TANDY

TRS-80

COMPUTER PRODUCTS

MODEL 2000 MS™-DOS 2.0 CAT. NO. 26-5255

MS™-FORTRAN

03.13.00

MODEL 2000

Custom manufactured in USA
by Tandy Corporation.

READ ME FIRST

All computer software is subject to change, correction, or improvement as the manufacturer receives customer comments and experiences. Radio Shack has established a system to keep you immediately informed of any reported problems with this software, and the solutions. We have a customer service network including representatives in many Radio Shack Computer Centers, and a large group in Fort Worth, Texas, to help with any specific errors you may find in your use of the programs. We will also furnish information on any improvements or changes that are "cut in" on later production versions.

To take advantage of these services, you must do three things:

(1) Send in the postage-paid software registration card included in this manual immediately. (Postage must be affixed in Canada.)

(2) If you change your address, you must send us a change of address card (enclosed), listing your old address exactly as it is currently on file with us.

(3) As we furnish updates or "patches", and you update your software, you must keep an accurate record of the current version numbers on the logs below. (The version number will be furnished with each update.)

Keep this card in your manual at all times, and refer to the current version numbers when requesting information or help from us. Thank you.

APPLICATIONS SOFTWARE VERSION LOG

03.13.00

OP. SYSTEM VERSION LOG

Cat. No. 26-5255

MS™-FORTRAN

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK AND TANDY COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Radio Shack and Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores and from Radio Shack franchisees and dealers at its authorized location.** The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. **Except as provided herein, Radio Shack makes no express warranties, and any implied warranty of merchantability or fitness for a particular purpose is limited in its duration to the duration of the written limited warranties set forth herein.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. **Except as provided herein, Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by "Equipment" or "Software" sold, leased, licensed or furnished by Radio Shack, including, but not limited to, any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of the "Equipment" or "Software". In no event shall Radio Shack be liable for loss of profits, or any indirect, special, or consequential damages arising out of any breach of this warranty or in any manner arising out of or connected with the sale, lease, license, use or anticipated use of the "Equipment" or "Software".**

Notwithstanding the above limitations and warranties, Radio Shack's liability hereunder for damages incurred by customer or others shall not exceed the amount paid by customer for the particular "Equipment" or "Software" involved.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

MS™-FORTRAN Compiler

User's Guide

Replacement Page
for the
MS-FORTRAN Reference Manual

The attached replacement page is provided to update your *MS-FORTRAN Reference Manual.* It contains the changes made to files on the MS-FORTRAN Compiler diskette.

Please remove and discard the original page and put this in its place.

Note to MS-FORTRAN users: If you get a disk full error during any pass of the compiler, you should immediately use the MS-DOS CHKDSK utility to insure the integrity of the file system.

Thank You!
Tandy Corporation

Operating
Fortran

Contents

Figures iii

Tables iii

Introduction iv

System Requirements iv
Package Contents iv
About This Manual v
Syntax Notation vii
Learning More About FORTRAN vii

1 / Getting Started 1

1.1 Preliminary Procedures 2
1.2 Program Development 3
1.3 Vocabulary 7

2 / A Sample Session 11

2.1 Preparing a FORTRAN Program 12
2.2 Compiling The FORTRAN Source File 13
2.3 Linking the Object File With the Runtime Library 17
2.4 Executing Your FORTRAN Program 19

3 / More About Compiling 21

3.1 Files Written by the Compiler 22
3.2 Filename Conventions 25
3.3 Starting the Compiler 29

4 / More About Linking 33

4.1 Files Read by the Linker 34
4.2 Files Written by the Linker 38
4.3 Linker Switches 40

5 / Using a Batch Command File 41

6 / Compiling and Linking Large Programs..................43

6.1 Avoiding Limits on Code Size..........................44
6.2 Avoiding Limits on Data Size..........................45
6.3 Working With Limits on Compile Time Memory46
6.4 Working With Floating-Point Operations..........49
6.5 Working With Limits on Disk Memory............54
6.6 Minimizing Load Module Size..........................59

7 / Using Assembly Language Routines........................61

7.1 Calling Conventions....................................62
7.2 Internal Representations of Data Types............65
7.3 Interfacing to Assembly Language Routines.......67

8 / Advanced Topics..73

8.1 The Structure of the Compiler........................74
8.2 An Overview of the File System......................80
8.3 Runtime Architecture...................................83

Appendices..97

A / FORTRAN File Control Block................................99

B / Real Number Conversion Utilities........................103

C / Structure of External FORTRAN Files...................105

D / FORTRAN Scratch File Names..............................107

E / Link Error Messages...109

Index..113

Figures

Figure 1.1	Program Development	4
Figure 7.1	Contents of the Frame	62
Figure 7.2	Two-Byte Return Value	67
Figure 7.3	Four-Byte Return Value	67
Figure 7.4	Stack Before Transfer to IADD	68
Figure 7.5	Stack Before Transfer to RADD	70
Figure 8.1	The Structure of the Compiler	74
Figure 8.2	The Unit U Interface	81
Figure 8.3	Memory Organization	85
Figure 8.4	FORTRAN Program Structure	89

Tables

Table 3.1	Files Used by the MS-FORTRAN Compiler	24
Table 3.2	Conventional Filename Extensions	26
Table 3.3	Default File Specifications	26
Table 4.1	Linker Defaults	36
Table 4.2	LINK Switches	40
Table 8.1	Front End Compilation Procedures	76
Table 8.2	Unit Identifier Suffixes	83
Table 8.3	Error Code Classification	93
Table 8.4	Runtime Values in BRTEQQ	94

Introduction

The FORTRAN Compiler, also referred to as the MS™-FORTRAN Compiler, accepts programs written according to the subset FORTRAN standard, described in the document American National Standard Programming Language FORTRAN, ANSI X3.1978. The MS-FORTRAN language is described in the accompanying *FORTRAN Reference Manual*. This User's Guide explains how to use the MS-FORTRAN Compiler v.3.10 implemented for the Microsoft Disk Operating System, MS-DOS.

System Requirements

The MS-FORTRAN Compiler can be used with the Tandy personal computer having one floppy disk drive and a minimum of 256K random access memory available. (The MS-DOS utility CHKDSK will tell you how much RAM is available.)

Your machine must run MS-DOS 2.0 and have MS-LINK. MS-LINK is a standard MS-DOS utility.

Package Contents

The MS-FORTRAN Compiler package includes one disk and one documentation binder, containing one user's guide and one reference manual.

Software

The software for the MS-FORTRAN Compiler contains the following files on disk:

File	Contents
FOR1.EXE	Pass one of the MS-FORTRAN Compiler
PAS2.EXE	Pass two of the MS-FORTRAN Compiler
PAS3.EXE	Pass three of the MS-FORTRAN Compiler

FORTRAN.LIB	The MS-FORTRAN runtime library, for emulating real number arithmetic in software
FORTRAN.PEM	A map of the MS-FORTRAN runtime library FORTRAN.LIB
LINK.EXE	MS-LINK Linker
CONUXU.OBJ	Provides MSDOS 2.0 file support to software
FORUXU.OBJ	Provides MSDOS 2.0 file support to software
FILUXU.OBJ	Provides MSDOS 2.0 file support to software
NULF.OBJ	The dummy file system
NULE6.OBJ	The dummy errror system
ENTX6L.ASM	The assembler source of the execution control module that initializes and terminates every program
DEMO.FOR	Bubble sort demonstration program
README.DOC	If present, this file contains documentation that is more up to date than the printed documentation described in the following section.

Documentation

Documentation for the MS-FORTRAN Compiler is provided in the following two manuals:

MS-FORTRAN Compiler User's Guide

This manual provides an introduction to compiling and linking, a sample session, and a technical reference for the MS-FORTRAN Compiler.

MS-FORTRAN Reference Manual

This manual describes the grammar and use of the MS-FORTRAN language. With the exception of any recent changes noted in a README.DOC file, this is the language supported by the MS-FORTRAN Compiler.

About This Manual

The *MS-FORTRAN Compiler User's Guide* describes the operation of the MS-FORTRAN Compiler, from the most rudimentary procedures to more advanced topics that may be of interest only to experienced programmers.

The document assumes that you have a working knowledge of the MS-FORTRAN language and MS-DOS. For information on programming in FORTRAN, see "Learning More About FORTRAN" in this introduction.

Chapter 1, "Getting Started," discusses several procedures to do before compiling and linking your first program. It also describes the process of program development, and provides a short vocabulary for those who may be unfamiliar with the terms used in this document. Those who are already familiar with compiling and linking may wish to skip most of this chapter, but should read Section 1.1, "Preliminary Procedures," before proceeding.

Chapter 2, "A Sample Session," provides a step-by-step walk-through of each of the steps that follow the writing of a program: compiling, linking, and running.

Chapter 3, "More About Compiling," and Chapter 4, "More About Linking," supplement the material in Chapter 2 on compiling and linking, respectively.

Chapters 1 through 4 should be read in their entirety by the first-time user of the MS-FORTRAN Compiler.

Chapter 5, "Using a Batch Command File," and Chapter 6, "Compiling and Linking Large Programs," provide information on these topics for the programmer who has moved beyond the basics.

Chapter 7, "Using Assembly Language Routines," provides information for the experienced programmer who requires supplementary routines written in assembly language.

Chapter 8, "Advanced Topics," provides additional technical information on compiler structure, the MS-FORTRAN file system, and runtime architecture.

Appendices A through D provide information specific to the MS-DOS implementation of this version of the compiler.

Appendix E, "MS-LINK Error Messages," lists the MS-LINK error messages.

Syntax Notation

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

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

< > — Angle brackets indicate user-supplied data. When the angle brackets enclose lowercase text, you must type in an entry defined by the text (for example, <filename>). When the angle brackets enclose uppercase text, you must press the key named by the text (such as <RETURN>).

[ ] — Square brackets indicate that the enclosed entry is optional.

... — Ellipses indicate that an entry may be repeated as many times as needed or desired.

All other punctuation, such as commas, colons, slash marks, parentheses, and equal signs, must be entered exactly as shown.

Pressing the RETURN (or ENTER) key is assumed at the end of every line you enter in response to a prompt. Only if this is the only response required is the <RETURN> shown.

Learning More About FORTRAN

The manuals in this package provide complete reference information for your implementation of the MS-FORTRAN Compiler. They do not, however, teach you how to write programs in FORTRAN. If you are new to FORTRAN or need help in learning to program, we suggest you read any of the following books:

Agelhoff, R., and Mojena, Richard. Applied FORTRAN 77, Featuring Structured Programming. Wadsworth, 1981.

Ashcroft, J., Eldridge, R. H., Paulson, R. W., and Wilson, G. A. Programming With FORTRAN 77.Granada, 1981.

Friedman, F., and Koffman, E. Problem Solving and Structured Programming in FORTRAN. Addision-Wesley, 2nd edition, 1981.

Wagener, J. L. FORTRAN 77: Principles of Programming. Wiley, 1980

Chapter 1

Getting Started

1.1 Preliminary Procedures..............................2

1.1.1 Backing Up Your MS-FORTRAN Disk.....2

1.1.2 Setting Up Your MS-FORTRAN Disk.......2

1.2 Program Development..............................3

1.3 Vocabulary..7

1.1 Preliminary Procedures

This section describes several preliminary procedures, some of which are required and some of which are highly recommended before you begin the sample session or compile any programs of your own. If you are unfamiliar with any of the MS-DOS procedures mentioned, consult your MS-DOS manual for instructions.

1.1.1 Backing Up Your MS-FORTRAN Disk

This step is optional but highly recommended.

