLARR1(1)=1.0
      RLARR2(1)=3.14159
C TWO DUMMY VALUES TO BE CLOBBERED BY SUBARR
      CALL SUBARR(RLARR1)
      CALL SUBARR(RLARR2)
```

```
C TWO CALLS OF SUBARR WITH DIFFERENT SIZE ARRAYS
      WRITE(*,100) RLARR1(1), RLARR2(1)
100   FORMAT(2F8.4)
      END
      SUBROUTINE SUBARR(R)
      DIMENSION R(*)
C WHEN AN ACTUAL ARGUMENT IS PASSED TO SUBARR AS R, IT MAY HAVE
C ANY NUMBER OF ELEMENTS.
      R(1)=2.0
C CLOBBER THE FIRST ELEMENT OF R
      END
```

All adjustable and assumed sized arrays must also be formal arguments to the program unit in which they appear. Additionally, an assumed size dimension declarator may only appear as the last dimension in an array declarator.

The order of array elements in memory is column-major order. That is, the left most subscript changes most rapidly in a memory sequential reference to all array elements. Note that this is the opposite of Pascal which has row-major order.

The form of an array element name is:

```
arr(sub [,sub]... )
```

where: arr is the name of an array.

sub is a subscript expression.

A subscript expression is an integer expression used in selecting a specific element of an array. The number of subscript expressions must match the number of dimensions in the array declarator. The value of a subscript expression must be between 1 and the upper bound for the dimension it represents.

The following is an example of an array element name:

MATRIX(2,3) referring to column 3, row 2

Type Statement

Type statements are used to specify the type of user defined names. A type statement may confirm or override the implicit type of a name. Type statements may also specify dimension information.

A user name for a variable, array, external function, or statement function may appear in a type statement. Such an appearance defines the type of that name for the entire program unit. Within a program unit, a name may not have its type explicitly specified by a type statement more than once. A type statement may confirm the type of an intrinsic function, but is not required. The name of a subroutine or main program cannot appear in a type statement.

The form of an INTEGER, REAL, or LOGICAL type statement is:

```
type var [,var]...
```

where: type is INTEGER, REAL, or LOGICAL.

var is a variable name, array name, function name, or an array declarator.

The following are examples of the TYPE statement:

INTEGER MATRIX	Does not include Dimension information
INTEGER MATRIX(16,10)	Includes Dimension information
REAL A	Declares that A holds a REAL value

The form of a CHARACTER type statement is:

```
CHARACTER [*nnn [,]] var [*nnn] [, var [*nnn] ]...
```

where: var is a variable name, array name, or an array declarator.

nnn is the length in number of characters of a character variable or character array element. It must be an unsigned integer in the range 1 to 255.

The following are examples of CHARACTER type statements:

CHARACTER*100,A	A holds up to 100 characters
CHARACTER*50,STRING	Variable name STRING can hold up to 50 characters

The length nnn following the type name CHARACTER is the default length for any name not having its own length specified. If not present, the default length is assumed to be one. A length immediately following a variable or array overrides the default length for that item only. For an array the length specifies the length of each element of that array.

COMMON Statement

The COMMON statement provides a method of sharing storage between two or more program units. Such program units can share the same data without passing it as arguments. The form of the COMMON statement is:

```
COMMON [/ [cname] /] nlist [[,] / [cname] / nlist]...
```

where:	cname	is a common block name. If a cname is omitted, then the blank common block is specified.
	nlist	is a list of variable names, array names, and array declarators separated by commas. Formal argument names and function names cannot appear in a COMMON statement.

The following is an example of the COMMON statement:

COMMON/SHARE/MATRIX,ARRAY,A	Block SHARE can refer to the variable names MATRIX, ARRAY, and A.

In the COMMON statement, all variables and arrays appearing in each nlist following a common block cname are declared to be in that common block. If the first cname is omitted, all elements appearing in the first nlist are specified to be in the blank common block.

Any common block name can appear more than once in COMMON statements in the same program unit. All elements in all nlists for the same common block are allocated storage sequentially in that common storage area in the order that they appear.

All elements in a single common area must be either all of type CHARACTER or none of type character. Furthermore, if two program units reference the same named common containing character data, association of character variables of different length is not allowed. Two variables are said to be associated if they refer to the same actual storage.

The size of a common block is equal to the number of bytes of storage required to hold all elements in that common block. If the same named common block is referenced by several distinct program units, the size must be the same in all program units.

EXTERNAL Statement

An EXTERNAL statement is used to identify a user defined name as an external subroutine or function. The form of an EXTERNAL statement is:

```
EXTERNAL name [,name]...
```

where: name is the name of an external subroutine of function.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure. Statement function names cannot appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement, then that name becomes the name of an external procedure, and the corresponding intrinsic function can no longer be called from that program unit. A user name can only appear once in an EXTERNAL statement.

INTRINSIC Statement

An INTRINSIC statement is used to declare that a user name is an intrinsic function. The form of an INTRINSIC statement is:

```
INTRINSIC name [,name]...
```

where: name is an intrinsic function name.

Each user name may only appear once in an INTRINSIC statement. If a name appears in an INTRINSIC statement, it cannot appear in an EXTERNAL statement. All names used in an INTRINSIC statement must be system-defined INTRINSIC functions. For a list of these functions, see Table 2 in Appendix C. Note that the use of the INTRINSIC statement is optional. The INTRINSIC function can be used without the INTRINSIC statement.

SAVE Statement

A SAVE statement is used to retain the definition of a common block after the return from a procedure that defines that common block. Within a subroutine or function, a common block that has been specified in a SAVE statement does not become undefined upon exit from the subroutine or function. The form of a SAVE statement is:

```
SAVE /name/ [,/name/]...
```

where: name is the name of a common block.

Note: In Apple FORTRAN, all common blocks are statically allocated, so the SAVE statement has no effect and is not normally used.

EQUIVALENCE Statement

An EQUIVALENCE statement is used to specify that two or more variables or arrays are to share the same storage. If the shared variables are of different types, the EQUIVALENCE does not cause any kind of automatic type conversion. The form of an EQUIVALENCE statement is:

```
EQUIVALENCE (nlist) [, (nlist)]...
```

where: nlist is a list of at least two variable names, array names, or array element names separated by commas.

Argument names may not appear in an EQUIVALENCE statement. Subscripts must be integer constants and must be within the bounds of the array they index.

An EQUIVALENCE statement specifies that the storage sequences of the elements that appear in the nlist have the same first storage location. Two or more variables are said to be associated if they refer to the same actual storage. Thus, an EQUIVALENCE statement causes its list of variables to become associated. An element of type character can only be associated with another element of type character with the same length. If an array name appears in an EQUIVALENCE statement, it refers to the first element of the array.

An EQUIVALENCE statement cannot specify that the same storage location is to appear more than once, such as:

```
     REAL R,S(10)
     EQUIVALENCE (R,S(1)),(R,S(5))
```

which forces the variable R to appear in two distinct memory locations.

Furthermore, an EQUIVALENCE statement cannot specify that consecutive array elements are not stored in sequential order. For example:

```
     REAL R(10),S(10)
     EQUIVALENCE (R(1),S(1)),(R(5),S(6))
```

is not allowed.

When EQUIVALENCE statements and COMMON statements are used together, several further restrictions apply. An EQUIVALENCE statement cannot cause storage in two different common blocks to become equivalenced. An EQUIVALENCE statement can extend a common block by adding storage elements following the common block, but not preceding the common block. For example:

```
     COMMON /ABCDE/ R(10)
     REAL S(10)
     EQUIVALENCE (R(1),S(10))
```

is not allowed because it extends the common block by adding storage preceding the start of the block.

DATA STATEMENTS

The DATA statement is used to assign initial values to variables. A DATA statement is a non-executable statement. If present, it must appear after all specification statements and prior to any statement function statements or executable statements. The form of a DATA statement is:

```
     DATA nlist / clist / [[,] nlist / clist /]...
```

where:	nlist	is a list of variable, array element, or array names.
	clist	is a list of constants or constants preceded by an integer constant repeat factor and an asterisk, such as: 5*3.14159 3*'Help' 100*0 A repeat factor followed by a constant is the equivalent of the value of the constant repeated a number of times that is equal to the repeat constant.

There must be the same number of values in each clist as there are variables or array elements in the corresponding nlist. The appearance of an array in an nlist is the equivalent to a list of all elements in that array in storage sequence order. Array elements must be indexed only by constant subscripts.

The type of each non-character element in a clist must be the same as the type of the corresponding variable or array element in the accompanying nlist. Each character element in a clist must correspond to a character variable or array element in the nlist, and must have a length that is less than or equal to the length of that variable or array element. If the length of the constant is shorter, it is extended to the length of the variable by adding blank characters to the right. Note that a single character constant cannot be used to define more than one variable or even more than one array element.

Only local variables and array elements can appear in a DATA statement. Formal arguments, variables in common, and function names cannot be assigned initial values with a DATA statement.

The following are examples of the DATA statement:

```
          DATA X,Y,Z,A,C/1.0,3.8,4.5,6.7,1.9/

          DATA MATRIX/1.5,2.0,2.5,3.0,10.5,3.2/
```

Note in the second example that the array called MATRIX must have 6 elements, one for each DATA constant.

ASSIGNMENT STATEMENTS

An assignment statement is used to assign a value to a variable or an array element. There are two kinds of assignment statements, computational assignment statements and label assignment statements.

Computational Assignment Statement

The form of a computational assignment statement is:

```
var = expr
```

where: var is a variable or array element name.

expr is an expression.

Execution of a computational assignment statement evaluates the expression and assigns the resulting value to the variable or array element appearing on the left. The type of the variable or array element and the expression must be compatible. They must both be either numeric, logical, or character, in which case the assignment statement is called an arithmetic, logical, or character assignment statement.

If the types of the elements of an arithmetic assignment statement are not identical, automatic conversion of the value of the expression to the type of the variable is done. The following table gives the conversion rules:

Type of variable or array element	Type of expression	
	integer	real
integer	expr	INT(expr)
real	REAL(expr)	expr

Table of Type Conversions for Arithmetic Assignment Statements

If the length of the expression does not match the size of the variable in a character assignment statement, it is adjusted so that it does. If the expression is shorter, it is padded with enough blanks on the right to make the sizes equal before the assignment takes place. If the expression is longer, characters on the right are truncated to make the sizes the same.

Label Assignment Statement

The label assignment statement is used to assign the value of a format or statement label to an integer variable. The form of the statement is:

```
ASSIGN label TO var
```

where: label is a format label or statement label.

var is an integer variable.

Execution of an ASSIGN statement sets the integer variable to the value of the label. The label can be either a format label or a statement label, and it must appear in the same program unit as the ASSIGN statement. When used in an assigned GOTO statement, a variable must currently have the value of a statement label. When used as a format specifier in an I/O statement, a variable must have the value of a format statement label. The ASSIGN statement is the only way to assign the value of a label to a variable.

CHAPTER 9

EXPRESSIONS

58 Introduction
58 Arithmetic Expressions
59 Integer Division
59 Type Conversions and Result Types
60 Character Expressions
60 Relational Expressions
61 Logical Expressions
62 Operator Precedence

INTRODUCTION

This chapter describes the four classes of expressions found in the FORTRAN language. They are the arithmetic, the character, the relational, and the logical expression. Note that any variable, array element, or function referenced in an expression must be defined at the time of the reference. Integer variables must be defined with an arithmetic value, rather than a statement label value as set by an ASSIGN statement.

ARITHMETIC EXPRESSIONS

An arithmetic expression produces a value which is either of type integer or type real. The simplest forms of arithmetic expressions are:

* Unsigned integer or real constant.

* Integer or real variable reference.

* Integer or real array element reference.

* Integer or real function reference.

The value of a variable reference or array element reference must be defined for it to appear in an arithmetic expression. Moreover, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value previously set in an ASSIGN statement.

Other arithmetic expressions are built up from the above simple forms using parentheses and these arithmetic operators:

Operator	Representing Operation	Precedence
**	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	
-	Subtraction or Negation	Lowest
+	Addition or Identity	

Table of Arithmetic Operators

All of the operators are binary operators appearing between their arithmetic expression operands. The + and - may also be unary, preceding their operand. Operations of equal precedence are left associative except exponentiation which is right associative. Thus, A / B * C is the same as (A / B) * C and A ** B ** C is the same as A ** (B ** C). Arithmetic expressions can be formed in the usual mathematical sense, as in most programming languages, except that FORTRAN prohibits two operators from appearing consecutively. Thus, A ** -B is prohibited, although A ** (-B) is permissible. Parenthesis may be used in a program to control the order of operator evaluation in an expression.

Certain arithmetic operations are illegal, since they are not mathematically meaningful; such as dividing by zero. Other prohibited operations are raising a zero-valued operand to a zero or negative power and raising a negative-valued operand to a power of type real.

Integer Division

The division of two integers results in a value which is the mathematical quotient of the two values, rounded toward 0. Thus, 7 / 3 evaluates to 2, (-7) / 3 evaluates to -2, 9 / 10 evaluates to 0 and 9 / (-10) evaluates to 0.

Type Conversions and Result Types

Arithmetic expressions may involve operations between operands which are of different types. The general rules for determining type conversions and the result type for an arithmetic expression are:

* An operation between two integers results in an expression of type integer.

* An operation between two reals results in an expression of type real.

* For any operator except **, an operation between a real and an integer converts the integer to type real and performs the operation, resulting in an expression of type real.

* For the operator **, a real raised to an integer power is computed without conversion of the integer, and results in an expression of type real. An integer raised to a real power is converted to type real and the operation results in an expression of type real. Note that for integer I and negative integer J, I ** J is the same as 1 / (I ** IABS(J)) which is subject to the rules of integer division. For example, 2 ** (-4) is 1 / 16 which is 0.

* Unary operators result in the same result type as their operand type.

The type which results from the evaluation of an arithmetic operator is not dependent on the context in which the operation is specified. For example, evaluation of an integer plus a real results in a real even if the value obtained is to be immediately assigned into an integer variable.

CHARACTER EXPRESSIONS

A character expression produces a value which is of type character. The forms of character expressions are:

- Character constant.
- Character variable reference.
- Character array element reference.
- Any character expression enclosed in parenthesis.

There are no operators which result in character expressions.

RELATIONAL EXPRESSIONS

Relational expressions are used to compare the values of two arithmetic expressions or two character expressions. It is not legal in Apple FORTRAN to compare an arithmetic value with a character value. The result of a relational expression is of type logical.

Relational expressions may use any of these operators to compare values:

Operator	Representing Operation
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Table of Relational Operators

All of the operators are binary operators, appearing between their operands. There is no relative precedence or associativity among the relational operands since an expression of the form A .LT. B .NE. C violates the type rules for operands. Relational expressions may only appear within logical expressions.

Relational expressions with arithmetic operands may have an operand of type integer and one of type real. In this case, the integer operand is converted to type real before the relational expression is evaluated.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. An operand is less than another if it appears earlier in the collating sequence, etc. If operands of unequal length are compared, the shorter operand is considered as if it were blank extended to the length of the longer operand.

LOGICAL EXPRESSIONS

A logical expression produces a value which is of type logical. The simplest forms of logical expressions are:

- Logical constant.
- Logical variable reference.
- Logical array element reference.
- Logical function reference.
- Relational expression.

Other logical expressions are built up from the above simple forms using parenthesis and these logical operators:

Operator	Representing Operation	Precedence
.NOT.	Negation	Highest
.AND.	Conjunction	
.OR.	Inclusive disjunction	Lowest

Table of Logical Operators

The .AND. and .OR. operators are binary operators, appearing between their logical expression operands. The .NOT. operator is unary, preceding its operand. Operations of equal precedence are left associative. For example, A .AND. B .AND. C is equivalent to (A .AND. B) .AND. C. As an example of the precedence rules, .NOT. A .OR. B .AND. C is interpreted the same as (.NOT. A) .OR. (B .AND. C). It is not permitted to have two .NOT. operators adjacent to each other, although A .AND. .NOT. B is an example of an allowable expression with two operators being adjacent.

The meaning of the logical operators is their standard mathematical semantics, with .OR. being nonexclusive, that is .TRUE. .OR. .TRUE. evaluates to the value .TRUE..

OPERATOR PRECEDENCE

When arithmetic, relational, and logical operators appear in the same expression, their relative precedences are:

Operator	Precedence
Arithmetic	Highest
Relational	
Logical	Lowest

Table of Operator Precedence

CHAPTER 10

CONTROL STATEMENTS

64 Introduction
64 Unconditional GOTO
64 Computed GOTO
65 Assigned GOTO
65 Arithmetic IF
66 Logical IF
66 Block IF...THEN...ELSE
68 Block IF
69 ELSEIF
69 ELSE
69 ENDIF
70 DO
71 CONTINUE
72 STOP
72 PAUSE
72 END

INTRODUCTION

Control statements are used to control the order of execution of statements in a FORTRAN program. This chapter describes the control statements UNCONDITIONAL GOTO, COMPUTED GOTO, ASSIGNED GOTO, ARITHMETIC IF, LOGICAL IF, BLOCK IF...THEN...ELSE, BLOCK IF, ELSEIF, ELSE, ENDIF, DO, CONTINUE, STOP, PAUSE, and END.

The two remaining statements which control the order of execution of statements are the CALL statement and the RETURN statement, both of which are described in Chapter 13.

UNCONDITIONAL GOTO

The format for an unconditional GOTO statement is:

```
GOTO s
```

where s is a statement label number of an executable statement that is found in the same program unit as the GOTO statement. The effect of executing a GOTO statement is that the next statement executed is the statement labeled s. You are not allowed to GOTO into a DO, IF, ELSEIF, or ELSE block from outside the block.

COMPUTED GOTO

The format for a computed GOTO statement is:

```
GOTO (s [, s] ...) [,] i
```

where i is an integer expression and each s is a statement label of an executable statement that is found in the same program unit as the computed GOTO statement. The same statement label may appear repeatedly in the list of labels. The effect of the computed GOTO statement can be explained as follows: Suppose that there are n labels in the list of labels. If i < 1 or i > n then the computed GOTO statement acts as if it were a CONTINUE statement, otherwise, the next statement executed will be the statement labeled by the ith label in the list of labels. It is illegal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block.

Here is an example:

```
GOTO(1,2,3,4)X
```

If X=2, the program will branch to the line labeled 2 since 2 happens to be the second, or Xth, label in the s list.

ASSIGNED GOTO

The format for an assigned GOTO statement is:

```
GOTO i [[,] (s [, s] ...)]
```

where i is an integer variable name and each s is a statement label of an executable statement that is found in the same program unit as the assigned GOTO statement. The same statement label may appear repeatedly in the list of labels. When the assigned GOTO statement is executed, i must have been assigned the label of an executable statement that is found in the same program unit as the assigned GOTO statement. The effect of the statement is that the next statement executed will be the statement labeled by the label last assigned to i. If the optional list of labels is present, a runtime error is generated if the label last assigned to i is not among those listed. It is illegal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block.

Example:

```
GOTO TARGET
```

If TARGET, which is a variable name in this example, = 100, the statement causes a branch to the statement with the label 100.

ARITHMETIC IF

The format for an arithmetic IF statement is:

```
IF (e) s1, s2, s3
```

where e is an integer or real expression and each of s1, s2, and s3 are statement labels of executable statements found in the same program unit as the arithmetic IF statement. The same statement label may appear more than once among the three labels. The effect of the statement is to evaluate the expression and then select a label based on the value of the expression. Label s1 is selected if the value of e is less than 0, s2 is selected if the value of e equals 0, and s3 is selected if the value of e exceeds 0. The next statement executed will be the statement labeled by the selected label. It is illegal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block.

Example:

```
IF (I) 100,200,300
```

If I evaluates to a negative number, the branch will be to the statement labeled 100. If I is 0, it will be to 200. If I is a positive number, the branch will be to the statement labeled 300.

LOGICAL IF

The format for a logical IF statement is:

```
     IF (e) st
```

where e is a logical expression and st is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement. The statement causes the logical expression to be evaluated and, if the value of that expression is true, then the statement, st, is executed. Should the expression evaluate to false, the statement st is not executed and the execution sequence continues as if a CONTINUE statement had been encountered.

BLOCK IF . . . THEN . . . ELSE

The following sections describe the block IF statement and the various related statements. These statements are new to FORTRAN 77 and can be used to dramatically improve the readability of FORTRAN programs and to cut down the number of GOTOs. As an overview of these sections, the following three code skeletons illustrate the basic concepts:

Skeleton 1 - Simple Block IF which skips a group of statements if the expression is false:

```
      IF(I.LT.10)THEN
        .
          .  Some statements executed only if I.LT.10
          .
      ENDIF
```

Skeleton 2 - Block IF with a series of ELSEIF statements:

```
      IF(J.GT.1000)THEN
        .
          .  Some statements executed only if J.GT.1000
          .
      ELSEIF(J.GT.100)THEN
        .
          .  Some statements executed only if J.GT.100 and J.LE.1000
          .
      ELSEIF(J.GT.10)THEN
        .
          .  Some statements executed only if J.GT.10 and J.LE.1000
          .   and J.LE.100
      ELSE
        .
          .  Some statements executed only if none of above conditions
          .   were true
      ENDIF
```

Skeleton 3 - Illustrates that the constructs can be nested. (Also, an ELSE statement can follow a block IF without intervening ELSEIF statements.) The indentation is solely to enhance readability.

```
      IF(I.LT.100)THEN
        .
          .  Some statements executed only if I.LT.100
          .
              IF(J.LT.10)THEN
                .
                  .  Some statements executed only if I.LT.100 and J.LT.1
                  .
              ENDIF
        .
          .  Some statements executed only if I.LT.100
          .
      ELSEIF(I.LT.1000)THEN
        .
          .  Some statements executed only if I.GE.100 and I.LT.1000
          .
              IF(J.LT.10)THEN
                .
                  .  Some statements executed only if I.GE.100 and I.LT.100
                  .   and J.LT.10
              ENDIF
        .
          .  Some statements executed only if I.GE.100 and I.LT.1000
          .
      ENDIF
```

In order to understand, in detail, the block IF and associated statements, the concept of an IF-level is necessary. For any statement, its IF-level is

```
n1 - n2
```

where n1 is the number of block IF statements from the beginning of the program unit that the statement is in up to and including that statement, and n2 is is the number of ENDIF statements from the beginning of the program unit) up to, but not including, that statement. The IF-level of every statement must be greater than or equal to 0 and the IF-level of every block IF, ELSEIF, ELSE, and ENDIF must be greater than 0. Finally, the IF-level of every END statement must be 0. The IF-level will be used to define the nesting rules for the block IF and associated statements and to define the extent of IF blocks, ELSEIF blocks, and ELSE blocks.

BLOCK IF

The format for a block IF statement is:

```
IF (e) THEN
```

where e is a logical expression. The IF block associated with this block IF statement consists of all of the executable statements, possibly none, that appear following this statement up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this block IF statement. The IF-level defines the notion of matching ELSEIF, ELSE, or ENDIF. The effect of executing the block IF statement is that the expression is evaluated. If it evaluates to true and there is at least one statement in the IF block, the next statement executed is the first statement of the IF block. Following the execution of the last statement in the IF block, the next statement to be executed will be the next ENDIF statement at the same IF-level as this block IF statement. If the expression in this block IF statement evaluates to true and the IF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF level as the block IF statement. If the expression evaluates to false, the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the block IF statement. Note that transfer of control into an IF block from outside that block is not allowed.

ELSEIF

The format of an ELSEIF statement is:

```
ELSEIF (e) THEN
```

where e is a logical expression. The ELSEIF block associated with an ELSEIF statement consists of all of the executable statements, possibly none, that follow the ELSEIF statement up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement. The execution of an ELSEIF statement begins by evaluating the expression. If its value is true and there is at least one statement in the ELSEIF block, the next statement executed is the first statement of the ELSEIF block. Following the execution of the last statement in the ELSEIF block, the next statement to be executed will be the next ENDIF statement at the same IF-level as this ELSEIF statement. If the expression in this ELSEIF statement evaluates to true and the ELSEIF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF level as the ELSEIF statement. If the expression evaluates to false, the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the ELSEIF statement. Note that transfer of control into an ELSEIF block from outside that block is not allowed.

ELSE

The format of an ELSE statement is:

```
ELSE
```

The ELSE block associated with an ELSE statement consists of all of the executable statements, possibly none, that follow the ELSE statement up to, but not including, the next ENDIF statement that has the same IF-level as this ELSE statement. The matching ENDIF statement must appear before any intervening ELSE or ELSEIF statements of the same IF-level. There is no effect in executing an ELSE statement. Note that transfer of control into an ELSE block from outside that block is not allowed.

ENDIF

The format of an ENDIF statement is:

```
ENDIF
```

There is no effect in executing an ENDIF statement. An ENDIF statement is required to match every block IF statement in a program unit in order to specify which statements are in a particular block IF statement.

DO

The format of a DO statement is:

```
     DO s [,] i=e1, e2 [, e3]
```

where s is a statement label of an executable statement. The label must follow this DO statement and be contained in the same program unit. In the DO statement, i is an integer variable, and e1, e2, and e3 are integer expressions. The statement labeled by s is called the terminal statement of the DO loop. It must not be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF, it may contain any executable statement EXCEPT those not permitted inside a logical IF statement.

A DO loop is said to have a range, beginning with the statement which follows the DO statement and ending immediately after the terminal statement of the DO loop. If a DO statement appears in the range of another DO loop, its range must be entirely contained within the range of the enclosing DO loop, although the loops may share a terminal statement. If a DO statement appears within an IF block, ELSEIF block, or ELSE block, the range of the associated DO loop must be entirely contained in the particular block. If a block IF statement appears within the range of a DO loop, its associated ENDIF statement must also appear within the range of that DO loop. The DO variable, i, may not be set by the program within the range of the DO loop associated with it. It is not allowed to jump into the range of a DO loop from outside its range.

The execution of a DO statement causes the following steps to happen in order:

1. The expressions e1, e2, and e3 are evaluated. If e3 is not present, it is as if e3 evaluated to 1; e3 must not evaluate to 0.

2. The DO variable, i, is set to the value of e1.

3. The iteration count for the loop is computed to be
   MAX0(((e2 - e1 + e3)/e3),0)
   which may be zero (Note: unlike FORTRAN 66) if either

   e1 > e2 and e3 > 0
   or
   e1 < e2 and e3 < 0.

4. The iteration count is tested, and if it exceeds zero, the statements in the range of the DO loop are executed.

Following the execution of the terminal statement of a DO loop, the following steps occur in order:

1. The value of the DO variable, i, is incremented by the value of e3 which was computed when the DO statement was executed.

2. The iteration count is decremented by one.

3. The iteration count is tested, and if it exceeds zero, the statements in the range of the DO loop are executed again.

The value of the DO variable is well defined regardless of whether the DO loop exits as a result of the iteration count becoming zero or as a result of a transfer of control out of the DO loop.

Example of final value of DO variable:

```
C This program fragment prints the number 1 to 11 on the CONSOLE:
      DO 200 I=1,10
200   WRITE(*,'(I5)')I
      WRITE(*,'(I5)')I
```

CONTINUE

The format of a CONTINUE statement is:

```
     CONTINUE
```

There is no effect associated with execution of a CONTINUE statement. The primary use for the CONTINUE statement is as a convenient statement to label, particularly as the terminal statement in a DO loop.

Here's an example of the CONTINUE statement in action:

```
C EXAMPLE OF CONTINUE STATEMENT
      DO 200 I=1,10
      WRITE(*,'(I5)')I
200   CONTINUE
      WRITE(*,'(I5)')I
      END
```

Note that CONTINUE simply acts as the terminator statement for the DO loop in the routine.

STOP

The format of a STOP statement is:

```
STOP [n]
```

where n is either a character constant or a string of not more than 5 digits. The effect of executing a STOP statement is to cause the program to terminate. The argument, n, if present, is displayed on the CONSOLE: upon termination.

Example: STOP 'DONE!'

The message DONE! will be displayed on the screen when the program executes the STOP statement.

PAUSE

The format of a PAUSE statement is:

```
PAUSE [n]
```

where n is either a character constant or a string of not more than 5 digits. The effect of executing a PAUSE statement is to cause the program to PAUSE until input is received from the keyboard. Execution will then continue. The contents of n, if present, are displayed as part of the prompt requesting input. When input is received, execution resumes as if a CONTINUE statement had been executed.

END

The format of an END statement is:

```
END
```

Unlike other statements, an END statement must be entered as an initial line. No other FORTRAN statement, such as the ENDIF statement, may have an initial line which APPEARS to be an END statement. The effect of executing the END statement in a subprogram is the same as execution of a RETURN statement and the effect in the main program is to terminate execution of the program. The END statement must appear as the last statement in every program unit.

CHAPTER 11

INPUT/OUTPUT OPERATIONS

74 I/O Overview
74 Records
75 Files
75 Formatted vs. Unformatted Files
75 Sequential vs. Direct Acccess
76 Internal Files
76 Units
77 Choosing a File Structure
79 I/O Limitations
79 I/O Statements
81 OPEN
83 CLOSE
83 READ
84 WRITE
85 BACKSPACE
85 ENDFILE
85 REWIND
85 Notes on I/O Operations

I/O OVERVIEW

Input/output (I/O) statements are all statements that transfer data between the program and any devices attached to your Apple, such as diskette drives, the Apple's keyboard and screen, a printer, and the like. Each device to be used as the source or target of I/O is assigned a unit number. The transfer of data takes place between the variables in your program and the appropriate device number, both of which must be properly specified in the I/O statements that indicate the direction of data transfer. FORMAT statements are used to edit the form of the data to be transferred.

This chapter discusses the FORTRAN I/O system and statements, and gives some general considerations that apply to handling files under the system. The I/O system provided by Apple FORTRAN is a superset of the ANSI Standard subset FORTRAN 77.

In order to fully understand the I/O statements, it is necessary to be familiar with a variety of terms and concepts related to the structure of the FORTRAN I/O system. Most I/O tasks can be accomplished without a complete understanding of this material and the reader is encouraged to use this chapter primarily for reference.

Records

The building block of the FORTRAN file system is the Record. A Record is a sequence of characters or a sequence of values. There are three kinds of records:

* Formatted
* Unformatted
* Endfile

A formatted record is a sequence of characters terminated by the character value 13 which corresponds to the RETURN key on the Apple. Formatted records are interpreted on input in the same way that the operating system and text editor interpret characters. Thus, reading characters from formatted records from FORTRAN is identical to other system programs and other languages on the system.

An unformatted record is a sequence of values, with no system alteration or interpretation. No physical representation for the end of record exists.

The system makes it appear as though an endfile record exists, but no actual record is there.

It should be noted that FORTRAN numbers records starting from 1, but Pascal numbers records from 0.

Files

FORTRAN files are sequences of records. FORTRAN files may be either internal or external.

An external FORTRAN file is a file on a device or a device itself. An internal FORTRAN file is a character variable that serves as the source or destination of some I/O action. From this point on, both FORTRAN files and the notion of a file (as known to the operating system and to the editor) will be referred to simply as files, with the context determining which meaning is intended. The OPEN statement provides the linkage between the two notions of files and, in most cases, the ambiguity disappears since after opening a file, the two notions are one and the same.

A file which is being acted upon by a FORTRAN program has a variety of properties as described below:

* A file may have a name. If present, a name is a character string identical to the name by which it is known to the UCSD file system. There may be more than one name for the same file, such as SYS:A.TEXT and #4:A.TEXT.

* A file has a position property which is usually set by the previous I/O operation. There is a notion of the initial point in the file, the terminal point in the file, the current record, the preceding record, and the next record of the file. It is reasonable to be between records in a file, in which case the next record is the successor to the previous record and there is no current record. The file position after sequential writes is at the end of file, but not beyond the endfile record. Execution of the ENDFILE statement positions the file beyond the endfile record, as does a read statement executed at the end of file (but not beyond the endfile record). Reading an endfile record may be trapped by the user using the END= option in a READ statement. Should the end of file record be detected in this manner, the program can then be directed to branch, or other appropriate action may be taken.

Formatted vs. Unformatted Files

An external file is opened as either formatted or unformatted. All internal files are formatted. Files which are formatted consist entirely of formatted records and files which are unformatted consist entirely of unformatted records. Files which are formatted obey all the structural rules of .TEXT files, so that they are fully compatible with the system editor.

Sequential vs. Direct Access

An external file is opened as either sequential or direct. Sequential files contain records with an order property determined by the order in which the records were written. These files must not be read or written using the REC= option which specifies a position for direct access I/O. The system will attempt to extend sequential access files if a record is written beyond the old terminating boundary of the file, but the success of this depends on the existence of room on the physical device at the end of the file.

Direct access files may be read or written in any order (they are random access files). Records in a direct access file are numbered sequentially, with the first record numbered one. All records in a direct access file have the same length, which is specified at the time the file is opened. Each record in the file is uniquely identified by its record number, which was specified when the record was written. It is entirely possible to write the records out of order, including, for example, writing record 9, 5, and 11 in that order without the records in between. It is not possible to delete a record once written, but it is possible to overwrite a record with a new value.

It is an error to read a record from a direct access file which has not been written, but the system will not detect this error unless the record which is being read is beyond the last record written in the file. Direct access files must reside on blocked peripheral devices such as diskettes, so that it is meaningful to specify a position in the file and reference it. The system will attempt to extend direct access files if an attempt is made to write to a position beyond the previous terminating boundary of the file, but the success of this depends on the existence of room on the physical device.

Internal Files

Internal files provide a mechanism for using the formatting capabilities of the I/O system to convert values to and from their external character representations. That is, reading a character variable converts the character values into numeric, logical, or character values and writing into a character variable allows values to be converted into their external character representation.

An internal file is a character variable or character array element. The file has exactly one record, which has the same length as the character variable or character array element. Should less than the entire record be written by a WRITE statement, the remaining portion of the record is filled with blanks. The file position is always at the beginning of the file prior to I/O statement execution. Only formatted, sequential I/O is permitted to internal files and only the I/O statements READ and WRITE may specify an internal file.

Units

A unit is a means of referring to a file. A unit specified in an I/O statement may be either an external unit specifier or an internal unit specifier.

External unit specifiers are either integer expressions which evaluate to positive values or the character * which stands for the CONSOLE: device. In most cases, external unit specifier values represent physical devices or files resident on those devices by name using the OPEN statement. After the OPEN statement, FORTRAN I/O statements refer to the unit number instead of the name of the external entity. This continues until an explicit CLOSE occurs or until the program terminates. The only exception to the above is that the unit value 0 is initially associated with the CONSOLE: device for reading and writing and no explicit OPEN is necessary.

CHOOSING A FILE STRUCTURE

FORTRAN provides a multitude of possible file structures. Choosing from among these may at first seem somewhat confusing. However, two kinds of files will suffice for most applications.

* An asterisk (*) which specifies the Apple console: This is a sequential, formatted file, also known as unit 0. This particular unit has the special property that an entire line terminated by the return key, must be entered when reading from it, and the various backspace and line delete keys familiar to the system user serve their normal functions. Note that reading from any other unit will not have these properties, even though that unit is bound to the console by an explicit OPEN statement.

* Explicitly opened external, sequential, formatted files: These files are bound to a system file by name in an OPEN statement. They can be read and written in the system text editor format.

Here is an example program which uses the kinds of files discussed in this chapter for reading and for writing. The various I/O statements are explained in detail later in this chapter.