The first thing you should do when you have unwrapped your MS-FORTRAN disk is to make copies to work with, saving the original disk for backup. Make the copies using the COPY or DISKCOPY utilities supplied with MS-DOS.

1.1.2 Setting Up Your MS-FORTRAN Disk

This step is required.

You must have the file COMMAND.COM on the backup of your MS-FORTRAN disk in order to use the disk in any drive after booting MS-DOS. Therefore, you must copy COMMAND.COM to the backup of your MS-FORTRAN disk from your system disk (with the MS-DOS command COPY).

1.2 Program Development

This section provides a brief introduction to program development, a multistep process which includes first writing the program, and then compiling, linking, and running it. For a brief explanation of terms that may be unfamiliar, see Section 1.3, "Vocabulary."

A microprocessor can execute only its own machine instructions; it cannot execute source program statements directly. Therefore, before you run a program, some type of translation, from the statements in your program, to the machine language of your microprocessor, must occur.

Compilers and interpreters are two types of programs that perform this translation. Depending on the language you are using, either or both types of translation may be available to you. MS-FORTRAN is a compiled language.

A compiler translates a source program and creates a new file called an object file. The object file contains relocatable machine code that can be placed and run at different absolute locations in memory.

Compilation also associates memory addresses with variables and with the targets of GOTO statements, so that lists of variables or of labels do not have to be searched during execution of your program.

Many compilers, including the MS-FORTRAN Compiler, are what are called "optimizing" compilers. During optimization, the compiler reorders expressions and eliminates common subexpressions, either to increase speed of execution or to decrease program size. These factors combine to measurably increase the execution speed of your program.

The MS-FORTRAN Compiler has a three-part structure. The first two parts, pass one and pass two, carry out the optimization and create the object code. Pass three is an optional step that creates an object code listing. Compiling is described in greater detail in Section 2.2, "Compiling the Source File," and in Chapter 3, "More about Compiling."

Before a successfully compiled program can be executed, it must be linked. Linking is the process in which MS-LINK computes absolute offset addresses for routines and variables in relocatable object modules and then resolves all external references by searching the runtime library. The linker saves your program on disk as an executable file, ready to run.

You may, at link time, link more than one object module, as well as routines written in assembly language or other high-level languages, and routines in other libraries. Linking is described in greater detail in Section 2.3, "Linking the Object File With the Runtime Library," and in Chapter 4, "More about Linking."

Figure 1.1 illustrates the entire program development process.

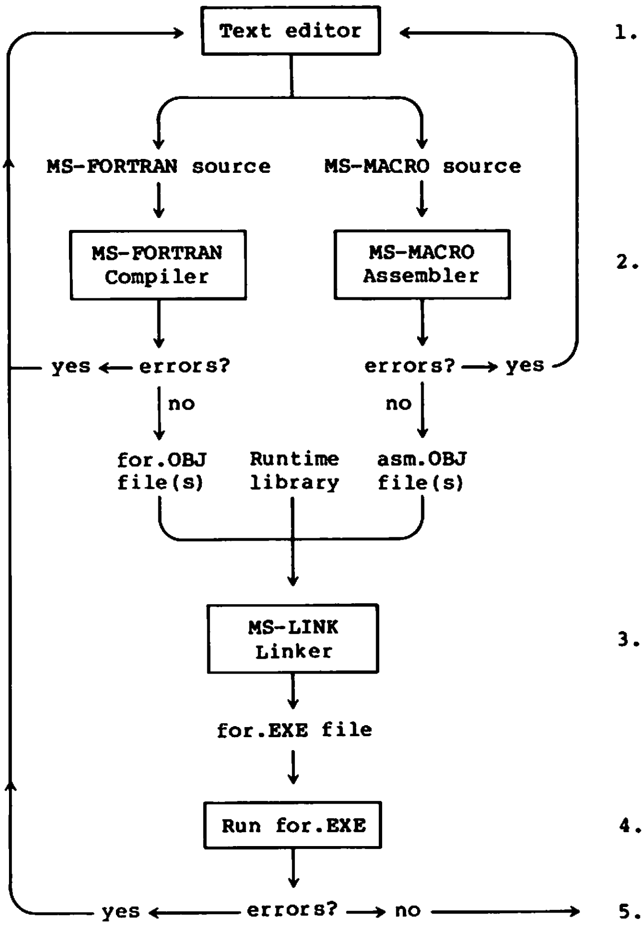


Figure 1.1. Program Development

1. Create and edit the MS-FORTRAN (and MS-Macro) source file.

   Program development begins when you write an MS-FORTRAN program; any general purpose text editor will serve the purpose. Use a text editor also to write any assembly language routines you may plan to include.

2. Compile the program with $DEBUG. Assemble the assembler source, if any.

   Once you have written a program, compile it with the MS-FORTRAN Compiler. The compiler flags all grammatical errors as it reads your source file. Include the $DEBUG metacommand in your source file to generate diagnostic calls for runtime errors. If compilation is successful, the compiler creates a relocatable object file.

   If you have written your own assembly language routines (for example, to increase the speed of execution of a particular algorithm),assemble those routines with the Microsoft Macro Assembler, MS-Macro.

   (You may have received MS-Macro as part of the utility package that came with your computer system. If not, it is available separately from your software dealer.)

3. Link the compiled (and assembled) OBJ files with the runtime library.

   A compiled (or assembled) object file is not executable and must be linked with one of the runtime libraries, using MS-LINK. Separately compiled MS-Pascal subroutines and functions can also be linked to your program at this time.

4. Run the EXE file.

   The linker links all modules needed by your program and produces as output an executable run file with .EXE as the extension. This file can be executed by simply typing its filename.

5. Recompile, relink, and rerun with $NODEBUG.

   Repeat these processes until your program has successfully compiled, linked, and run without errors. Then recompile, relink, and rerun with $NODEBUG to reduce the amount of time and space required. Chapter 6, "Compiling and Linking Large Programs," discusses how to work within the various physical limits you may encounter in compiling, linking, and executing a program.

1.3 Vocabulary

This section reviews some of the vocabulary that is commonly used in discussing the steps in program development. The definitions given are intended primarily for use with this manual. Thus, neither the individual definition nor the list of terms is comprehensive.

An MS-FORTRAN program is more commonly called a "source program" or "source file." The source file is the input file to the compiler and must be in ASCII format. The compiler translates this source and creates, as output, a new file called a "relocatable object file." The source and object files generally have the default extensions .FOR and .OBJ, respectively. After compiling, the object file must be linked with the runtime library to produce an executable program or run file. The run file has the extension .EXE.

Some other terms you should know are related to stages in the development and execution of a compiled program. These stages are:

1. Compile time

   The time during which the compiler is executing and during which it compiles an MS-FORTRAN source file and creates a relocatable object file.

2. Link time

   The time during which the linker is executing and during which it links together relocatable object files and library files.

3. Runtime

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

The following terms pertain to the linking process and the runtime library:

1. Module

   A general term for a discrete unit of code. There are several types of modules, including relocatable and executable modules.

The object files created by the compiler are said to be "relocatable", that is, they do not contain absolute addresses. Linking produces an "executable" module, that is, one that contains the necessary addresses to proceed with loading and running the program.

2. Routine

   Code, residing in a module, that represents a particular subroutine or function. More than one routine may reside in a module.

3. External reference

   A variable or routine in one given module that is referred to by a routine in another module. The variable or routine is often said to be "defined" in the module in which it resides.

   The linker tries to resolve external references by searching for the declaration of each such reference in other modules. If such a declaration is found, the module in which it resides is selected to be part of the executable module (if it is not already selected) and becomes part of your executable file. These other modules are usually library modules in the runtime library.

   If the variable or routine is found, the address associated with it is substituted for the reference in the first module, which is then said to be "bound." When a variable is not found, it is said to be "undefined" or "unresolved."

4. Relocatable module

   One whose code can be loaded and run at different locations in memory. Relocatable modules contain routines and variables represented as offsets relative to the start of the module. These routines and variables are said to be at "relative" offset addresses.

   When the module is processed by the linker, an address is associated with the start of the module. The linker then computes an absolute offset address that is equal to the associated address plus the relative offset for each routine or variable. These new computed values become the absolute offset addresses that are used in the executable file. Compiled object files and library files are all relocatable modules.

These offset addresses are still relative to a "segment," which corresponds to an 8086 segment register. Segment addresses are not defined by the linker; rather, they are computed when your program is actually loaded prior to execution.

5. Runtime library

   Contains the runtime routines needed to implement the MS-FORTRAN language. A library module usually corresponds to a feature or subfeature of the MS-FORTRAN language.

Chapter 2

A Sample Session

2.1 Preparing a MS-FORTRAN Program.............12
2.2 Compiling the Source File.........................13
2.2.1 Pass One..13
2.2.2 Pass Two..15
2.2.3 Pass Three..16
2.3 Linking the Object File With the Runtime Library..17
2.4 Executing Your MS-FORTRAN Program........19

You can easily see how to use your MS-FORTRAN package if you run a program.

To do this, follow these steps:

(1) Prepare a MS-FORTRAN program.
(2) Check the program for errors and then create a relocatable object file (by compiling it).
(3) Link the compiled OBJ files with the runtime library.
(4) Execute the program.

We assume that you have completed the necessary preliminary procedures described in section 1.1 and that you have one disk drive (A).

2.1 Preparing a MS-FORTRAN Program

Start up your computer and enter the date and time. MS-DOS prompts:

```
A>
```

Insert your backup of the MS-FORTRAN disk in Drive A.

You can create MS-FORTRAN programs with any available text editor. Source files should have the .FOR extension. For this introductory session, use the program DEMO.FOR as your source file, which is included on your MS-FORTRAN backup disk.

Remove the system disk from Drive A and insert the MS-FORTRAN backup disk.

The source file DEMO.FOR is now on the current drive.

2.2 Compiling the Source File

You must compile your source file so that it is translated into an object file that contains relocatable machine code. Compilation also associates memory addresses with variables and with the targets of GOTO statements.

Also, each time you run a program, it's good to check for syntax errors, which are usually simple spelling and word order errors in your statement lines. You can check by compiling the program.

Compiling your source code takes two or three passes. In this sample session we complete three passes.

2.2.1 Pass One

Pass one supplies the compiler with the filenames that you specify for the Object file, Source listing file, and Object listing file. It also writes the source listing to the specified file and detects any errors that prevent your program from running correctly.

To start Pass one type the following at the system prompt:

```
A:FOR1
```

FOR1 is the executable run file for Pass one of the MS-FORTRAN Compiler. The computer loads the compiler, displays the date and version number, and then prompts you for the name of the program:

```
Source filename [.FOR]:
```

For this session, beside the prompt type:

```
DEMO
```

This indicates that the source file is A:DEMO.FOR. All the following prompts provide the default filename in brackets.

Next the compiler prompts:

```
Object filename [DEMO.OBJ]:
```

Type **<RETURN>** to

indicate that the object filename is DEMO.OBJ.

Next the compiler prompts:

```
Source listing [NUL.LST]:
```

Type:

```
DEMO
```

to indicate that you want the source listing written to a file called DEMO.LST instead of the default of the NUL file (no file is created).

Next the compiler displays the final prompt:

```
Object listing [NUL.COD]:
```

When you answer this prompt, you must decide whether or not you wish to run the optional Pass 3 of the compiler, which creates the object code listing. If you wish to run Pass 3, specify a file other than the default (no file at all).

In this session we run the Pass 3, therefore type:

```
DEMO
```

This indicates that you want the object listing written to a disk file called DEMO.COD.

Compilation begins as soon as you respond to all four prompts. If your program is error free, the screen displays the following message when Pass one is complete:

```
Pass One        No Errors Detected.
```

In this case all the files (Object file, Source listing file, and Object listing file) are created and written to the file's whose names you previously requested by answering the prompts.

If the compiler detects errors during compilation, the screen shows a message similar to the following on your screen:

```
Pass One        3 Errors Detected.
```

Pass one creates two intermediate files, PASIBF.SYM and PASIBF.BIN. The compiler saves both these files on the current drive for use during Pass two. If the program contains errors, the compiler deletes these two files and does not let you run Pass two.

Refer to the source listing file, DEMO.LST, for the error messages that apply to your program. Next, check to see that you have correctly carried out all the required procedures described in Section 1.1, "Preliminary Procedures," and carefully redo each step of this sample session up to this point. See Appendix C, "Error Messages," in the *MS-FORTRAN Reference Manual* for a complete listing of the error messages you may encounter in MS-FORTRAN.

When your program is error free, of all errors you can run Pass two.

2.2.2 Pass Two

Pass two performs the following:

1. It reads the intermediate files PASIBF.SYM and PASIBF.BIN created in Pass one.
2. It writes the object file.
3. It deletes the intermediate files created in Pass one.
4. It writes two new intermediate files, PASIBF.TMP and PASIBF.OID, for use in Pass three if you requested an object listing in Pass one. Otherwise the compiler writes and later deletes just one new intermediate file, PASIBF.TMP. It writes these files to the current drive.

To start Pass two type:

```
A:PAS2
```

When Pass two is complete, your screen displays a message similar to the following:

```
Code Area Size  =  #05EC  ( 1516)
Cons Area Size  =  #00E6  (  230)
Data Area Size  =  #0264  (  612)

Pass Two     No Errors Detected.
```

The Lines 1-3 indicate, first in hexadecimal and then in decimal notation, the amount of space used by executable code (Code), constants (Cons), and variables (Data). The message concerning the number of errors refers to Pass two only, not to the entire compilation.

The object file is now on your diskette in the current drive. This object file contains relocatable machine code that can be placed and run at different absolute locations in memory.

2.2.3 Pass Three

Pass three runs successfully only if you requested an object listing by answering the last prompt in Pass one with any name other than the default NUL.COD.

This Pass reads PASIBF.TMP and PASIBF.OID, the intermediate files created during Pass two, and, because of your earlier response to the object listing prompt, writes the object code listing to the file DEMO.COD.

To start Pass three, at the system prompt type:

```
A:PAS3
```

When Pass three is complete, your screen displays the system prompt.

Pass three deletes the two temporary files. If, after requesting an object listing, you choose not to run Pass three, delete the temporary files (to save space).

See Chapter 3, "More About Compiling," for detailed information about filename conventions and how to respond to the compiler prompts.

2.3 Linking the Object File With the Runtime Library

Now you are ready to link your program. Linking converts the relocatable object file into an executable program by assigning absolute addresses and setting up calls to the runtime library.

To start the linker type:

```
A:LINK
```

Your computer loads the linker, displays a heading, and then prompts for the name of your relocatable object file (or files):

```
Object Modules [.OBJ]:
```

For this session beside the prompt type:

```
DEMO
```

This indicates that the file DEMO.OBJ, created during compilation, will link with FORTRAN.LIB during the linking process.

Next the linker prompts:

```
Run File [DEMO.EXE]:
```

Type <**RETURN**> to

indicate that you want the executable file to be named DEMO.EXE.

Next the linker prompts:

```
List File [NUL.MAP]:
```

For this session, beside the prompt type:

<**RETURN**>

indicates that you wish to accept the default, which is the NUL file, that is, no file at all.

If, when linking your programs, you wish to display the list file on your screen, but not write it to a diskette file, type

CON beside this prompt. If you want the linker map written to a diskette file, respond to this prompt with a name for the file.

Next the linker displays the final prompt, which is for the location of the libraries:

Libraries [.LIB]:

Type:

<RETURN>

To indicate that FORTRAN.LIB is on Drive A.

Now the linker links your compiled program, DEMO.OBJ, with the necessary modules in the MS-FORTRAN runtime library, A:FORTRAN.LIB. This linking process creates an executable file, named DEMO.EXE, on the diskette in the current drive. When linking is complete, the operating system prompt appears.

See Chapter 4, "More About Linking," for more information on the Linker.

2.4 Executing Your FORTRAN Program

Now you are ready to run your program. To run the sample program, just type:

```
DEMO
```

This command directs MS-DOS to load the executable file DEMO.EXE, fix segment addresses to their absolute value (based on the address at which the file is loaded), and start execution.

Assuming the program runs correctly, it prompts you to enter ten numbers. The program sorts these numbers and displays them on your screen in sorted order, from lowest to highest.

Chapter 3

More About Compiling

3.1 Files Written by the Compiler.......................22

3.1.1 The Object File.......................................22

3.1.2 The Source Listing File..........................22

3.1.3 The Object Listing File...........................23

3.1.4 The Intermediate Files...........................23

3.2 Filename Conventions...................................25

3.3 Starting the Compiler....................................29

3.3.1 Giving No Parameters on the Command Line.......................................29

3.3.2 Giving All Parameters on the Command Line.......................................3Ø

3.3.3 Giving Some Parameters on the Command Line.......................................31

This chapter provides additional procedural information on the compiler, supplementing the discussion in Section 2.2, "Compiling the Source File." For a more technical discussion of the compiler, see Section 8.1, "The Structure of the Compiler."

3.1 Files Written by the Compiler

In addition to creating several intermediate files, which it later reads and deletes, the compiler writes one required file and two optional files that represent your program in various ways. The object file is the one permanent file that must be created. The source listing and object listing files are optional; you may request that either or both of these be displayed or printed instead of being written to a disk file.

3.1.1 The Object File

The object file is written to disk after the completion of pass two of the compiler. It is a relocatable module, which contains relative rather than absolute addresses. Normally created with the .OBJ extension, the object module must be linked with the MS-FORTRAN runtime library to create an executable module containing absolute addresses.

3.1.2 The Source Listing File

The source listing file is a line-by-line account of the source file(s), with page headings and messages. Each line is preceded by a number that is referred to by any error messages that pertain to that source line.

Compiler error messages, shown in the source listing, are also displayed on your terminal screen. See Appendix C, "Error Messages," in the *MS-FORTRAN Reference Manual* for a complete list of MS-FORTRAN error messages.

If you include files in the compilation with the $INCLUDE metacommand, these files are also shown in the source listing. (For information on the $INCLUDE metacommand, see the entry for $INCLUDE in Section 6.2, "Metacommand Directory," in the *MS-FORTRAN Reference Manual.)*

The various flags, level numbers, error message indicators, and symbol tables in the source listing make it useful for error checking and debugging. Many programmers prefer a printout of the source listing file rather than of the source file itself as a working copy of the program.

3.1.3 The Object Listing File

The object listing file, a symbolic, assembler-like listing of the object code, lists addresses relative to the start of the program or module. Absolute addresses are not determined until the object file itself is linked with the runtime library.

The object listing file is used less often than the source listing file, but may be a useful tool during program development:

1. You can look at it simply to see what code the compiler generates and to familiarize yourself with it.
2. You can check to see whether a different construct or assembly language would improve program efficiency.
3. You use it as a guide when debugging your program with the MS-DOS DEBUG utility.

3.1.4 The Intermediate Files

Pass one creates two intermediate files, PASIBF.SYM and PASIBF.BIN, which incorporate information from your source file for use in creating the object file during Pass two. These two intermediate files are always written to the default drive.

Pass two reads and then deletes PASIBF.SYM and PASIBF.BIN. Pass two itself creates one or two new intermediate files, depending on whether or not you've requested an object listing. If, as for the sample session, you plan to run Pass three to produce the object listing, Pass two writes the two intermediate files, PASIBF.TMP and PASIBF.OID.

If in Pass one you do not request an object listing, Pass two writes and later deletes just one new intermediate file, PASIBF.TMP.

PAS2.EXE assumes that the intermediate files created in Pass one are on the default drive. If you have switched disks so that they are on another drive, you must indicate their location on the command that starts Pass two. For example:

```
A:PAS2 A/PAUSE
```

The "A" immediately following the command tells the compiler that PASIBF.BIN and PASIBF.SYM are on Drive A, instead of the default Drive B:. The "/PAUSE" tells the compiler to pause before continuing so that you can insert the disk that contains them into Drive A:.

After pausing, Pass two prompts as follows:

```
Press enter key to begin Pass two.
```

When you have inserted the new disk in Drive A:, press the RETURN key and the compiler proceeds with Pass two.

PASIBF.TMP and PASIBF.OID are deleted from the default drive during Pass three. If you change your mind after requesting an object listing file and decide not to run Pass three, be sure to delete these files to recover the space on your disk.

Table 3.1 is a summary of the files read and written by each of the three passes of the compiler.

Table 3.1. Files Used by the MS-FORTRAN Compiler

Pass	Reads	Writes	Deletes
1	DEMO.FOR	DEMO.LST PASIBF.SYM PASIBF.BIN	
2	PASIBF.SYM PASIBF.BIN	DEMO.OBJ PASIBF.OID* PASIBF.TMP	PASIBF.SYM PASIBF.BIN
3	PASIBF.OID* PASIBF.TMP	DEMO.COD	PASIBF.OID* PASIBF.TMP

* Refers to the action of the file as optional.

3.2 Filename Conventions

When you start up the compiler, it prompts you for the names of four files: your source file, the object file, the source listing file, and the object listing file. The only one of these names you must supply is the source filename.

This section describes how the compiler constructs the remaining filenames from the source filename and how you can override these defaults.

A complete filename specification under MS-DOS has three parts:

1. Device name

   The name of the disk drive where the file is or will be. On a single-drive machine, all device names default to A:. On multidrive machines, if you do not specify a device, the compiler assumes the currently logged drive.

2. Filename

   The name you give to a file. Consult your operating system manual for any limitations on assigning filenames.

3. Filename extension

   Added to the filename for further identification of the file. The extension consists of up to three alphanumeric characters and must be preceded by a period. Although you may give any extension to a filename, the MS-FORTRAN Compiler and MS-LINK recognize and assign certain extensions by default, as shown in Table 3.1.

Table 3.2. Conventional Filename Extensions

Extension	Function of File
.FOR	MS-FORTRAN source file
.PAS	MS-Pascal source file
.OBJ	Relocatable object file
.LST	Source listing file
.COD	Object listing file
.ASM	Assembler source file
.MAP	Linker map file
.LIB	Library file
.EXE	Executable run file

If you give unique extensions to your filenames, you must include the extension as part of the filename in response to a prompt. If you do not specify an extension, the MS-FORTRAN Compiler supplies one of those shown in Table 3.2.

Table 3.3. Default File Specifications

File	Device	Extension	Full File Spec
Source file	dev:	.FOR	dev:filename.FOR
Object file	dev:	.OBJ	dev:filename.OBJ
Source listing	dev:	.LST	dev:NUL.LST
Object listing	dev:	.COD	dev:NUL.COD

Table 3.2 also shows the default file specifications supplied by the compiler if you give a name for the source file and then press the RETURN key in response to each of the remaining compiler prompts.

The device "dev:" is the currently logged drive. Even if you specify a device with the source filename, the remaining file specifications will default to the currently logged drive. You must explicitly specify the name of another drive if that is where you want a particular file to go.