```
C Copy a file with three columns of integers, each 7 columns wide
C from a file whose name is input by the user to another file named
C OUT.TEXT reversing the positions of the first and second column.
      PROGRAM COLSWP
      CHARACTER*23 FNAME
C Prompt to the CONSOLE: by writing to *
      WRITE(*,900)
900   FORMAT('Input file name - '$)
C Read the file name from the CONSOLE: by reading from *
      READ(*,910) FNAME
910   FORMAT(A)
C Use unit 3 for input, any unit number except 0 will do
      OPEN(3,FILE=FNAME)
C Use unit 4 for output, any unit number except 0 and 3 will do
      OPEN(4,FILE='OUT.TEXT',STATUS='NEW')
C Read and write until end of file
100   READ(3,920,END=200)I,J,K
      WRITE(4,920)J,I,K
920   FORMAT(3I7)
      GOTO 100
200   WRITE(*,910)'Done'
      END
```

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of the intended usages for them follows: If the I/O is to be random access, such as in maintaining a database, direct access files are probably necessary. If the data is to be written by FORTRAN and reread by FORTRAN on the same type of system, unformatted files are more efficient both in file space and in I/O overhead. The combination of direct and unformatted is ideal for a database created, maintained, and accessed exclusively by FORTRAN. If the data must be transferred without any system interpretation, especially if all 256 possible bytes will be transferred, unformatted I/O will be necessary, since .TEXT files may contain only the printable character set as data.

A good example of unformatted I/O would be the control of a device which has a single byte, binary interface. Formatted I/O would interpret certain characters, such as the ASCII representation for carriage return, and fail to pass them through to the program unaltered. Internal files are not I/O in the conventional sense but rather provide certain character string operations and conversions.

Use of formatted direct access files requires special caution. FORTRAN formatted files attempt to comply with the operating system rules for .TEXT files. The FORTRAN I/O system is able to enforce these rules for sequential files but it cannot always enforce them for direct access files. Direct access files are not necessarily legal .TEXT files since any unwritten record "holes" contain undefined values which do not follow .TEXT file conventions. Direct files do obey FORTRAN I/O rules.

A file opened in FORTRAN is either old or new. An old file just means one that already exists, while a new one is being used for the first time. There is no concept of opened for reading as distinguished from opened for writing. Therefore, you may open old files and write to them, with the effect of modifying existing files. Similarly, you may alternately write and read to the same file, providing that you avoid reading beyond end of file or trying to read unwritten records in a direct file. A write to a sequential file effectively deletes any records which had existed beyond the freshly written record. Normally, when a device is opened as a file, such as CONSOLE: or PRINTER:, it makes no difference whether the file is opened as old or new. With diskette files, opening a file with STATUS='NEW' creates a new temporary file. If that file is closed using the keep option, or if the program is terminated without doing a CLOSE on that file, a permanent file is created with the name given when the file was opened. If a previous file existed with the same name, it is deleted. If closed using the delete option, the newly created temporary file is deleted, and any previous file of the same name is left intact. Opening a diskette file as old that does not exist, will generate a run-time error. Note that within FORTRAN, it is safer to explicitly CLOSE a file that was OPENed.

I/O LIMITATIONS

Within the FORTRAN I/O system, there are limitations pertaining to direct access files, backspacing, and function calls within I/O statements. These limitations are discussed in this section.

The operating system has two kinds of devices, blocked and sequential. A sequential file may be thought of as a stream of characters, with no explicit motion allowed except reading and/or writing. The CONSOLE: and PRINTER: are examples of sequential devices. Blocked devices, such as diskette files, have the additional operation of seeking a specific location. They can be accessed either sequentially or randomly and thus can support direct files. Since there is no notion of seeking a position on a file which is not blocked, the FORTRAN I/O system does not allow direct file access to sequential devices.

Sequential devices cannot be backspaced meaningfully under the Apple Pascal operating system, so the FORTRAN I/O system disallows backspacing a file on a sequential device.

BACKSPACE may not be used with unformatted sequential files. It was not possible to implement BACKSPACE on unformatted sequential files because there is no indication in the file itself of the record boundaries. It would be possible to append end of record marks to unformatted sequential files, but this would conflict with the idea of an unformatted file being a pure sequence of values. It would also interfere with the most common usage for such files; the direct control of an external instrument. Direct files contain records of fixed and specified length, so it is possible to backspace direct unformatted files. Refer to the BACKSPACE Section in this chapter for more information on the BACKSPACE statement.

There is also a limitation on calling functions within an individual I/O statement. During the course of executing any I/O statement, the evaluation of an expression may cause a function to be called. That function call must not cause any I/O statement to be executed.

I/O STATEMENTS

I/O statements that are available from FORTRAN are as follows: OPEN, CLOSE, READ, WRITE, BACKSPACE, ENDFILE, and REWIND.

In addition, there is an I/O intrinsic function called EOF that returns a logical value indicating whether the file associated with the unit specifier passed to it is at end of file. A familiarity with the FORTRAN file system, units, records, and access methods as described in the previous sections is assumed for the purpose of describing these statements.

The various I/O statements use certain parameters and arguments which specify sources and destinations of data transfer, as well as other facets of the I/O operation. The abbreviations for these are used in the descriptions of the statements and explained below.

The unit specifier, u, can take one of these forms in an I/O statement:

* An asterisk (*) refers to the CONSOLE:.

* An integer expression refers to external file with unit number equal to the value of the expression (* is unit number 0).

* A name of a character variable or character array element refers to the internal file which is the character datum.

The format specifier, f, can take one of these forms in an I/O statement:

* A statement label that refers to the FORMAT statement labeled by that label.

* An integer variable name that refers to the FORMAT label which that integer variable has been assigned to using the ASSIGN statement.

* A character expression that is specified as the current value of the character expression.

The input-output list, or iolist, specifies the entities whose values are transferred by READ and WRITE statements. An iolist is a comma-separated list of items which consist of:

* Input or Output entities

* Implied DO lists

An input entity may be specified in the I/0 list of a READ statement and is of one of these forms:

* Variable name

* Array element name

* Array name which is a means of specifying all of the elements of the array in storage sequence order.

An output entity may be specified in the iolist of a WRITE statement and is of one of these forms:

* Variable name.

* Array element name.

* Array name: This is a means of specifying all of the elements of the array in storage sequence order.

* Any other expression not beginning with the character '(' to distinguish implied DO lists from expressions.

Implied DO lists may be specified as items in the iolist of READ and WRITE statements and are of the form:

```
     (iolist, i = e1, e2 [, e3])
```

where the iolist is as above, including nested implied DO lists, and i, e1, e2 and the optional e3 are as defined for the DO statement. That is, i is an integer variable and e1, e2, and e3 are integer expressions. In a READ statement, the DO variable i or any associated entity must not appear as an input list item in the embedded iolist, but may have been read in the same READ statement outside of the implied DO list. The embedded iolist is effectively repeated for each iteration of i with appropriate substitution of values for the DO variable i.

The following I/O statements are supported in the FORTRAN system. The possible form for each statement is specified first, with an explanation of the meanings for the forms following. Certain items are specified as required if they must appear in the statement and are specified as optional if they need not appear. The defaults are indicated for optional items. All punctuation marks, parentheses, and the like must be entered exactly as shown. Optional parameters and other items are enclosed in brakets [ ]. Commas within brackets are required if the bracketed parameters are used. The single quotes within the brackets are likewise required if the optional parameter is used. Lower case items in the statement descriptions are explained.

OPEN

```
OPEN(u,FILE=fname[,ACCESS=' '][,STATUS=' '][,FORM=' '][,RECL=rl])
```

OPEN(u	Required, must appear as the first argument. Must not be internal unit specifier.
,FILE=fname	The file name, fname, is a character expression. This argument to OPEN is required and must appear as the second argument.

The following arguments are all optional and may appear in any order. The options are character constants with optional trailing blanks except RECL=. Defaults are indicated.

,STATUS='OLD' or	Default, for reading or writing existing files.
'NEW'	For writing new files.
,ACCESS='SEQUENTIAL' or 'DIRECT'	Default.
,FORM='FORMATTED' or 'UNFORMATTED'	Default.
,RECL=rl)	The record length rl is an integer expression. This argument to OPEN is for direct access files only, for which it is required.

The OPEN statement binds a unit number with an external device or file on an external device by specifying its file name. If the file is to be direct, the RECL=rl option specifies the length of the records in that file.

Note that STATUS='OLD' is the default. To open a new file, you must specify STATUS='NEW' in the OPEN statement.

If a file is to be written to a printer, the format of the OPEN statement must be:

OPEN(X,FILE='PRINTER:') where X is an integer.

Example program fragment 1:

```
     C Prompt user for a file name
           WRITE(*,'(A$)') 'Specify output file name - '
     C Presume that FNAME is specified to be CHARACTER*23
     C Read the file name from the CONSOLE:
           READ(*,'(A)') FNAME
     C Open the file as formatted sequential as unit 7, note that the
     C ACCESS specified need not have appeared since it is the default.
           OPEN(7,FILE=FNAME,ACCESS='SEQUENTIAL',STATUS='NEW');
```

Example program fragment 2:

```
     C Open an existing file created by the editor called DATA3.TEXT
     C as unit #3
           OPEN(3,FILE='DATA3.TEXT')
```

CLOSE

CLOSE(u[,STATUS=' '])

CLOSE(u	Required. Must appear as the first argument. Must not be internal unit specifier.
,STATUS='KEEP') ,STATUS='DELETE')	Optional argument which applies only to files opened NEW, default is KEEP. The option is character constant.

CLOSE disconnects the unit specified and prevents subsequent I/O from being directed to that unit unless the same unit number is reopened, possibly bound to a different file or device. Files opened NEW are temporaries and discarded if STATUS='DELETE' is specified. Normal termination of a FORTRAN program automatically closes all open files as if CLOSE with STATUS='KEEP' had been specified. It is generally safer, however, to explicitly CLOSE all files.

Example program fragment:

```
     C Close the file opened in OPEN example, discarding the file
           CLOSE(7,STATUS='DELETE')
```

READ

READ(u[,f][,REC=rn][,END=s])iolist

READ(u	Required, must be first argument.
,f	Required for formatted read as second argument, must not appear for unformatted read.
,REC=rn	For direct access only, otherwise error. Positions to record number rn, where rn is a positive integer expression. If omitted for direct access file, reading continues from the current position in the file.
,END=s)	Optional, statement label. If not present, reading end of file results in a run time error. If present, encountering an end of file condition results in the transfer to the executable statement labeled s which must be in the same program unit as the READ statement.
iolist	See description above. Note closing parentheses follows last of above parameters and immediately precedes the iolist.

The READ statement sets the items in iolist, assuming that no end of file or error condition occurs. If the read is internal, the character variable or character array element specified is the source of the input, otherwise the external unit is the source.

Example program fragment:

```
C Need a two dimensional array for the example
      DIMENSION IA(10,20)
C Read in bounds for array off first line, hopefully less than
C 10 and 20. Then read in the array in nested implied DO lists
C with input format of 8 columns of width 5 each.
      READ(3,990)I,J,((IA(I,J),J=1,J),I=1,I,1)
990   FORMAT(2I5/,(8I5))
```

WRITE

WRITE(u[,f][,REC=rn])iolist

WRITE(u	Required, must be first argument.
,f	Required for formatted write as second argument, must not appear for unformatted write.
,REC=rn	For direct access only, otherwise error. Positions to record number rn, where rn is a positive integer expression. If omitted for direct access file, writing continues at the current position in the file.
)iolist	Note parentheses after last parameter and immediately before iolist.

The WRITE statement transfers the iolist items to the unit specified. If the write is internal, the character variable or character array element specified is the destination of the output, otherwise the external unit is the destination.

Example program fragment:

```
C Place message: "One = 1, Two = 2, Three = 3" on the CONSOLE:
C not doing things in the simplest way!
      WRITE(*,980)'One =',1,1+1,'ee = ',+(1+1+1)
980   FORMAT(A,I2,', Two =',1X,I1,', Thr',A,I1)
```

BACKSPACE

BACKSPACE u	Unit is not internal unit specifier. Only useable with blocked devices that are set up for direct access or are both sequential and formatted.

BACKSPACE causes the file connected to the specified unit to be positioned before the preceding record. The file position is not changed if there is no preceding record. If the preceding record is the endfile record, however, the file becomes positioned before the endfile record.

ENDFILE

ENDFILE u	Unit is not an internal unit specifier.

ENDFILE writes an end of file record as the next record of the file that is connected to the specified unit. The file is then positioned after the end of file record. This prohibits any further sequential data transfer until either a BACKSPACE or REWIND is executed. If an ENDFILE is written on a direct access file, all records written beyond the position of the new end of file disappear.

REWIND

REWIND u	Unit is not an internal unit specifier.

Execution of a REWIND statement causes the file associated with the specified unit to be positioned at its initial point.

NOTES ON I/O OPERATIONS

1. Any function referenced in an expression within an I/O statement cannot cause any I/O statement to be executed.

2. The ANSI Standard subset FORTRAN 77 language includes only formatted sequential files and unformatted direct (random access) files. As in the full ANSI Standard FORTRAN 77, Apple FORTRAN has all combinations of formatted, unformatted, sequential, and direct access files, with these restrictions:

* Unformatted sequential access files do not support the BACKSPACE operation.

* Direct access files must be connected to blocked devices such as disk drives, since only these devices can implement random access.

* The BACKSPACE operation is only supported when using files connected to blocked devices, since it depends on random access to the files.

3. Since the Apple system is interactive, it is sometimes desirable to be able to write or read partial records in formatted READ and WRITE statements. In order to accomplish this, the dollar sign format control inhibits advancing to the next record when the next record is also the last format control executed in a READ or WRITE operation. This allows interactive prompting and reading from the console on the same line of the screen instead of having to prompt on one line and take user input from the next one. This also gives you the ability to read and write formatted files in units smaller than one record. On input, formatted records are almost infinitely extended with blanks (ASCII space character, decimal code 32) to satisfy multiple read operations until the next record is explicitly called for.

For convenient interaction with the console, you will find that several features have been included in the FORTRAN system. ANSI FORTRAN specifies that devices may be preconnected without an OPEN statement, and that one such device may be given the special unit number 0 and may also be referred to with the character *. In Apple FORTRAN, the preconnected unit * is connected to the CONSOLE: device, for reading and writing to the user of the system when a FORTRAN program is executing. Reading from this unit will continue until terminated by a RETURN character (ASCII CR, decimal 13). In addition, this unit supports the backspace key (ASCII DEL, decimal 127) to delete one character at a time as well as the line rubout key (ASCII DLE, decimal 16) to delete the entire line entered since the last RETURN.

The preconnected unit feature in conjunction with the end of record inhibitor $ for writing on the CONSOLE:, the infinite blank extension, and the standard BN format control option allow for very convenient interaction. Here's an example of one-line prompting:

```
      WRITE(*,900)'Input a five digit integer: '
900   FORMAT(A$)
      READ(*,910)I
910   FORMAT(BN,I5)
```

This example prompts without a terminating carriage return after the first WRITE statement and the cursor is left one space beyond the word 'integer:'. If the program user then typed the digits 123 and then pressed RETURN, it would be interpreted as the number 123, since the record is automatically blank-extended two columns to satisfy the I5, and the BN edit control right-justifies the 123 input in the edit field.

Don't confuse the use of the * in FORTRAN I/O statements with the use of the same character by the Pascal operating system to specify the volume name of the system or boot diskette. There is no real ambiguity here, because the context of these two usages is different. The unit * always means the CONSOLE: when it appears in READ and WRITE statements; the unit * will never appear in OPEN statements because it is preconnected. The volume * can and does appear in OPEN statements where it can be a part of a complete file name specification. The * boot diskette volume name will never appear in a READ or WRITE statement because only the unit number associated to it will appear there.

4. To make interactive access to the Pascal operating system file manager possible, the OPEN statement contains a reference to a Pascal operating system filename. The following program fragment prompts for a filename to be used as an input file to be connected to unit 3:

```
      CHARACTER*23 FNAME
      WRITE(*,920)'File name for input:'
920   FORMAT(A$)
      READ(*,930)FNAME
930   FORMAT(A)
      OPEN(3,FILE=FNAME)
```

Unformatted files are ideal for control of I/O devices which require or produce arbitrary bit-patterns. The files are "pure" to FORTRAN, with no end-of-record marks provided or expected by the system, and with no character interpretation done at the system level. Patterns such as those that correspond to characters with values 13 (RETURN on the keyboard) and 16 (DLE), which ordinarily have special meanings when treated as ASCII characters, are passed directly through the I/O system. Thus, instrument control or monitoring applications can be programmed in a straightforward manner. Unformatted I/O is also quite a bit more efficient than the formatted I/O procedures in execution time and in file-space required. Data bases that will be reread by FORTRAN should usually be written as unformatted files.

CHAPTER 12

FORMATTED I/O

90 Introduction
90 Formatting I/O
91 Formatting and the I/O List
92 Nonrepeatable Edit Descriptors
92 Apostrophe Editing
93 H Hollerith Editing
93 X Positional Editing
93 / Slash Editing
93 $ Dollar Sign Editing
94 P Scale Factor Editing
94 BN/BZ Blank Interpretation
94 Repeatable Edit Descriptors
95 I Integer Editing
95 F Real Editing
95 E Real Editing
96 L Logical Editing
96 A Character Editing

INTRODUCTION

This chapter describes formatted I/O and the FORMAT statement. Some familiarity with the FORTRAN file system, units, records, access methods, and I/O statements as described in the previous chapter is assumed.

FORMATTING I/O

If a READ or WRITE statement specifies a format, in parentheses immediately following the READ or WRITE statement, it is considered a formatted, rather than an unformatted I/O statement. Such a format may be specified in one of three ways, as explained in the previous chapter. Two ways refer to FORMAT statements and one is an immediate format in the form of a character expression containing the format itself. The following are all valid and equivalent means of specifying a format:

```
      WRITE(*,990)I,J,K
990   FORMAT(2I5,I3)

      ASSIGN 990 TO IFMT
990   FORMAT(2I5,I3)
      WRITE(*,IFMT)I,J,K

      WRITE(*,'(2I5,I3)')I,J,K

      CHARACTER*8 FMTCH
      FMTCH = '(2I5,I3)'
      WRITE(*,FMTCH)I,J,K
```

The format specification itself must begin with a left or opening parenthesis, possibly following initial blank characters. It must end with a matching closing or right parenthesis. Characters beyond the closing parenthesis are ignored.

FORMAT statements must be labeled, and like all nonexecutable statements, may not be the target of a branching operation.

Between the initial and terminating parentheses is a list of items, separated by commas, each of which is one of these:

[r] ed - repeatable edit descriptors

ned - nonrepeatable edit descripors

[r] fs - a nested format specification. At most 3 levels of nested parentheses are permitted within the outermost level.

where r is an optional, nonzero, unsigned, integer constant called a repeat specification. The comma separating two list items may be omitted if the resulting format specification is still unambiguous, such as after a P edit descriptor or before or after the / edit descriptor.

The repeatable edit descriptors, explained in detail below, are:

```
Iw
Fw.d
Ew.d
Ew.dEe
Lw
A
Aw
```

where I, F, E, L, and A indicate the manner of editing and,

w and e are nonzero, unsigned, integer constants, and
d is an unsigned integer constant.

The nonrepeatable edit descriptors, also explained in detail below, are:

```
'xxxx' - character constants of any length
nHxxxx - another means of specifying character constants
nX
/
$
kP
BN
BZ
```

where apostrophe, H, X, slash, dollar sign, P, BN, and BZ indicate the manner of editing and,

x is any ASCII character,
n is a nonzero, unsigned, integer constant, and
k is an optionally signed integer constant.

FORMATTING AND THE I/O LIST

Before describing in greater detail the manner of editing specified by each of the above edit descriptors, it should be understood how the format specification interacts with the input/output list (iolist) in a given READ or WRITE statement.

If an iolist contains one or more items, at least one repeatable edit descriptor is required in the format specification. In particular, the empty edit specification, (), may be used only if no items are specified in the iolist, in which case the only action caused by the I/O statement is the implicit record skipping action associated with formats. Each item in the iolist will become associated with a repeatable edit descriptor during the I/O statement execution. In contrast, the remaining format control items interact directly with the record and do not become associated with items in the iolist.

The items in a format specification are interpreted from left to right. Repeatable edit descriptors act as if they were present r times; omitted r is treated as a repeat factor of 1. Similarly, a nested format specification is treated as if its items appeared r times.

The formatted I/O process proceeds as follows: The format controller scans the format items in the order indicated above. When a repeatable edit descriptor is encountered, either:

* A corresponding item appears in the iolist in which case the item and the edit descriptor become associated and I/O of that item proceeds under format control of the edit descriptor, or

* The format controller terminates I/O.

If the format controller encounters the matching right parentheses of the format specification and there are no further items in the iolist, the format controller terminates I/O. If, however, there are further items in the iolist, the file is positioned at the beginning of the next record and the format controller continues by rescanning the format starting at the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, the format controller will rescan the format from the beginning. Within the portion of the format rescanned, there must be at least one repeatable edit descriptor. Should the rescan of the format specification begin with a repeated nested format specification, the repeat factor is used to indicate the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or BN or BZ blank control in effect. When the format controller terminates, the remaining characters of an input record are skipped or an end of record is written on output, except as noted under the $ edit descriptor.

NONREPEATABLE EDIT DESCRIPTORS

Here are the detailed explanations of the various format specification descriptors, beginning with the nonrepeatable edit descriptors.

Apostrophe Editing

The apostrophe edit descriptor has the form of a character constant. Embedded blanks are significant and double '' are interpreted as a single '. Apostrophe editing may not be used for input (READ). It causes the character constant to be transmitted to the output unit.

H Hollerith Editing

The nH edit descriptor causes the following n characters, including blanks, to be transmitted to the output. Hollerith editing may not to be used for input (READ).

Examples of Apostrophe and Hollerith editing:

```
C Each write outputs characters between the slashes: /ABC'DEF/
      WRITE(*,970)
970   FORMAT('ABC''DEF')
      WRITE(*,'(''ABC''''DEF'')')
      WRITE(*,'(7HABC''DEF)')
      WRITE(*,960)
960   FORMAT(7HABC'DEF)
```

X Positional Editing

On input (READ), the nX edit descriptor causes the file position to advance over n characters, thus the next n characters are skipped. On output (WRITE), the nX edit descriptor causes n blanks to be written, providing that further writing to the record occurs, otherwise, the nX descriptor results in no operation.

/ Slash Editing

The slash indicates the end of data transfer on the current record. On input, the file is positioned to the beginning of the next record. On output, an end of record is written and the file is positioned to write on the beginning of the next record.

$ Dollar Sign Editing

Normally when the format controller terminates, the end of data transmission on the current record occurs. If the last edit descriptor encountered by the format controller is the dollar sign, this automatic end of record is inhibited. This allows subsequent I/O statements to continue reading or writing out of or into the same record. The most common use for this mechanism is to prompt the user to respond on the keyboard, and to READ a response off the same line as in:

```
      WRITE(*,'(A$)') 'Input an integer -> '
      READ(*,'(BN,I6)') I
```

The dollar sign edit descriptor does not inhibit the automatic end of record generated when reading from the * unit. Input from the CONSOLE: must always be terminated by the return key. This permits the backspace character and the line delete key to function properly.

P Scale Factor Editing

The kP edit descriptor is used to set the scale factor for subsequent F and E edit descriptors until another kP edit descriptor is encountered. At the start of each I/O statement, the scale factor begins at 0. The scale factor effects format editing in the following ways:

* On input, with F and E editing, providing that no explicit exponent exists in the field, and F with output editing, the externally represented number equals the internally represented number multiplied by 10**k.

* On input, with F and E editing, the scale factor has no effect if there is an explicit exponent in the input field.

* On output, with E editing, the real part of the quantity is output multiplied by 10**k and the exponent is reduced by k, effectively altering the column position of the decimal point but not the value output.

BN/BZ Blank Interpretation

These edit descriptors specify the interpretation of blanks in numeric input fields. The default, BZ, is set at the start of each I/O statement. This makes blanks, other than leading blanks, identical to zeros. If a BN edit descriptor is processed by the format controller, blanks in subsequent input fields will be ignored until a BZ edit descriptor is processed. The effect of ignoring blanks is to take all the non-blank characters in the input field, and treat them as if they were right-justified in the field with the number of leading blanks equal to the number of ignored blanks. For instance, the following READ statement accepts the characters shown between the slashes as the value 123 where <cr> indicates hitting the return key:

```
      READ(*,100) I
100   FORMAT(BN,I6)
```

```
     /123      <cr>/,
     /123   456<cr>/,
     /123<cr>/, or
     /   123<cr>/.
```

The BN edit descriptor, in conjunction with infinite blank padding at the end of formatted records, makes interactive input very convenient.

REPEATABLE EDIT DESCRIPTORS

The I, F, and E edit descriptors are used for I/O of integer and real data. The following general rules apply to all three of them:

* On input, leading blanks are not significant. Other blanks are interpreted differently depending on the BN or BZ flag in effect, but all blank fields always become the value 0. Plus signs are optional.

* On input, with F and E editing, an explicit decimal point appearing in the input field overrides the edit descriptor specification of the decimal point position.

* On output, the characters generated are right justified in the field with padding leading blanks if necessary.

* On output, if the number of characters produced exceeds the field width or the exponent exceeds its specified width, the entire field is filled with asterisks.

I Integer Editing

The edit descriptor Iw must be associated with an iolist item which is of type integer. The field width is w characters in length. On input, an optional sign may appear in the field. The general rules of numeric editing apply to the I edit descriptor.

F Real Editing

The edit descriptor Fw.d must be associated with an iolist item which is of type real. The width of the field is w positions, the fractional part of which consists of d digits. The input field begins with an optional sign followed by a string of digits optionally containing a decimal point. If the decimal point is present, it overrides the d specified in the edit descriptor, otherwise the rightmost d digits of the string are interpreted as following the decimal point. Leading blanks are converted to zeros if necessary. Following this is an optional exponent which is one of these:

* Plus or minus followed by an integer.

* E or D followed by zero or more blanks followed by an optional sign followed by an integer. E and D are treated identically.

The output field occupies w digits, d of which fall beyond the decimal point and the value output is controlled both by the iolist item and the current scale factor. The output value is rounded rather than truncated.

The general rules of numeric editing apply to the F edit descriptor.

E Real Editing

An E edit descriptor either takes the form Ew.d or Ew.dEe. In either case the field width is w characters. The e has no effect on input. The input field for an E edit descriptor is identical to that described by an F edit descriptor with the same w and d. The form of the output field depends on the scale factor set by the P edit descriptor that is in effect. For a scale factor of 0, the output field is a minus sign if necessary, followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent, exp, of one of the following forms:

Ew.d	-99 <= exp <= 99	E followed by plus or minus followed by the two digit exponent.
Ew.dEe	-((10**e) - 1) <= exp <= (10**e) -1	E followed by plus or minus followed by e digits which are the exponent with possible leading zeros.

The form Ew.d must not be used if the absolute value of the exponent to be printed exceeds 999.

The scale factor controls the decimal normalization of the printed E field. If the scale factor, k, is in the range -d < k <= 0 then the output field contains exactly -k leading zeros after the decimal point and d + k significant digits after this. If 0 < k < d+2 then the output field contains exactly k significant digits to the left of the decimal point and d - k - 1 places after the decimal point. Other values of k are errors.

The general rules of numeric editing apply to the E edit descriptor.

L Logical Editing

The edit descriptor is Lw, indicating that the field width is w characters. The iolist element which becomes associated with an L edit descriptor must be of type logical. On input, the field consists of optional blanks, followed by an optional decimal point, followed by T (for .TRUE.) or F (for .FALSE.). Any further characters in the field are ignored, but accepted on input, so that .TRUE. and .FALSE. are valid inputs. On output, w - 1 blanks are followed by either T or F as appropriate.

A Character Editing

The forms of the edit descriptor are A or Aw, in which the former acquires an implied field width, w, from the number of characters in the iolist item with which it becomes associated. The iolist item must be of the character type if it is to be associated with an A or Aw edit descriptor. On input, if w exceeds or equals the number of characters in the iolist element, the rightmost characters of the input field are used as the input characters, otherwise the input characters are left justified in the input iolist item and trailing blanks are provided. On output, if w should exceed the characters produced by the iolist item, leading blanks are provided, otherwise, the leftmost w characters of the iolist item are output.

CHAPTER 13

PROGRAM UNITS

98 Introduction
98 Main Programs
98 Subroutines
98 SUBROUTINE Statement
99 CALL Statement
100 Functions
100 External Functions
101 Intrinsic Functions
102 Table of Intrinsic Functions
105 Statement Functions
106 The RETURN Statement
106 Parameters

INTRODUCTION

This chapter describes the format of program units. A program unit is either a main program, a subroutine, or a function program unit. The term procedure is used to refer to either a function or a subroutine. This chapter also describes the CALL and RETURN statements as well as function calls.

MAIN PROGRAMS

A main program is any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. It may have a PROGRAM statement as its first statement. The execution of a FORTRAN program always begins with the first executable statement in the main program. Consequently, there must be one and only one main program in every executable program. The form of a PROGRAM statement is:

```
     PROGRAM pname
```

where: pname is a user defined name that is the name of the main program.

The name, pname, is a global name. Therefore, it cannot be the same as another external procedure's name or a common block's name. It is also a local name to the main program, and must not conflict with any other local name. The PROGRAM statement may only appear as the first statement of a main program.

SUBROUTINES

A subroutine is a program unit that can be called from other program units by a CALL statement. When called, it performs the set of actions defined by its executable statements, and then returns control to the statement immediately following the statement that called it. A subroutine does not directly return a value, although values can be passed back to the calling program unit via parameters or common variables.

SUBROUTINE Statement

A subroutine begins with a SUBROUTINE statement and ends with the first following END statement. It may contain any kind of statement other than a PROGRAM statement or a FUNCTION statement. The form of a SUBROUTINE statement is:

```
     SUBROUTINE sname [( [farg [, farg]...] )]
```

where: sname is the user defined name of the subroutine.

farg is a user defined name of a formal argument.

The name, sname, is a global name, but it is also local to the subroutine it names. The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

CALL Statement

A subroutine is executed as a consequence of executing a CALL statement in another program unit that references that subroutine. The form of a CALL statement is:

```
     CALL sname [( [arg [,arg]... ] )]
```

where: sname is the name of a subroutine.

arg is an actual argument.

An actual argument may be either an expression or the name of an array. The actual arguments in the CALL statement must agree in type and number with the corresponding formal arguments specified in the SUBROUTINE statement of the referenced subroutine. If there are no arguments in the SUBROUTINE statement, then a CALL statement referencing that subroutine must not have any actual arguments, but may optionally have a pair of parentheses following the name of the subroutine. Note that a formal argument may be used as an actual argument in another subprogram call.

Execution of a CALL statement proceeds as follows: All arguments that are expressions are evaluated. All actual arguments are associated with their corresponding formal arguments, and the body of the specified subroutine is executed. Control is returned to the statement following the CALL statement upon exiting the subroutine, by executing either a RETURN statement or an END statement in that subroutine.

A subroutine specified in any program unit may be called from any other program unit within the same executable program. Recursive subroutine calls, however, are not allowed in FORTRAN. That is, a subroutine cannot call itself directly, nor can it call another subroutine that will result in that subroutine being called again before it returns control to its caller.

FUNCTIONS

A function is referenced in an expression and returns a value that is used in the computation of that expression. There are three kinds of functions: external functions, intrinsic functions, and statement functions. This section describes the three kinds of functions.

A function reference may appear in an arithmetic expression. Execution of a function reference causes the function to be evaluated, and the resulting value is used as an operand in the containing expression. The form of a function reference is:

```
     fname ( [arg [,arg]...] )
```

where: fname is the name of an external, intrinsic, or statement function.

arg is an actual argument.

An actual argument may be an arithmetic expression or an array. The number of actual arguments must be the same as in the definition of the function, and the corresponding types must agree.

External Functions

An external function is specified by a function program unit. It begins with a FUNCTION statement and ends with an END statement. It may contain any kind of statement other that a PROGRAM statement, a FUNCTION statement, or a SUBROUTINE statement. The form of a FUNCTION statement is:

```
     [type] FUNCTION fname ( [farg [, farg]...] )
```

where: type is one of INTEGER, REAL, or LOGICAL.

fname is the user defined name of the function.

farg is a formal argument name.

The name, fname, is a global name, and it is also local to the function it names. If no type is present in the FUNCTION statement, the function's type is determined by default and any subsequent IMPLICIT or type statements that would·determine the type of an ordinary variable. If a type is present, then the function name cannot appear in any additional type statements. In any event, an external function cannot be of type character. The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Neither argument names nor fname can appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The function name must appear as a variable in the program unit defining the function. Every execution of that function must assign a value to that variable. The final value of this variable, upon execution of a RETURN or an END statement, defines the value of the function. After being defined, the value of this variable can be referenced in an expression, exactly like any other variable. An external function may return values in addition to the value of the function by assignment to one or more of its formal arguments.

Intrinsic Functions

Intrinsic functions are functions that are predefined by the FORTRAN compiler and are available for use in a FORTRAN program. The table following this section gives the name, definition, number of parameters, and type of the intrinsic functions available in Apple FORTRAN 77. An IMPLICIT statement does not alter the type of an intrinsic function. For those intrinsic functions that allow several types of arguments, all arguments in a single reference must be of the same type.

An intrinsic function name may appear in an INTRINSIC statement, but only those intrinsic functions listed in the table may do so. An intrinsic function name also may appear in a TYPE statement, but only if the type is the same as the standard type of that intrinsic function.

Arguments to certain intrinsic functions are limited by the definition of the function being computed. For example, the logarithm of a negative number is mathematically undefined, and therefore not allowed.

TABLE OF INTRINSIC FUNCTIONS

Intrinsic Function	Definition	No. Args	Name	Type of	
				Argument	Function
Type Conversion	Conversion to Integer int(a) See Note 1	1	INT IFIX	Real Real	Integer Integer
	Conversion to Real See Note 2	1	REAL FLOAT	Integer Integer	Real Real
	Conversion to Integer See Note 3	1	ICHAR	Character	Integer
	Conversion to Character	1	CHAR	Integer	Character
Truncation	int(a) See Note 1	1	AINT	Real	Real
Nearest Whole Number	int(a+.5) a>=0 int(a-.5) a<0	1	ANINT	Real	Real
Nearest Integer	int(a+.5) a>=0 int(a-.5) a<0	1	NINT	Real	Integer
Absolute Value	|a|	1 1	IABS ABS	Integer Real	Integer Real
Remaindering	a1-int(a1/a2)*a2 See Note 1	2	MOD AMOD	Integer Real	Integer Real
Transfer of Sign	|a1| if a2>=0 -|a1| if a2<0	2	ISIGN SIGN	Integer Real	Integer Real
Positive Difference	a1-a2 if a1>a2 0 if a1<=a2	2	IDIM DIM	Integer Real	Integer Real

TABLE OF INTRINSIC FUNCTIONS - Continued

Intrinsic Function	Definition	No. Args.	Name	Type of	
				Argument	Function
Choosing Largest Value	max(a1,a2,...)	>=2	MAX0 AMAX1	Integer Real	Integer Real
			AMAX0 MAX1	Integer Real	Real Integer
Choosing Smallest Value	min(a1,a2,...)	>=2	MIN0 AMIN1	Integer Real	Integer Real
			AMIN0 MIN1	Integer Real	Real Integer
Square Root	a**0.5	1	SQRT	Real	Real
Exponential	e**a	1	EXP	Real	Real
Natural Logarithm	log(a)	1	ALOG	Real	Real
Common Logarithm	log10(a)	1	ALOG10	Real	Real
Sine	sin(a)	1	SIN	Real	Real
Cosine	cos(a)	1	COS	Real	Real
Tangent	tan(a)	1	TAN	Real	Real
Arcsine	arcsin(a)	1	ASIN	Real	Real
Arccosine	arccos(a)	1	ACOS	Real	Real
Arctangent	arctan(a)	1	ATAN	Real	Real
	arctan(a1/a2)	2	ATAN2	Real	Real
Hyperbolic Sine	sinh(a)	1	SINH	Real	Real
Hyperbolic Cosine	cosh(a)	1	COSH	Real	Real

TABLE OF INTRINSIC FUNCTIONS - Continued

Intrinsic Function	Definition	No. Args.	Name	Type of	
				Argument	Function
Hyperbolic Tangent	tanh(a)	1	TANH	Real	Real
Lexically Greater Than or Equal	a1 >= a2 See Note 4	2	LGE	Character	Logical
Lexically Greater Than	a1 > a2 See Note 4	2	LGT	Character	Logical
Lexically Less Than or Equal	a1 <= a2 See Note 4	2	LLE	Character	Logical
Lexically Less Than	a1 < a2 See Note 4	2	LLT	Character	Logical
End of File	End_Of_File(a) See Note 5	1	EOF	Integer	Logical

The number of each of the notes that follow refers to the number in column 2 of the Table.

(1) For a of type real, if a >= 0 then int(a) is the largest integer not greater than a, if a < 0 then int(a) in the most negative integer not less than a. IFIX(a) is the same as INT(a).

(2) For a of type integer, REAL(a) is as much precision of the significant part of a as a real value can contain. FLOAT(a) is the same as REAL(a).

(3) ICHAR converts a character value into an integer value. The integer value of a character is the ASCII internal representation of that character, and is in the range 0 to 127. For any two characters, c1 and c2, (c1 .LE. c1) is true if and only if (ICHAR(c1) .LE. ICHAR(c2)) is true.

(4) LGE(a1,a2) returns the value true if a1 = a2 or if a1 follows a2 in the ASCII collating sequence; otherwise, it returns false.

LGT(a1,a2) returns true if a1 follows a2 in the ASCII collating sequence; otherwise, it returns false.

LLE(a1,a2) returns true if a1 = a2 or if a1 precedes a2 in the ASCII collating sequence; otherwise, it returns false.

LLT(a1,a2) returns true if a1 precedes a2 in the ASCII collating sequence; otherwise, it returns false.

The operands or LGE, LGT, LLE, and LLT must be of the same length.

(5) EOF(a) returns the value true if the unit specified by its argument is at or past the end of file record, otherwise it returns false. The value of a must correspond to an open file, or to zero which indicates the CONSOLE: device.

(6) All angles are expressed in radians.

(7) All arguments in an intrinsic function reference must be of the same type.

Statement Functions

A statement function is a function that is defined by a single statement. It is similar in form to an assignment statement. A statement function statement can only appear after the specification statements and before any executable statements in the program unit in which it appears. A statement function is not an executable statement; since it is not executed in order as the first statement in its particular program unit. Rather, the body of a statement function serves to define the meaning of the statement function. It is executed, as any other function, by the execution of a function reference. The form of a statement function is:

```
    fname ( [arg [, arg]...] ) = expr
```

where

fname is the name of the statement function.
arg is a formal argument name.
expr is an expression.

The type of the expr must be assignment compatible with the type of the statement function name. The list of formal argument names serves to define the number and type of arguments to the statement function. The scope of formal argument names is the statement function. Therefore, formal argument names may be used as other user defined names in the rest of the program unit containing the statement function definition. The name of the statement function, however, is local to its program unit, and must not be used otherwise, except as the name of a common block, or as the name of a formal argument to another statement function. The type of all such uses, however, must be the same. If a formal argument name is the same as another local name, then a reference to that name within the statement function defining it always refers to the formal argument, never to the other usage.

Within the expression expr, references to variables, formal arguments, other functions, array elements, and constants are allowed. Statement function references, however, must refer to statement functions that have been defined prior to the statement function in which they appear. Statement functions cannot be recursively called, either directly or indirectly.

A statement function can only be referenced in the program unit in which it is defined. The name of a statement function cannot appear in any specification statement, except in a type statement which may not define that name as an array, and in a COMMON statement as the name of a common block. A statement function cannot be of type character.

The RETURN Statement

A RETURN statement causes return of control to the calling program unit. It may only appear in a function or subroutine. The form of a RETURN statement is:

```
     RETURN
```

Execution of a RETURN statement terminates the execution of the enclosing subroutine or function. If the RETURN statement is in a function, then the value of that function is equal to the current value of the variable with the same name as the function. Execution of an END statement in a function or subroutine is treated in exactly the same way as is execution of a RETURN statement.

PARAMETERS

This section discusses the relationship between formal and actual arguments in a function or subroutine call. A formal argument refers to the name by which the argument is known within the function or subroutine, and an actual argument is the specific variable, expression, array, and so forth, passed to the procedure in question at any specific calling location.

Arguments are used to pass values into and out of procedures. Variables in common can be used to perform this task as well. The number of actual arguments must be the same as formal arguments, and the corresponding types must agree.

Upon entry to a subroutine or function, the actual arguments become associated with the formal arguments, much as an EQUIVALENCE statement associates two or more arrays or variables, and COMMON statements in two or more program units associate lists of variables. This association remains in effect until execution of the subroutine or function is terminated. Thus, assigning a value to a formal argument during execution of a subroutine or function may alter the value of the corresponding actual argument. If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the corresponding formal argument is not allowed, and may have some strange side effects.

If an actual argument is an expression, it is evaluated immediately prior to the association of formal and actual arguments. If an actual argument is an array element, its subscript expression is evaluated just prior to the association, and remains constant throughout the execution of the procedure, even if it contains variables that are redefined during the execution of the procedure.

A formal argument that is a variable may be associated with an actual argument that is a variable, an array element, or an expression. A formal argument that is expressed as an array may be associated with an actual argument that is an array or an array element. The number and size of dimensions in a formal argument may be different than those of the actual argument, but any reference to the formal array must be within the limits of the storage sequence in the actual array. While a reference to an element outside these bounds is not detected as an error in a running FORTRAN program, the results are unpredictable.

CHAPTER 14

COMPILATION UNITS

110 Introduction
110 Units, Segments, Partial Compilation
111 Linking
112 $USES Compiler Directive
113 Separate Compilation
113 FORTRAN Overlays

INTRODUCTION

This chapter describes the relationship between FORTRAN and the Apple Pascal segment mechanism. In normal use, the user need not be aware of such intricacies. However, if you want to interface FORTRAN with Pascal, to create overlays, or to take advantage of separate compilation or libraries, the details contained here are helpful.

The first section of this chapter discusses the general form of a FORTRAN program in terms of the operating system object code structure. The next section deals with linking FORTRAN programs. The third describes the $USES compiler directive. This directive provides access to libraries or already compiled procedures, and provides overlays in FORTRAN. The next section describes how you link FORTRAN with Pascal.

UNITS, SEGMENTS, PARTIAL COMPILATION

When a full FORTRAN program is compiled in one piece, the .CODE file created contains 2 distinct segments and a reference to a third. Unit number 1, called MAINSEGX, contains code to manage all other segments, defines named common blocks, initializes the run time system, etc. No actual user code resides in the segment MAINSEGX. It must, however, remain as a distinct unit in order for the linker to properly define named common data areas and file support for the run time system. Unit number 7, given the name of the main program, contains all of the user code. A reference to unit number 8, RTUNIT, is also contained in the .CODE file. This is the FORTRAN run-time system.

If a FORTRAN compilation contains no main program, then it is output as if it were a Pascal unit compilation. The unit is given the name U, followed by the name of its first procedure. For example:

```
C --- No PROGRAM statement present
      SUBROUTINE X
      ...
      END
      SUBROUTINE Y
      ...
      END
      ...
      SUBROUTINE Z
      ...
      END
```

would be compiled into a single unit named UX. Assume for later examples that the object code is output to file X.CODE. All procedures called from within unit UX must be defined within unit UX, unless a $USES or a $EXT statement has shown them to reside in another unit.

Similarly, procedures in unit UX cannot be called from other units unless the other units contain a $USES UX statement. Thus, a typical main program that would call X might be:

```
C
C --- This is the main program BIGGIE
C
$USES UX IN X.CODE
      PROGRAM BIGGIE
      ...
      CALL X
      ...
      END
      SUBROUTINE W
      ...
      CALL Y
      ...
      END
```

If the $USES statement were not present, the FORTRAN compiler would expect subroutines X and Y to appear in the same compilation, somewhere after subroutine W. Assume that the object code for this compilation is output to the file BIGGIE.CODE.

Thus, the user can create libraries of functions and partial compilations, and save compilation time and disk space, by a simple use of the $USES statement. More inforation on the $USES statement, will be found later on in this chapter.

LINKING

Since the FORTRAN run time library must be linked into any user program in order that it may be executed, you must always L(ink a program before it can be executed. Normally, you specify the file containing the main program as the 'host file' and the SYSTEM.LIBRARY as one of the 'lib file' entries. In addition, you must specify the files containing any user defined units referenced via the $USES statement as 'lib file' entries. Thus, to link the program BIGGIE you would run the linker by using the L(ink command, and respond as shown below.

```
Linker II.1 [A4]
Host file? BIGGIE                  User inputs the name of the file
Opening BIGGIE.CODE                    containing the main program.
Lib file? SYSTEM.LIBRARY           File containing run time library.
Opening SYSTEM.LIBRARY
Lib file? X                        File containing user defined unit.
Opening X.CODE
Lib file?
Map name?
Reading MAINSEGX
Reading BIGGIE
Reading RTUNIT
Reading UX
Output file? BIG.CODE              File for linked object code
Linking BIGGIE   # 7
Linking RTUNIT   # 8
Linking UX       # 9
Linking MAINSEGX # 1
```

You could then eX(ecute the code file called BIG.CODE.

$USES COMPILER DIRECTIVE

The $USES compiler directive provides several distinct functions. It allows procedures and functions in separately compiled units, such as the system library, to be called from FORTRAN. It provides a relatively secure form of separate compilation for FORTRAN programs. It allows us to call Pascal routines that have been compiled into Pascal units. It also provides an overlay mechanism to the FORTRAN user that is somewhat more general than that provided in the Pascal language.

The format of the $USES control statement is:

```
$USES unitname [ IN filename ] [ OVERLAY ]
```

where
unitname is the name of a unit.
filename is a valid file name.

As with all $ control statements, the $ must appear in column one. This compiler directive directs the compiler to open the .CODE file filename, or the SYSTEM.LIBRARY if the filename is absent, locate the unit unitname, and process the INTERFACE information associated with that unit, generating a reasonable FORTRAN equivalent declaration for the FORTRAN compilation in progress. There cannot be any global variables in the INTERFACE portion of a Pascal unit. All $USES commands must appear before any FORTRAN statements, specification or executable, but they are allowed to follow comment lines and other $ control lines. If the optional 'IN filename' is present, the name filename is used as the file to process. If it is not, the file *SYSTEM.LIBRARY is used as a default. If the optional field OVERLAY is present, the unit in question is treated as an overlay. It is only present in memory when one of its procedures is active. If the OVERLAY field is not present, the unit is loaded into memory before the user program is executed, and remains there until execution is over.

Warning: If a FORTRAN main program $USES a Pascal unit, that Pascal unit cannot have any global variables in the INTERFACE part of its declarations.

SEPARATE COMPILATION

Separate compilation is accomplished by compiling a set of subroutines and functions without any main program. Each such compilation creates a code file containing a single unit. Then, when the main program is compiled, possibly along with many subroutines or functions, it $USES the separately compiled units. The routines compiled with the main program obtain the correct definition of each externally compiled procedure through the $USES directive.

In the simplest form, when no $USES statements appear in any of the separate compilations, the user simply $USES all separately compiled FORTRAN units in the main program. However, this limits the procedure calls in each of the separately compiled units to procedures defined in the that particular unit. If there are calls to procedures in unit A from unit B, then unit B must contain a $USES A statement. The main program must then contain a $USES A statement as its first $USES statement, followed by a $USES B statement. This is necessary for the compiler to get the unit numbers allocated consistently.

In more complex cases, the user must insure that all references to procedures in outside units are preceded by the proper $USES statement in the same order, and are not missing any units. If unit B $USES unit A, and unit C $USES unit B, then unit C must first $USES unit A. Likewise, if units D and E both $USES unit F, they both must contain exactly the same $USES statements prior to the $USES F statement.

FORTRAN OVERLAYS

The FORTRAN overlay mechanism is slightly more general than the Pascal mechanism. In Pascal, an overlay procedure is specified by the reserved word SEGMENT appearing prior to that procedure's name. The meaning is that the procedure and all nested procedures are to become an overlay. Thus, whenever that procedure is active, the segment is present in memory, and not otherwise! There is no way to combine two or more procedures into a single overlay such that the calling of either one causes the overlay to be loaded into memory, due to the fact that the static nesting of Pascal procedures hides any sub-procedures from any outside caller. The FORTRAN mechanism allows many such procedures to be visible to outside procedures, thus overcoming this limitation.

CHAPTER 15

BI-LINGUAL PROGRAMS

116 Introduction
116 Pascal in FORTRAN Main Programs
118 FORTRAN in Pascal Main Programs
119 I/O from Bilingual Programs
120 Calling Machine Code Routines

INTRODUCTION

As was discussed in Chapter 3, Programs in Pieces, if you want to mix FORTRAN and Pascal code, you must first separately compile all the subprograms that will be needed using the compiler of their native language. For instance, in order to call Pascal functions and procedures from a FORTRAN program, the Pascal routines must first be compiled into a Pascal unit. The FORTRAN program must then contain a $USES compiler directive statement for that unit as described in Chapter 4.

In attempting to interface the two languages, there are some fundamental differences which must be pointed out. For instance, the exceedingly rich type and data structures in Pascal are not available in FORTRAN. Also, the I/O systems of FORTRAN and Pascal are not compatible. The fact that they both execute P-code on the Apple Pascal operating system overcomes most of the other problems, however. This section is designed to help you interface the two languages.

PASCAL IN FORTRAN MAIN PROGRAMS

Since there are Pascal types that have no FORTRAN equivalent, the way FORTRAN looks at Pascal data structures is somewhat limited. Thus, when a FORTRAN program $USES a Pascal unit, the FORTRAN compiler must make some translations of the kinds of data it finds there. The table below shows how these are mapped into FORTRAN data types.

Ordinary FORTRAN compilers do not recognize the passing of an argument by value to a subprogram; they only recognize passing arguments by reference. It should be noted that FORTRAN does not recognize global variables declared in the INTERFACE portion of a Pascal unit. If there is a global variable in a unit called by FORTRAN, the Linker will gemerate the error message: PUBLIC <varname> UNDEFINED.

The difference between value and reference arguments is that, for a variable passed to a subprogram by reference, the address of the variable is passed to the subroutine, so that the subroutine can then fetch the contents of that variable, and possibly replace its contents with another value. When a variable is passed to a subroutine by value, the contents of the variable is first copied into a special temporary location before the subroutine is called. The subroutine is only given the address of this temporary cell, which allows the original variable to be protected from the subroutine.

It should be understood that the Apple FORTRAN compiler cannot create FORTRAN subroutine argument calls by value, but that if, via a $USES statement, it encounters a Pascal procedure or function which does have value parameters in its argument list, it will generate the correct calling sequence for that Pascal procedure.

The following table shows how FORTRAN views Pascal declarations that it finds via a $USES statement:

	Pascal	FORTRAN
DECLARATIONS:		
	CONST anything ... ;	Ignored
	TYPE anything ... ;	Ignored
	VAR anything ... ;	Ignored
	PROCEDURE X(arg-list);	SUBROUTINE X(arglist)
	FUNCTION X(arg-list): type;	type FUNCTION X(arglist)

Note: Type of FUNCTION may only be INTEGER, LOGICAL, or REAL.

	Pascal	FORTRAN
DATA TYPES:		
	REAL	REAL
	BOOLEAN	LOGICAL
	CHAR	CHARACTER*1
	any other identifier	INTEGER

Note: Be aware that the results of passing some of the more esoteric Pascal data types to FORTRAN INTEGER data types can be tricky. You should do trials first to determine the exact results.

Note: There is no proper FORTRAN equivalent to value parameters, but the FORTRAN compiler does generate the correct calling sequence for Pascal routines with value parameters.

The following FORTRAN program calls a Pascal unit called PUNIT found in Z:PAS.CODE. If your diskette is not designated Z:, you will have to change the FORTRAN program to give the correct name.

```
C  FORTRAN PROGRAM TO CALL PASCAL ROUTINE
$USES PUNIT IN Z:PAS.CODE
      I=ADDONE(3)
      WRITE(*,100)I
100   FORMAT(I8)
      END
```

This is the Pascal unit called by the FORTRAN program:

```
(*$S+*)
UNIT PUNIT;
INTERFACE
  FUNCTION ADDONE(INT:INTEGER): INTEGER;
IMPLEMENTATION
  FUNCTION ADDONE;
    BEGIN
      ADDONE:=INT+1;
    END
  BEGIN
  (* NO INITIALIZATION CODE IN THIS EXAMPLE *)
  END
```

FORTRAN IN PASCAL MAIN PROGRAMS

The following table gives the data type correspondences:

FORTRAN	Pascal
DECLARATIONS:	
SUBROUTINE X(arg-list)	PROCEDURE X(arg-list);
type FUNCTION X(arg-list)	FUNCTION X(arg-list): type;
DATA TYPES:	
INTEGER	INTEGER
REAL	REAL
LOGICAL	BOOLEAN
CHARACTER*	CHAR
argument list:	
(I)	(VAR I: type)
type I	

When a Pascal program USES a FORTRAN unit, it is the responsibility of the Pascal program to make sure that any needed type declarations for the string or packed array of CHAR types are properly defined for the FORTRAN unit. This cannot consistently be done by FORTRAN as it would lead to duplicate type definitions should a program use two FORTRAN units in which each declare the same entity.

Note: Pascal stores its multidimensional arrays by row-major order, while FORTRAN stores them by column-major order.

The following Pascal program is used to call a FORTRAN function called ADDONE:

```
PROGRAM CALLFORTRAN;
(*$U Z:FOR.CODE*)
USES UADDONE;
BEGIN
WRITELN(ADDONE(3));
END
```

This is the FORTRAN function saved as Z:FOR.CODE:

```
INTEGER FUNCTION ADDONE(I)
ADDONE=I+1
END
```

Note that the FORTRAN unit got the name UADDONE automatically from the concatenation of U to the first function or subroutine name encountered in the file, ADDONE.

I/O FROM BILINGUAL PROGRAMS

Because the I/O systems of FORTRAN and Pascal are not compatible, it is not always possible to do everything that is desired. This section does, however, help the user to do what is possible in interfacing the two languages.

The FORTRAN compiler assumes that the run time support unit RTUNIT is assigned unit number 8. Therefore, it is generally a good idea for Pascal programs that use FORTRAN units to USES RTUNIT in such a manner that it will be assigned number 8. For this to happen, RTUNIT must be the second unit used by the Pascal program. While not all FORTRAN units actually call run time support routines that reside in RTUNIT, the absence of RTUNIT in such a case can lead to very mysterious results.

It is not generally possible to do I/O from Pascal routines called from a main program that is written in FORTRAN. Normal Pascal I/O to and from the console, however, can always be done from Pascal routines providing that there is no file name in the I/O statement. The Pascal routines RESET, REWRITE, CLOSE, etc., should not be called from Pascal routines running under a FORTRAN program.

It is possible to do I/O from a FORTRAN procedure that is called from a Pascal main program. In general, however, this practice should be avoided. The following information is provided to allow the user who absolutely must mix I/O operations from both languages to do what is possible.

There are several precautions that the user must take for FORTRAN I/O to work from a Pascal program. The Pascal main program must USES the FORTRAN run time unit RTUNIT. This must be done in such a manner that RTUNIT is assigned unit number 8 by the Pascal main program. Prior to any FORTRAN I/O operations, the Pascal program must call the procedure RTINITIALIZE. After all FORTRAN I/O is completed, the Pascal program must call the procedure RTFINALIZE. Both of these procedures exist in the FORTRAN run time unit. The FORTRAN I/O procedures use the heap for the allocation of file related storage, so the user should not force the deallocation of heap memory via calls to MARK and/or RELEASE. If the user USES TURTLEGRAPHICS in the Pascal program, then INITTURTLE must be called prior to calling RTINITIALIZE. This is due to the way that TURTLEGRAPHICS handles the heap marker. Other restrictions may apply in special cases.

CALLING MACHINE CODE ROUTINES

The following example uses a machine code function and a subroutine.

```
;
;  SAMPLE MACRO POPS 16 BIT ARGUMENT
;
        .MACRO POP
        PLA
        STA %1
        PLA
        STA %1+1
        .ENDM


        .FUNC PADDLE,1  ;ONE WORD OF PARAMETERS
;_________________________________________
;
; SAMPLE GAME PADDLE FUNCTION FOR PASCAL
; (This function provided in APPLESTUFF unit.)
;
; FUNCTION PADDLE(SELECT: INTEGER): INTEGER;
;
;_________________________________________
RETURN  .EQU    0       ;TEMP VAR FOR RETURN ADDR
                        ;note: 0..35 hex available
TEMP    .EQU    2       ;TEMP VAR FOR ARGUMENT ADDR

        POP RETURN      ;SAVE PASCAL RETURN ADDR
        PLA             ;DISCARD 4 BYTES STACK BIAS
        PLA             ;( ONLY DO FOR .FUNC )
        PLA
        PLA
        POP TEMP        ;GET ARGUMENT ADDR
        LDY #0
        LDA (TEMP),Y    ;LOAD ARGUMENT'S VALUE
        AND #3          ;FORCE INTO RANGE 0..3
        TAX
        LDA 0C070       ;TRIGGER PADDLES
        LDY #0          ;INIT COUNT IN Y REG
        NOP             ;COMPENSATE FIRST COUNT
        NOP
PREAD2  LDA 0C064,X     ;TEST PADDLE
        BPL DONE        ;BRANCH IF TIMER DONE
        INY             ;ELSE INC Y EVERY 12 USEC
        BNE PREAD2      ;LOOP UNLESS 255 EXCEEDED
        DEY             ;MAKE 0 INTO 255 (MAX COUNT)
DONE    LDA #0
        PHA             ;PUSH MSB OF RETURN VALUE=0
        TYA
        PHA             ;PUSH LSB OF RETURN VALUE
        LDA RETURN+1    ;RESTORE PASCAL RETURN ADDR
        PHA
        LDA RETURN
        PHA
        RTS             ;AND RETURN TO PASCAL CALLER


        .PROC TTLOUT,2  ;TWO WORDS OF PARAMETERS
;_________________________________________
;
; ROUTINE TO SET OR CLEAR ONE OF THE TTL I/O BITS
; (This procedure provided in APPLESTUFF unit.)
;
; PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);
;
RETURN  .EQU    0       ;TEMP RETURN ADDR

        POP RETURN      ;SAVE PASCAL RETURN ADDRESS
                        ;POP PARAMETERS, LAST FIRST
        PLA             ;GET LSB BOOLEAN DATA 1=TRUE
        LSR A           ;SAVE BOOLEAN IN CARRY
        PLA             ;DISCARD MSB BOOLEAN DATA
        PLA             ;GET LSB SELECT
        AND #03         ;TREAT IT MOD 4
        ROL A           ;DOUBLE, ADD DATA FOR INDEX
        TAY             ;PUT I/O STROBE INDEX IN Y
        LDA 0C058,Y     ;ACTIVATE I/O STROBE
        PLA             ;DISCARD MSB SELECT PARAM
        LDA RETURN+1    ;RESTORE PASCAL RETURN ADDR
        PHA
        LDA RETURN
        PHA
        RTS             ;GO BACK TO PASCAL



        .END            ;END OF ASSEMBLY
```

The $EXT statement can be used to call machine language routines from a FORTRAN program. The following example calls the machine language routines listed above. You should note a couple of things here. First, we don't use the normal $USES statement, but substitute $EXT. Secondly, we don't have to CALL the routine called PADDLE because it is a function. We do, of course, CALL TTLOUT.

```
$EXT INTEGER FUNCTION PADDLE 1
$EXT SUBROUTINE TTLOUT 2
C
      PROGRAM CALASM
      DO 100 I=1, 100
      WRITE(*,400)PADDLE(0),PADDLE(1)
100   CONTINUE
      CALL TTLOUT(0,.TRUE.)
400   FORMAT(2I12)
      END
```

The example simply reads the two control paddles and writes the values returned to the screen. This is a handy routine for game programming, and the incredible speed of the machine language operations can be very useful in such real-time applications.

CHAPTER 16

SPECIAL UNITS

124 The Turtle Graphics Unit
124   The Apple Screen
124   The INITTU Subroutine
125   The GRAFMO Subroutine
125   The TEXTMO Subroutine
125   The VIEWPO Subroutine
126   Subroutines for Using Color
127   Cartesian Graphics
127   Turtle Graphic Subroutines
128   Turtle Graphic Functions
129   Sending an Array to the Screen
130   Text on the Graphic Screen
131 The Applestuff Unit
132   RANDOM Function/RANDOI Subroutine
132   Using the Game Controls
134   Making Music: the NOTE Subroutine
134   The KEYPRE Function

THE TURTLE GRAPHICS UNIT

There is a CODE unit in SYSTEM.LIBRARY which contains a set of subroutines that have been designed to enable the use of fancy color-graphics on your Apple.

The following compiler directive statement must appear near the top of the program or subprogram that uses this CODE unit:

```
$USES TURTLEGRAPHICS
```

The statement must come before any executable statement or specification statement. It may appear after other compiler directive statements or comment lines.

If this statement appears, the graphics subroutines and functions described in this section can be used. This statement tells the FORTRAN system to get the graphics subprograms from the library. These subprograms are loaded in at run-time, which means that the library file must be available to the system when any program using TURTLEGRAPHICS or APPLESTUFF is executed.

Incidentally, this graphics package is called Turtle Graphics since it is based on the turtles devised by S. Papert and his co-workers at the Massachusetts Institute of Technology. To make graphics easy for children who might have difficulty understanding Cartesian coordinates, Papert et al. invented the idea of the turtle who could walk a given distance and turn through a specified angle while dragging a pencil along. Very simple algorithms in this system, which could be called relative polar coordinates, can give more interesting images than an algorithm of the same length in Cartesian coordinates.

The Apple Screen

The Apple screen is a rectangle with the origin (X=0,Y=0) at the lower left corner. The upper right corner has the coordinates (X=279,Y=191). Since points may only be placed at integral coordinates, all arguments to the graphics functions are integers.

There are two different screen images stored in the Apple's memory. One of them holds text, the other holds a graphic image. There are three statements that switch between the modes. They are INITTU, TEXTMO and GRAFMO.

The INITTU Subroutine

This subroutine has no parameters. It clears the screen, and allows the screen to be used for graphics rather than text. It is a good idea to use this routine before starting any graphics.

INITTU does a few other things as well: the turtle (more about it later) is placed in the center of the screen facing right, the pen color is set to NONE (more about this later too) and the viewport is set to full screen.

The GRAFMO Subroutine

The GRAFics MOde subroutine has no parameters. It switches the monitor or TV to show the graphics screen, without the other initialization that INITTU does. It is usually used to show graphics in a program that switches between graphics and text display.

The TEXTMO Subroutine

The TEXT MOde subroutine has no parameters. It switches from graphics mode, obtained by INITTU or GRAFMO, to showing text. It is a very, very good idea to conclude any graphics program with a return to text mode. If you forget to do this, you may not be able to see the usual COMMAND: prompt or any other text. When you switch to text mode, the image that you saw in GRAFMO is not lost, but will still be there when you use GRAFMO to go into graphics mode again, unless you deliberately changed it.

The VIEWPO Subroutine

The VIEWPOrt subroutine has the form

```
VIEWPO (left, right, bottom, top)
```

where the four parameters are integers which give the boundaries you want the viewport to have. If you don't use this subroutine, Apple FORTRAN assumes that you want to use the whole screen for your graphics.