The NUL file is equivalent to creating no file at all; thus, by default, the compiler creates neither a source listing file nor an object listing file. If, in response to either of the last two prompts, you enter any part of a file specification, the remaining parts default as follows:

Source listing	dev:filename.LST
Object listing	dev:filename.COD

Neither listing file is created unless you explicitly request it. If you specify any non-null file for the object listing, Pass two leaves PASIBF.TMP and PASIBF.OID, the input files for Pass three, on your work disk until you delete them, either explicitly or by running Pass three.

The general rules for filenames may be summarized as follows:

1. All lowercase letters in filenames are changed into uppercase letters. For example, the following three names are all considered equivalent to ABCDE.FGH:

   abcde.fgh   AbCdE.FgH   ABCDE.fgh

2. To enter a filename that has no extension in response to a prompt, type the name followed by a period.

   For example, typing "ABC" in response to the source filename prompt gives a filename of ABC.FOR; typing "ABC." instructs the compiler to accept ABC with no extension as the name.

3. Leading and trailing spaces are permitted, so the following is an acceptable response to the source file prompt:

   ```
   ABC     ;
   ```

   The filename itself must not contain spaces.

4. You may override any defaults by typing all or part of the name instead of pressing the RETURN key. For example, if the currently logged drive is B: and you want the object file to be written to the disk in Drive A:, type "A:" in response to the following prompt:

   ```
   Object Filename [ABC.OBJ]:
   ```

   This results in a full filename of A:ABC.OBJ for the object file.

5. Listing files default to null. However, if you specify any part of a legal filename, the default changes so

that the compiler creates a filename with the same default rules that apply to the source and object files. Specifically, if you give a drive or extension, then the base name is the base name of the source file. For example, typing "B:" in response to the object listing prompt gives a filename of B:ABC.COD.

6. Typing a semicolon after the source filename or in response to any of the later prompts tells the compiler to assign the default filenames to all the remaining files. This is the quickest way to start the compiler if you don't need either of the listing files. For example, typing "ABC;" in response to the source file prompt eliminates the remaining prompts and results in the following filenames:

Source file	B:ABC.FOR
Object file	B:ABC.OBJ
Source listing	B:NUL.LST
Object listing	B:NUL.COD

   You may not enter a semicolon to specify a source file, since the source file has no default filename.

7. To send either listing file to your screen (console), use one of the special filenames USER or CON. USER is recognized only by MS-FORTRAN (and MS-Pascal) and writes to the screen immediately as the listing is created. CON is recognized by all MS-DOS programs, but saves the console output and writes it in blocks of 512 bytes.

3.3 Starting the Compiler

You can start the MS-FORTRAN Compiler in one of three ways:

1. You can let the compiler prompt you for each of the four filenames (as in the sample session).
2. You can give all four filenames on the command line.
3. You can give some of the filenames on the command line and let the compiler prompt you for the rest.

Each of these methods is discussed in the following sections. The second method, giving all four filenames on the command line, is particularly useful when you plan to use a batch command file. See Chapter 5, "Using a Batch Command File," for information.

3.3.1 Giving No Parameters on the Command Line

To start the compiler without giving any of the necessary parameters (filenames) on the command line, simply type the following:

```
A:FOR1
```

As in the sample session, the compiler prompts you for each of the four filenames it needs. A typical session might look like this (your responses are shown underlined):

```
Source filename    [.FOR]: MYFILE
Object filename    [MYFILE.OBJ]: <RETURN>
Source listing     [NUL.LST]: MYFILE
Object listing     [NUL.COD]: <RETURN>
```

This sequence of responses would give you an object file called B:MYFILE.OBJ, a source listing file called B:MYFILE.LST, and no object listing file.

Note: Pressing the RETURN key means that you accept the default shown in brackets; giving any part of a file specification creates a file with the same default rules that apply to other files.

3.3.2 Giving All Parameters on the Command Line

Instead of letting the compiler prompt you for each of the four filenames in turn, you may implicitly or explicitly give all four names on the same command line with which you start the compiler. This eliminates prompting for the filenames and is particularly useful when you are using the MS-DOS batch file facility. See Chapter 5, "Using a Batch Command File," for information on creating a batch command file for use with the compiler.

The general form of the command line that includes all of the compiler parameters is as follows:

```
A:FOR1 <source>,<object>,<sourcelist>,<objectlist>;
```

The same default naming conventions apply here as when you are prompted for the filenames.

You must separate each filename with a comma; spaces are optional. Put a semicolon at the end of the line to indicate that you do not want additional prompting.

If you omit a filename after a comma, the file by default is given the same filename as the source, the default device designation, and the default extension. Thus, these two command lines are equivalent:

```
A:FOR1 DATABASE,DATABASE,DATABASE,DATABASE;
A:FOR1 DATABASE,,,;
```

Both result in the following four filenames being assigned:

Source file	B:DATABASE.FOR
Object file	B:DATABASE.OBJ
Source listing	B:DATABASE.LST
Object listing	B:DATABASE.COD

If you want the normal defaults, with NUL listing files, use the semicolon (;) following the source filename. Thus, these command lines are equivalent:

```
A:FOR1 YOYO,YOYO,NUL,NUL;
A:FOR1 YOYO;
```

You may include spaces before or after filenames, but not within them.

3.3.3 Giving Some Parameters on the Command Line

You may also start the compiler by giving one or more of the required filenames on the command line and letting the compiler prompt you for the rest. This feature of the compiler makes it relatively failsafe to use.

For example, if you give only the names of the source file and the object file on the command line, the compiler will prompt you for the names of the source listing and the object listing (your responses are shown underlined):

```
Source listing [NUL.COD]:
Object listing [NUL.COD]:
```

This sequence of responses results in the following filenames:

Source file	B:TEST.FOR
Object file	B:TEST.OBJ
Source listing	B:TEST.LST
Object listing	B:NUL.COD

Chapter 4

More About Linking

4.1 Files Read by the Linker 34
4.1.1 Object Modules 34
4.1.2 Libraries 37
4.2 Files Written by the Linker 38
4.2.1 The Run File 38
4.2.2 The Linker Listing File 38
4.2.3 VM.TMP 39
4.3 Linker Switches 40

4.1 Files Read by the Linker

A successful MS-FORTRAN compilation produces a relocatable object file. Linking, the next step in program development, is the process of converting one or more relocatable object files into an executable program.

4.1.1 Object Modules

Object files can come from any of the following sources:

1. MS-FORTRAN compilands (programs, subroutines, or functions)
2. MS-Pascal compilands (programs, modules, or units)
3. user code in other high-level languages
4. assembly language routines
5. routines in standard runtime modules that support facilities such as error handling, heap variable allocation, or input/output

Interfacing to MS-Pascal or other high-level language routines is quite straight forward. All procedures that are referenced in an MS-FORTRAN routine and that are not defined in the same program unit are automatically considered to be external. No additional EXTERNAL declarations are required. For information on how to specify in another language that a routine is public, see the appropriate reference and user manuals for that language.

Calling conventions and function returns between MS-FORTRAN and other languages may differ. (See Chapter 7, "Using Assembly Language Routines," for MS-FORTRAN calling conventions and interface requirements.) You may need to write assembly language interface routines to interface between MS-FORTRAN and other languages. Whatever the language, it must be able to produce linkable object modules.

For further information on MS-LINK, see the appropriate chapter in your MS-DOS manual.

The ability to link together programs and subroutines of MS-

FORTRAN source code, as well as assembly language and library routines, allows you to develop a program incrementally. Separate compilation and later linking of separate parts of a program not only reduces the need for continual recompilation, it also allows you to create programs that contain more than 64K bytes of code. (See Chapter 6, "Compiling and Linking Large Programs.") Separate compilation may increase the size of your object module and run file, but will have no effect on the size of your executable program.

For now, assume that you have created a program that uses one MS-FORTRAN main program and one subroutine and also contains two assembly language external procedures. Assume further that these files have already been compiled or, in the case of the assembly language routines, already assembled and that the files thus created are the following:

```
PROG.OBJ
SUBR.OBJ
ASM1.OBJ
ASM2.OBJ
```

To link these all together, invoke the linker by typing the following:

```
A:LINK
```

Like the compiler, the linker gives a sequence of four prompts. Before linking can proceed, you must explicitly or implicitly supply the following pieces of information:

1. the name(s) of the object modules to be linked
2. the name to be given to the executable run file
3. the linker listing file
4. the names of any libraries to be searched (other than FORTRAN.LIB)

As with the compiler, responses to all except the first prompt may be supplied by defaults.

In response to the first linker prompt, enter the names of the object files, separated by plus signs as shown:

```
PROG + SUBR + ASM1 + ASM2
```

The first object file listed must be an MS-FORTRAN object file, although it need not be the main program. Do not put any assembly language module first; doing so may result in segments being ordered incorrectly. After the initial MS-FORTRAN object file, you may list the other subroutines or assembly language routines in any order.

Typing a semicolon after the name of the last object file you wish to link tells the linker to omit the remaining prompts and to supply defaults, as shown in Table 4.1, for all remaining parameters.

Table 4.1. Linker Defaults

Prompt	Default Response
Object modules	None
Run file	prog.EXE
List map	NUL.MAP
Libraries	FORTRAN.LIB

4.1.2 Libraries

A runtime library contains runtime modules that are required during linking to resolve references made during compilation. The MS-FORTRAN Compiler generates space for instructions for most floating-point operations. It also issues fixup information in the object file. During linking, these instructions are resolved using information in the runtime library. The instructions are transformed into emulator interrupts, which are serviced by code automatically linked in with your program. (This code is also in FORTRAN.LIB.)

If you press the RETURN key in response to the final linker prompt, the linker will automatically search for a library called FORTRAN.LIB on the default drive. If FORTRAN.LIB is not on the default drive, the following message will appear on your screen:

```
Cannot find library FORTRAN.LIB
Enter new drive letter:
```

Switch disks if necessary, and then type the name of the drive that does contain FORTRAN.LIB.

If instead you respond by just pressing the RETURN key, linking will proceed without a library search. You can achieve the same effect by using the linker option switch, /NO (short for /NODEFAULTLIBRARYSEARCH), to override the automatic search for FORTRAN.LIB. This will produce unresolved reference error messages unless you replace every required runtime routine with a routine of your own. (Most MS-FORTRAN programmers never require this capability.)

To instruct the linker to search other libraries (for example, PASCAL.LIB) as well as FORTRAN.LIB, give the library names, separated by plus signs, in response to the "Libraries" prompt. See your MS-DOS manual for complete information on using different libraries with MS-LINK.

4.2 Files Written by the Linker

The primary output of the linking process is an executable run file. You may also request a linker map or listing file, which serves much the same purpose as the compiler listing files. The linker, if need be, also writes and later deletes one temporary file.

4.2.1 The Run File

The run file produced by the linker is your executable program.

The default filename, given in brackets as part of the prompt, is taken from the name of the first module listed in response to the first prompt. To accept this prompt, press the RETURN key. To specify another run filename, type in the name you want. All run files receive the extension .EXE, even if you specify something else.

The linker ordinarily saves the run file, with the extension .EXE, on the disk in the default drive. To specify another drive, which may be necessary if your program is large, type a drive name in response to the run file prompt.

4.2.2 The Linker Listing File

The linker map, also called the linker listing file, shows the addresses, relative to the start of the run module, for every code or data segment in your program. If you request it, with the /MAP switch, the linker map can also include all EXTERN and PUBLIC variables. (See Section 4.3, "Linker Switches," for information on the /MAP switch).

The linker map defaults to the NUL file, unless you specifically request that it be printed, displayed on the screen, or saved on disk. In the early stages of program development, you may find it useful to inspect the linker map in these two instances:

1. when using the debugger to set breakpoints and locate routines and variables
2. to find out why a load module is so large (for example,

what routines are loaded, how big they are, and what's in them)

As the prompt indicates, the default for the linker map is the NUL file, that is, no file at all. Press the RETURN key to accept this default. If you wish to see the linker map but not have it written to a disk file, type "CON" in response to the list file prompt. If you want the file written to disk, give a device or filename.

4.2.3 VM.TMP

Linking begins after you have responded to all of the linker prompts. If the linker needs more memory space to link your program than is available, it will create a file called VM.TMP on the disk in the default drive and will display a message like the following:

```
VM.TMP has been created.
Do not change disk in drive B:.
```

If the additional space is used up or if you remove the disk that contains VM.TMP before linking is complete, the linker will abort.

When the linker has finished, VM.TMP will be erased from the disk, and any errors that occurred during linking will be displayed. (For a list of MS-LINK error messages, see Appendix F, "MS-LINK Error Messages.")

If the linker aborts, use the MS-DOS command DIR to check the contents of your disk to make sure that VM.TMP has been deleted. Then, to make sure the space has been released, use the CHKDSK program (supplied with MS-DOS). CHKDSK will reclaim any available space from unclosed files and tell you the total amount of available space on the disk.

4.3 Linker Switches

After any of the linker prompts, you may give one or more linker switches. Table 4.2 summarizes the linker switches you may use with MS-FORTRAN. See your MS-DOS manual for more information on linker switches and when and how to use them.

Table 4.2. LINK Switches

Switch	Action
/DSALLOCATE	Loads data at the high end of the data segment. For MS-FORTRAN and MS-Pascal programs, this switch is required and supplied automatically by the compiler.
/LINENUMBERS	Includes source listing line numbers and associated addresses in the linker listing, which allows you to correlate machine addresses with source lines when debugging. This correlation is also available on the object listing.
/MAP	Includes all EXTERN and PUBLIC variables in the linker list file.
/NO	Tells the linker to not automatically search FORTRAN.LIB. (/NO is short for NODEFAULTLIBRARYSEARCH.)
/PAUSE	Tells MS-LINK to display the following message: About to generate .EXE file Change disks <press RETURN> You may then change disks before the linker continues.

The /PAUSE switch is particularly useful for linking large programs, since it allows you to switch disks before writing the run file. However, if a VM.TMP file is created, you must not switch the disk in the default drive.

Note: For MS-FORTRAN and MS-Pascal programs, do not use either of the additional linker switches /HIGH or /STACK.

Chapter 5

Using A Batch Command File

The MS-DOS batch file facility lets you create a batch file for executing a series of commands. This facility is described fully in your MS-DOS manual. This chapter provides a brief description of command files in the context of compiling, linking, and running an MS-FORTRAN program.

A batch command file is a text file of lines that are MS-DOS commands. If a batch file is open when MS-DOS is ready to process a command, the next line in the file becomes the command line. After processing all batch command lines (or if batch processing is otherwise terminated), MS-DOS goes back to reading command lines from the screen.

Batch file lines cannot be read by the compiler, the linker, or a user program. Thus, you cannot put responses to filename or other prompts in a batch file. All compiler parameters must be given on the command line, as described in Section 3.3.2, "Giving All Parameters on the Command Line."

The batch file may contain dummy parameters that you replace with actual parameters when you invoke it. The symbol %1 refers to the first parameter on the line, %2 to the second parameter, and so on. The limit is %9. A batch command file must have the extension .BAT and should be kept on either the program disk or the utility disk.

The PAUSE command, followed by the text of the prompt, tells the operating system to pause, display a prompt (which you have defined), and wait for some further input before continuing.

If your program is already debugged and you are making only minor changes to it, you can speed up the compilation process by creating a batch file that issues the compile, link, and run commands.

For example, use the line editor in MS-DOS to create the following batch file, COLIGO.BAT:

```
A:FOR1 %1,,;
PAUSE ...If no errors, insert PAS2 disk in drive A:.
A:PAS2
PAUSE ...Insert runtime libraries disk in drive A:.
A:LINK %1;
%1
```

To execute this file, type:

```
COLIGO DEMO
```

DEMO is the name of the source program you want to compile, link, and run.

1. The first line of the batch file runs Pass one of the compiler.
2. The second line generates a pause and prompts you to insert the Pass two disk.
3. The third line runs Pass two.
4. The fourth line generates a pause and prompts you to insert the runtime library.
5. The fifth line links the object file.
6. The sixth line runs the executable file.

A BAT file is only executed if there is neither a COM file or EXE file with the same name. Thus, if you keep your source file and BAT file on the same disk, give them different filenames.

For more information about batch command files, see your MS-DOS manual.

Chapter 6

Compiling And Linking Large Programs

6.1 Avoiding Limits on Code Size....................47
6.2 Avoiding Limits on Data Size....................45
6.3 Working With Limits on Compile Time Memory....................46
6.3.1 Identifiers....................46
6.3.2 Complex Expressions....................47
6.4 Working With Floating Point Operations...49
6.4.1 Floating-Point Overflows....................49
6.4.2 "Short" Arithmetic....................49
6.4.3 $FLOATCALLS and $NOFLOATCALLS Metacommands....................5Ø
6.4.4 Environment Control And Exception Handling For Real Math....................51
6.5 Working With Limits on Disk Memory......54
6.5.1 Pass One....................54
6.5.2 Pass Two....................55
6.5.3 Linking....................56
6.5.4 A Complex Example....................57
6.6 Minimizing Load Module Size....................59
6.6.1 I/O....................59
6.6.2 Runtime Error Handling....................6Ø
6.6.3 Real Number Operations....................6Ø
6.6.4 Debugging....................6Ø

Occasionally, you may find that a large program exceeds one or more physical limits on the size of program the compiler, the linker, or your machine can handle. This chapter describes some ways to avoid or work within such limits.

6.1 Avoiding Limits on Code Size

The upper limit on the size of object code that can be generated at once by the MS-FORTRAN Compiler is 64K bytes. This limit applies only to generated code; data size limits are discussed in the following section.

Since you can compile any number of compilands separately and link them together later, the real limit on program size is not 64K but the amount of main memory available. For example, you can separately compile six different compilands of 50K bytes each. Linking them together produces a program with a total of 300K bytes of code.

In practice, a source file large enough to generate 64K bytes of code would be thousands of lines long and unwieldy both to edit and to maintain. A better practice is to break a large program into subroutines and functions and compile logical groups of them separately. Separate compilation will have no effect on final program size, but may increase the total size of object files.

6.2 Avoiding Limits on Data Size

Overall program limits for data are as follows:

1. 64K for all "local" variables, constants, blank COMMON blocks, the stack, and the heap. The heap is used for dynamically allocated files (634 bytes per file) and for some entry and exit information if the $DEBUG metacommand is on.

   The stack contains arguments, return addresses, and certain temporary variables. This data will reside in one segment, the default data segment (DS register).

2. 64K bytes for each named COMMON block. Each named COMMON block will be allocated a separate segment. There are no additional limits for data for separate compilations. References to data in named COMMON blocks are usually less efficient than references to other data.

6.3 Working With Limits on Compile Time Memory

During compilation, large programs are most often limited in the number of identifiers in any one source file. They are occasionally limited by the complexity of the program itself. If one of these limits is reached, you will see the following error message:

```
Compiler Out Of Memory
```

There is no particular limit on number of bytes in a source file. The number of lines is limited to 32767, but in practice, any source file this big will run into other limits first.

6.3.1 Identifiers

Pass one of the compiler can handle a maximum of around 1000 identifiers, assuming your memory is big enough to provide a full data segment of 64K. In MS-FORTRAN, identifer entries are created for the following objects:

1. the program
2. subroutines and functions declared in the program unit
3. subroutines and functions referenced in the program unit
4. COMMON blocks
5. common variables
6. statement functions
7. formal parameters
8. "local" variables

Identifiers of objects 5 through 8 are required only while the subroutine or function that contains them is being compiled. These identifiers are discarded at the end of the subroutine and the space they used is made available for other identifiers.

Hence, you can create much bigger programs by splitting up your code into more subroutines and functions. Such a practice allows the "local" identifier space to be shared.

You can go even further by placing the subroutines and functions in files of their own and compiling them separately, since this usually reduces the number of identifiers in groups being used per compiland.

Remember that you may have to create data items in common to communicate between the new procedures, or preferably, from the point of view of good program structure, write communication subroutines. However, either of these may tend to defeat the purpose of breaking up the program in the first place.

6.3.2 Complex Expressions

It is also possible to run out of memory in Pass one with any of the following cases:

1. a very complex statement or expression (i.e., one that is very deeply nested)
2. a large number of error messages
3. a very large block of specification statements (EQUIVALENCE statements in particular)

Usually, if a program gets through Pass one without running out of memory, it will get through Pass two. The major exception occurs with complex basic blocks, as in either of the following:

1. sequences of statements with no labels or other breaks
2. sequences of statements containing very long expressions or parameter lists (especially with I/O statements)

Also, Pass two uses symbol table entries for objects 1 through 4 in Section 6.3.1. Unlike Pass one, Pass two also creates entries for many of the transcendental functions that are called by a program. However, these are limited in number. In any case, Pass two makes a smaller number of symbol table entries than Pass one.

If Pass two runs out of memory, it displays the message:

```
Compiler Out Of Memory
```

The error message will give a line number reference. If there is a particularly long expression or parameter list near this line, break it up by assigning parts of the expression to local variables (or using multiple WRITE calls). If this does not work, add labels to statements to break the basic block.

6.4 Working With Floating-Point Operations

MS-FORTRAN version 3.1 by default, generates in-line, floating-point instructions to perform floating-point operations. In-line code provides the most efficient use of memory. You must have these instructions emulated by interrupt-driven software.

Interrupt handling and address translation with the emulator library significantly contributes (up to 25 percent) to the execution time of basic floating-point operations. The REAL arithmetic support routines provided by FORTRAN.LEM contribute approximately 8K bytes to your program. Note that if your program does not use floating point operations you can use NULR7.OBJ to regain this space (see section 6.6.3 of this manual).

6.4.1 Floating-Point Overflows

A floating point overflow, in either direct or emulated mode, generates a "not-a-number" (NAN) error. The message appears in the output field as asterisks (*) or the letters "NAN", depending on the choice of formats.

6.4.2 "Short" Arithmetic

By default, the emulator library does all its arithmetic with 64 bits of precision (corresponding to an 80-bit representation) as specified by the IEEE standard. However, 64-bit arithmetic is relatively slow and may be more precise than you require. MS-FORTRAN version 3.1 supports an alternative 23-bit precision arithmetic. You can select the precision you want by using one of the following subroutines which are provided in the emulator library:

```
SUBROUTINE MPSRQQ
```

The MPSRQQ subroutine causes all subsequent floating-point operations to be carried out to 23 bits of precision, even if either or both of their arguments is double precision.

SUBROUTINE MPBRQQ

The MPBRQQ subroutine causes the precision of subsequent operations to be 23 bits if, and only if, both arguments are single precision or the result of a previous 23-bit operation.

SUBROUTINE MPDRQQ

The MPDRQQ subroutine restores the default state where all operations are performed to 64-bit precision.

INTEGER*2 FUNCTION MPIRQQ

The MPIRQQ function returns a value indicating the current precision status as follows:

0: default 64-bit
1: the precision depends on the operands
2: unconditional 23-bit arithmetic

You may call these subroutines anywhere in your program. "Short" arithmetic is about 25 percent faster for addition and subtraction, 50 percent faster for multiplication, and 75 percent faster for division. The code for "short" arithmetic is only included in your program if you call one of the previously listed subroutines.