```
VIEWPO (130, 150, 86, 106)
```

This example would allow the screen-plotting of all points whose X-coordinates are from 130 through 150 and whose Y-coordinates are from 86 through 106. For further information on VIEWPO see the descriptions of the line drawing subroutines, FILLSC and DRAWBL.

Clipping: When a line is drawn using any of the graphic commands, it is automatically clipped so that only the portion which lies within the current viewport is displayed. Points whose coordinates are not in the current viewport, even those points that would not be on the screen at all, are legal but are ignored.

This allows some dramatic effects. It also allows you to plot off-screen all day, and never see a thing or get an error message. Clipping cannot be disabled.

Subroutines for Using Color

The PENCOL and FILLSC subroutines are used for color in Turtle Graphics. The PENCOL subroutine sets the pen color. It has the form

```
      PENCOL (PENMODE)
```

where penmode is an integer which corresponds to a particular color or other mode as described in the table below.

Integer	PENMODE color
0	NONE Drawing with this "color" produces no change on the screen. You can consider it as drawing with the color that happens to be there already, or as invisible ink.
1	WHITE
2	BLACK
3	REVERSE Drawing with REVERSE changes BLACK to WHITE and WHITE to BLACK. It also changes WHITE1 to BLACK1, WHITE2 to BLACK2, GREEN to VIOLET and ORANGE to BLUE and vice versa. It is rather a magical pen. It allows you to draw, say, a line across a complex background and have it still show up.
4	RADAR This "color" has been left unused for future applications.
5	BLACK1 (two dots wide, for use with green and violet)
6	GREEN
7	VIOLET
8	WHITE1 (two dots wide, for use with green and violet)
9	BLACK2 (two dots wide, for use with orange and blue)
10	ORANGE
11	BLUE
12	WHITE2 (two dots wide, for use with orange and blue)

If you'd like the drawing to be in GREEN, you would use the statement:

```
      CALL PENCOL (6)
```

Now, it may seem strange that aside from WHITE, BLACK, GREEN, VIOLET, ORANGE, and BLUE, there are two additional flavors of WHITE and BLACK. These are due to the intricate, not to say bizarre, way that color television sets concoct their color, interacting with the technique that Apple uses to get a lot of color very economically. Rather than explaining how this all works, suffice it to say here that WHITE and BLACK give the finest lines possible, and the colors give a wider line in order to make the colors show. If you wish to make a white or black line that corresponds exactly in position and width with a green or violet line then you should use WHITE1 or BLACK1. If you wish to make a white or black line that corresponds exactly in position and width with an orange or blue line, then you should use WHITE2 or BLACK2.

On a black-and-white monitor or TV set, just use WHITE and BLACK for your colors.

The FILLSC subroutine has the form

```
      FILLSC (PENMODE)
```

where PENMODE is any of the integers standing for colors described above. FILLSC fills the entire viewport with the color indicated by PENMODE. For example

```
      FILLSC (2)
```

clears the viewport. The statement

```
      FILLSC (3)
```

makes a color negative of the contents of the viewport.

Cartesian Graphics

The MOVETO subroutine has the form

```
      MOVETO (X, Y)
```

where X and Y are integer screen coordinates. MOVETO creates a line in the current penmode from the last point drawn to the coordinates given by (X,Y). When you INITTU, the turtle moves (with color NONE) to the center of the screen.

The direction of the turtle, as described below, is not changed by MOVETO.

Turtle Graphic Subroutines

To understand turtle graphics, first imagine a small turtle sitting at the center of the screen, facing right. This turtle can turn or it can walk in the direction it is facing. As it walks, it leaves behind a trail of the current pen color.

The TURNTO subroutine has the form

```
TURNTO (DEGREES)
```

where DEGREES is an integer. It is treated modulo 360, and thus never gets out of the range -359 through 359. When invoked, this subroutine causes the turtle to turn from its present angle to the indicated angle. Zero is exactly to the right, and counterclockwise rotation represents increasing angles. This command never causes any change to the image on the screen. A negative argument causes clockwise rotation; a positive argument causes counterclockwise rotation.

The TURN subroutine has the form

```
TURN (DEGREES)
```

where DEGREES is again an integer number treated modulo 360. This subroutine causes the turtle to rotate counterclockwise from its current direction through the specified angle. It causes no change to the image on the screen.

The MOVE subroutine has the form

```
MOVE (DISTANCE)
```

where DISTANCE is an integer. This subroutine makes the turtle move in the direction in which it is pointing a distance given by the integer DISTANCE. It leaves a trail in the current pen color. The sequence of statements:

```
CALL PENCOL (1)
CALL MOVE (50)
CALL TURN (120)
CALL MOVE (50)
CALL TURN (120)
CALL MOVE (50)
```

draws an equilateral triangle, for instance.

Turtle Graphic Functions

The functions TURTLX, TURTLY, TURTLA and SCREEN allow you to ask your Apple about the current state of the turtle and the screen. Note that any functions specified without parameters must have ( ) following the function name.

The TURTLX and TURTLY functions, no parameters, return integers giving the current X and Y coordinates of the turtle.

The TURTLA function, no parameters, returns an integer giving the current turtle angle as a positive number of degrees modulo 360.

The SCREEN function has the form

```
SCREEN (X,Y)
```

where X and Y are screen coordinates. This function returns the logical value true if the specified location on the screen is not black, and false if it is black. It doesn't tell you what color is at that point, but only whether there is a turtle-mark, anything nonblack, there.

Sending an Array to the Screen

The DRAWBL subroutine has the form

```
DRAWBL (SOURCE, ROWSIZE, XSKIP, YSKIP, WIDTH, HEIGHT, XSCREEN,
YSCREEN, MODE)
```

where the SOURCE parameter is the name without subscripts of a two-dimensional array of type LOGICAL. All the other parameters are integers.

DRAWBL copies an array of dots in memory or a portion of the array onto the screen to form a screen image. You may choose to copy the entire SOURCE array, or you may choose to copy any specified window from the array, using only those dots in the array from XSKIP to XSKIP+WIDTH and from YSKIP to YSKIP+HEIGHT. Furthermore, you can specify the starting screen position for the copy, at (XSCREEN, YSCREEN).

The DRAWBL subroutine parameters have the following meaning:

SOURCE is the name of the two-dimensional BOOLEAN array to be copied.

ROWSIZE is the number of bytes per row in the array.

XSKIP tells how many horizontal dots in the array to skip over before the copying process is started.

YSKIP tells how many vertical dots in the array to skip over before beginning the copying process. Note that copies are made starting from the bottom up. The array, in effect, gets turned upside down.

WIDTH tells how many dots width of the array, starting at XSKIP, will be used.

HEIGHT tells how many dots height of the array, starting at YSKIP, will be used.

XSCREEN and YSCREEN are the coordinates of the lower left corner of the area to be copied into. The WIDTH and HEIGHT determine the size of the rectangle.

MODE ranges from 0 through 15. The MODE determines what appears on the portion of the screen specified by the other parameters. It is a powerful option which can simply send white or black to the screen, irrespective of what is in the array, copy the array literally, or combine the contents of the array and the screen and send the result to the screen. The following table specifies what operation is performed on the data in the array and on the screen, and thus what appears on the screen. The algebraic notation uses A for the array, and S for the screen. The symbol ~ means NOT.

MODE	EFFECT
0	Fills the area on the screen with black.
1	NOR of array and the screen. (A NOR S)
2	ANDs array with the complement of the screen. (A AND ~S)
3	Complements the screen. (~S)
4	ANDs the complement of array with the screen. (~A AND S)
5	Complements array. (~A)
6	XORs array with the screen. (A XOR S)
7	NANDs array with the screen. (A NAND S)
8	ANDs array and the screen. (A AND S)
9	EQUIVALENCEs array and the screen. (A = S)
10	Copies array to the screen. (A)
11	ORs array with the complement of the screen. (A OR ~S)
12	Screen replaces screen. (S)
13	ORs complement of array with screen. (~A OR S)
14	ORs array with screen. (A OR S)
15	Fills area with white.

Text on the Graphic Screen

Two subroutines, WCHAR and CHARTY, allow you to annotate graphics. If the turtle is at (X,Y) you can use these subroutines to put a character or string on the screen with its lower left corner at (X,Y). The WCHAR subroutine uses an array stored in the file SYSTEM.CHARSET. This array contains all the characters used, and is read in by the initialization routine when your program $USES TURTLEGRAPHICS. The subroutine DRAWBL is then used to copy each character from the array onto the screen. (Note that WSTRING is not available in FORTRAN because its argument is a string.)

If you make up a file containing your own character set, you should rename the old SYSTEM.CHARSET and then name your new array SYSTEM.CHARSET.

The WCHAR subroutine has the form

```
WCHAR (CH)
```

where CH is an expression of type CHAR. This subroutine places the character on the screen with its lower left corner at the current location of the turtle. When this subroutine is used, the turtle is shifted to the right 7 dots from its old position. For example, this puts an X in the center of the screen:

```
CALL PENCOL (0)
CALL MOVETO (137,90)
CALL WCHAR ('X')
```

In this example, note that it was not necessary to specify a new PENCOL before calling WCHAR. The character is not plotted with the current pen color; rather it depends on the current MODE, just as DRAWBL does. For details, see CHARTY below.

The CHARTY subroutine has the form

```
CHARTY (MODE)
```

where MODE is an integer selecting one of the 16 modes described above for DRAWBL. MODE defines the way characters get written on the screen. The default MODE is 10, which places each character on the screen in white, surrounded by a black rectangle. One of the most useful other MODEs is 6, which does an exclusive OR of the character with the current contents of the screen. Note that redrawing a character in exclusive OR mode effectively erases the character, leaving the original image unaffected. This is especially useful for user messages in a graphics oriented program.

Another useful MODE is 5, which gives inverse characters. Lastly, inverted exclusive OR would be a MODE of 9.

THE APPLESTUFF UNIT

This section tells you how to generate random numbers, how to use the control paddle and button inputs, how to read the cassette audio input, how to switch the control's TTL outputs and how to generate sounds on the Apple's speaker. To use these special Apple features from FORTRAN, you first have to place the statement

```
        $USES APPLESTUFF
```

before any executable statements in your program. The $ must appear in column 1. This compiler directive statement may appear after other compiler directive statements or comment statements. If you wish to use both TURTLEGRAPHICS and APPLESTUFF you would say both:

```
        $USES TURTLEGRAPHICS
        $USES APPLESTUFF
```

RANDOM Function/RANDOI Subroutine

RANDOM is an integer function with no parameters. It returns a value from 0 through 32767. If RANDOM is called repeatedly, the result is a psuedo-random sequence of integers. The following routine will display a random integer on the screen that is between the indicated limits:

```
C DEMO PROGRAM OF RANDOM FUNCTION
$USES APPLESTUFF
      INTEGER HI,LO,RESULT
      HI=100
      LO=10
      DO 100 I=1,10
      X=(HI-LO)/32767.0
      RESULT=X*RANDOM()+LO
100   WRITE(*,200)RESULT
200   FORMAT(I8)
      END
```

RANDOI is a subroutine with no parameters. Each time you run a given program using RANDOM, you will get the same random sequence unless you use RANDOI.

RANDOI uses a time-dependent memory location to generate a starting point for the random generator. The starting point changes each time you do any input or output operation in your program. If you use no I/O, the starting point for the random sequence does not change.

Using the Game Controls

The PADDLE and BUTTON functions and the TTLOUT subroutine are known as the game controls.

The PADDLE function has the form

```
      PADDLE (SELECT)
```

where SELECT is an integer treated modulo 4 to select one of the four paddle inputs numbered 0, 1, 2, and 3. PADDLE returns an integer in the range 0 to 255 which represents the position of the selected paddle. A 150K ohm variable resistance can be connected in place of any of the four paddles.

If you try to read two paddles too quickly in succession, the hardware may not be able to keep up. PADDLE data will be clipped and the PADDLE function will not return the correct results. A suitable delay is given by using a do-nothing loop as illustrated in the following example. This program reads the paddles and loops until a key is pressed.

```
C DEMO OF PADDLE FUNCTION
C
C HERE WE ARE USING A DO (NOTHING) LOOP TO SLOW DOWN
C
$USES APPLESTUFF
300   I=PADDLE(0)
      DO 200 K=0,3
200   CONTINUE
      J=PADDLE(1)
      WRITE(*,100)I,J
      IF (.NOT. KEYPRE()) GOTO 300
100   FORMAT(2I8)
      END
```

The BUTTON function has the form

```
      BUTTON (SELECT)
```

where SELECT is an integer treated modulo 4 to select one of the three button inputs numbered 0, 1, and 2, or the audio cassette input numbered 3. The BUTTON function returns a logical value of true if the selected game-control button is pressed, and false otherwise.

When BUTTON(3) is used to read the audio cassette input, it samples the cassette input, which changes from true to false and vice versa at each zero crossing of the input signal.

There are four TTL level outputs available on the game connector along with the button and paddle inputs. The TTLOUT subroutine is used to turn these outputs on or off. TTLOUT has the form

```
      TTLOUT (SELECT, DATA)
```

where SELECT is an integer treated modulo 4 to select one of the four TTL outputs numbered 0, 1, 2, and 3. DATA is a logical expression.

If DATA is true, then the selected output is turned on. It remains on until TTLOUT is invoked with the DATA set to false.

Making Music: the NOTE Subroutine

The NOTE subroutine has the form

```
    NOTE (PITCH, DURATION)
```

where PITCH is an integer from 0 through 50 and DURATION is an integer from 0 through 255.

A PITCH of 0 is used for a rest, and 2 through 48 yield a tempered (approximately) chromatic scale. DURATION is in arbitrary units of time.

NOTE (1,1) gives a click.

A musical scale is played by the following program:

```
C PROGRAM PLAYS MUSICAL SCALE
C
$USES APPLESTUFF
      PROGRAM MUSIC
      INTEGER PITCH
      DO 100 PITCH=2,48
100   CALL NOTE(PITCH,10)
      END
```

The KEYPRE Function

The KEYPRE function returns a value of true if a key is pressed from the console. Refer to the program in the Using the Game Controls Section for an example of the KEYPRE function.

APPENDIX A—PART ONE

SINGLE-DRIVE OPERATION

136 Introduction
136 Configuring Your System
138 System Startup
139 Changing the Date
140 Making Backup Diskettes
  140 How We Make Backups
  141 Formatting Diskettes
  143 Making the Actual Copies
145 Using the System
  147 And Now, Some Fun
  151 Executing a Program
  151 Writing a Program
  152 What to Leave in the Drive

INTRODUCTION

Appendix A Part One covers configuring your Apple FORTRAN System, booting Pascal and FORTRAN for the first time, and using the Editor and Filer programs of the Apple Pascal Operating System to format and make backup copies of diskettes. This appendix is for the user of a single-drive system.

The procedures described in the section immediately following, Configuring Your System, are a tutorial on how to configure your Apple FORTRAN System. The sections after that provide step-by-step instructions for those of you unfamiliar with the Pascal Operating System.

CONFIGURING YOUR SYSTEM

To run Apple FORTRAN, you should have the following equipment:

* Your 48K Apple computer, with a Language Card installed, and one disk drive attached to the connector marked DRIVE 1 on the disk controller card. The disk controller card must have the new PROMs, P5A and P6A which came with the Language System, and must be installed in the Apple's peripheral device slot 6.

* A TV set or video monitor properly connected to your Apple.

* The following diskettes and at least two blank diskettes:

FORT1:
FORT2:
APPLE1:
APPLE2:
APPLE3:

Your new Apple FORTRAN System consists of the following diskettes:

* Two identical diskettes labeled FORT2: each containing two files: SYSTEM.COMPILER and SYSTEM.LIBRARY. The SYSTEM.COMPILER is protected from being copied.

* One diskette labeled FORT1: containing the file FORTLIB.CODE.

To create an Apple FORTRAN System, you must transfer certain files from diskettes APPLE1: and APPLE2: of your Apple Pascal System to the two FORTRAN diskettes, FORT1: and FORT2:. This section explains how to transfer the required files from APPLE1: and APPLE2: to create a working FORTRAN system. It is our recommendation for an Apple FORTRAN System; it is not the only way to do this.

First, plug in the TV or monitor and turn it on. Then plug in the Apple. Put diskette APPLE1: in the disk drive and turn on the Apple.

You need to transfer the following files from APPLE1: to FORT1:

SYSTEM.APPLE
SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.CHARSET
SYSTEM.FILER
SYSTEM.EDITOR

You also need to transfer the following file from APPLE2: to FORT1:

SYSTEM.LINKER

Type F to enter the Filer, and then type T to Transfer files from one diskette to another. The screen asks TRANSFER ?, and you type
SYSTEM.APPLE
The screen then asks TO WHERE ?, and you respond by typing
FORT1:$
The dollar sign means to use the same file name for the file on diskette FORT1:. The name of the file after being transferred is FORT1:SYSTEM.APPLE.

The screen then says:

```
PUT IN FORT1:
TYPE <SPACE> TO CONTINUE
```

Put in FORT1: and press the spacebar. Take the diskettes in and out of the disk drive as instructed to do so by the directions on the screen. Note that the disk drive is known as unit #4. After the entire file has been transferred, the Filer prompt line appears on the screen. Put APPLE1: back in the disk drive, and type T for Transfer. Repeat the transfer process just described for all the files on APPLE1: listed above.

When all the required files have been transferred from APPLE1: to FORT1:, put APPLE2: in the disk drive and type T for Transfer. When the screen asks TRANSFER ?, you respond by typing
APPLE2:SYSTEM.LINKER
The screen then asks TO WHERE ?, and you type
FORT1:$
The transfer process proceeds as described above. When this file is transferred, you have a complete FORT1: diskette.

You now need to transfer the following files from APPLE1: to FORT2:

SYSTEM.APPLE
SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.CHARSET

Note that FORT2: came to you with two files on it:

```
SYSTEM.COMPILER
SYSTEM.LIBRARY
```

Transfer the four required files from APPLE1: to FORT2:, following the procedures described above. When those files are transferred, you have a complete FORT2: diskette.

At the conclusion of the transfer process, FORT1: and FORT2: should contain the following files:

FORT1:	FORT2:
SYSTEM.APPLE	SYSTEM.APPLE
SYSTEM.PASCAL	SYSTEM.PASCAL
SYSTEM.MISCINFO	SYSTEM.MISCINFO
SYSTEM.CHARSET	SYSTEM.CHARSET
SYSTEM.FILER	SYSTEM.COMPILER
SYSTEM.LINKER	SYSTEM.LIBRARY
SYSTEM.EDITOR	
FORTLIB.CODE	

SYSTEM STARTUP

To start Apple FORTRAN running on your system, first insert the diskette marked FORT2: into the disk drive. As you will soon discover, you can boot FORTRAN using either of the two diskettes provided. If you'll remember that the Filer, Linker, and Editor are on FORT1:, and the Compiler and Library are on FORT2:, you will easily be able to decide which diskette should be in your drive to provide the functions you want to use.

Close the door to the disk drive, and turn on the Apple. First, the message

```
APPLE II
```

appears at the top of your TV or monitor screen, and the disk drive's IN USE light comes on. Then this message appears:

```
WELCOME FORT2, TO
U.C.S.D. PASCAL SYSTEM II.1
CURRENT DATE IS 26-JUL-79
```

The date may be different. This is followed in a second or so by a line at the top of the screen:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

This line at the top of the screen is called a prompt line. When you see this prompt line, you know that your Apple computer is running the Apple FORTRAN System.

CHANGING THE DATE

The date that comes on the diskette will probably not be correct. It is a good habit to reset the date the first time you use the Apple FORTRAN System on any given day. It only takes a few seconds. First put FORT1: in the drive. Press F on the keyboard without pressing the RETURN key or any other keys. The screen goes blank, and then this line appears at the top:

```
FILER: G, S, N, L, R, C, T, D, Q [C.1]
```

This is a new prompt line. Prompt lines are named after their first word. The prompt line you first saw was the Command prompt line. This one is the Filer prompt line. Sometimes we say that you are "in the Filer" when this line is at the top of the screen. Each of the letters on the prompt line represents a task that you can ask the system to do. For example, to change the date, press D. Again, just type the single key, without pressing RETURN or any other key.

When you do, another message is put on the screen. It says:

```
DATE SET: <1..31>-<JAN..DEC>-<00..99>
TODAY IS 26-JUL-79
NEW DATE ?
```

It doesn't really mean that today is 26-JUL-79 or whatever date your screen shows, but that the Apple thinks that is today's date. Since it isn't, you can change the date to be correct. The correct form for typing the date is shown on the second line of the message: one or two digits giving the day of the month, followed by a minus sign, followed by the first three letters of the name of the month, followed by another minus sign, followed by the last two digits of the current year. Then press the key marked RETURN. If the date is correct, or if you change your mind and decide not to change the date, just press the RETURN key.

If the month and year are correct, as they will often be when you change the date, all you have to do is type the correct day of the month, and press the RETURN key. The system will assume that you mean to keep the same month and year displayed by the message. If you type a day and a month, the system will assume you mean to keep only the year the same. Go ahead and make the date correct. Your new date is saved on the diskette, so the system remembers this date the next time you turn the Apple on.

Usually, at the top of the screen there will be a prompt line which represents several choices of action. When you type the first letter of one of the choices, either you will be shown a new prompt line giving a further list of choices, or else the system will carry out the desired action directly. If you type a letter that does not correspond to one of the choices, the prompt line blinks but otherwise nothing happens. Remember to type only a single letter to indicate your choice; it is not necessary to press the RETURN key afterward.

Sometimes, as when setting the date, you are asked to type a response of several characters. You tell the system that your response is complete by pressing the RETURN key. If you make a typing error before pressing the RETURN key, you can back up and correct the error by pressing the left-arrow key. You should experiment by making deliberate errors in entering a date, and then erasing the errors with the left-arrow key.

MAKING BACKUP DISKETTES

Ask yourself this question: What would happen to your system if you were to lose or damage one of the system diskettes? Without the diskettes, you don't own a FORTRAN system.

The first thing you should do, therefore, is to make a backup copy of FORT1:. Afterward, you should never use the original, but put it someplace where the temperature is moderate, where there is no danger of it getting wet, and where magnetic fields cannot get at it. Since the FORTRAN compiler on diskette FORT2: is protected from being copied, Apple provides a backup copy of FORT2:.

It is a good idea to have two backup copies of each original. That way, you will need to use an original only in the very rare case when both of its backup copies are lost. When one copy is lost or damaged, another backup copy is made from the surviving backup copy. If your backups were damaged or erased while in use, find out why they were destroyed before inserting your only surviving copy. If you can't figure out what the problem is, bring your system to the dealer to make sure it is working correctly.

How We Make Backups

The Apple FORTRAN system can copy all or any portion of information from one diskette onto another diskette unless the information is protected from being copied. But the system cannot store information on a new diskette without first preparing that diskette for use on the Apple. Therefore, the FORTRAN/Pascal system includes a program that allows you to purchase any 5-inch floppy diskette and format it so that it will work with the Apple FORTRAN system. But remember that you cannot use diskettes that you formatted for BASIC (using DOS) with your FORTRAN system. These are quite different. Of course, if you have an old BASIC diskette you'd like to convert to store FORTRAN programs, you can reformat it for the purpose, just as if it were a new, blank diskette. The old BASIC programs or data on the diskette will be lost, however.

If you have been following this discussion by carrying out the instructions on your Apple, diskette FORT1: should be in your drive, and the FILER prompt line should be showing at the top of the screen:

```
        FILER: G, S, N, L, R, C, T, D, Q [C.1]
```

Put FORT2: in the disk drive and type Q to Quit the Filer. When you Quit the Filer, the disk whirrs, and you see the Command prompt line again:

```
        COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

There is actually more of this prompt line, off to the right of your TV or monitor. To see the rest of the screen, hold down the CTRL key and, while holding it down, press the A key right alongside it. In the future we will simply say CTRL-A to abbreviate this procedure.

You now see

```
        K, X(ECUTE, A(SSEM, D(EBUG,? [II.1]
```

This is simply the rest of the line that began COMMAND:. All together, the full prompt line would look like this:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```

The Apple FORTRAN system displays information on a screen that is 80 characters wide, but your TV or monitor shows only the leftmost 40 characters or the rightmost 40 characters at any one time. You use CTRL-A whenever you wish to see if there is more information on the other half of the screen. Repeated pressing of CTRL-A flips back and forth between the left half of the screen and the right half. If your TV screen appears to be completely blank, it might mean that you are just staring at the empty right half of the screen.

Formatting Diskettes

Insert diskette APPLE3: from the Pascal package. Now, type

```
X
```

and the screen responds:

```
        EXECUTE WHAT FILE?
```

You type

```
APPLE3:FORMATTER
```

and press the RETURN key. The disk whirrs a bit and the screen says:

```
        APPLE DISK FORMATTER PROGRAM
        FORMAT WHICH DISK (4, 5, 9..12) ?
```

Take all the new, blank diskettes that you are going to use with the Apple FORTRAN System. Do not, of course, take any diskettes that have precious information on them, such as the diskettes that came with the Apple FORTRAN System. Place the diskettes in a pile. Their labels should be blank. Make sure that you don't have any diskettes with data in a non-FORTRAN or non-Pascal format, such as BASIC diskettes.

Remove APPLE3: from the disk drive, and place one of the blank diskettes into the drive. Type

```
4
```

and press the RETURN key. The number 4 is the volume number of your disk drive. Note for information only that if you had four drives, these would be volumes 4, 5, 11, and 12 in the FORTRAN-Pascal Language System.

If the diskette in the drive has already been formatted, you will receive a warning. For example, if you have left APPLE3: in the drive you will be warned with the message

```
DESTROY DIRECTORY OF APPLE3: ?
```

At this point you can type

```
N
```

(which stands for No) without pressing the RETURN key, and your diskette will not be destroyed.

Let's assume that you have placed a new, unformatted diskette in the disk drive. Then you will not get any warning, but the Apple will place this message on the screen:

```
NOW FORMATTING DISKETTE IN DRIVE 4
```

The drive will make some clickings and buzzings and begin to whirr. The process takes about 30 seconds. When formatting is complete, the screen again shows the message

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

Now you have a formatted diskette. We suggest that you write the word FORTRAN in small letters at the top of the diskette's label, using a marking pen.

Never use a pencil or ballpoint pen, as the pressure may damage the diskette. The label will let you know that the diskette is formatted for use with the Apple FORTRAN system, and you can distinguish it from unformatted diskettes, BASIC diskettes, or diskettes for use with other systems. Diskettes that are formatted for FORTRAN will also store Pascal files and vice-versa because the formatting used for FORTRAN and Pascal is identical.

While you are at it, repeat this formatting process on all the new diskettes that you want to use with the Apple FORTRAN System. With each new diskette, place it in the disk drive, type 4, and press the RETURN key. When each diskette is removed from the drive, label it.

You may wonder why your one-and-only disk drive is called "4". It has to do with the way the Pascal operating system deals with all of its peripherals. Your disk drive is just one of the many peripherals, such as printers, other disk drives, the keyboard, the TV monitor, and so

forth. It just happened that among these other peripherals, the first disk drive connected to the system got the number 4.

When you have finished formatting all your new diskettes, and have written the word FORTRAN on each of them, answer the question

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

with a simple press of the key marked RETURN. You get the message

```
PUT SYSTEM DISK IN #4 AND PRESS RETURN
```

By SYSTEM DISK the Apple means the FORT2: diskette. Sometimes your disk drive is called DRIVE 4 and sometimes #4:, but it's all the same thing.

Do as it says, place the FORT2: diskette in the disk drive and press the RETURN key.

The Apple says:

```
THAT'S ALL FOLKS...
```

And if you watch the top of the screen, the line:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```

appears. Of course, it doesn't all appear, but you know it's there, and can check with CTRL-A.

Making the Actual Copies

As you have seen, you can get into the Filer by typing F when you have the Command prompt line on the screen. You must have diskette FORT1: in the disk drive when you type F for the Filer, or you will get the message

```
NO FILE FORT1:SYSTEM.FILER
```

If this happens, just put FORT1: in the disk drive and type F again.

The Filer is that portion of the system that allows you to manipulate information on diskettes. One of the Filer's abilities is to transfer information from one diskette to another. To invoke this facility, once you have the Filer prompt line on the screen, type T for T(ransfer.

This is what you see:

```
TRANSFER ?
```

Answer the question as follows:
FORT1:
which means that you want to transfer the entire contents of the source diskette called FORT1:. After you have specified which diskette's information you want transferred and pressed the key marked RETURN, the computer checks to make sure the correct diskette is in the disk drive. If you have forgotten to put diskette FORT1: in the drive, you will see the message

```
FORT1:
NO SUCH VOL ON LINE <SOURCE>
```

In that case you must type T for Transfer again, and repeat the process. With the correct source diskette in the drive, the Transfer process continues and the computer asks

```
TO WHERE ?
```

Answer this question by typing
BLANK:
This is the name of the destination diskette, onto which you want FORT1:'s information transferred. When a diskette is formatted it is automatically given the name BLANK:. Incidentally, those colons (:) are very important. You use them to indicate that you are referring to an entire diskette, and not just a part of one.

After you have told the computer where you want FORT1:'s information transferred and pressed the key marked RETURN, it says:

```
TRANSFER 280 BLOCKS ? (Y/N)
```

This message is mainly there to give you a chance to abandon the transfer if you made a typing error in the names of the source or the destination diskettes. The phrase "280 BLOCKS" means merely the whole diskette. In any case, you type
Y
The disk whirrs a few times, and you see the message:

```
PUT IN BLANK:
TYPE <SPACE> TO CONTINUE
```

By the colon, you know that it means to put the diskette called BLANK: into the disk drive. The second line tells you to press the space bar when the diskette is in place and the door closed.

Some of the information which is on diskette FORT1:, including the diskette's name, is now in the Apple's memory and will be copied onto diskette BLANK:, completely overwriting BLANK:. Therefore, the computer warns you that you are about to lose any information that might be stored on BLANK:. It says

```
DESTROY BLANK: ?
```

Since you want to turn BLANK: into a perfect copy of FORT1:, the answer is Y. The process is under way. The computer will tell you to first put in one diskette and then the other. Follow the instructions. Your screen will look like this after a while:

```
PUT FORT1: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT BLANK: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT FORT1: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT BLANK: IN UNIT #4
TYPE <SPACE> TO CONTINUE
```

and so on. You will have to insert the two diskettes a total of 20 times, and press the spacebar 20 times, to copy the entire diskette. When copying is done, the screen says

```
FORT1:                    --> BLANK:
```

By this remark, the computer is telling you that the contents of FORT1:, including the diskette's name, have been copied onto the diskette that used to be called BLANK:. Now label the diskette and store the original in a cool, safe place.

When you are through making backup copies, be sure to put FORT2: back into the disk drive, before typing Q to Quit the Filer. If you forget to do this, the system will stop responding to the keyboard after you type Q; you will have to turn the Apple off and repeat the entire startup procedure.

USING THE SYSTEM

You are now ready to use the Apple FORTRAN system to run a program. The first thing for you to know is that you should always start or boot the system using FORT2: in the drive. FORT2: will thus be known as your system diskette or boot diskette, as it is often called. We'll assume that you have never used the Editor to create any new text files on it before.

But let's check and make sure! Put FORT1: in the drive and type F to get into the F(iler. Now type E for Extended listing. When your Apple asks you which diskette you want to examine, type
FORT1:
and press the RETURN key.

There should be no files on the diskette except SYSTEM files. (If any TEXT or CODE files are there, R(emove them.) While you're at it, do a listing of FORT2:, and R(emove any files that don't begin with SYSTEM. Leave FORT2: in the drive and type Q(uit to leave the Filer.

Remember that FORT2: must always be in the drive when you leave either the Editor or the Filer. Going the other way, into the Editor or Filer, FORT1: must be in the drive, as it contains the Filer and Editor programs.

Soon you'll get the Command prompt. Now put FORT1: in the drive and type

```
E
```

to call the Editor. When you see the Editor's prompt line, it should look like this:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT)
:
```

As usual, you must use CTRL-A to see the right half of the message. The prompt message gives you some information and some choices. The first word, >EDIT:, tells you that you are now in the Editor. The next sentence, NO WORKFILE IS PRESENT, tells you that you have not yet used the Editor to create a workfile, which is a scratchpad diskette copy of a program you are working on. If there had been a workfile on FORT1:, that file would have been read into the Editor automatically.

Since there was no workfile to read in, the Editor asks you, FILE? If you now typed the name of a .TEXT file stored on FORT1:, that textfile would be read into the Editor. However, there are no .TEXT files on FORT1: yet, and besides, you want to write a new program. In parentheses, you are shown how to say that you don't want to read in an old file: <RET> FOR NO FILE. This means that, if you press the Apple's RETURN key, no file will be read in and you can start a new file of your own. That's just what you want to do, so press the Apple's RETURN key. The rest of the message says if you first press the ESC key and then press the RETURN key, you'll be sent back to the Command prompt line. When you have pressed only the RETURN key, the full Edit prompt line appears:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP
```

Refer to the Pascal documentation (of whichever Pascal manual you have) for an explanation of these commands. The first one you will use here is

```
          I(NSRT
```

which selects the Editor's mode for inserting new text. Type I to select Insert mode, and yet another prompt line appears:

```
>INSERT: TEXT [<BS> A CHAR,<DEL> A LINE] [<ETX> ACCEPTS, <ESC> ESCAPES]
```

The meaning of this prompt message is simply that anything you type will be placed on the screen, just to the left of the white square cursor. If the cursor is in the middle of a line, the rest of the line is pushed over to make room for the new text. If you make a mistake, just use the left-arrow key to backspace over the error, and then retype. At any time during an insertion, if you press the Apple's ESC key your insertion will be erased. If you press CTRL-C at any time during an insertion, the insertion is made a permanent part of your file, safe from being erased by ESC or by the left-arrow key. You can then type I to enter Insert mode and type more text.