6.4.3 $FLOATCALLS and $NOFLOATCALLS Metacommands

You can compile your program with either the $FLOATCALLS metacommand or the $NOFLOATCALLS metacommand (the default). When you use the $FLOATCALLS metacommand the compiler generates subroutine calls to do floating-point operations. This is faster than using in-line interrupt instructions, but requires more code space. To increase precision, but decrease execution speed, call the MPSRQQ, MPBRQQ or MPDRQQ emulator library subroutines.

When you use the $NOFLOATCALLS metacommand the compiler generates interrupt instructions to do floating-point operations. This results in compact, executable programs, but may be slow in execution speed.

You can increase execution speed at the cost of some precision by calling the MPSRQQ or MPBRQQ emulator library subroutines. Likewise you can restore precision at the cost of execution speed by calling the MPDRQQ emulator library subroutine.

For more information on the $FLOATCALLS and $NOFLOATCALLS metacommands refer to section 6.2.3 of your MS-FORTRAN Reference Manual.

6.4.4 Environmental Control And Exception Handling For Real Math

The five exceptions required by the IEEE Real Math Standard are supported by the Real Math support routines. By default they are disabled. Contrary to the description in the *MS-FORTRAN Reference Manual* (section 6.2.1) the $DEBUG metacommand does not control the handling of these exceptions; however, you should continue to use it to control the Integer arithmetic errors. Instead, there are two memory locations that control both processors. These are called the CONTROL and STATUS words. The effect of these words is discussed below. You can read or set their values using the subroutines in the following program:

```
      SUBROUTINE SCRWQQ (CW)
      INTEGER*2 CW
C   Sets the control word to the value in CW

      SUBROUTINE LCWRQQ (CW)
      INTEGER*2 CW
C   CW is set to the value of the control word

      SUBROUTINE SSWRQQ (SW)
      INTEGER*2 SW
C   Sets the status word to the value in SW

      SUBROUTINE LSWRQQ (SW)
      INTEGER*2 SW
C   SW is set to the value of the status word
```

The five IEEE Real Math Standard exceptions are:

1. Invalid Operation - Any operation with a NAN (not a number), square root (-1), Ø * INF, etc. Generally returns a NAN.

2. Divide by zero - Returns properly signed INF.
3. Overflow - Occurs when any number is greater than the maximum representable number. Returns INF.
4. Underflow - Occurs when any number is smaller than the smallest valid representable number. Returns a Denormal or a zero.
5. Precision - Occurs whenever a result is subjected to rounding error. Informs that the result is not exact. Returns properly rounded result.

When one of these exceptional conditions occurs the appropriate bit in the status word is set. This flag remains set to indicate that the exception occured until cleared by the user. If the bit in the control word relating to a given exception is set then that exception is masked and the operation proceeds with a supplied default. If the bit is unset any exception of that type generates an error message, halts the operation and your program will stop. In either case the exception is ORed into the STATUS word.

The CONTROL word is also used to set modes for the internal arithmetic required by the IEEE standard. These are:

Rounding Control - round to nearest (or even), Up, Down, or Chop

Precision Control - Determines at which bit of the mantissa rounding should take place. (24, 53, or 64). Note all results are done to 64 bits regardless of the precision control. It only affects the rounding in the internal form. On storage any result is again rounded to the storage precision.

Infinity Control - Affine mode is the familiar + and - INF style of arithmetic. Projective mode is a mode where + and - INF are considered to be the same number. The principal effect is to change the nature of comparisons (Projective INF does not compare with anything but itself).

Format for STATUS BYTE and CONTROL WORD

```
               15                   8  7  6  5   4   3   2   1   0
STATUS  | hi byte unused            |  |  |PE|UE|OE|ZE|   |IE|
  Precision Exception-------------------------┘  |  |  |      |
  Underflow Exception----------------------------┘  |  |      |
  Overflow Exception--------------------------------┘  |      |
  Zero Divide Exception--------------------------------┘      |
  Invalid Exception-------------------------------------------┘
        (All other bits unused, may be either 1 or 0)
```

```
           15  14  15  12  11-10  9-8  7  6  5   4   3   2   1   0
CONTROL|   |   |   |IC|  RC  | PC |  |  |PM|UM|OM|ZM|   |IM|
  Infinity Control---------┘      |      |     |  |  |  |      |
  Round Control-------------------┘      |     |  |  |  |      |
  Precision Control----------------------┘     |  |  |  |      |
  Precision Mask-------------------------------┘  |  |  |      |
  Underflow Mask----------------------------------┘  |  |      |
  Overflow Mask--------------------------------------┘  |      |
  Zero Divide Mask--------------------------------------┘      |
  Invalid Mask-------------------------------------------------┘
        (All other bits unused, may be either 1 or 0)
```

Infinity Control
- 0 = Projective
- 1 = Affine

Round Control
- 00 = Round nearest or even
- 01 = Round down (toward -INF)
- 10 = Round up(toward +INF)
- 11 = Chop (Truncate toward 0)

Precision Control
- 00 = 24 bits of mantissa
- 01 = (reserved)
- 10 = 53 bits of mantissa
- 11 = 64 bits of mantissa

6.5 Working With Limits on Disk Memory

Another type of limit you may encounter is in the number of disk drives on your computer or the maximum file size on one disk. As with other limits, there are several possible solutions discussed in the following sections.

The simplest method of avoiding these limits is to first load a compiler pass, then switch disks and run the Pass.

6.5.1 Pass One

For FOR1.EXE, just type "FOR1" (or "dev:FOR1" if necessary) to load Pass one. When the "Source File" prompt appears, you can remove the disk containing FOR1.EXE. If you have a single-drive system, replace the system disk with the disk containing your source file. FOR1.EXE will write its intermediate files on the same disk.

If you have a two-drive system, insert your source file in the nondefault drive. Since the intermediate files are always written to the default drive, you will need to give an explicit device (i.e., drive) letter for your source file. Typically a source listing file would go on the same drive as the source.

If your source file will not fit on one disk, you can break it into pieces and use the $INCLUDE metacommand to compile the pieces as a group.

These $INCLUDE files can be typed at your screen (or console). Just give "USER" as the name of your source file, and type your $INCLUDE metacommands directly, one per line. You will need to type <CONTROL-Z> (end-of-file) to end the compilation.

If your source file doesn't fit on one disk, your source listing file will not fit either, so you will need to send it directly to the printer.

Another way to control a large listing file is by including the $NOLIST metacommand at the beginning of your source file, and then using the $LIST and $NOLIST metacommands to bracket only those portions of the source for which a source listing is required. In particular, you may want to exclude $INCLUDEd files when compiling subprograms.

6.5.2 Pass Two

Two command line parameters available with Pass two can help you with disk limitations.

1. You can indicate a drive letter on which your intermediate files, PASIBF.SYM and PASIBF.BIN, can be found.
2. The /PAUSE switch tells Pass two to pause while you remove the disk containing PAS2.EXE and insert some other disk.

For example, if you have a single-drive system, insert your PAS2.EXE disk and type "PAS2 /PAUSE". After PAS2.EXE is loaded, you will see the message:

```
Press ENTER key to begin Pass two
```

Take out the PAS2.EXE disk and insert the disk with the intermediate file from Pass one. Now press the ENTER (i.e., RETURN) key, and Pass two will run.

If you have two drives, but you run out of disk memory when executing Pass two you need to have the files PASIBF.SYM and PASIBF.BIN on one drive and the intermediate file PASIBF.TMP (and PASIBF.OID if you are making an object listing file) on the other drive.

The PASIBF.TMP file (and the PASIBF.OID file used in Pass three) are always written to the default drive.

Give Pass two a drive letter to specify the drive containing the PASIBF.SYM and PASIBF.BIN files; for example, "PAS2 B". Normally, you would also need the pause command; for example, "PAS2 B/PAUSE". Pass two will respond with a message such as the following:

```
PASIBF.SYM and PASIBF.BIN are on B:
```

This message is followed by the pause prompt:

```
Press enter key to begin Pass two
```

When you run Pass two with the PASIBF files on two disks, the object file should usually go on the same disk as PASIBF.TMP (and PASIBF.OID); that is, in the default drive. If it doesn't quite fit, and you are making an object listing file, you could compile your program twice, once without the object listing but with the object file itself, and once with an object listing but with NUL used for the object file.

6.5.3 Linking

If you are making a large program with small disks (or only one disk drive), you may run into similar problems when you link your program. Since you can split your program into pieces and compile them separately, but you must link the entire program at one time, you may run into disk limitations in the linker but not the compiler.

The linker will prompt you for any object files and/or libraries it cannot find, so you can swap in the correct disk and continue linking. Also, the /PAUSE switch makes the linker wait after linking but before writing the run (EXE) file, so you can create a run file that fills an entire disk. However, creation of the virtual file, VM.TMP, and the link map limit the amount of disk swapping you can do.

On a single-drive system:

1. Load the linker by typing "LINK".
2. Remove the disk containing LINK.EXE and insert the disk containing your object file(s) and, if there is room, any libraries.
3. Respond normally to the linker prompts, except to include the /PAUSE switch with the run file if you want the run file on another disk.

Unless all object files, libraries, and the run file will fit on one disk, you must not write the linker listing to a disk file. Instead, send the linker map to NUL, CON, or directly to your printer. Since the map is written at various points in the linking process, you cannot swap the disk on which the map is written.

The linker will prompt you when it needs an object file, a library file, or is about to write the run file; exchange disks as necessary when this happens. If the linker gives a message that it is creating VM.TMP, its virtual memory file, you cannot switch disks anymore, so you may not be able to link without more memory or a second disk drive.

With two disk drives, you can devote one drive (the default) to the VM.TMP file (and to the link map, if you want one). Use the other drive for your object files, libraries, and run file (using the /PAUSE switch). With this method, you can link very large programs.

The linker makes two passes through the object files and libraries: one to build a symbol table and allocate memory, and one to actually build the run file. This means you will insert a disk containing object files or libraries twice, and finally insert the disk that will receive your run file.

6.5.4 A Complex Example

The following example illustrates compiling and linking a very large program. The example assumes that the machine has two drives and that the programmer doesn't want any of the listing files.

Pass one

1. Log onto Drive B: and insert an empty disk in B:.
2. Insert the disk containing FOR1.EXE in Drive A:, type "A:FOR1", and wait for the "Source File" prompt.
3. Remove the disk containing FOR1.EXE from A: and insert the disk containing the source file LARGE.PAS.
4. Respond to the "Source File" prompt with "A:LARGE,A:LARGE", and wait for Pass one to run.

Pass two

1. Log onto Drive A:. Remove source disk from A:.
2. Insert the disk containing PAS2.EXE in A:, type "PAS2 B/PAUSE" and wait for the Pass two prompt.
3. Remove the disk containing PAS2.EXE from A:, insert an empty disk (to which the object file will be written).
4. Respond to the pass two prompt by pressing the RETURN key, and wait for Pass two to run.
5. Remove the disk containing the object file from A:.

Linking

1. Log onto Drive B: (which contains a now-empty disk).
2. Insert LINK.EXE in A:. Type "A:LINK" and wait for the "Object Modules" prompt.
3. Remove the disk containing LINK.EXE from A: and insert the disk containing the object file(s).
4. Respond to the "Object Modules" prompt by typing "A:LARGE" (plus any other object files).
5. Respond to the "Run File" prompt by typing "LARGE/PAUSE".
6. Respond to the "List File" prompt by pressing the RETURN key, or type "B:LARGE" to get a linker map.
7. Respond to the "Libraries" prompt by pressing the RETURN key or with a library name (the library must be on A:).
8. Wait for the linker to run, swapping the A: disk after prompts as necessary.