And Now, Some Fun

Now for our program. With the Insert prompt line showing, press the RETURN key a couple of times, to move the cursor down the screen a bit, and then type

```
C FOOTRAM DEMO
```

and press CTRL-C.

Your insertion so far is made permanent, and the Edit prompt line reappears. But that's not how to spell FORTRAN. Since you have already pressed CTRL-C, it is too late to backspace over your errors and re-type them.

Fortunately, there are other ways. First, let's delete the extra O in FORTRAN. Using the left arrow key, move the cursor left until it is sitting directly on the second O. Then type D to enter the Editor's D(LETE mode. Then press the right-arrow key once and the superfluous O will disappear. Now press CTRL-C to make this deletion permanent.

Now let's insert the missing R. The cursor is already correctly placed to make this insertion, right on top of the T. Type I to enter Insert mode. Ignore the fact that the remainder of the line seems to have suddenly disappeared, and type the missing letter R. When you press CTRL-C to make this insertion permanent, the rest of the line returns:

```
C FORTRAM DEMO
```

The letter M is certainly not correct, so move the cursor right using the right-arrow key until it is sitting directly on the M . Now type X to select the Editor's eXchange option. When the eXchange prompt line appears, press the N key and the offending M is instantly transformed. Here's another thing to know: for all these modes, I(nsert, D(elete and eXchange, if you press the ESC key instead of CTRL-C, the alteration is undone, as if it had never happened. If you press CTRL-C, the alteration is made a permanent part of your file. To change that M permanently, press CTRL-C. Finally we have:

```
C FORTRAN DEMO
```

Now you know how to use the Editor's Insert, Delete and eXchange modes to write text and to correct your errors. Try typing the rest of the program that follows into your file for practice. Be sure to accept your insertions, from time to time, by pressing CTRL-C. That way, you minimize your loss if you accidentally press the ESC key. Here is the complete program:

```
C FORTRAN DEMO
      DO 10 I=1,100
      X=I
10    WRITE(*,1)I,X,X*X
1     FORMAT (I6,3X,F7.2,3X,F8.2)
      END
```

When you are typing this program, the punctuation and spelling must be exactly as shown. The indentation of the lines is important in FORTRAN. Note especially that the indented lines must be in the seventh vertical column. Put them under the second R in FORTRAN. You will notice that, once you have started a new indentation, the Editor maintains that indentation for you. To move back to the left, just press the left-arrow key before you type anything on the new line.

Now you want to write this program to the diskette. With the Edit prompt line showing, type Q to select the Q(uit option. The following message appears:

```
>QUIT:
     U(PDATE THE WORKFILE AND LEAVE
     E(XIT WITHOUT UPDATING
     R(ETURN TO THE EDITOR WITHOUT UPDATING
     W(RITE TO A FILE NAME AND RETURN
```

Type W to tell it you want to write to a file. Now it will prompt you to enter the name of the file you want to use. Before you answer the Apple's question, put in diskette FORT2:. This is done for two reasons: First, you are leaving the Editor, and the boot diskette must be in the drive. Secondly, you are going to put your file onto FORT2:, because that's where the Compiler is. With FORT2: in the drive, you can answer the Apple's query with: FORT2:DEMO1 which names the file DEMO1 and stores it on FORT2:.

Your Apple responds with the message:

```
WRITING...
YOUR FILE IS 127 BYTES LONG.
```

The number of bytes in your file may be different. Then you will see the question:

```
DO YOU WANT TO E(XIT FROM OR R(ETURN TO THE EDITOR?
```

Type E to exit from the Editor. The Command prompt line reappears.

There are three separate but related steps that must now be taken to run your program. First it must be compiled, that is, translated into the pseudo-code used by the interpreter. Second, the code must be linked to the SYSTEM.LIBRARY routines needed by virtually all FORTRAN programs. Finally, the compiled, linked P-code program must be executed by the Apple in its own native language.

Your file DEMO1 is on FORT2: and so is the Compiler program. All you do at this point is type
C
to get into the Compiler. Almost immediately, you will see:

```
COMPILE WHAT TEXT ?
```

to which you respond:
FORT2:DEMO1

Now it asks you:

```
TO WHAT CODEFILE ?
```

and you type
FORT2:DEMO1

It whirrs a very short while, then asks you about a LISTING FILE. The single-drive user cannot create a listing file, but the listing file can be sent to the console or printer by typing CONSOLE: or PRINTER:. If no listing file is to be sent to the console or printer, we reply by simply pressing the RETURN key.

Now the compilation takes place. If the Compiler discovers mistakes, it will give you a message that will resemble the following:

```
FORTRAN COMPILER II.1 [1.0]

<   0>...
RTEST  [3577 WORDS]
<   3>.......
***** ERROR NUMBER: 161 IN LINE:  10
<SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

Put in FORT1: and type E to go back to the Editor. Since the file was not saved in a workfile, it is necessary to reenter the file. When the file is reentered, type Q to Q(uit, write the file back to FORT2:, and try compiling again.

When your program has been successfully compiled, it is ready for linking. The Linker routine is on FORT1:. You can load the Linker routine into your Apple's memory by putting FORT1: in the drive and typing L. It then asks you

```
HOST FILE ?
```

By this question, the Apple means that it wants you to tell it the name of the file to be linked. Reply with:
FORT2:DEMO1
without pressing the RETURN key yet.

Now put FORT2: in your drive and press RETURN. It will ask you for more information. The routine goes like this, with your responses and the Apple's prompts in ALL CAPITAL LETTERS and any comments in lower case and in parentheses:

```
OPENING FORT2:DEMO1.CODE
LIB FILE ?                       (Any library routines?)
   *                             (* means the system's library)
OPENING *SYSTEM.LIBRARY
LIB FILE ?                       (In case you have more than one)
<RETURN>                         (You don't, so just hit RETURN)
MAP NAME ?
<RETURN>                         (No, just press the RETURN key)
```

Now your Apple flashes red lights. It then puts these messages on the screen:

```
READING MAINSEGX
READING NONAME                   (Your program)
READING RTUNIT                   (Another library routine)
```

Finally it asks

```
OUTPUT FILE?                     (Where shall I put the linked
                                  code?)
```

You can use the same name, but you must now specify that it is to be a CODE file, not a TEXT or DATA file. So you answer
FORT2:DEMO1.CODE

Now the Apple begins showing you these messages:

```
LINKING NONAME        #7       (Your program)
LINKING RTUNIT        #8
LINKING MAINSEGX      #1
```

The Command prompt line comes back. You are now ready to execute the program. To get it running, just type X. You should still have FORT2: in the drive. It will ask you for the file name to be executed. You respond
DEMO1
or whatever FORT2: filename you gave it.

It will come up with the message:

```
RUNNING....
```

and you will get a screenful of squares. Numbers, not the four sided kind!

Try making changes to the program by altering the equation at the end of program line 10. Where it now reads X*X, substitute SQRT(X) to get a sequence of square roots. But first DEMO1 must be transferred to FORT1: to be read in to the Editor. To do this, put in FORT1: and type F to enter the F(iler. Then type T to T(ransfer FORT2:DEMO1 to FORT1:DEMO1. Follow the prompts for putting the diskettes into the drive and typing the file names. When the file is transferred, FORT1: will be in the drive. Type Q to Q(uit the Filer and E to enter the E(ditor.

Use I(nsert and D(lete to make changes, and then Q(uit, put in FORT2:, and write the file to FORT2:DEMO1. Then Compile, Link, and Execute again. This cycle of Edit-Run-Edit-Run is the basis of program development with the Apple FORTRAN System.

Diskette FORT2: contains the text and the code versions of your program. You can use the T(ransfer command in the F(iler to move your developed program onto another diskette for storage. When this is done, clear the workfile by using the N(ew command that is also in the Filer program on FORT1:.

Executing a Program

To execute a previously compiled program, put FORT1: into the disk drive. With the Command prompt line showing, enter the Filer by typing F. When the Filer prompt line appears, put the diskette containing the program codefile that you wish to execute into the disk drive. Then type T for T(ransfer. To the question TRANSFER ? from the system, reply by typing the name of the name of the program's diskette and codefile. For example:
APPLE3:GRAFDEMO.CODE
To the next question from the system, TO WHERE ?, reply with the name of your system diskette, FORT2:, and the same filename or another name, if you wish. For example:
FORT2:GRAFDEMO.CODE

When you are prompted PUT IN FORT2:, follow the instruction and press the spacebar. The program is then transferred onto your system diskette, which is where it must be to be executed. Now type Q to Q(uit the Filer. When the Command prompt appears, type X to eX(ecute the program. When the Apple prompts EXECUTE WHAT FILE?, answer by typing:
FORT2:GRAFDEMO
The program should now run.

Writing a Program

To start a new file in the Editor, put FORT1: into the disk drive. With the Command prompt line showing, type F to enter the Filer. Then type N for N(ew. If you are asked THROW AWAY CURRENT WORKFILE ? type Y for Y(es. When you see the message WORKFILE CLEARED, put in FORT2: and then type Q to Q(uit the Filer. Then put in FORT1: and type E to enter the Editor. This message appears:

```
>EDIT: NO WORKFILE IS PRESENT.
FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
```

Press the RETURN key, and the full Edit prompt line appears. You can now insert text at the cursor position by typing I for I(nsert and then typing your program. Conclude each insertion by pressing CTRL-C.

Delete text at the cursor position by typing D for D(elete and then moving the cursor to erase text. Conclude each deletion by pressing CTRL-C . When you have written a version of your program, type Q to Q(uit the Editor, then type W(rite to tell it you want to write your file out to a new filename.

When it asks you for the name of the file, answer with a FORT2: file name to get your text file and all the rest of the routines you need to compile it on the same diskette. First put in FORT2:, and then press RETURN to start the writing. Then type E to E(xit from the Editor.

With the Command prompt line showing, you can then type C to start the Compiler. Answer its prompts in the appropriate manner, and if you have questions, refer to the earlier part of this appendix.

With compilation complete, and still in Command mode, put in FORT1: and type L to load the Linker routine into memory. Now put in FORT2: and answer the questions as the Apple asks them. When it finishes the task of linking your program to the various library files, it signals completion presenting you with the Command prompt. To run your program, (you are still in the Command mode), just type X and answer the Apple´s question EXECUTE WHAT FILE ?

When a version of your program is complete, you can T(ransfer the text and code forms of the routine to another diskette for later use. Type

```
T
```

while you are in the Filer. The Apple says:

```
        TRANSFER WHAT FILE ?
```

which you answer by entering the name of the diskette and the file name as in the following example. Note you must specify whether it is the CODE or TEXT version you are transferring to a new diskette from FORT2:

```
FORT2:MYFILE.TEXT
```

Now it asks:

```
        TO WHERE?
```

And you key in the message

```
MYDISK:MYFILE.TEXT
```

and press RETURN and take the diskettes in and out of the disk drive as the Apple instructs.

What to Leave in the Drive

When you turn the Apple off, it is a good idea to leave diskette FORT2: in the disk drive. If another diskette or no diskette is in the drive when the Apple is turned on, the drive may spin indefinitely. This will cause unnecessary wear on the drive and the diskette.

APPENDIX A—PART TWO

MULTI-DRIVE OPERATION

154 Introduction
154 Equipment You Will Need
155   More Than Two Disk Drives
155   Numbering the Disk Drives
155 Configuring Your System
157 FORTRAN in Seconds
157 Changing the Date
159 Making Backup Diskettes
159   How We Make Backups
160   Formatting Diskettes
162   Making the Actual Copies
164 Using the System
165   And Now, Some Fun
167   Executing a Program
168   Writing a Program
170   What to Leave in the Drive

INTRODUCTION

Appendix A Part Two covers configuring your Apple FORTRAN System, booting Pascal and FORTRAN for the first time, and using the Editor and Filer programs of the Apple Pascal Operating System to format and make backup copies of diskettes. This appendix is for the user of a system with two or more disk drives.

The procedures described in the section, Configuring Your System, are a tutorial on how to configure your Apple FORTRAN System. The sections after that provide step-by-step instructions for those of you unfamiliar with the Pascal Operating System.

EQUIPMENT YOU WILL NEED

To run Apple FORTRAN, you should have the following equipment:

* Your 48K Apple computer, with a Language Card installed, and at least two disk drives. The first two should be attached to a disk controller card in slot 6. All your disk controller cards should have the PROMs, P5A and P6A, that came with the Language System.

* A TV set or video monitor connected to your Apple.

* The following diskettes and at least two blank diskettes:

FORT1:
FORT2:
APPLE1:
APPLE2:
APPLE3:

Your new Apple FORTRAN System consists of the following diskettes:

* Two identical diskettes labeled FORT2: each containing two files: SYSTEM.COMPILER and SYSTEM.LIBRARY. The SYSTEM.COMPILER is protected from being copied.

* One diskette labeled FORT1: containing the file FORTLIB.CODE.

To create an Apple FORTRAN System, you must transfer certain files from diskettes APPLE1: and APPLE2: of your Apple Pascal System to the two FORTRAN diskettes, FORT1: and FORT2:. The Configuring Your System Section of this appendix explains how to transfer the required files from APPLE1: and APPLE2: to create a working FORTRAN system. It is our recommendation for an Apple FORTRAN System; it is not the only way to do this.

More Than Two Disk Drives

If your system has more than two disk drives, the third drive gets connected to the DRIVE 1 pins on the second controller, which goes in slot 5. A fourth drive is connected to the DRIVE 2 pins on the second controller in slot 5. A fifth and even a sixth drive can be connected to a controller in slot 4, using the DRIVE 1 and DRIVE 2 pins, respectively.

Numbering the Disk Drives

Apple FORTRAN assigns a volume number to each of the disk drives. It is not a bad idea to place tags with these numbers on your disk drives. Here's how the volume numbers are assigned to the various disk drives:

Apple disk drive	FORTRAN volume #
Slot 6, Drive 1	#4:
Slot 6, Drive 2	#5:
Slot 5, Drive 1	#11:
Slot 5, Drive 2	#12:
Slot 4, Drive 1	#9:
Slot 4, Drive 2	#10:

You will find that you can refer to any diskette by either the name of the diskette (e.g., APPLE3:) or by the volume number of the drive in which it is installed (e.g., #11:).

CONFIGURING YOUR SYSTEM

First, plug in the TV or monitor and turn it on. Then plug in the Apple. Put diskette APPLE1: in drive 1 (volume #4) and diskette FORT1: in drive 2 (volume #5). Turn on the Apple.

You need to transfer the following files from APPLE1: to FORT1:

SYSTEM.APPLE
SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.CHARSET
SYSTEM.FILER
SYSTEM.EDITOR

You also need to transfer the following file from APPLE2: to FORT1:

SYSTEM.LINKER

Type F to enter the Filer, and then type T to Transfer files from one diskette to another. The screen asks TRANSFER ?, and you respond by typing
APPLE1:?,FORT1:$
The question mark means ask before transferring if the file is to be transferred. The dollar sign means give the file the same file name on the new diskette. The screen then asks TRANSFER SYSTEM.APPLE, and you respond by typing
Y
The file is then transferred from APPLE1: to FORT1:.

Reply Y to the question to transfer files SYSTEM.PASCAL, SYSTEM.MISCINFO, SYSTEM.EDITOR, SYSTEM.CHARSET, and SYSTEM.FILER. Reply N to the question to transfer files SYSTEM.LIBRARY and SYSTEM.SYNTAX.

Now put APPLE2: in drive 1 and leave FORT1: in drive 2. Type T and respond to the question TRANSFER ?, by typing
APPLE2:SYSTEM.LINKER,FORT1:$
This transfers the file SYSTEM.LINKER to FORT1:. When that file is transferred, you have a complete FORT1: diskette.

You now need to transfer the following files from APPLE1: to FORT2:

SYSTEM.APPLE
SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.CHARSET

Note that FORT2: came to you with two files on it:

SYSTEM.COMPILER
SYSTEM.LIBRARY

Put APPLE1: in drive 1 and FORT2: in drive 2. Type T for Transfer and respond to the question TRANSFER ?, by typing
APPLE1:?,FORT2:$
Answer Y to the question to transfer files SYSTEM.APPLE, SYSTEM.PASCAL, SYSTEM.MISCINFO, and SYSTEM.CHARSET. After transferring those files, you now have a complete FORT2: diskette.

At the conclusion of the transfer process, FORT1: and FORT2: should contain the following files:

FORT1:	FORT2:
SYSTEM.APPLE	SYSTEM.APPLE
SYSTEM.PASCAL	SYSTEM.PASCAL
SYSTEM.MISCINFO	SYSTEM.MISCINFO
SYSTEM.CHARSET	SYSTEM.CHARSET
SYSTEM.FILER	SYSTEM.COMPILER
SYSTEM.LINKER	SYSTEM.LIBRARY
SYSTEM.EDITOR	
FORTLIB.CODE	

FORTRAN IN SECONDS

To start Apple FORTRAN running on your system, place the diskette marked FORT2: in disk drive #4: (slot 6, drive 1). Please note at this point that diskette FORT2: will always be your boot or system diskette. Later you may find exceptions to this rule.

Close the door to disk drive #4:, and turn on the Apple. The message

```
APPLE II
```

appears at the top of your TV or monitor screen, and the IN USE light for disk drive #4: comes on. The disk drive emits a whirring sound to let you know that everything is working. Then the message

```
WELCOME FORT2,  TO
U.C.S.D. PASCAL SYSTEM II.1
CURRENT DATE IS 26-JUL-79
```

appears. The date may be different, and is almost certainly wrong. This is followed in a second or so by a line at the top of the screen:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

This line at the top of the screen is called a prompt line. When you see this prompt line, you know that your Apple computer is running the Apple FORTRAN system.

Starting the system depends only on having either FORTRAN diskette in disk drive #4:. This time, you left the other drives empty; but you will soon discover that the system starts more quickly and quietly if the other drives have FORTRAN diskettes in them. For now, you could put any properly formatted diskettes in the empty disk drives. For example, you could put FORT1: in drive volume #5, APPLE3: in drive volume #11, and a blank but formatted diskette in volume #12. You should get in the habit of always having FORT2: in volume #4: and FORT1: in volume #5:.

Make sure you never put two diskettes with the same names into the system at the same time. This may cause the directories of those diskettes to get scrambled.

CHANGING THE DATE

The date that comes on the diskette will not be correct. It is a good habit to reset the date the first time you use the FORTRAN System on any given day. It only takes a few seconds. Put FORT1: diskette into disk drive #5:. Press F on the keyboard without pressing the RETURN key or any other keys. The screen goes blank, and then this line appears at the top:

```
FILER: G, S, N, L, R, C, T, D, Q [C.1]
```

This is a new prompt line. Prompt lines are named after their first word. The prompt line you first saw was the Command prompt line. This one is the Filer prompt line. Sometimes we say that you are "in the Filer" when this line is at the top of the screen. Each of the letters on the prompt line represents a task that you can ask the Apple to do. For example, to change the date, press D. Again, just type the single key without pressing RETURN.

When you do, another message is put on the screen. It says:

```
DATE SET: <1..31>-<JAN..DEC>-<00..99>
TODAY IS 26-JUL-79
NEW DATE ?
```

It doesn't really mean that today is 26-JUL-79 or whatever date your screen shows, but that the Apple thinks that is today's date. Since you know better, you can change the date to be correct. The correct form for typing the date is shown on the second line of the message: one or two digits giving the day of the month, followed by a minus sign, followed by the first three letters of the name of the month, followed by another minus sign, followed by the last two digits of the current year. Then press the key marked RETURN.

If the month and year are correct, as they will often be when you change the date, all you have to do is type the correct day of the month, and press the RETURN key. The system will assume that you mean to keep the same month and year displayed by the message. If you type a day and a month, the system will assume you mean to keep only the year the same. Go ahead and make the date correct. Your new date is saved on diskette FORT1:, or whatever diskette you have in volume #4, so the system remembers this date the next time you turn the Apple on.

In general, at the top of the screen there will be a prompt line which represents several choices of action. When you type the first letter of one of the choices, either you will be shown a new prompt line giving a further list of choices, or the system will carry out the desired action directly. If you type a letter that does not correspond to one of the choices, the prompt line blinks, but otherwise nothing happens. Type only a single letter to indicate your choice; it is not necessary to press the RETURN key afterward.

Sometimes, as when setting the date, you are asked to type a response of several characters. You tell the system that your response is complete by pressing the RETURN key. If you make a typing error before pressing the RETURN key, you can back up and correct the error by pressing the left-arrow key. You should experiment by making deliberate errors in entering a date, and then erasing the errors with the left-arrow key.

MAKING BACKUP DISKETTES

Ask yourself this question: What would happen to your system if you were to lose or damage one of the system diskettes? Without your system diskettes, you don't own a FORTRAN programming capability. The first thing you should do, therefore, is to make backup copies of FORT1:. Afterward, you should never use the original, but put it someplace where the temperature is moderate, where there is no danger of it getting wet, and where magnetic fields cannot get at it. Since the FORTRAN compiler on FORT2: is protected from being copied, Apple provides a backup copy of FORT2:.

A truly cautious person will keep on hand two backup copies of FORT1:. That way, you will need to use the original only in the very rare case when both of its backup copies are lost. When one copy is lost or damaged, another backup copy is made from the surviving backup copy. If your backups were damaged or erased while in use, find out why they were destroyed before inserting your only surviving copy. If you can't figure out what the problem is, bring your system to the dealer to make sure it is working correctly.

How We Make Backups

The FORTRAN system can copy all or any portion of information from one diskette onto another diskette unless the information is protected from being copied. But the system cannot store information on a raw diskette, just as that diskette comes from the store. Therefore, the system is supplied with a program that allows you to take any 5-inch floppy diskette and format it so that it will work with the Apple FORTRAN system.

If you have been following this chapter by carrying out the instructions on your Apple, the Filer prompt line should be showing at the top of the screen:

```
FILER: G, S, N, L, R, C, T, D, Q [C.1]
```

Type Q on the keyboard to Quit the Filer. When you Quit the Filer, the drives whirr, and soon you see the Command prompt line again:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

There is actually more of this prompt line, off to the right of your TV or monitor. To see the rest of the screen, hold down the key marked CTRL and, while holding it, press the A right alongside it. Or, to be brief, we say press CTRL-A.

You now see

```
K, X(ECUTE, A(SSEM, D(EBUG,? [II.1]
```

This is simply the rest of the line that began COMMAND:. All together, the full prompt line would look like this:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```

The Apple FORTRAN system displays information on a screen that is 80 characters wide, but your TV or monitor shows only the leftmost 40 characters or the rightmost 40 characters at any one time. You use CTRL-A whenever you wish to see if there is more information on the other half of the screen. Repeated pressing of CTRL-A flips back and forth between the left half of the screen and the right half.

Also, sometimes the TV display will seem to be blank. This might mean that you are just staring at the empty right half of the screen. Before you come to the conclusion that something is wrong, always try CTRL-A.

Formatting Diskettes

Place diskette APPLE3: in any available disk drive except drive #4:. This has to be done because the Formatter program is on APPLE3:. Now, with the Command prompt line at the top of the screen, type
X
and the screen responds:

```
          EXECUTE WHAT FILE?
```

You type
APPLE3:FORMATTER
and press the RETURN key.

The disk drive containing APPLE3: whirrs a bit and the screen says:

```
          FOR DISK FORMATTER PROGRAM
          FORMAT WHICH DISK (4, 5, 9..12) ?
```

Now take all the new, blank diskettes that you are going to use with the FORTRAN system. Do not, of course, take any diskettes that have precious information on them, such as the diskettes that came with the FORTRAN system. Place the diskettes in a pile. Their labels should be blank. Make sure that you don't have any diskettes with data in a non-FORTRAN or non-Pascal format, such as BASIC diskettes, unless of course you wish to throw away their contents and reformat them for use with FORTRAN.

Remove the diskette in disk drive #5:. If yours is a two-drive system, you will be removing diskette APPLE3:. Put one of the new, blank diskettes into disk drive #5. Then type
5
and press the key marked RETURN.

If the diskette in drive #5: has already been formatted, you will receive a warning. For example, if you have left APPLE3: in that drive you will be warned with the message

```
          DESTROY DIRECTORY OF APPLE3 ?
```

At this point you can type
N
(which stands for No) without pressing the RETURN key, and your diskette will not be destroyed. Let's assume that you have a new, unformatted diskette. Then you will not get any warning, but the Apple will place this message on the screen:

```
          NOW FORMATTING DISKETTE IN DRIVE 5
```

Disk drive #5: will make some clickings and begin to whirr. The process takes about 30 seconds. When formatting is complete, the screen again shows the message

```
          FORMAT WHICH DISK (4, 5, 9..12) ?
```

Now you have a formatted diskette. We suggest that you write FORTRAN in small letters at the top of the diskette's label, using a marking pen.

Never use a pencil or ballpoint pen, as the pressure of writing may damage the diskette. The label will let you know that the diskette is formatted for use with the Apple FORTRAN system, and you can distinguish it from unformatted diskettes, BASIC diskettes, or diskettes for use with other systems. Diskettes that are formatted for FORTRAN will also store Pascal files and vice-versa because the formatting for FORTRAN and Pascal is identical.

While you are at it, repeat this formatting process on all the new diskettes that you want to use with the Apple FORTRAN System. With each new diskette, place it in drive #5:, type 5 and press the RETURN key.

Note: If you have more than two drives, you can simplify the procedure by putting the next diskette to be formatted into any unoccupied drive. Then, when the system asks

```
          FORMAT WHICH DISK (4, 5, 9..12) ?
```

just type the correct volume number of the drive containing your new, blank diskette, and then press the RETURN key.

When the pile of unformatted diskettes is depleted, and you have written the word FORTRAN on each diskette, answer the question

```
          FORMAT WHICH DISK (4, 5, 9..12) ?
```

with a simple press of the key marked RETURN. You get the message

```
          THAT'S ALL FOLKS...
```

And if you watch the top of the screen, the line

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```

appears. Of course, it doesn't all appear; but you know it's there, and can check with CTRL-A.

Making the Actual Copies

As you have seen, you can get into the Filer by typing F when you have the Command prompt line on the screen. You must have diskette FORT1: in one of the disk drives when you type F to enter the Filer. Put FORT1: in disk drive #5.

The Filer is that portion of the system which allows you to manipulate information on diskettes. One of the Filer's abilities is to transfer information from one diskette to another. To invoke this facility, once you have the Filer prompt line on the screen, type T for T(ransfer.

This is what you see:

```
TRANSFER ?
```

Let's say that you want to make a backup copy of diskette FORT1: by copying it onto one of your newly formatted diskettes. Put FORT1: into any available disk drive, and put a newly formatted diskette into any other drive. Now, answer the question by typing the name of the source diskette to be copied:

```
FORT1:
```

When you press the RETURN key, the computer checks to see that diskette FORT1: is in one of the disk drives. If it is not, you will see the message

```
FORT1:
NO SUCH VOL ON-LINE <SOURCE>
```

In that case, just put FORT1: in a disk drive and type T for Transfer again. If the Apple succeeds in finding FORT1:, it asks you the question:

```
TO WHERE ?
```

Answer this question by typing the name of the diskette that is to become an exact backup copy of FORT1:

```
BLANK:
```

Remember that BLANK: is the name given to all newly formatted diskettes by the Formatter program. The colons (:) that appear after the diskette names are quite significant: They indicate that you are referring to the entire diskette.

After you have told the computer where you want FORT1:'s information transferred and pressed the key marked RETURN, it checks to see that BLANK: is also in one of the disk drives. If it is not, you will see the message

```
PUT IN BLANK:
TYPE <SPACE> TO CONTINUE
```

In that case, put BLANK: into any disk drive except the one containing FORT1:, and press RETURN. When the computer succeeds in finding both the source and the destination diskettes, it says

```
TRANSFER 280 BLOCKS ? (Y/N)
```

This message is mainly there to give you a chance to abandon the transfer if you made a typing error in the names of the source or the destination diskettes. The phrase "280 BLOCKS" means merely the whole diskette. In any case, you type

```
Y
```

All the information on diskette FORT1:, including the diskette's name, will be copied onto diskette BLANK:, completely overwriting BLANK:. Therefore, the computer warns you that you are about to lose any information that might be stored on BLANK:. It says

```
DESTROY BLANK: ?
```

Since you want to turn BLANK: into a perfect copy of FORT1:, the answer is

```
Y
```

The process is under way. It takes about two minutes to copy and check the entire diskette. When copying is done the screen says:

```
FORT1: --> BLANK:
```

By this remark the computer is telling you that the contents of FORT1:, including the diskette's name, have been copied onto the diskette that used to be called BLANK:.

There are now two diskettes with the same name, both in the system at once. This is a risky situation, so remove one of the copies right away. Write FORT1: on the new diskette's label.

Before you type Q to Quit the Filer and return to the Command prompt line, be sure that diskette FORT2: is still in drive #4:. If you Quit the Filer or Editor without FORT2: in place, the computer may stop responding to its keyboard after you type Q; even the RESET key will have no effect. You will have to turn your Apple off, put FORT2: in drive #4:, and reboot the system.

USING THE SYSTEM

To use the Apple FORTRAN System, put FORT2: in the boot drive, volume #4: and FORT1: in drive #5:. With the Command prompt line showing, type E to select the E(dit option. Soon, this message appears:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
:
```

As usual, you must use CTRL-A to see the right half of the message. This message gives you some information and some choices. The first word, >EDIT:, tells you that you are now in the Editor. The next sentence, NO WORKFILE IS PRESENT, tells you that you have not yet used the Editor to create a workfile, which is a scratchpad diskette copy of a program you are working on. If there had been a workfile on FORT2:, that file would have been read into the Editor automatically.

Since there was no workfile to read in, the Editor asks you, FILE? If you now typed the name, including the drive's volume number or the diskette's name, of a .TEXT file stored on either diskette, that textfile would be read into the Editor. However, there are no .TEXT files on FORT1: or FORT2: yet, and besides, you want to write a new program. In parentheses, you are shown how to say that you don't want to read in an old file: <RET> FOR NO FILE . This means that, if you press the Apple's RETURN key, no file will be read in and you can start a new file of your own. That's just what you want to do, so press the Apple's RETURN key. The rest of the message says if you first press the ESC key and THEN press the RETURN key, you'll be sent back to the Command prompt line. When you have pressed the RETURN key, the full Edit prompt line appears:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP
```

The Pascal documention on the Editor (in whichever Pascal manual you have) explains all of these command options in detail; for now you will only need a few of them. The first one you will use is I(nsert, which selects the Editor's mode for inserting new text. Type I to select Insert mode, and yet another prompt line appears:

```
>INSERT: TEXT [<BS> A CHAR,<DEL> A LINE] [<ETX> ACCEPTS, <ESC> ESCAPES]
```

This line of symbols means that anything you type will be placed on the screen, just to the left of the white square cursor. If the cursor is in the middle of a line, the rest of the line is pushed over to make room for the new text. If you make a mistake, just use the left-arrow key to backspace over the error, and then retype. At any time during an insertion, if you press the Apple's ESC key your insertion will be erased. At any time during an insertion, if you press CTRL-C the insertion will be made a permanent part of your file, safe from being erased by the ESC or left-arrow key. You can then type I to enter Insert mode and type more text.

And Now, Some Fun

Now for our program. With the Insert prompt line showing, press the RETURN key a couple of times, to move the cursor down the screen a bit, and then type

C FOOTRAM DEMO

Now press CTRL-C. Your insertion so far is made permanent, and the Edit prompt line reappears. But that's not how to spell FORTRAN. Since you have already pressed CTRL-C, it is too late to backspace over your errors and retype them.

Fortunately, there are other ways. First, let's delete the extra O in FORTRAN. Using the left arrow key, move the cursor left until it is sitting directly on the second O. Then type D to enter the Editor's D(lete mode. Then press the right-arrow key once and the superfluous O will disappear. Now press CTRL-C to make this deletion permanent.

Now let's insert the missing R. The cursor is already correctly placed to make this insertion, being right on top of the T. Type I to enter Insert mode. Ignore the fact that the remainder of the line seems to have suddenly disappeared, and type the missing letter R. When you press CTRL-C to make this insertion permanent, the rest of the line returns:

```
C FORTRAM DEMO
```

The letter M is certainly not correct, so move the cursor right, using the right-arrow key, until it is sitting directly on the M. Now type X to select the Editor's eXchange option. When the eXchange prompt line appears, press the N key and the offending M is instantly transformed. For Insert, Delete and eXchange modes, if you press the ESC key, the alteration is undone, as if it had never happened. If you press CTRL-C, the alteration is made a permanent part of your file. To change that M permanently, press CTRL-C. Finally we have:

```
C FORTRAN DEMO
```

Now you know how to use the Editor's Insert, Delete and eXchange modes to write text and to correct your errors. Try typing the rest of the program that follows into your file for practice. Be sure to accept your insertions, from time to time, by pressing CTRL-C. That way, you minimize your loss if you accidentally press the ESC key. Here is the complete program:

```
C FORTRAN DEMO
      DO 10 I=1,100
      X=I
10    WRITE(*,1)I,X,X*X
1     FORMAT (I6,3X,F7.2,3X,F8.2)
      END
```

When you are typing this program, the punctuation and spelling must be exactly as shown. The indentation of the lines is important in FORTRAN. The indented lines must start in column seven. In the case of our example, start the indented lines under the second R in FORTRAN. You will notice that once you have started a new indentation, the Editor maintains that indentation for you. To move back to the left, just press the left-arrow key before you type anything on the new line.

Now you should save this program. With the Edit prompt line showing, type Q to select the Q(uit option. The following message appears:

```
>QUIT:
     U(PDATE THE WORKFILE AND LEAVE
     E(XIT WITHOUT UPDATING
     R(ETURN TO THE EDITOR WITHOUT UPDATING
     W(RITE TO A FILE NAME AND RETURN
```

Type U to create a workfile diskette copy of your program. Future versions of this file will be Updates. This workfile is a file on your boot diskette FORT2:, called SYSTEM.WRK.TEXT. The Apple says

```
WRITING...
YOUR FILE IS 127 BYTES LONG.
```

The number of bytes in your file may be a little different. Then the Command prompt line appears. Now type R to select the R(un option. This automatically calls the FORTRAN compiler for you, since the workfile contains text. If you have typed the program perfectly, the following message appears:

```
COMPILING...
```

The compiler assumes that it is the workfile (SYSTEM.WRK.TEXT) on the boot diskette that is to be compiled.

If the compiler discovers mistakes, it will give you a message such as

```
FORTRAN COMPILER II.1 [1.0]

<   0>...
RTEST  [3577 WORDS]
<   3>.......
***** ERROR NUMBER: 161 IN LINE:  10
<SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

Don't despair; just type E for E(dit. Your workfile will be automatically read back into the Editor so that you can make repairs. A brief error message corresponding to the error message above will appear at the top of the screen, along with instructions which say to press the spacebar when you want to start editing. Make any necessary changes using I(nsert and D(elete. Then Q(uit, U(pdate the workfile, and R(un your program again, by typing Q U R. You can type them in rapid succession if you like, since the Apple can store up several commands in advance.