6.6 Minimizing Load Module Size

Some MS-FORTRAN load modules can be reduced in size by eliminating runtime modules your program doesn't use. Reductions can be made in several areas:

1. I/O
2. runtime error messages
3. real number operations
4. debugging

6.6.1 I/O

Because most MS-FORTRAN programs perform I/O, they require linking to the MS-FORTRAN file system in the runtime library. However, some programs do not perform I/O and others perform I/O by directly calling MS-FORTRAN "Unit U" file routines or calling operating system I/O routines. (For more information on Unit U, see Section 8.2, "An Overview of the File System.")

Nonetheless, all programs include calls to INIVQQ and ENDYQQ, the procedures that initialize and terminate the file system. These calls increase the size of the load module by linking and loading routines that may never be used.

If a program doesn't need the file system routines, you can eliminate unnecessary file support by declaring dummy INIVQQ and ENDYQQ subroutines in your program, as follows:

```
SUBROUTINE INIVQQ
END

SUBROUTINE ENDYQQ
END
```

The linker will still load the Unit U procedures necessary to access the terminal (INIUQQ, ENDUQQ, PTYUQQ, PLYUQQ, and GTYUQQ), so that it can write any runtime error messages.

However, if you do include the dummy subroutines described, and the linker produces any error messages for global names that end with the "VQQ" or "UQQ" suffix, your program requires the file system and the process described above will not work.

6.6.2 Runtime Error Handling

If runtime error messages are not required, the load module can be further reduced in size by eliminating the error message module and the Unit U procedures. Two null object modules are provided as replacements: NULF.OBJ and NULE6.OBJ. NULF.OBJ contains the dummy subroutines for INIVQQ and ENDYQQ, as well as dummies for INIUQQ and ENDUQQ. NULE6.OBJ replaces EMSEQQ and provides simple termination of a program if an error occurs.

6.6.3 Real Number Operations

If an MS-FORTRAN program does no real number operations, it doesn't require INIX87 and ENDX87 modules that initialize and terminate the real number support system. The dummy object module NULR7.OBJ provides dummy routines for these two modules.

6.6.4 Debugging

Compiling and linking a program with the $DEBUG metacommand may generate up to 40 percent more code than with $NODEBUG. Therefore, after a program has been successfully compiled, linked, and run, remove the $DEBUG from your source file and repeat the entire process to create a program that will run considerably faster.

Chapter 7

Using Assembly Language Routines

7.1 Calling Conventions....62

7.2 Internal Representations of Data Types.....65

7.3 Interfacing to Assembly Language Routines....67

This chapter first describes the MS-FORTRAN calling conventions and internal representations of data types, and then shows how to interface 8086 assembly language routines to MS-FORTRAN programs. The information in this chapter is not required for most MS-FORTRAN programs and is intended primarily for the experienced programmer who is familiar with the following material:

1. the EXTERNAL statement (see Section 3.2.17, "The EXTERNAL Statement," in the *MS-FORTRAN Reference Manual* for a description of the statement)
2. subroutine and function arguments (see Section 3.2.4, "The CALL Statement," in the *MS-FORTRAN Reference Manual* for a description of the statement)
3. MS-Macro Assembler (see your MS-DOS manual)

7.1 Calling Conventions

At runtime, each active subroutine or function has a "frame" allocated on the stack. The frame contains the data shown in Figure 7.1.

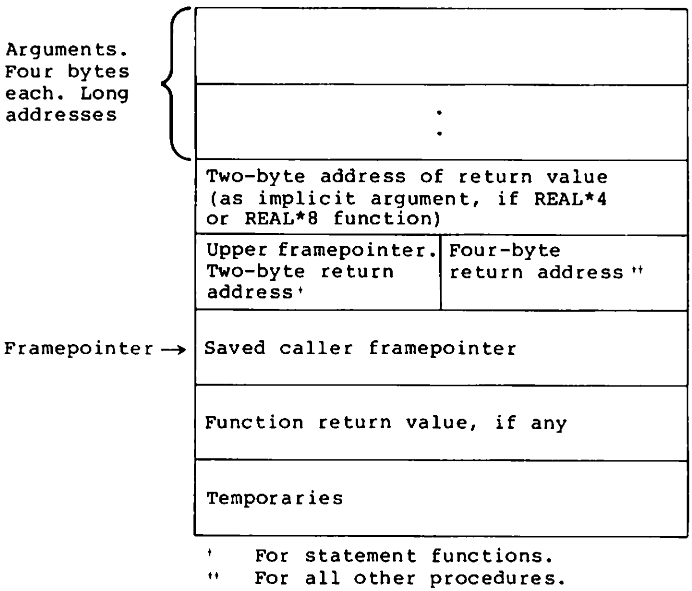


Figure 7.1. Contents of the Frame

The framepointer points at the saved caller framepointer, below the return address, and is used to access frame data. A statement function nested within another subroutine or function has an upper framepointer, so that it can access variables in the enclosing frame.

The following takes place during a procedure or function call:

1. The caller saves any registers it needs (except the framepointer).
2. The caller pushes parameters in the same order as they are declared in the source and then performs the call.
3. The called routine pushes the old framepointer, sets up its new framepointer, and allocates any other stack locations needed.

To return to the calling routine, the called routine restores the caller's framepointer, releases the entire frame, and returns. Not all of these steps need necessarily be taken in an assembly language routine. You must only ensure that the framepointer is not modified and that the entire frame, including all parameters, is popped off the stack before returning. For further information on the assembly language interface, see Section 7.3, "Interfacing to Assembly Language Routines."

A function always returns its value in registers. For REAL*4 and REAL*8, the caller allocates a frame temporary for the result and passes the address to the function like a parameter. When the called routine returns, it places the address back in the normal return register.

In MS-FORTRAN, all such subroutines and functions are PUBLIC or EXTERN. All calls to subroutines or functions are long calls (i.e., have four-byte addresses). All calls to statement functions are short calls (i.e., have two-byte addresses).

The called routine must save the BP register, which contains the MS-FORTRAN framepointer as well as the DS segment register. The SS register is used by interrupt routines, both user-declared and 8087 support, to locate the default data segment, and so must not be changed (at least, if interrupts are enabled). Other registers (AX, BX, CX, DX, SI, DI, and ES) need not be saved.

Functions return a one-byte value in AL, a two-byte value in AX, and a four-byte value in DX:AX (high part:low part, or segment:offset).

7.2 Internal Representations of Data Types

This section describes the internal representation of MS-FORTRAN data types. Programmers who use both MS-FORTRAN and MS-Pascal should pay particular attention to the data type and parameter passing differences when passing data between the two languages. For internal representations of MS-Pascal data types, see the *MS-Pascal Compiler User's Guide.*

1. Integer (INTEGER*2 and INTEGER*4)

   INTEGER*2 values are 16-bit two's complement numbers; INTEGER*4 values are 32-bit two's complement numbers.

2. Real (REAL*4) and double precision (REAL*8)

   Reals are IEEE 4-byte real numbers. They have a sign bit, an 8-bit excess 127 binary exponent, and a 24-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives an exponent range of 10^38 and 7 digits of precision. The maximum real number is normally 1.701411E37. The most significant byte contains the sign bit and the most significant bits of the exponent. The least significant byte contains the least significant bits of the mantissa.

   Double precision real numbers are IEEE 8-byte real numbers. They have a format similar to 4-byte REAL numbers. The exponent is 11-bits (excess 1023), and the mantissa has 52 bits (plus the implied high-order 1 bit).

3. Logical (LOGICAL*2 and LOGICAL*4)

   LOGICAL*2 values occupy two bytes. The least significant (first) byte either is 0 (.FALSE.) or 1 (.TRUE.); the most significant byte is undefined. LOGICAL*4 variables occupy two words, the least significant (first) of which contains a LOGICAL*2 value. The most significant word is undefined.

4. Character

   Character values occupy 8 bits and correspond to the ASCII collating sequence.

5. Files

   MS-FORTRAN files use file control blocks (of type FCBFQQ), allocated dynamically on the heap. File control blocks for MS-FORTRAN are not identical to file control blocks for MS-Pascal. See Appendix A, "The MS-FORTRAN File Control Block," for a complete listing.

6. Procedural parameters (subroutine and function parameters)

   Procedural parameters contain a reference to the location of the subroutine or function. The parameter always contains two words: the first word is zero, and the second word contains a data segment offset address. This is an offset to two words in the constant area that contain the segmented address of the actual routine.

7.3 Interfacing to Assembly Language Routines

All subroutines and functions in MS-FORTRAN are external. They need not be declared as external with the EXTERNAL statement. When a subroutine or function is called, the addresses of the actual parameters are first pushed on the stack in the order that they are declared. MS-FORTRAN always uses calls by reference, even if the actual parameters are expressions or constants.

If the procedure called is a function and if the function return type is real or double precision, an additional implicit parameter for the function is pushed on the stack. This parameter is the two-byte address of a temporary variable created by the calling program.

After all parameters have been pushed, the return address is pushed. If the procedure called is a function, the return value is expected as follows:

1. If the return value is a two-byte integer or logical value, that value is expected in the AX register, as shown in Figure 7.2:

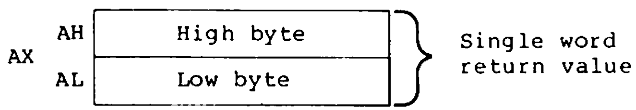


Figure 7.2 Two-Byte Return Value

2. If the return value is a four-byte integer or logical value, that value is expected in the DX, AX pair, as shown in Figure 7.3:

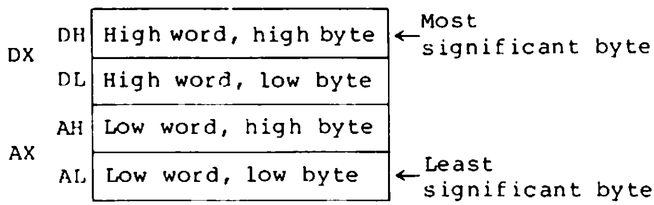


Figure 7.3. Four-Byte Return Value

3. If the return value is a four-byte or eight-byte REAL value, that value is expected in the temporary variable created by the calling program. The two-byte address of this temporary variable is the last parameter pushed on the stack. It is always at BP + 6 (see Example 2).