During the compilation process, you will be asked LISTING FILE? by your Apple. Listing files are discussed in Chapter 4 of this manual. For now, just press <RETURN> to go on. Note that the listing file cannot be written to the same volume as the output file.

The Apple will tell you that compilatiion is successful with the message: "NNN LINES. 0 ERRORS." Upon successful compilation of your program, the Linker is called to link your coded program to the other files needed to make up a complete, executable program in P-Code. After linking, the program is run.

Try making changes in the TEXT file. Just get into E(dit, make the changes, Q(uit the Editor by U(pdating the workfile, and then R(un the changed program. This cycle of Edit-Run-Edit-Run is the basis of all program development in the Apple FORTRAN System.

The workfile on FORT2: now contains the text version of your program in a file named SYSTEM.WRK.TEXT, and the compiled P-code version of your program in another file named SYSTEM.WRK.CODE. When your program is running as you want it to, you should save the text and code workfile under other file names. With the Command promptline showing, type F to enter the Filer. When the Filer prompt line appears, type S for S(ave. You will be asked

```
SAVE AS ?
```

and you should respond by typing a filename.

For example, you might type
DEMO
This changes the names of the workfile from SYSTEM.WRK.TEXT to DEMO.TEXT, and from SYSTEM.WRK.CODE to DEMO.CODE. If you want to keep a permanent copy of your program on another diskette, you should now use the T(ransfer command to transfer DEMO.TEXT and DEMO.CODE, one at a time, to the other diskette. Put the source diskette in one drive and the destination diskette in the other drive.

Executing a Program

To execute a previously compiled and linked program in FORTRAN, just leave FORT2: in the boot drive, and place your program diskette in any other drive. Now type
X
to start eXecution. The Apple will ask you

```
EXECUTE WHAT PROGRAM ?
```

Tell it the diskette name or number, and the file name of the program. Here are two examples:

```
DISK1:FILE2
   #5:FILE3
```

Either one will work. The Apple will load the codefile into memory and begin execution automatically.

Writing a Program

With multi-drive systems, there are two possible ways to create and run a program. Both begin by calling up the Editor and typing in the text of a FORTRAN program, as you did for the demonstration program in this appendix. When this is done, Q(uit and U(pdate the workfile, which will be automatically stored on FORT2: in the boot drive. At this point, you have a choice: You can either R(un the program or separately compile, link, and execute it. R(un is simply a command that automatically first compiles, then links, and finally executes the workfile called FORT2:SYSTEM.WRK.CODE. When the compilation part is done, the result is stored as FORT2:SYSTEM.WRK.CODE. After linking to the necessary library or other routines, the final code, ready for execution, is stored in the same FORT2: codefile and automatically executed.

The second option is to type C for Compile instead of R. This starts compilation in a less automatic mode. The Apple will ask you first

```
COMPILE WHAT TEXT ?
```

You would then tell it
SYSTEM.WRK.TEXT
It knows that it's on FORT2: because that's your boot diskette.

You will be asked for the name of the output file:

```
OUTPUT FILE ?
```

You can give it any name you prefer, but usually you will supply it with the workfile:
SYSTEM.WRK.CODE

Soon it will ask you

```
LISTING FILE ?
```

You don't need one here, so you press RETURN, or type CONSOLE: or PRINTER: to send the listing file to the console or printer.

When compilation is complete, you will see the Command prompt line reappear on the screen at the top. Now you are ready to link, so type
L
to call the Linker routine. Soon the linking process starts, and you will be asked

```
HOST FILE ?
```

Give it the name under which your program is stored:
SYSTEM.WRK
and soon it will ask you:

```
LIB FILE ?
```

You will normally reply:
* or SYSTEM.LIBRARY or FORT2:SYSTEM.LIBRARY
and the question will be repeated. This time, just type RETURN with no other entry. Now it will ask you

```
MAP NAME ?
```

Just press RETURN, as you don't want to do anything here.

Finally, it will ask you

```
OUTPUT FILE ?
```

Again, you can specify any diskette and filename, but for our purposes enter
SYSTEM.WRK.CODE

Now the linking process is automatic. When it is complete, the Command prompt will appear once more at the top of your screen.

To test your compiled and linked program, type
X
When the Apple asks you

```
EXECUTE WHAT PROGRAM ?
```

Answer with the name of the codefile:
SYSTEM.WRK

Once you have your program running and debugged, you may store it on any formatted diskette by using the T(ransfer command to transfer only the codefile. If you want to save the textfile and the codefile on the same disk, the suggested technique is to use the S(ave command from the Filer. Get into the Filer and type S.

The Apple soon asks

```
SAVE AS ?
```

and you answer with the diskette and file name, but do not specify either CODE or TEXT. For example, you could type
DISK34:FOO2U2
The Apple will now store both your text and code in two separate files on DISK34:. The code will be under DISK34:FOO2U2.CODE, while the text, your FORTRAN source program, will be filed under DISK34:FOO2U2.TEXT.

What to Leave in the Drive

When you turn the Apple off, it is a good idea to leave diskette FORT2: in the boot drive. If some other diskette or no diskette is in the drive when the Apple is turned on, the drive will spin indefinitely. This places unnecessary wear on the drive and the diskette.

Note that you can boot with FORT1: as your boot diskette. If you do this, you cannot switch the two diskettes later without rebooting. Also note that FORT2: containing the SYSTEM.LIBRARY must be in the boot drive when Linking if the Run option is used, and that if you are going to run programs from the workfile, that workfile must be on FORT2:. If you want to link manually, FORT2: can be in any drive.

APPENDIX B

FORTRAN ERROR MESSAGES

172 Compile-time Error Messages
176 Run-time Error Messages

COMPILE-TIME ERROR MESSAGES

1 Fatal error reading source block
2 Nonnumeric characters in label field
3 Too many continuation lines
4 Fatal end of file encountered
5 Labeled continuation line
6 Missing field on $ compiler directive line
7 Unable to open listing file specified on $ compiler directive line
8 Unrecognizable $ compiler directive
9 Input source file not valid textfile format
10 Maximum depth of include file nesting exceeded
11 Integer constant overflow
12 Error in real constant
13 Too many digits in constant
14 Identifier too long
15 Character constant extends to end of line
16 Character constant zero length
17 Illegal character in input
18 Integer constant expected
19 Label expected
20 Error in label
21 Type name expected (INTEGER, REAL, LOGICAL, or CHARACTER[*n])
22 Integer constant expected
23 Extra characters at end of statement
24 '(' expected
25 Letter IMPLICIT'ed more than once
26 ')' expected
27 Letter expected
28 Identifier expected
29 Dimension(s) required in DIMENSION statement
30 Array dimensioned more than once
31 Maximum of 3 dimensions in an array
32 Incompatible arguments to EQUIVALENCE
33 Variable appears more than once in a type specification statement
34 This identifier has already been declared
35 This intrinsic function cannot be passed as an argument
36 Identifier must be a variable
37 Identifier must be a variable or the current FUNCTION
38 '/' expected
39 Named COMMON block already saved
40 Variable already appears in a COMMON block
41 Variables in two different COMMON blocks cannot be equivalenced
42 Number of subscripts in EQUIVALENCE statement does not agree with variable declaration
43 EQUIVALENCE subscript out of range
44 Two distinct cells EQUIVALENCE'd to the same location in a COMMON block
45 EQUIVALENCE statement extends a COMMON block in the negative direction
46 EQUIVALENCE statement forces a variable to two distinct locations, not in a COMMON block
47 Statement number expected
48 Mixed CHARACTER and numeric items not allowed in same COMMON block
49 CHARACTER items cannot be EQUIVALENCE'd with non-character items
50 Illegal symbol in expression
51 Can't use SUBROUTINE name in an expression
52 Type of argument must be INTEGER or REAL
53 Type of argument must be INTEGER, REAL, or CHARACTER
54 Types of comparisons must be compatible
55 Type of expression must be LOGICAL
56 Too many subscripts
57 Too few subscripts
58 Variable expected
59 '=' expected
60 Size of EQUIVALENCE'd CHARACTER items must be the same
61 Illegal assignment - types do not match
62 Can only call SUBROUTINES
63 Dummy parameters cannot appear in COMMON statements
64 Dummy parameters cannot appear in EQUIVALENCE statements
65 Assumed-size array declarations can only be used for dummy arrays
66 Adjustable-size array declarations can only be used for dummy arrays
67 Assumed-size array dimension specifier must be last dimension
68 Adjustable bound must be either parameter or in COMMON prior to appearance
69 Adjustable bound must be simple integer variable
70 Cannot have more than 1 main program
71 The size of a named COMMON must be the same in all procedures
72 Dummy arguments cannot appear in DATA statements
73 COMMON variables cannot appear in DATA statements
74 SUBROUTINE names, FUNCTION names, INTRINSIC names, etc. cannot appear in DATA statements
75 Subscript out of range in DATA statement
76 Repeat count must be >= 1
77 Constant expected
78 Type conflict in DATA statement
79 Number of variables does not match number of values in DATA statement list
80 Statement cannot have label
81 No such INTRINSIC function
82 Type declaration for INTRINSIC function does not match actual type of INTRINSIC function
83 Letter expected
84 Type of FUNCTION does not agree with a previous call
85 This procedure has already appeared in this compilation
86 This procedure has already been defined to exist in another unit via a $USES command
87 Error in type of argument to an INTRINSIC FUNCTION
88 SUBROUTINE/FUNCTION was previously used as a FUNCTION/SUBROUTINE
89 Unrecognizable statement
90 Functions cannot be of type CHARACTER
91 Missing END statement

92 A program unit cannot appear in a $SEPARATE compilation
93 Fewer actual arguments than formal arguments in FUNCTION/SUBROUTINE call
94 More actual arguments than formal arguments in FUNCTION/SUBROUTINE call
95 Type of actual argument does not agree with type of format argument
96 The following procedures were called but not defined:
97 This procedure was already defined by a $EXT directive
98 Maximum size of type CHARACTER is 255, minimum is 1

100 Statement out of order
101 Unrecognizable statement
102 Illegal jump into block
103 Label already used for FORMAT
104 Label already defined
105 Jump to format label
106 DO statement forbidden in this context
107 DO label must follow DO statement
108 ENDIF forbidden in this context
109 No matching IF for this ENDIF
110 Improperly nested DO block in IF block
111 ELSEIF forbidden in this context
112 No matching IF for ELSEIF
113 Improperly nested DO or ELSE block
114 '(' expected
115 ')' expected
116 THEN expected
117 Logical expression expected
118 ELSE statement forbidden in this context
119 No matching IF for ELSE
120 Unconditional GOTO forbidden in this context
121 Assigned GOTO forbidden in this context
122 Block IF statement forbidden in this context
123 Logical IF statement forbidden in this context
124 Arithmetic IF statement forbidden in this context
125 ',' expected
126 Expression of wrong type
127 RETURN forbidden in this context
128 STOP forbidden in this context
129 END forbidden in this context
131 Label referenced but not defined
132 DO or IF block not terminated
133 FORMAT statement not permitted in this context
134 FORMAT label already referenced
135 FORMAT must be labeled
136 Identifier expected
137 Integer variable expected
138 'TO' expected
139 Integer expression expected
140 Assigned GOTO but no ASSIGN statements
141 Unrecognizable character constant as option
142 Character constant expected as option
143 Integer expression expected for unit designation
144 STATUS option expected after ',' in CLOSE statement
145 Character expression as filename in OPEN
146 FILE= option must be present in OPEN statement
147 RECL= option specified twice in OPEN statement
148 Integer expression expected for RECL= option in OPEN statement
149 Unrecognizable option in OPEN statement
150 Direct access files must specify RECL= in OPEN statement
151 Adjustable arrays not allowed as I/O list elements
152 End of statement encountered in implied DO, expressions beginning with '(' not allowed as I/O list elements
153 Variable required as control for implied DO
154 Expressions not allowed as reading I/O list elements
155 REC= option appears twice in statement
156 REC= expects integer expression
157 END= option only allowed in READ statement
158 END= option appears twice in statement
159 Unrecognizable I/O unit
160 Unrecognizable format in I/O statement
161 Options expected after ',' in I/O statement
162 Unrecognizable I/O list element
163 Label used as format but not defined in format statement
164 Integer variable used as assigned format but no ASSIGN statements
165 Label of an executable statement used as a format
166 Integer variable expected for assigned format
167 Label defined more than once as format
169 Function calls require '( )'

200 Error in reading $USES file
201 Syntax error in $USES file
202 SUBROUTINE/FUNCTION name in $USES file has already been declared
203 FUNCTIONS cannot return values of type CHARACTER
204 Unable to open $USES file
205 Too many $USES statements
206 No .TEXT info for this unit in $USES file
207 Illegal segment kind in $USES file
208 There is no such unit in this $USES file
209 Missing UNIT name in $USES statement
210 Extra characters at end of $USES directive
211 Intrinsic units cannot be overlayed
212 Syntax error in $EXT directive
213 A SUBROUTINE cannot have a type
214 SUBROUTINE/FUNCTION name in #EXT directive has already been define

400 Code file write error
401 Too many entries in JTAB
402 Too many SUBROUTINES/FUNCTIONS in segment
403 Procedure too large (code buffer too small)
404 Insufficient room for scratch file on system disk
405 Read error on scratch file

RUN-TIME ERROR MESSAGES

600 Format missing final ')'
601 Sign not expected in input
602 Sign not followed by digit in input
603 Digit expected in input
604 Missing N or Z after B in format
605 Unexpected character in format
606 Zero repetition factor in format not allowed
607 Integer expected for w field in format
608 Positive integer required for w field in format
609 '.' expected in format
610 Integer expected for d field in format
611 Integer expected for e field in format
612 Positive integer required for e field in format
613 Positive integer required for w field in A format
614 Hollerith field in format must not appear for reading
615 Hollerith field in format requires repetition factor
616 X field in format requires repetition factor
617 P field in format requires repetition factor
618 Integer appears before '+' or '-' in format
619 Integer expected after '+' or '-' in format
620 P format expected after signed repetition factor in format
621 Maximum nesting level for formats exceeded
622 ')' has repetition factor in format
623 Integer followed by ',' illegal in format
624 '.' is illegal format control character
625 Character constant must not appear in format for reading
626 Character constant in format must not be repeated
627 '/' in format must not be repeated
628 '$' in format must not be repeated
629 BN or BZ format control must not be repeated
630 Attempt to perform I/O on unknown unit number
631 Formatted I/O attempted on file opened as unformatted
632 Format fails to begin with '('
633 I format expected for integer read
634 F or E format expected for real read
635 Two '.' characters in formatted real read
636 Digit expected in formatted real read
637 L format expected for logical read
639 T or F expected in logical read
640 A format expected for character read
641 I format expected for integer write
642 w field in F format not greater than d field + 1
643 Scale factor out of range of d field in E format
644 E or F format expected for real write
645 L format expected for logical write
646 A format expected for character write
647 Attempt to do unformatted I/O to a unit opened as formatted
648 Unable to write blocked output, possibly no room on device for file
649 Unable to read blocked input
650 Error in formatted textfile, no <cr> in last 512 bytes
651 Integer overflow on input
652 Too many bytes read out of direct access unit record
653 Incorrect number of bytes read from a direct access unit record
654 Attempt to open direct access unit on unblocked device
655 Attempt to do external I/O on a unit beyond end of file record
656 Attempt to position a unit for direct access on a nonpositive record number
657 Attempt to do direct access to a unit opened as sequential
658 Attempt to position direct access unit on unblocked device
659 Attempt to position direct access unit beyond end of file for reading
660 Attempt to backspace unit connected to unblocked device
661 Attempt to backspace sequential, unformatted unit
662 Argument to ASIN or ACOS out of bounds (ABS(X) .GT. 1.0)
663 Argument to SIN or COS too large (ABS(X) .GT. 10E6)
664 Attempt to do unformatted I/O to internal unit
665 Attempt to put more than one record into internal unit
666 Attempt to write more characters to internal unit than its length
667 EOF called on unknown unit
697 Integer variable not currently assigned a format label
698 End of file encountered on read with no END= option
699 Integer variable not ASSIGNed a label used in assigned goto

1000+ Compiler debug error messages - should never appear in correct programs

APPENDIX C

TABLES

180 Unit Identifiers
181 Intrinsic Functions
184 Transcendental Functions
185 Lexical Comparisons
186 ASCII Character Codes

UNIT IDENTIFIERS

The identifiers listed below are declared or defined only if your program $USES the unit under which they are listed. If your program does not use the particular unit, you can use the identifier names for other purposes.

TURTLEGRAPHICS UNIT IDENTIFIERS

CHARTY
DRAWBL
FILLSC
GRAFMO
INITTU
MOVE
MOVETO
PENCOL
SCREEN
TEXTMO
TURN
TURNTO
TURTLA
TURTLX
TURTLY
VIEWPO
WCHAR

APPLESTUFF UNIT IDENTIFIERS

BUTTON
PADDLE
TTLOUT
KEYPRE
RANDOM
NOTE
RANDOI

INTRINSIC FUNCTIONS

This is the list of intrinsic functions available in Apple FORTRAN. The type of the result is listed first, followed by the name of the function in all caps, and the type of the argument(s) in parentheses.

TYPE CONVERSION

integer INT (real)
integer IFIX (real)

Converts from real to integer.

real REAL (integer)
real FLOAT (integer)

Converts integer to real.

integer ICHAR (character)

Converts the first character of the argument string to its corresponding integer value, according to the ASCII collating sequence. CHAR is the reverse of ICHAR.

TRUNCATION

real AINT (real)

Removes the fractional part of a real variable, returning the result as a real.

NEAREST WHOLE NUMBER

real ANINT (real)

Finds nearest whole number, that is: INT(argument+.5) if the argument is .GE. 0, otherwise INT(argument-.5).

NEAREST INTEGER

integer NINT (real)

Finds nearest integer: INT(argument+.5) if argument .GE. 0, otherwise INT(argument-.5).

ABSOLUTE VALUE

integer IABS (integer)
real ABS (real)

Returns absolute value.

REMAINDERING

integer MOD (integer_a, integer_b)
real AMOD (real_a, real_b)

Returns the result of a-INT(a/b)*b.

TRANSFER OF SIGN

integer ISIGN (integer_a, integer_b)
real SIGN (real_a, real_b)

Converts sign of a according to sign of b. Result is $|a|$ if b .GE. 0, otherwise result is $-|a|$.

POSITIVE DIFFERENCE

integer IDIM (integer_a, integer_b)
real DIM (real_a, real_b)

Takes the positive difference of the two arguments. Result is a-b if a .GE. b, otherwise result is 0.

MAGNITUDE COMPARISON

integer MAX0 (integer_a, integer_b,... integer_n)
real AMAX1 (real_a, real_b,... real_n)
real AMAX0 (integer_a, integer_b,... integer_n)
integer MAX1 (real_a, real_b,... real_n)

Returns the largest (most positive) of all the actual arguments.

integer MIN0 (integer_a, integer_b,... integer_n)
real AMIN1 (real_a, real_b,... real_n)
real AMIN0 (integer_a, integer_b,... integer_n)
integer MIN1 (real_a, real_b,... real_n)

Returns the smallest (least positive) of all the actual arguments.

SQUARE ROOT

real SQRT (real)

Returns the square root of the argument.

TRANSCENDENTAL FUNCTIONS

The transcendental functions all return real results, and all arguments are real. The arguments to SIN, COS, TAN, SINH, COSH and TANH are in radians. The results of ASIN, ACOS, ATAN and ATAN2 are in radians.

EXP(a)	Exponential
ALOG(a)	Natural logarithm
ALOG10(a)	Common logarithm
SIN(a)	Sine
COS(a)	Cosine
TAN(a)	Tangent
ASIN(a)	Arcsine
ACOS(a)	Arccosine
ATAN(a)	Arctangent
ATAN2(a, b)	Arctan(a/b)
SINH(a)	Hyperbolic Sine
COSH(a)	Hyperbolic Cosine
TANH(a)	Hyperbolic Tangent

LEXICAL COMPARISONS

The lexical functions all return logical results, and all arguments are character strings.

LGE(a, b)

Returns true if the two strings are identical, or if the first non-identical character in string a has an ASCII collating sequence number greater than or equal to the corresponding character in string b.

LGT(a, b)

Returns true if the first non-identical character in string a has an ASCII collating sequence number greater than the corresponding character in string b.

LLE(a, b)

Returns true if the two strings are identical, or if the first non-identical character in string a has an ASCII collating sequence number less than the corresponding character in string b.

LLT(a, b)

Returns true if the first non-identical character in string a has an ASCII collating sequence number less than the corresponding character in string b.

ASCII CHARACTER CODES

Code		Char	Code		Char	Code		Char	Code		Char
Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	32	20	SP	64	40	@	96	60	
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	89	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

APPENDIX D

FORTRAN SYNTAX DIAGRAMS

FORTRAN SYNTAX DIAGRAMS

The following charts describe the syntax of Apple FORTRAN. They are only an informal guide to its use, however; they are not a rigorous definition of the language.

These charts are revisions of the charts given in the ANSI 77 manual, but made to conform to the subset rather than the full language, and to take into account those small areas where Apple FORTRAN is different from the subset. As with the ANSI charts, these charts are not accurate with respect to the use of blanks, the use of continuation lines, comment lines or context dependent features. Wherever there is a conflict between these charts and the formal description earlier in this manual, the formal description is to be taken instead.

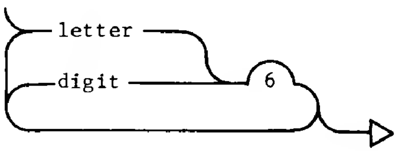
These charts are sometimes called "railroad diagrams," and in fact it is more truth than not. You typically enter any given chart from the upper left, where the name of the syntactical object represented by the chart is given. The lines connecting the different objects in the chart indicate all the possible ways that these objects can be connected in an Apple FORTRAN program. Where they branch you may take any fork of the branch; where two paths unite, you must go in the direction that the line bends when it unites.

Any numbers embedded in half-circles on a line indicate that the line may be traversed at most that many times. If a number appears in a complete circle, then that path must be traversed exactly that many times.

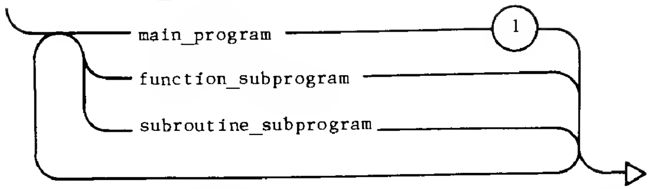
Objects appearing in upper case are FORTRAN key words, such as DO, or OPEN. They must appear in programs as written (although they may be written in lower case). Other uncapitalized objects are names for syntactic categories. For instance, a label is a kind of syntactic category in FORTRAN. The names chosen for these categories are adopted from the ANSI standard where possible, but even the standard sometimes abreviates terms, such as where "statement label" is shortened to "label," or "expression" to "expr."

For instance, the syntax of a symbolic name is the list of all legal character sequences that can be called symbolic names. Specifically, a symbolic name must start with a letter, and may be followed by up to 5 more letters or digits. Here is its syntax diagram:

symbolic_name:


letter
digit
6

1 executable_program:


main_program
1
function_subprogram
subroutine_subprogram

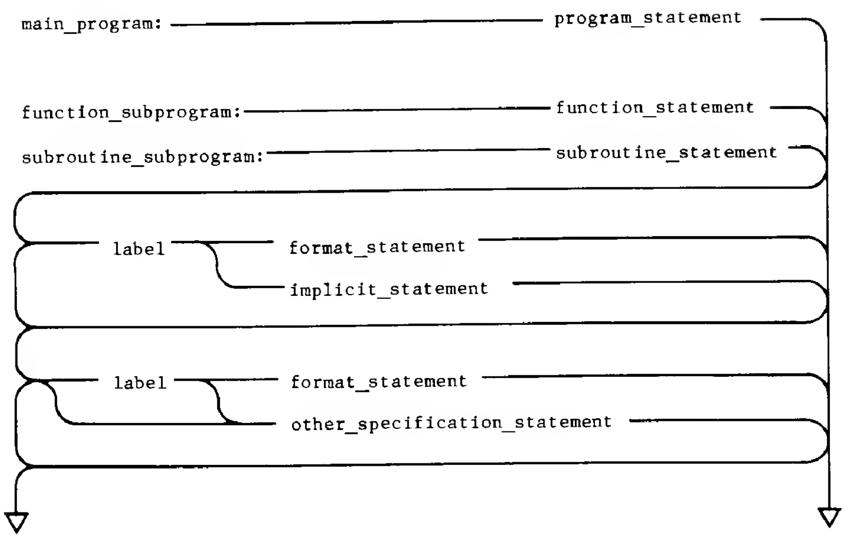
(1) An executable program must contain one and only one main program.

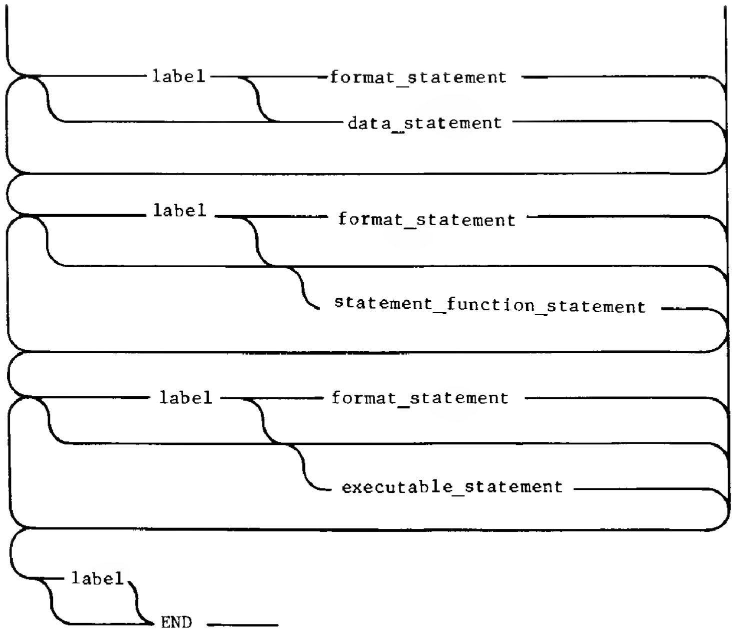
An executable program may contain external procedures specified by means other than FORTRAN.

2

3

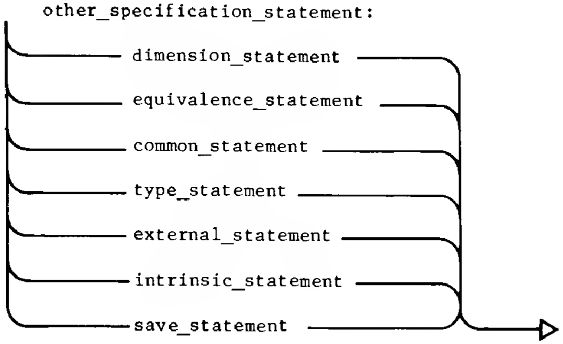
4


main_program:
program_statement
function_subprogram:
function_statement
subroutine_subprogram:
subroutine_statement
label
format_statement
implicit_statement
label
format_statement
other_specification_statement

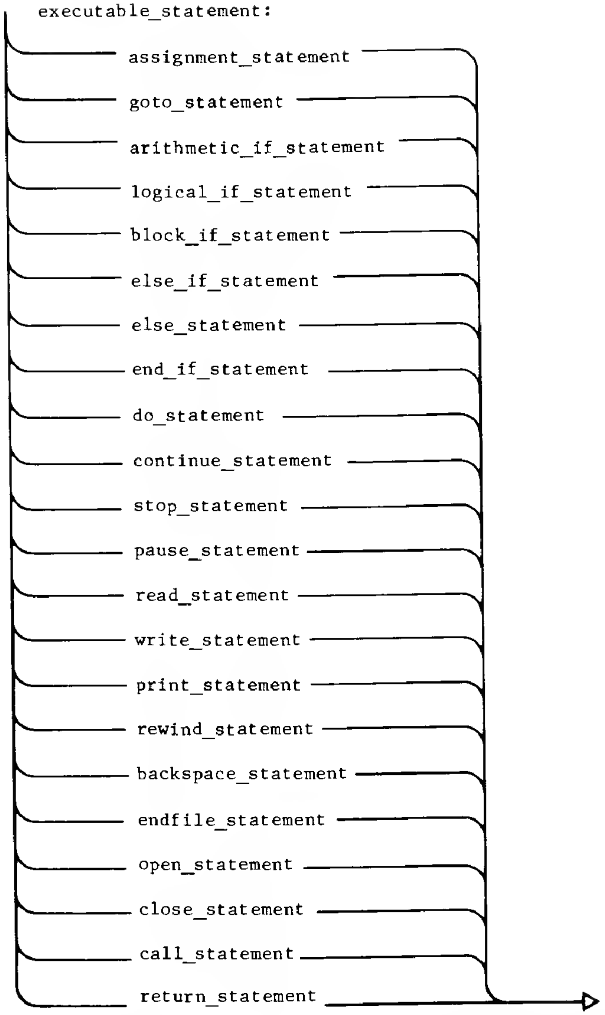

label
format_statement
data_statement
label
format_statement
statement_function_statement
label
format_statement
executable_statement
label
END

(2) A main program may not contain a RETURN statement.

5 other_specification_statement:

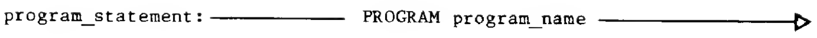

dimension_statement
equivalence_statement
common_statement
type_statement
external_statement
intrinsic_statement
save_statement

6 executable_statement:

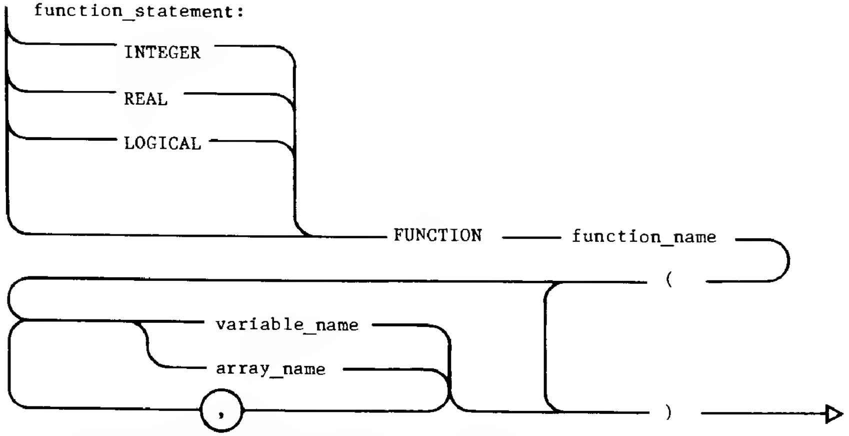

assignment_statement
goto_statement
arithmetic_if_statement
logical_if_statement
block_if_statement
else_if_statement
else_statement
end_if_statement
do_statement
continue_statement
stop_statement
pause_statement
read_statement
write_statement
print_statement
rewind_statement
backspace_statement
endfile_statement
open_statement
close_statement
call_statement
return_statement

(6) An END statement is also an executable statement and must appear as the last statement of a program unit, followed by exactly one carraige return.

7

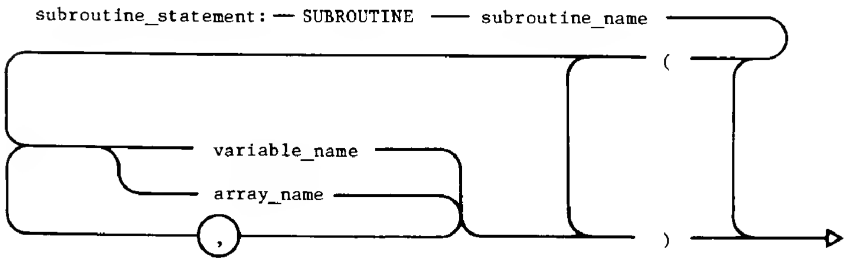

program_statement:
PROGRAM program_name

8

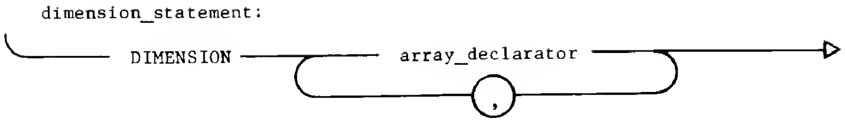

function_statement:
INTEGER
REAL
LOGICAL
FUNCTION
function_name
(
variable_name
array_name
,
)

(8) The parentheses must appear in a FUNCTION statement, even if there are no arguments.

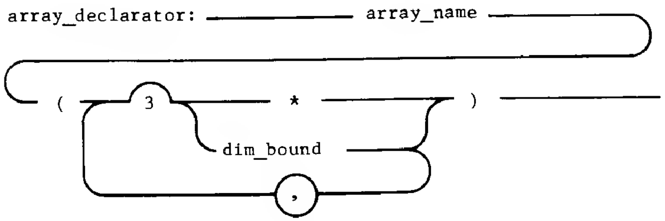
9


subroutine_statement:
SUBROUTINE
subroutine_name
(
variable_name
array_name
,
)

10

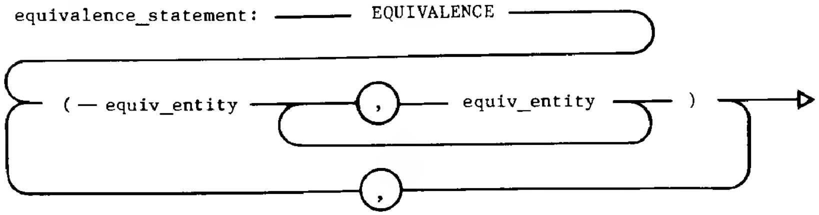

dimension_statement:
DIMENSION
array_declarator
,

11

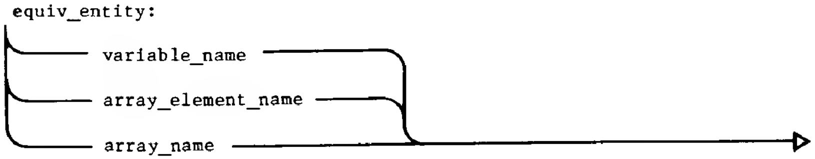

array_declarator:
array_name
(
3
*
)
dim_bound
,

(11) Only a dummy array declarator may contain an asterisk.

12

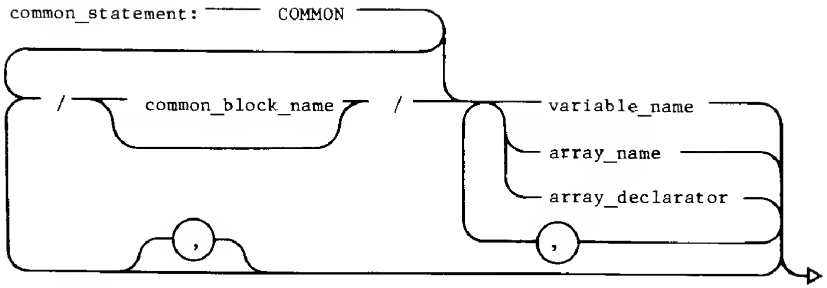

equivalence_statement:
EQUIVALENCE
(— equiv_entity
,
equiv_entity
)
,

13

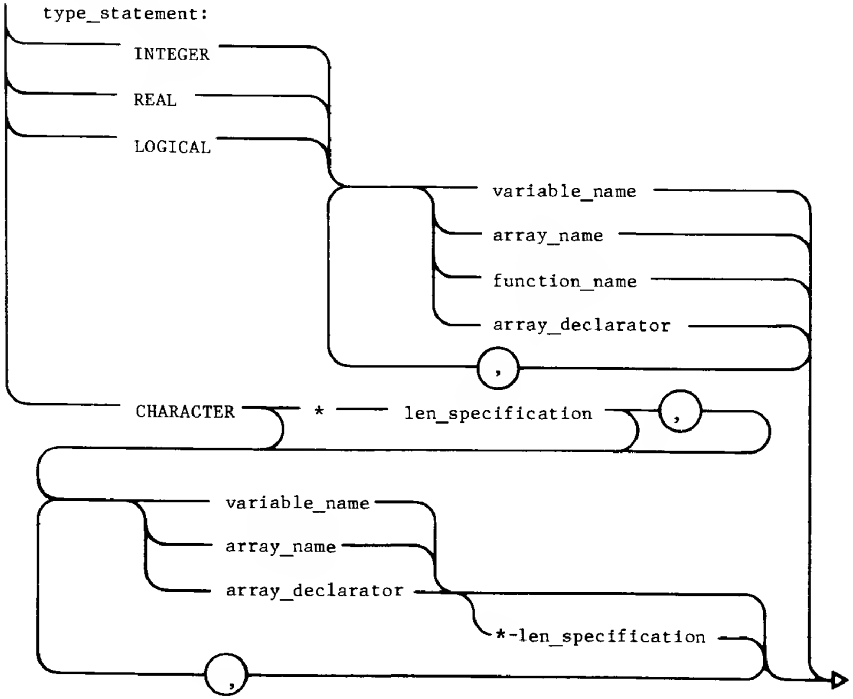

equiv_entity:
variable_name
array_element_name
array_name

(13) A subscript expression in an EQUIVALENCE statement must be an integer constant.

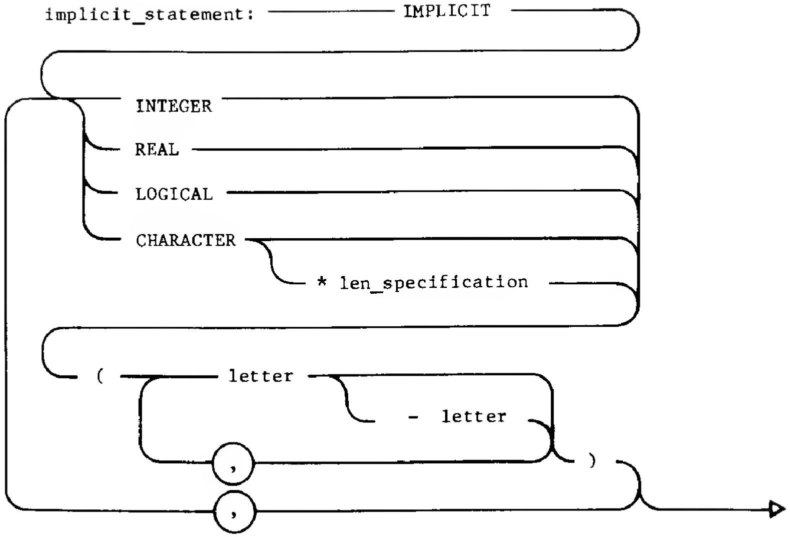
14


common_statement:
COMMON
/
common_block_name
/
variable_name
array_name
array_declarator
,
,

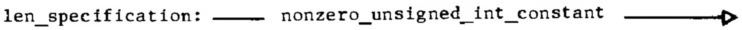
15


type_statement:
INTEGER
REAL
LOGICAL
variable_name
array_name
function_name
array_declarator
,
CHARACTER
*
len_specification
,
variable_name
array_name
array_declarator
*-len_specification
,

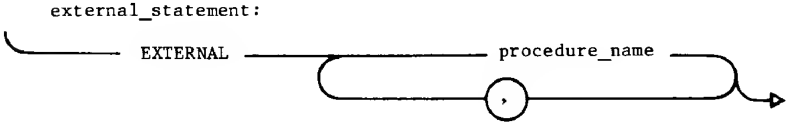
16


implicit_statement:
IMPLICIT
INTEGER
REAL
LOGICAL
CHARACTER
* len_specification
(
letter
- letter
,
)
,

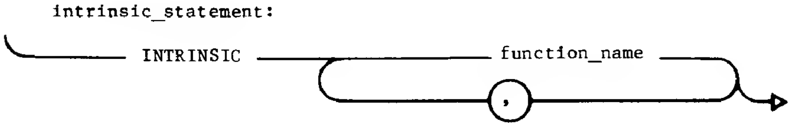
17


len_specification:
nonzero_unsigned_int_constant

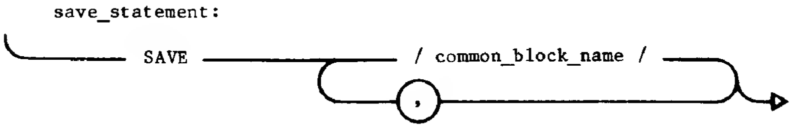
18


external_statement:
EXTERNAL
procedure_name
,

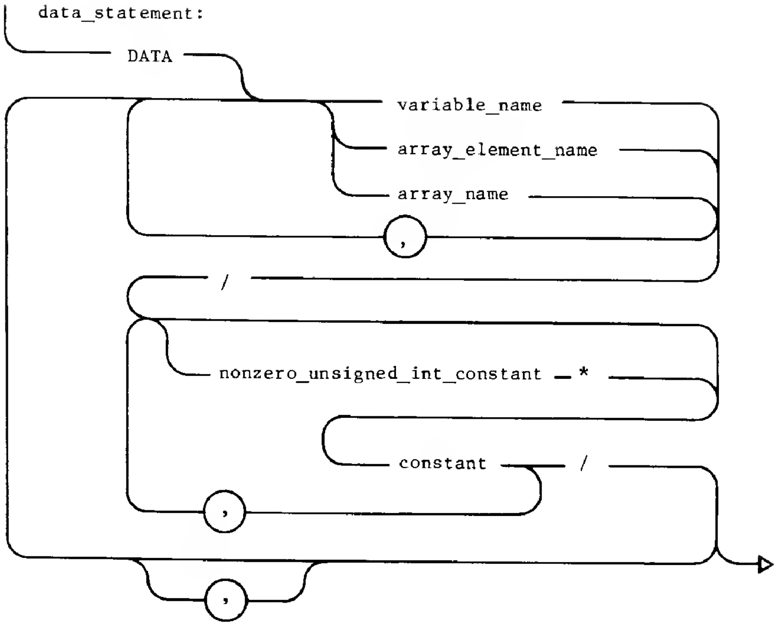
19


intrinsic_statement:
INTRINSIC
function_name
,

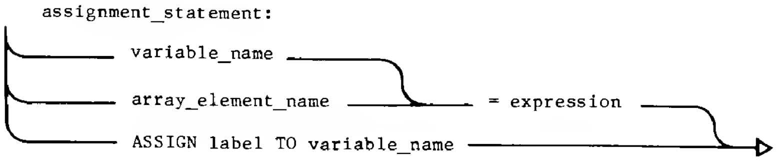
20


save_statement:
SAVE
/ common_block_name /
,

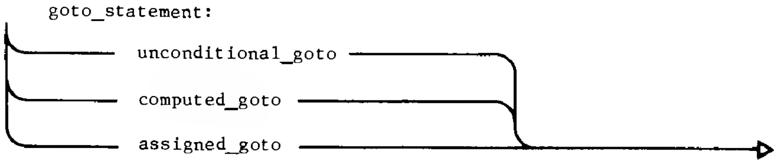
21


data_statement:
DATA
variable_name
array_element_name
array_name
,
/
nonzero_unsigned_int_constant
*
constant
/
,
,

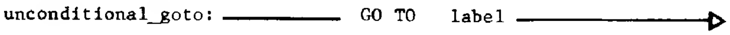
22


assignment_statement:
variable_name
array_element_name
= expression
ASSIGN label TO variable_name

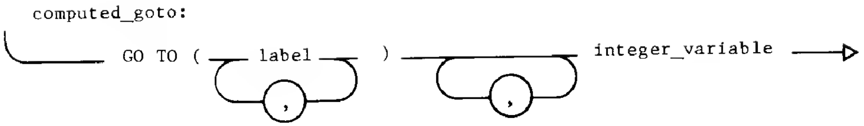
23


goto_statement:
unconditional_goto
computed_goto
assigned_goto

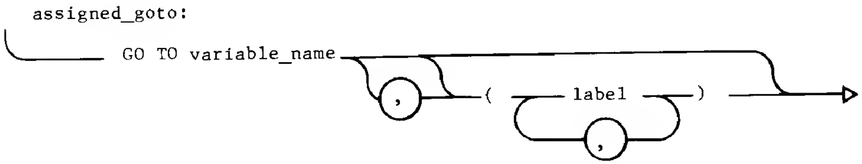
24


unconditional_goto:
GO TO
label

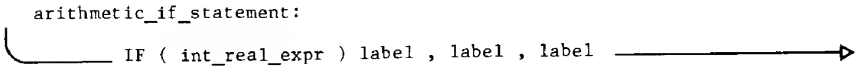
25


computed_goto:
GO TO (
label
,
)
,
integer_variable

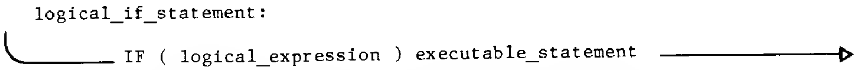
26


assigned_goto:
GO TO variable_name
,
(
label
,
)

27

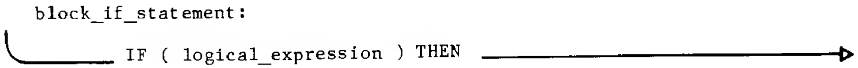

arithmetic_if_statement:
IF (int_real_expr) label , label , label

28

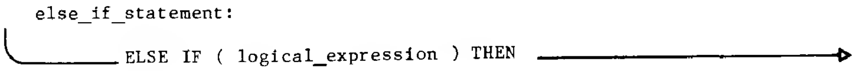

logical_if_statement:
IF (logical_expression) executable_statement

(28) The executable statement of a logical IF statement must not be a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

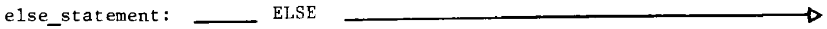
29


block_if_statement:
IF (logical_expression) THEN

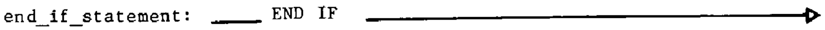
30


else_if_statement:
ELSE IF (logical_expression) THEN

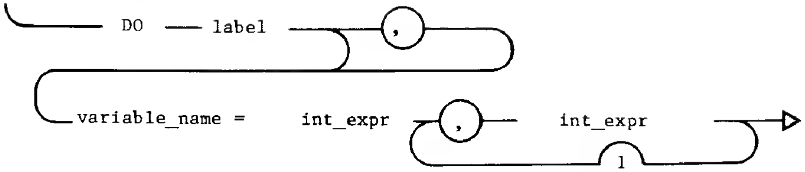
31


else_statement:
ELSE

32

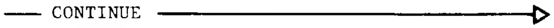

end_if_statement:
END IF

33 do_statement:


DO
label
,
variable_name =
int_expr
,
int_expr
1

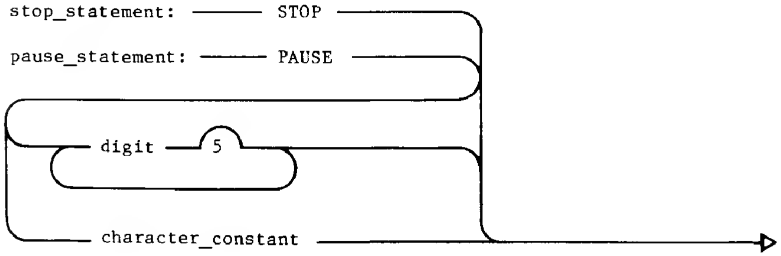
(33) The control parameters must be integer expressions. If the iteration count of the DO statement is zero, the statements in the range of the DO statement will not be executed, unlike ANSI 66 where they would still be executed once.

34 continue_statement:



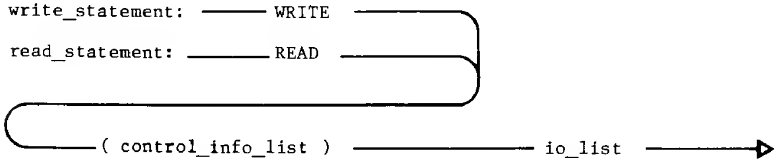
35 stop_statement:

36 pause_statement:

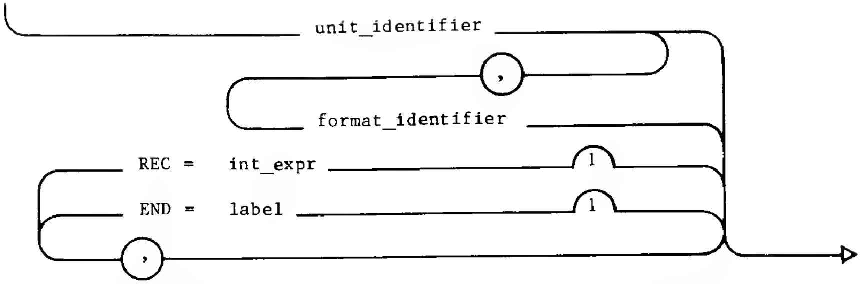

STOP
PAUSE
digit
5
character_constant

37 write_statement:

38 read_statement:

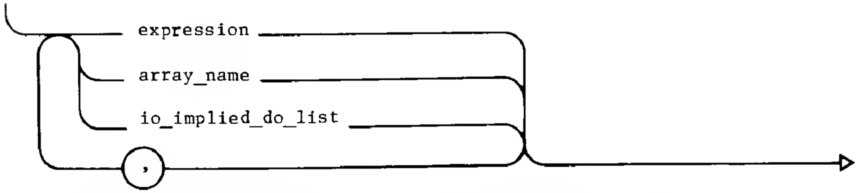

WRITE
READ
(control_info_list)
io_list

39 control_info_list:


unit_identifier
,
format_identifier
REC = int_expr
1
END = label
1
,

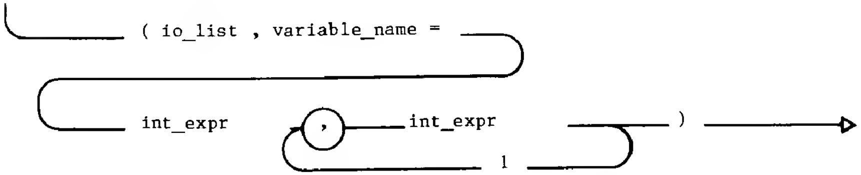
(39) A control_info_list must contain exactly one unit_identifier. An END= specifier must not appear in a WRITE statement.

40 io_list:


expression
array_name
io_implied_do_list
,

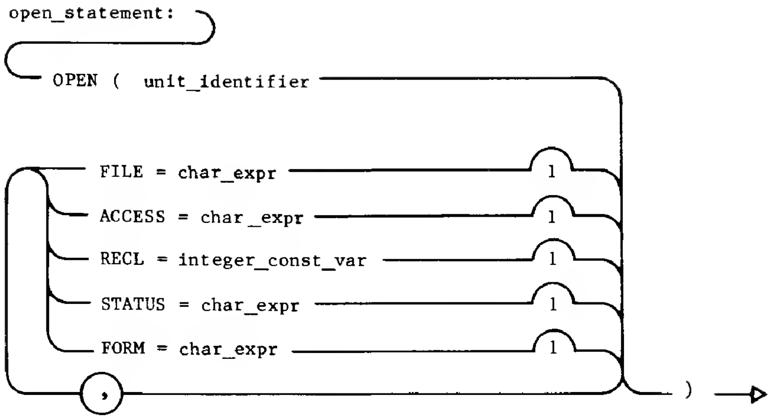
(40) In a WRITE statement, an I/O list expression must not begin with "(".

41 io_implied_do_list:

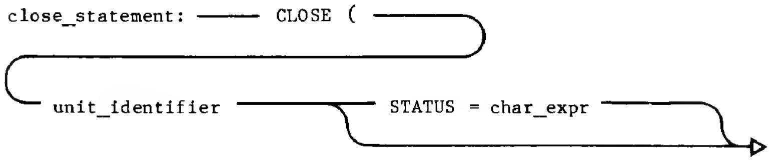

(io_list , variable_name =
int_expr
,
int_expr
1
)

(41) The control parameters must be integer expressions. If the iteration count of the I/O implied DO list is zero, the statements in the range of the DO list will not be executed, unlike ANSI 66 where they would still be executed once.

42


open_statement:
OPEN (
unit_identifier
FILE = char_expr
ACCESS = char_expr
RECL = integer_const_var
STATUS = char_expr
FORM = char_expr
1
1
1
1
1
,
)

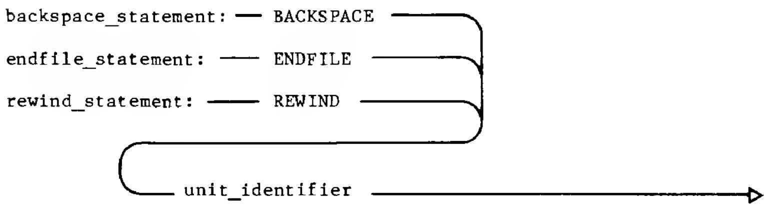
43


close_statement:
CLOSE (
unit_identifier
STATUS = char_expr

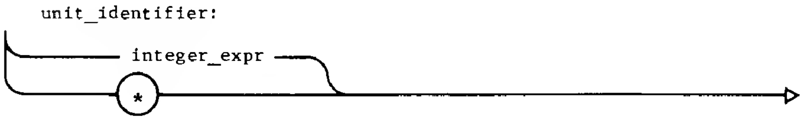
44

45

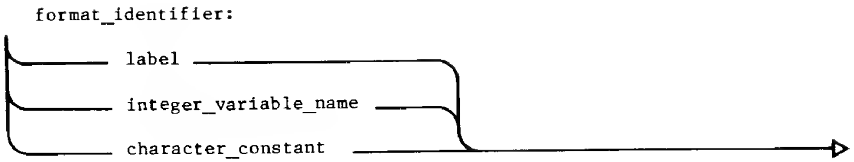
46


backspace_statement:
BACKSPACE
endfile_statement:
ENDFILE
rewind_statement:
REWIND
unit_identifier

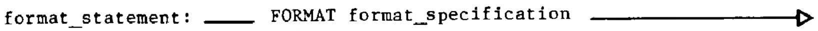
47


unit_identifier:
integer_expr
*

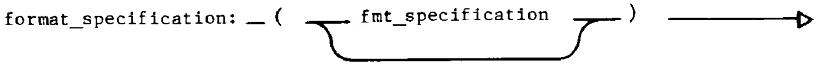
48


format_identifier:
label
integer_variable_name
character_constant

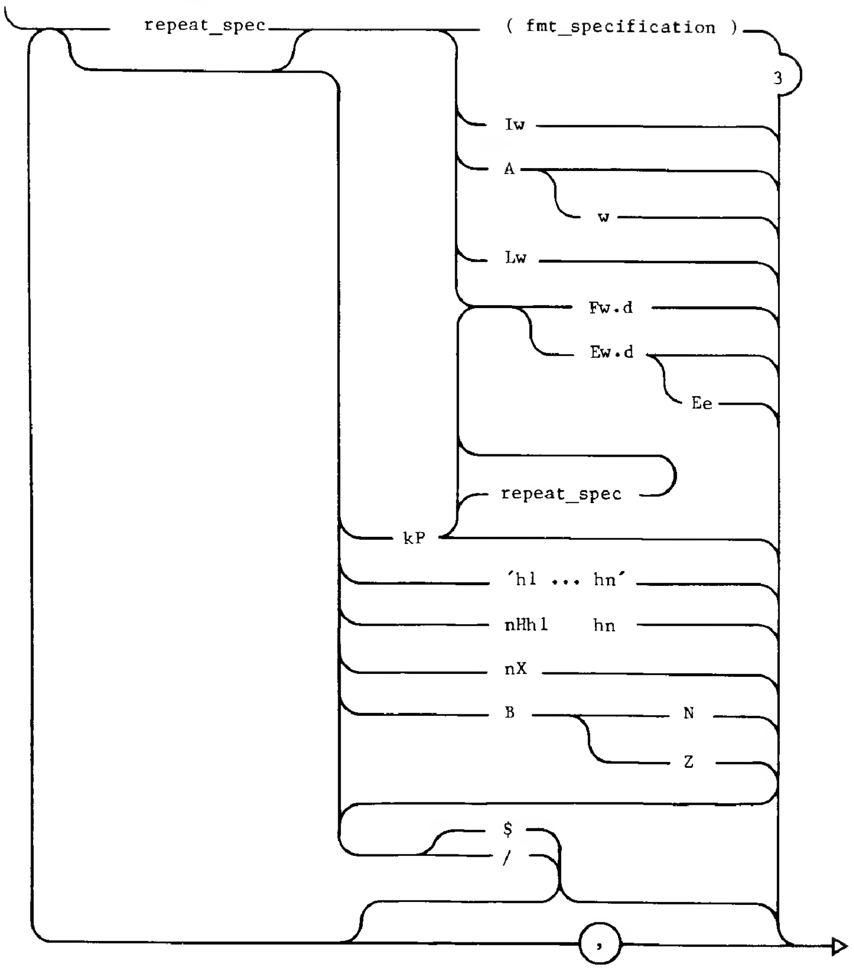
49

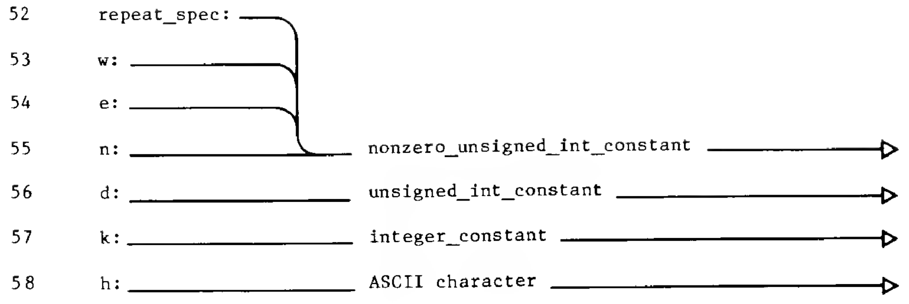

format_statement:
FORMAT format_specification

50

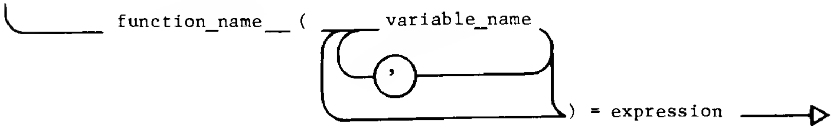

format_specification:
(
fmt_specification
)

51 fmt_specification:

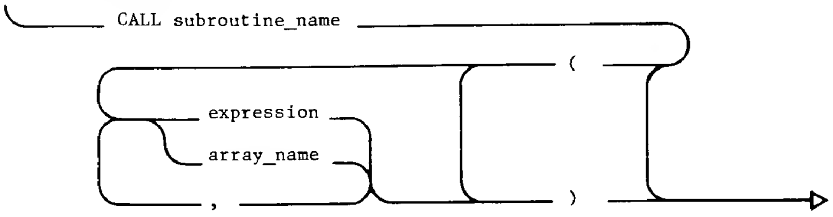

repeat_spec
(fmt_specification)
3
Iw
A
w
Lw
Fw.d
Ew.d
Ee
repeat_spec
kP
'h1 ... hn'
nHh1 hn
nX
B
N
Z
$
/
,


52 repeat_spec:
53 w:
54 e:
55 n:
nonzero_unsigned_int_constant
56 d:
unsigned_int_constant
57 k:
integer_constant
58 h:
ASCII character

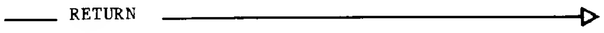
59 statement_function_statement


function_name (
variable_name
,
) = expression

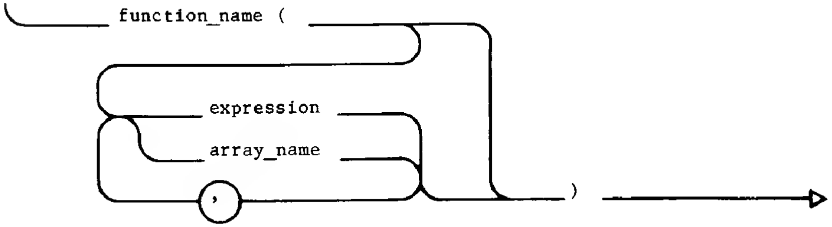
60 call_statement:


CALL subroutine_name
(
expression
array_name
,
)

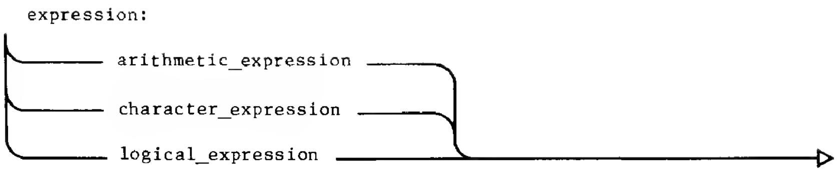
61 return_statement:



62 function_reference:


function_name (
expression
array_name
,
)

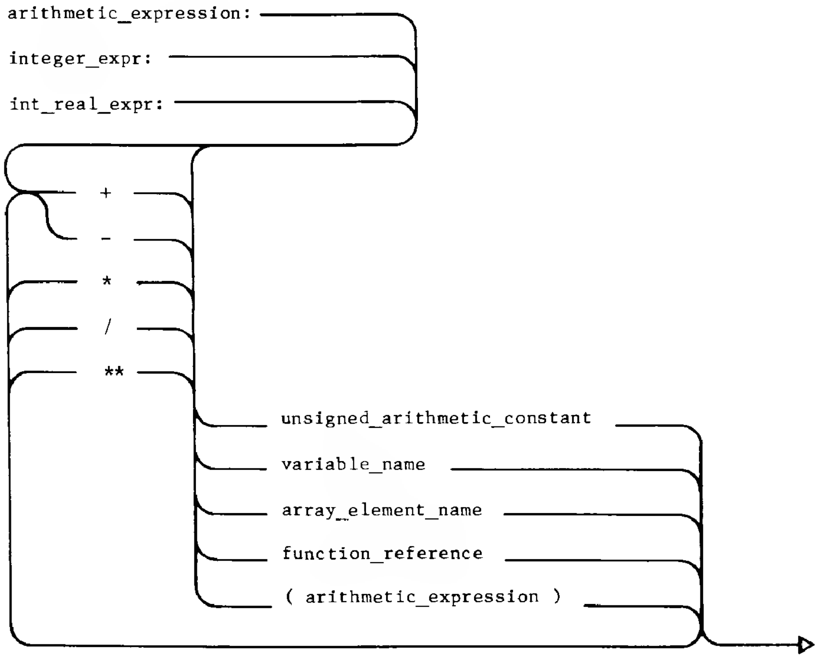
63


expression:
arithmetic_expression
character_expression
logical_expression

64

65

66

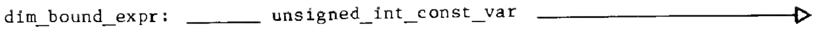

arithmetic_expression:
integer_expr:
int_real_expr:
+
-
*
/
**
unsigned_arithmetic_constant
variable_name
array_element_name
function_reference
(arithmetic_expression)

(64) A variable name, array element name, or function reference in an arithmetic expression must be of type integer or real.

(65) An integer expression is an arithmetic expression of type integer.

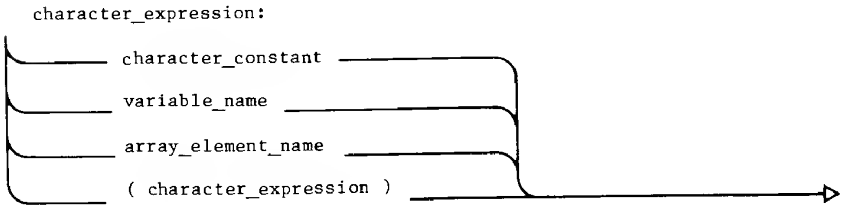
(66) An int_real_expression is an arithmetic expression of type integer or real.

69


dim_bound_expr:
unsigned_int_const_var

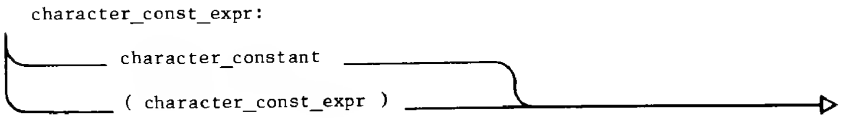
(69) Variables in a dimension bound expression must be of type integer, unsigned, and be either a constant or variable name.

70

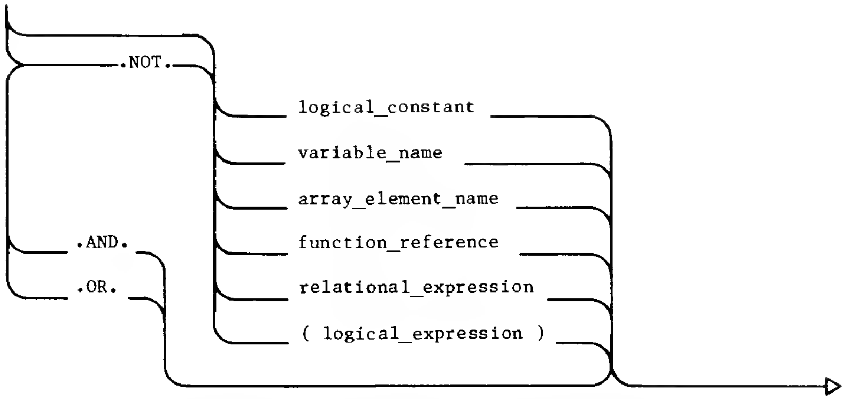

character_expression:
character_constant
variable_name
array_element_name
(character_expression)

(70) A constant name, variable name or array element name must be of type character in a character expression. There is no concatenation operator in the subset.

71

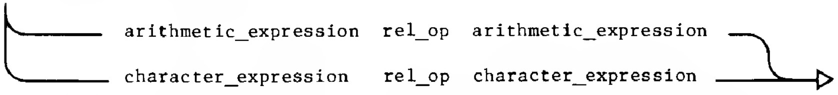

character_const_expr:
character_constant
(character_const_expr)

72 logical_expression:


.NOT.
logical_constant
variable_name
array_element_name
function_reference
relational_expression
(logical_expression)
.AND.
.OR.

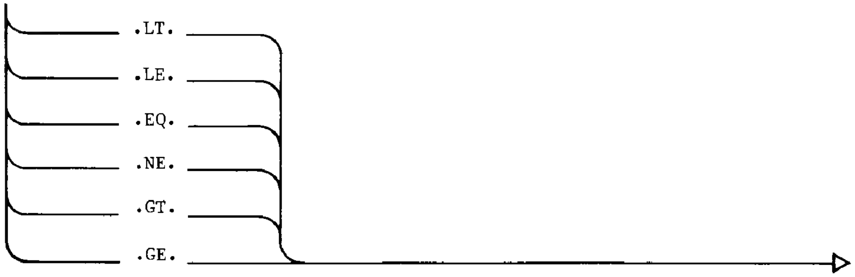
(72) A constant name, variable name, array element name or function reference must be of type logical in a logical expression. .EQV. and .NEQV. are not in the subset.

73 relational_expression:

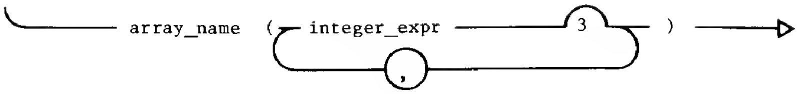

arithmetic_expression rel_op arithmetic_expression
character_expression rel_op character_expression

(73) Character expressions in relational expressions are evaluated in a lexicographical comparison, using the ASCII character collating sequence.

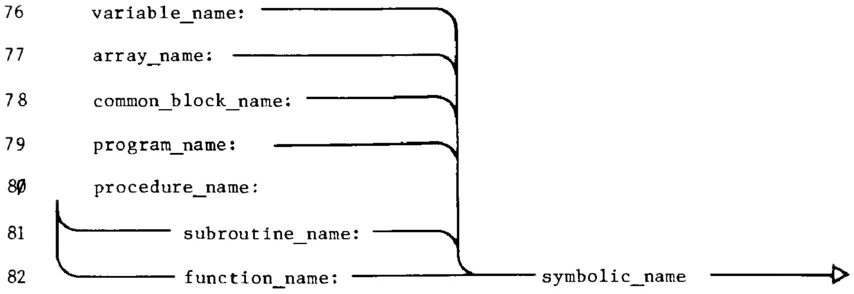
74 rel_op:


.LT.
.LE.
.EQ.
.NE.
.GT.
.GE.

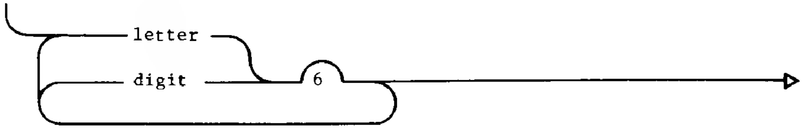
75 array_element_name:


array_name
(
integer_expr
3
,
)

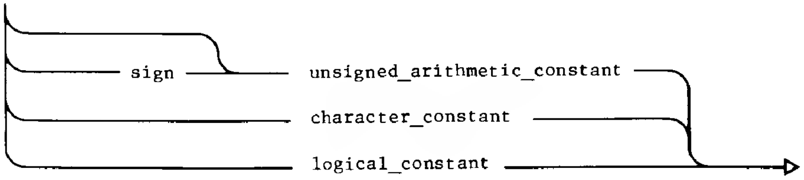
(75) Arrays are restricted to three dimensions in the subset.


76 variable_name:
77 array_name:
78 common_block_name:
79 program_name:
80 procedure_name:
81 subroutine_name:
82 function_name:
symbolic_name

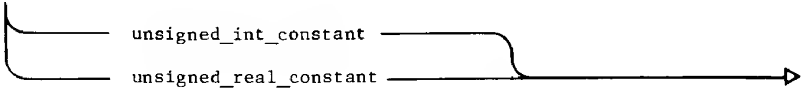
83 symbolic_name:


letter
digit
6

84 constant:


sign
unsigned_arithmetic_constant
character_constant
logical_constant

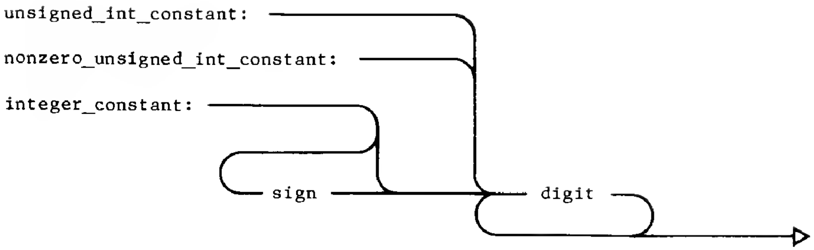
85 unsigned_arithmetic_constant:


unsigned_int_constant
unsigned_real_constant

86

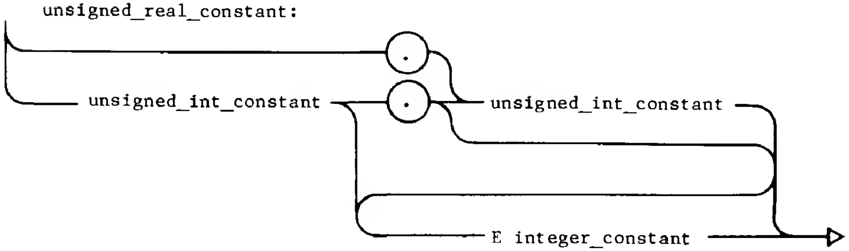
87

88

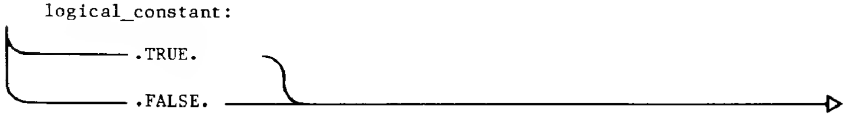

unsigned_int_constant:
nonzero_unsigned_int_constant:
integer_constant:
sign
digit

(88) A nonzero, unsigned, integer constant must contain a nonzero digit.

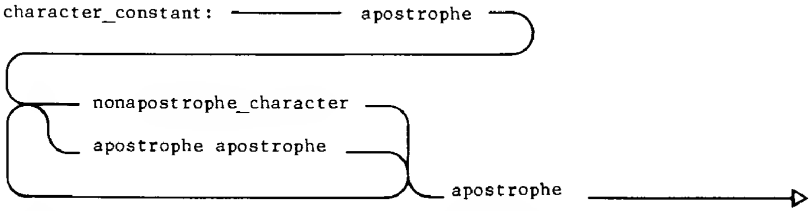
89


unsigned_real_constant:
.
unsigned_int_constant
.
unsigned_int_constant
E integer_constant

90

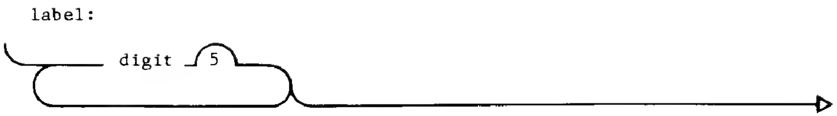

logical_constant:
.TRUE.
.FALSE.

91


character_constant:
apostrophe
nonapostrophe_character
apostrophe apostrophe
apostrophe

(91) An apostrophe within a character constant is represented by two consecutive apostrophes with no intervening blanks.

92


label:
digit
5

(92) A label must contain a nonzero digit.

APPENDIX E

FORTRAN STATEMENT SUMMARY

In the following summary of Apple FORTRAN statements, FORTRAN reserved words are capitalized; other entities that are required by the statement, such as arguments and type qualifiers, are in lower case. Items enclosed in square brackets: [ ] are optional. An ellipsis indicates that the previous item may be repeated. For instance in the CALL statement shown below, the named subroutine can have no arguments, in which case the name of the subroutine is not followed by anything, or it can have one argument, in which case the argument is enclosed in parentheses, or it can have more than one argument in which case the argument list is enclosed in parentheses and the individual arguments are separated by commas.

The type identifier below can be any of INTEGER, REAL, LOGICAL, or CHARACTER[*length]. The length argument specifies the number of characters that the entity can store, and is an unsigned, nonzero, integer constant. I/O device unit numbers may be integer expressions.

```
ASSIGN statement label TO integer variable

BACKSPACE unit

CALL subroutine [([argument [,argument]...])]

CHARACTER [*length[,]] name [,name]...

COMMON [/[common_block]/]name_list[[,]/[common_block]/name_list]...

CONTINUE

DATA name_list/constant_list/ [[,]name_list/constant_list/]...

DIMENSION array(dimension) [,array(dimension)]...

DO statement [,] integer_variable=expression_1, expression_2
     [,expression_3]

ELSE

ELSE IF (expression) THEN

END

END IF

ENDFILE unit

EQUIVALENCE (name_list) [,(name_list)]...

EXTERNAL procedure [,procedure]...

FORMAT format_specification
```

```
function ([dummy_argument [,dummy_argument]...]) = expression

[type] FUNCTION function ([dummy_argument [,dummy_argument]...])

GO TO integer_variable [[,][statement_label [,statement_label]...)]

GO TO statement_label

GO TO (statement_label [,statement_label]...)[,] integer_variable

IF (expression) statement

IF (expression) statement1, statement2, statement3

IF (expression) THEN

IMPLICIT type (a [,a]...) [,type (a [,a]...)]...

INTEGER variable_name [,variable_name]

INTRINSIC function [,function]...

LOGICAL variable_name [,variable_name]...

OPEN (open_list)

PAUSE [character_constant or integer]

PROGRAM program_name

READ (control_information_list) [i/o_list]

REAL variable_name [,variable_name]

RETURN

REWIND unit

SAVE a[,a]...

STOP [character_constant or integer]

SUBROUTINE subroutine [([dummy_argument [,dummy_argument]...])]

arithmetic_variable = arithmetic_expression

logical_variable = logical_expression

character_variable = character_expression

WRITE (control_information_list) [i/o_list]
```

APPENDIX F

ANSI FORTRAN 66 VS. 77

216 Introduction
216 Conflicts
216 Line Format
216 Hollerith
216 Arrays
216 I/O
217 Intrinsic Functions
217 Other Conflicts
217 Adapting Programs
218 Additions

INTRODUCTION

This Appendix contains a brief description of the major differences between the new ANSI 77 Standard FORTRAN and the earlier, much more common ANSI 66 Standard FORTRAN that you are probably more familiar with. These differences break down into two categories, first, changes that cleaned up those undefined areas remaining in the ANSI 66 specification, and secondly changes that added capabilities not available in ANSI 66.

CONFLICTS

There are some conflicts between ANSI 66 and ANSI 77 FORTRAN. These are listed in the sections immediately following. The additions made to the language are described under ADDITIONS.

Line Formats

A line that contains only blanks which are defined in Apple FORTRAN to be the standard ASCII SPACE character in columns 1 to 72 is treated as a comment line whereas ANSI 66 treated it as the initial line of a statement.

Columns 1 through 5 of a continuation line must now contain blanks. ANSI 66 made no requirement except that column 1 could not contain a C unless it were to be treated as a comment. Noncontinuation lines must be blank in column 6.

Hollerith

Hollerith constants and Hollerith data have been deleted. A new character type has been substituted in their place, see below under ADDITIONS. There is still an H edit descriptor, but it is not a Hollerith constant.

Arrays

Each array subscript expression must not exceed its corresponding upper bound. ANSI 66 allowed this under some circumstances.

You must now always specify all the dimensions of an array element. ANSI 66 allowed multi-dimensional arrays to be specified with a one-dimensional subscript in EQUIVALENCE statements.

I/O

No records may be written after an endfile record in a sequential file. (This is made to be impossible by Apple FORTRAN I/O.)

Only positive values are allowed for I/O unit identifiers. ANSI 66 did not specifically prohibit negative values.

You may not read into an H edit descriptor in a FORMAT statement.

Intrinsic Functions

An intrinsic function must appear in an INTRINSIC statement prior to its use as an actual argument. ANSI 66 allowed it to appear in an EXTERNAL statement instead. The intrinsic function class includes the basic external function class of ANSI 66.

Naming an intrinsic function in a type-statement that conflicts with the type of the intrinsic function does not remove the function from the class of intrinsic functions. For ANSI 66, this would have been sufficient to remove it.

There are now more intrinsic functions available than defined in ANSI 66. See Appendix B for a list of these.

Other Conflicts

This section summarizes the other conflicts between ANSI 66 and ANSI 77.

* It is illegal to specify the type of an identifier more than once.

* The range of a DO loop may be entered only by execution of a DO statement. The concept of extended range of a DO as described in ANSI 66 no longer exists and will be trapped by the compiler. Also, in ANSI 66 a large number of systems extended the standard by allowing the terminal parameter in a DO statement to be less than the initial parameter, but executed the DO loop one time, rather than zero times as specified in ANSI 77.

* A labeled END statement could conflict with the initial line of a statement in ANSI 66.

* The E or D output FORMAT edit descriptors will now append a plus or minus before the exponent field. This is an ANSI 77 feature not present in ANSI 66.

ADAPTING PROGRAMS

Here are some problem areas that you should consider if you are contemplating translating programs into Apple FORTRAN from other versions.

Subprograms written in languages other than FORTRAN or Pascal will need to be rewritten. This is especially true of machine language subprograms. It may be possible, however, to use a program written in Pascal 6502 assembler code for the Apple without rewriting it.

FORTRAN has never specified the collating sequence of the character set used. Apple FORTRAN uses both the ASCII encoded binary representation as shown in Appendix B and its collating sequence. Character relational expressions may not necessarily have the same value if the translated program used another character collating sequence. It is possible that the character set of the source program contains characters not in the ASCII set.

* File name formats may be different.

* I/O capabilities may be different.

* The program may utilize full language features not available in the ANSI Standard subset.

* There are usually some language extensions introduced in most versions of FORTRAN that may not be available in Apple FORTRAN which is subset standard conforming.

* The program may utilize aspects of ANSI 66 FORTRAN that have been deleted from ANSI 77.

ADDITIONS

The three major additions to ANSI 77 are the IF statement constructs, the CHARACTER data type, and the standardizations to I/O. I/O is treated extensively in Chapter 11. The IF statement is discussed in Chapter 10, and the CHARACTER statement is discussed in Chapter 7.

ANSI FORTRAN 66 had only an Arithmetic IF statement. ANSI 77 has extended this to include the Logical IF statement, the Block IF statement and the ELSE IF, ELSE and END IF statements. Together these statements provide a vastly improved method of clearly and accurately specifying the flow of program control. Refer to Chapter 10 for a discussion of these IF statements.

APPENDIX G

APPLE FORTRAN VS. ANSI 77

220 Introduction
220 Unsupported Features
220 Full-Language Features
221 Extensions to the Standard

INTRODUCTION

This appendix is directed at the reader who is familiar with the ANSI Standard FORTRAN 77 Subset language as defined in ANSI X3.9-1978. It describes how Apple FORTRAN 77 differs from the standard language. The differences fall into three general categories, unsupported features, full-language features, and extensions to the Standard.

UNSUPPORTED FEATURES

There are two significant places where Apple FORTRAN 77 does not comply with the Standard. One is that procedures cannot be passed as formal parameters and the other is that INTEGER and REAL data types do not occupy the same amount of storage. Both differences are due to limitations of the UCSD P-Code architecture.

Parametric procedures are not supported simply because there is no practical way to do so in the UCSD P-Code. The instruction set does not allow the loading of a procedure's address onto the stack, and more significantly, does not allow the calling of a procedure whose address is on the stack.

REAL variables require 4 bytes (32 bits) of storage while INTEGER and LOGICAL variables only require 2 bytes. This is due to the fact that the UCSD P-Code supported operations on those types are implemented in those sizes.

FULL-LANGUAGE FEATURES

There are several features from the full language that have been included in this implementation for a variety of reasons. Some were done at either minimal or zero cost, such as allowing arbitrary expressions in subscript calculations. Others were included because it was felt that they would significantly increase the utility of the implementation, especially in an engineering or laboratory application. In all cases, a program which is written to comply with the subset restrictions will compile and execute, since the full language includes the subset constructs. A short description of full language features included in the implementation follows.

The subset does not allow function calls or array element references in subscript expressions, but the full language and this implementation do.

The subset restricts expressions that define the limits of a DO statement, but the full language does not. Apple FORTRAN also allows full integer expressions in DO statement limit computations. Similarly, arbitrary integer expressions are allowed in implied DO loops associated with READ and WRITE statements.

Apple FORTRAN allows an I/O unit to be specified by an integer expression, as does the full language.

The subset does not allow expressions to appear in an I/O list whereas the full language does allow expressions in the I/O list of a WRITE statement. Apple FORTRAN allows expressions in the I/O list of a WRITE statement providing that they do not begin with an initial left parenthesis. User note: the expression (A+B)*(C+D) can be specified in an output list as +(A+B)*(C+D) which, incidentally, does not generate any extra code to evaluate the leading plus sign.

Apple FORTRAN allows an expression for the value of a computed GOTO, consistent with the full language rather than the subset language.

Apple FORTRAN allows both sequential and direct access files to be either formatted or unformatted. The subset language requires direct access files to be unformatted and sequential to be formatted. Apple FORTRAN also contains an augmented OPEN statement which takes additional parameters that are not included in the subset. There is also a form of the CLOSE statement, which is not included at all in the subset. I/O is described in more detail in Chapters 11 and 12.

Apple FORTRAN includes the CHAR intrinsic function. CHAR (i) returns the character in the ith position of the ASCII collating sequence.

```
ICHAR(CHAR(i))=i
```

EXTENSIONS TO THE STANDARD

The language implemented has several minor extensions to the full language standard. These are briefly described below.

Compiler directives have been added to allow the programmer to communicate certain information to the compiler. An additional kind of line, called a compiler directive line, has been added. It is characterized by a dollar sign ($) appearing in column 1. Certain directives are restricted to appear in certain places. A compiler directive line is used to convey certain compile-time information to the FORTRAN system about the nature of the current compilation. The set of directives is briefly described below:

$INCLUDE filename	Include textually the file filename at this point in the source. Nested includes are implemented to a depth of nesting of five files. Thus, for example, a program may include various files with subprograms, each of which includes various files which describe common areas; this would be a depth of nesting of three files.
$USES ident [IN filename] [OVERLAY]	Similar to the USES command in the UCSD Pascal compiler. The already compiled FORTRAN subroutines or Pascal procedures contained in the file filename, or in the file *SYSTEM.LIBRARY if no file name is present, become callable from the currently compiling code. This directive must appear before the initial non-comment line.
$XREF	Produce a cross-reference listing at the end of each procedure compiled.
$EXT SUBROUTINE name #parms or $EXT [type] FUNCTION name #params	The subroutine or function called name is an assembly language routine. The routine has exactly #params reference parameters.

The edit control character $ can be used in formats to inhibit the normal advance to the next record which is associated with the completion of a READ or a WRITE statement. This is particularly useful when prompting to an interactive device, such as the CONSOLE:, so that a response can be on the same line as the prompt.

An intrinsic function, EOF, has been provided. The function accepts a unit specifier as an argument and returns a logical value which indicates whether the specified unit is at its end of file.

Upper and lower case source input is allowed. In most contexts, lower case characters are treated as indistinguishable from their upper case counterparts. Lower case is significant in character constants and hollerith fields.

BIBLIOGRAPHY

The FORTRAN readings suggested here provide information on the full FORTRAN language.

Brainerd, Walter S., Charles H. Goldberg, and Jonathan L. Gross. "FORTRAN 77 Programming," New York, N.Y.: Harper & Row Publishers Inc., 1978.

Brainerd, Walter S. "FORTRAN 77." Communications of the ACM, Vol. 21, No. 10 (Oct. 1978).

Katzan, Harry, Jr. "FORTRAN 77," New York, N. Y.: Van Nostrand Reinhold Company, 1978.

Meissner, Loren P., and Elliott I. Organick. "FORTRAN 77," Reading, Mass.: Addison-Wesley Publishing Co., 1980.

Wagener, Jerrold L. "FORTRAN 77 Principles of Programming," New York, N. Y.: John Wiley and Sons, 1980.

INDEX

A

- A edit descriptor 96
- ANSI FORTRAN 66 4, 6, 216-218
- ANSI FORTRAN 77 4, 6, 216-222
- APPLESTUFF unit
  - BUTTON function 133
  - game controls 132, 133
  - KEYPRE function 134
  - NOTE subroutine 134
  - PADDLE function 132, 133
  - RANDOI subroutine 132
  - RANDOM function 132
- APPLE1 diskette 136-138, 154-156
- APPLE2 diskette 136-138, 154-156
- APPLE3 diskette 10
- arguments
  - by reference 116
  - by value 116
- arithmetic expressions
  - integer division 59
  - operators 58, 59
  - result type 59
  - type conversions 59
- arithmetic IF statement 65
- arrays
  - ANSI 66 vs. ANSI 77 216
  - assumed size 48
  - asterisk dimension 47
  - element name 48
  - number of dimensions 47
  - order of elements 48
  - storage 48, 118
  - subscript expression 48
  - Turtle Graphics 129, 130
- ASMDEMO program 10, 120-122
- ASSCII Character Codes Table 186
- Assembly language routines 120-122
- ASSIGN statement 54, 55
- assigned GOTO statement 65
- assignment statements
  - computational 54
  - label 54, 55

B

- BACKSPACE statement 79, 85
- bilingual programs 119-122
- block IF statement 66-68
- BN edit descriptor 86, 94
- BUTTON function 133
- BZ edit descriptor 94

C

- CALL statement 98, 99
- Cartesian coordinates 125, 127
- CHAR intrinsic function 5, 221
- character collating sequence 35, 186
- character data type 41, 42
- character expressions 60
- character set 34, 35
- CHARACTER type statement 49
- CHARTY subroutine 131
- CLOSE statement 78, 83
- CODE files 8, 12-15
- comment lines 36
- COMMON statement 23, 49, 50
- compilation
  - CODE file 12-15
  - error messages 20, 21, 26, 172-175
  - modules 12, 13
  - organizing programs 12
  - partial 12
  - same name option 20
  - separate 13, 113
  - TEXT file 12, 13
- Compiler
  - input requirements 18
  - operation 18-26
  - sample listing 25
- compiler directives 5, 221, 222
  - $EXT 14, 23, 24, 121
  - $INCLUDE 23
  - $USES 13, 14, 23, 110-113, 116, 121
  - $XREF 23, 26
- compile-time error messages 172-175
- computational assignment statement 54
- computed GOTO statement 64
- configuring Apple FORTRAN
  - multi-drive user 155, 156
  - single-drive user 136-138
- CONSOLE: 10, 76, 80, 82, 84, 86, 100, 105
- CONTINUE statement 71
- control statements
  - arithmetic IF 65
  - assigned GOTO 65
  - block IF 66-68
  - CONTINUE 71
  - DO 70, 71
  - END 72
  - logical IF 65
  - PAUSE 72
  - STOP 72

D

- DATA statements 37, 38, 52, 53
- data type correspondence 118
- data types
  - character 41
  - integer 40
  - logical 41
  - real 40, 41
- database 78, 87
- DIMENSION statement
  - asterisk array dimension 47, 48
  - dimension declarator 47
  - form 47
- direct access files 75, 76, 78, 85
- diskettes
  - formatting 141-143, 160-162
  - making backups 140-145, 159-163
- DO loop range 217
- DO statement 70, 71
- DO variable expression 5
- DRAWBL subroutine 129-131

E

- E edit descriptor 95
- edit descriptors
  - apostrophe 92
  - blank interpretation 86, 94
  - character 96
  - dollar sign 86, 93, 222
  - Hollerith 93, 216
  - integer 95
  - logical 96
  - nonrepeatable 92-94
  - positional 93
  - real 95
  - repeatable 94-96
  - scale factor 94, 95
  - slash 93
- Editor 138-152, 157-170
- ELSE statement 69
- ELSEIF statement 69
- END statement 22, 37, 38, 72, 99
- ENDFILE statement 85
- ENDIF statement 69
- EOF 5, 79, 222
- EQUIVALENCE statement 51, 52
- error messages 172-177
- expressions
  - arithmetic 58-60
  - character 60
  - logical 61, 62
  - operator precedence 62
  - relational 60, 61
- external files 75-77
- external FUNCTION 100
- EXTERNAL statement 50

F

- F edit descriptor 95
- Filer 138-152, 157-170
- files
  - direct access 5, 75, 76, 78, 85
  - external 75-77
  - internal 75, 76
  - name 75
  - sequential 5, 75, 77, 79, 85
- FILLSC subroutine 127
- formal parameters 5
- FORMAT statement 37, 90-92
- formatted files 75, 77, 78
- formatted I/O 90-96
- FORTLIB.CODE 10, 15, 30
- FORTRAN
  - ANSI 66 vs. 77 216-218
  - Apple unsupported features 220
  - Apple vs. ANSI 77 220-222
  - Pascal interface 8, 116-120
  - program development facility 2
  - running a compiled program 151, 167
  - running a new program 145-151, 165-167
  - transferring programs 217, 218
  - writing a program 151, 152, 168
- FORTRAN statement summary 212, 213
- FORTRAN statements
  - assignment 53-55
  - continuation 37
  - control 64-72

  defined 37
  initial line 37
  ordering 37, 38
  specification 45-52
  statement label 36
FORTRAN syntax diagrams 188-209
FORT1:
  configuring 136-138, 155, 156
  system files 138, 156
FORT2:
  configuring 136-138, 155, 156
  system files 138, 156
FUNCTION statement 13, 37, 38, 98-100
functions
  calling with I/O statements 79
  external 100
  formal and actual arguments 106, 107
  intrinsic 101-105

G

game controls 132, 133
global names 44, 45, 100
global symbol table 26
GOTO statements 64, 65
GRAFMO subroutine 125

H

H edit descriptor 93
heap marker 120

I

I edit descriptor 95
identifiers 20, 28
IMPLICIT statement 38, 46
INITTU subroutine 124, 125
INITTURTLE 120
integer data type 4, 40
integer division 59
INTEGER type statement 49
internal files 75, 76
intrinsic function 101-105
  CHAR 5
  EOF 5, 79
  placement in statement 216
Intrinsic Functions Table 181-183
INTRINSIC statement 51
iolist
  defined 80
  expressions in 5
  formatting 91, 92
  implied DO list 81
I/O device
  blocked 76
  external files 75
I/O statements
  BACKSPACE 85, 91
  CLOSE 83
  ENDFILE 85
  OPEN 81, 82
  READ 83, 84
  REWIND 85
  WRITE 84
I/O System 74-87
I/O unit number 5
I/O unit specifier 80

J

K

KEYPRE function 134
keywords 44

L

L edit descriptor 96
label assignment statement 54, 55
Lexical Comparisons Table 185
library 29-31
Linker
  mapfile 31
  operation 28-32, 111
  system files used 28
local scope names 44, 45
logical data type 41
logical expressions 61, 62
logical IF statement 65
LOGICAL type statement 49

M

main program 12-15, 98
MAINSEGX 110, 112
memory after compilation 26
modules 12, 13
MOVE subroutine 128
MOVETO subroutine 127
multi-drive user
  compilation 19
  Linker 28
  system configuration 155, 156

N

names
  common data blocks 45
  global scope 44, 45
  integers 45
  keywords 44
  local scope 44, 45
  undeclared 45
  variables 45
notation conventions 34
NOTE subroutine 134

O

OPEN statement 81, 82, 86, 87
operator precedence 62
overlay 14, 15, 23, 113

P

P edit descriptor 94, 95
P-code 8, 18
PADDLE function 132, 133
partial compilation 110, 111
Pascal
  FORTRAN interface 8, 116-120
  INTERFACE 116
  RTFINIALIZE 120
  RTINITIALIZE 119, 120
  USES 118
Pascal documentation 9, 10
Pascal Operating System 2, 3
PAUSE statement 72
pen colors 126
PENCOL subroutine 126, 127
preconnected unit 86
PRINTER: 78
program identifier 20
program input
  blanks 36
  character set 34, 35
  columns 35
  comment lines 36
  END statement 22
  form 21
  line length 22
  upper and lower case 21, 222
PROGRAM statement 37, 98
program units 37, 98-107

Q

R

RANDOI subroutine 132
RANDOM function 132
READ statement 83, 84
real data type
  basic real constant 41
  defined 4, 40
  real constant 41
REAL type statement 49
record
  endfile 74, 75
  formatted 74-78
  kinds of 74
  unformatted 74-79, 85
recursive subroutine calls 99
relational expressions 60, 61
REMIN 21
RETURN character 38
RETURN statement 99, 106
REWIND statement 85
RTUNIT 8, 10, 15, 29, 110, 119
RUN Command 19
run-time error messages 176, 177

S

SAVE statement 51
SCREEN function 129
sequential files 75, 77, 79, 85

single-drive user
  compilation 18, 19
  Linker 28
  system configuration 136-138
specification statements
  COMMON 49, 50
  DIMENSION 47
  EQUIVALENCE 51, 52
  EXTERNAL 50
  IMPLICIT 46
  INTRINSIC 51
  SAVE 51
  statement ordering 37, 38
  type 48, 49
statement function 105, 106
statement label 64, 65
STOP statement 72
subprograms 12-15
SUBROUTINE statement 13, 37, 38, 98-100
subscript expressions 5
symbol table 45
SYSTEM.COMPILER 8, 18, 19
SYSTEM.EDITOR 18, 19
SYSTEM.LIBRARY 8, 10, 14, 29, 111
SYSTEM.WRK.CODE 19-21
SYSTEM.WRK.TEXT 18, 19

T

TEXT files 8, 12, 13
TEXTMO subroutine 125
Transcendental Functions Table 184
TTLOUT subroutine 133
TURN subroutine 128
turnkey system 2, 9
TURNTO subroutine 128
TURTLA function 128
Turtle Graphics
  Apple screen coordinates 124
  arrays 129, 130
  Cartesian graphics 127
  CHARTY subroutine 131
  DRAWBL subroutine 129-131
  FILLSC subroutine 127
  GRAFMO subroutine 125
  INITTU subroutine 124, 125
  MOVE subroutine 128
  MOVETO subroutine 127
  Pascal programs 120
  PENCOL subroutine 126, 127
  SCREEN function 129
  text on screen 130, 131
  TEXTMO subroutine 125
  TURN subroutine 128
  TURNTO subroutine 128
  TURTLA function 128
  TURTLX function 128
  TURTLY function 128
  VIEWPO subroutine 125
  WCHAR subroutine 130, 131
TURTLX function 128
TURTLY function 128

U

unconditional GOTO statement 64
unformatted files 75, 78, 79, 85, 87
Unit Identifiers Table 180

V

VIEWPO subroutine 125

W

WCHAR subroutine 28, 130, 131
WRITE statement 84

X,Y, Z

6502 Assembly Language 8-10, 18