Example 1. INTEGER*4 Add Routine

Assume the following MS-FORTRAN program has been compiled:

```
PROGRAM  EXAMPL1
INTEGER I, TOTAL, IADD
I = 10
TOTAL = IADD (I,15)
WRITE (*,'1X,F16)') TOTAL
END
```

At runtime, just prior to the transfer to IADD, the stack would be as shown in Figure 7.4:

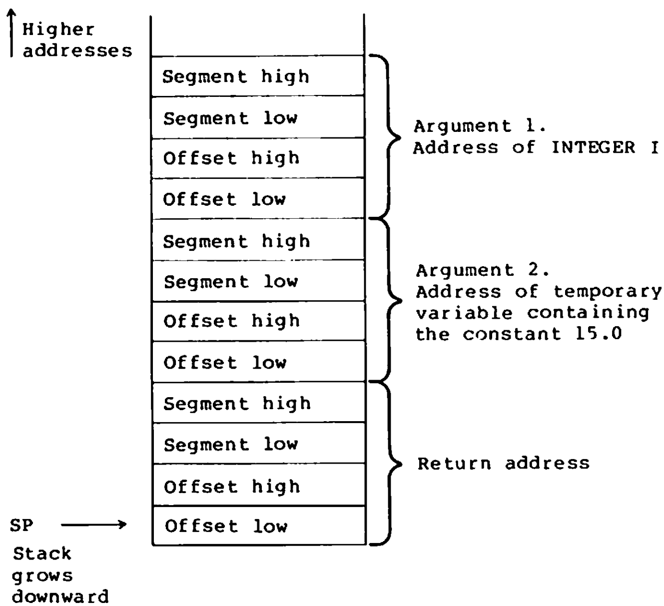


Figure 7.4. Stack Before Transfer to IADD

An example of an assembly language routine that implements the integer ADD function, IADD, is illustrated in the following routine. Note that the function return value is AX,DX.

```
DATA      SEGMENT PUBLIC 'DATA'
          ;See Note at end of section
DATA      ENDS

DGROUP    GROUP PUBLIC DATA ;See Note
CODE      SEGMENT 'CODE'
          ASSUME CS:CODE,DS:DGROUP,SS:DGROUP ;See Note
PUBLIC    IADD
IADD      PROC    FAR
          PUSH    BP               ;Save framepointer on stack
          MOV     BP,SP
          LES     BX,[BP + 10]     ;ES,BX : = addr of 1st param
          MOV     AX,ES:[BX]       ;AX,DX: = addr of 1st param
          MOV     DX,ES:[BX] +2
          LES     BX,[BP + 6]      ;ES,BX: = addr of 2nd param
          ADD     AX,ES:[BX]       ;AX,DX: = 1st parameter plus
          ADC     DX,ES:[BX] +2      ;2nd parameter
          MOV     SP,BP
          POP     BP               ;Restore the framepointer
          RET     AH               ;Return, pop 8 bytes

          IADD    ENDP
          CODE    ENDS
END
```

Example 2.   REAL*4 Add Routine

Assume the following FORTRAN program has been compiled:

```
PROGRAM   EXAMPL2
REAL R, TOTAL, RADD
R = 10.0
TOTAL = RADD (15.0,R)
WRITE (*,'1X,F10.3)') TOTAL
END
```

At runtime, just prior to the transfer to RADD, the stack would be as shown in Figure 7.5:

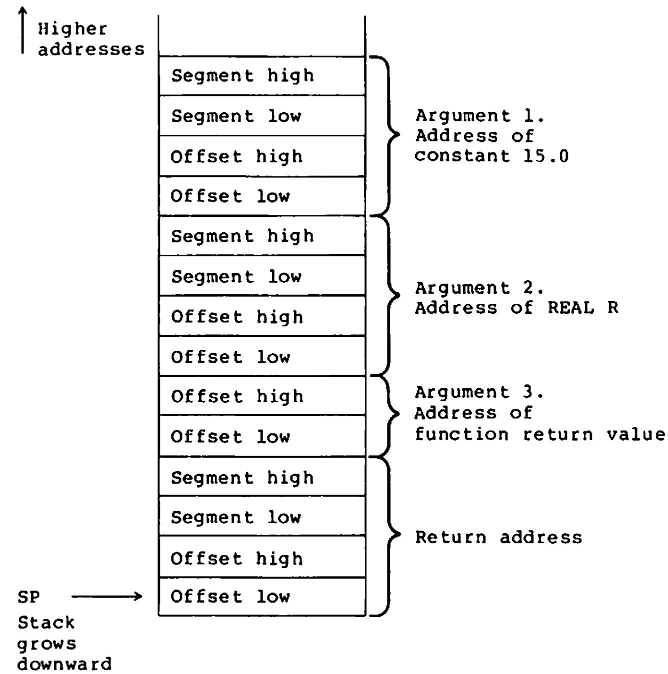


Figure 7.5. Stack Before Transfer to RADD

An example of an assembly language routine that implements the real add function, RADD, is illustrated in the following routine. Note that the function return value is in the location specified by BP + 6.

```
DATA      SEGMENT PUBLIC 'DATA'
          ;See Note at end of section
DATA      ENDS

DGROUP    GROUP PUBLIC DATA ;See Note
CODE      SEGMENT'CODE'
          ASSUME CS:CODE,DS:DGROUP,SS:DGROUP ;See Note
PUBLIC    RADD
RADD      PROC    FAR
          PUSH    BP              ;Save framepointer on stack
          MOV     BP,SP
          LES     BX,[BP + 12]    ;ES,BX : = addr of 1st param
          FLD     ES:[BX]         ;Push value of 1st param
                                  ;on 8087 stack
          LES     BX,[BP + 8]     ;ES,BX : = addr of 2nd param
          FLD     ES:[BX]         ;Push value of 2nd param
                                  ;on 8087 stack
          FADDP   ST(1),ST        ;Add first two items
                                  ;on 8087 stack
          MOV     DI,[BP + 6]     ;DI : = addr of funct return
          FSTP    [DI]            ;Store result on 8087 stack
                                  ;at funct return location
          FWAIT
          MOVS    P,BP            ;Restore the framepointer
          POP     BP
          RET     0AH             ;Return, pop 10 bytes

          RADD    ENDP
          CODE    ENDS

END
```

Note: Data used by assembly language routines must be placed in a segment whose name is DATA, whose classname is 'DATA', and which is grouped in DGROUP. The ASSUME statement is required.

Chapter 8

Advanced Topics

8.1 The Structure of the Compiler....74
8.1.1 The Front End 76
8.1.2 The Back End....77
8.1.2.1 Pass Two....77
8.1.2.2 Pass Three....79
8.2 An Overview of the File System....80
8.3 Runtime Architecture....83
8.3.1 Runtime Routines....83
8.3.2 Memory Organization....84
8.3.3 Initialization and Termination....87
8.3.3.1 Machine Level Initialization....90
8.3.3.2 Program Level Initialization....91
8.3.3.3 Program Termination....91
8.3.4 Error Handling....92
8.3.4.1 Machine Error Context....93
8.3.4.2 Source Error Context....95

This chapter contains advanced technical information that will be of interest primarily to experienced programmers. Since MS-FORTRAN and MS-Pascal (but not FORTRAN-80) have the same compiler back end, and share a common file and runtime system, much of the information that follows refers to both languages. Differences, where they exist, are noted.

8.1 The Structure of the Compiler

The compiler is divided into three phases, or passes, each of which performs a specific part of the compilation process. Figure 8.1, which follows, illustrates the basic structure of the compiler and its relationship to the files that it reads and writes.

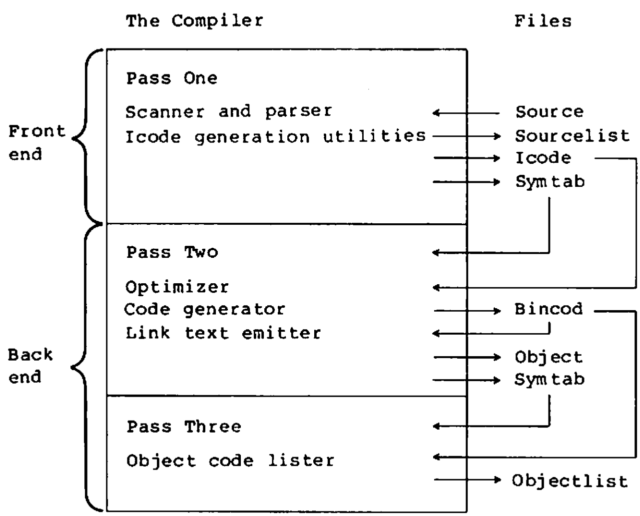


Figure 8.1. The Structure of the Compiler

Pass one, which normally corresponds to a file named FOR1.EXE, constitutes the front end of the compiler and performs the following actions:

1. reads the source program
2. compiles the source into an intermediate form
3. writes the source listing file
4. writes the symbol table file
5. writes the intermediate code file

Passes two and three (the files PAS2.EXE and PAS3.EXE) together make up the back end, which does the following:

1. optimizes the intermediate code
2. generates target code from intermediate code
3. writes and reads the intermediate binary file
4. writes the object (link text) file
5. writes the object listing file

Both the front and back end of the compiler are written in MS-Pascal, in a source format that can be transformed into either relatively standard Pascal or into system level MS-Pascal. (See the *MS-Pascal Reference Manual* for a discussion of implementation levels in MS-Pascal.)

All intermediate files contain MS-Pascal records. The front and back ends include a common constant and type definition file called PASCOM, which defines the intermediate code and symbol table types. The back ends use a similar file for the intermediate binary file definition. Formatted dump programs for all intermediate files and object files are available for special purpose debugging.

The symbol table record is relatively complex, with a variant for every kind of identifier (variables, procedures, intrinsic functions, and common blocks). The intermediate code (or Icode) record contains an Icode number, opcode, and up to four arguments; an argument can be the Icode number of another Icode to represent expressions in tree form, or something else (such as a symbol table reference, constant, or length). The intermediate binary code record contains several variants for absolute code or data bytes, public or external references, label references and definitions, etc.

8.1.1 The Front End

The MS-FORTRAN front end can be divided into several parts:

1. the scanner
2. various utilities
3. EXECSTMTS, which processes executable statements
4. DECLSTMTS, which processes declarative statements

The front end is driven by recursive descent syntax analysis, using a set of procedures such as EXPRESSION (for expressions) and VARIABLE (for variables). Parsing is performed on a strict statement basis. The scanner procedure GETSTMT gets the next MS-FORTRAN statement into the statement buffer.

Overall compilation control depends on a series of states, handled as shown in Table 8.1.

Table 8.1. Front End Compilation Procedures

Name	Function
INITSTATE	Initialize procedure
HEADSTATE	Process subroutine header
IMPSTATE	Process IMPLICIT statements
SPECSTATE	Process specification statements
DATASTATE	Process DATA statements
STMTFUNSTATE	Process statement functions
EXECSTATE	Process executable statements
ENDSTATE	End procedure

After initialization in INITSTATE, the current state cycles from HEADSTATE through EXECSTATE for the program and for all subroutines and functions. The final procedure, which carries out program termination, is ENDSTATE.

MS-FORTRAN intermediate files are written in the same manner as for the MS-Pascal front end. A few of the intermediate code operations are specific to MS-FORTRAN, particularly those concerned with assigned GOTO and DO statements. The symbol table contains special flags for COMMON and EQUIVALENCE variables, since these affect common subexpression optimization.

8.1.2 The Back End

Of the separate passes that make up the back end of the compiler, pass two is required while Pass three is optional.

8.1.2.1 Pass Two

The optimizer reads the interpass files in the following order: first the symbol table for a block, then the intermediate code for the block. Optimization is performed on each "basic block," i.e., each block of intermediate code up to the first internal or user label or up to a fixed maximum number of Icodes, whichever comes first.

Within a block, the optimizer can reorder and condense expressions as long as the intent of the program(mer) is preserved. For instance, in the following program fragment, the array address A (J, K) need be calculated only once:

```
      A(J,K) = A(J,K) + 1
C     J = J-1
      IF (A(J,K) .EQ. MAX) CALL PUNT
```

However, if the above preceding is rewritten to include the assignment to J, shown in the fragment as a comment, the array address in the IF statement must be partially recalculated.

This optimization is called common subexpression elimination. The optimizer also reorders expressions so that the most complicated parts are done first, when more registers for temporary values are available. It also does several other optimizations, such as:

1. constant folding not done by the front end
2. strength reduction (changing multiplications and divisions into shifts when possible)
3. peephole optimization (removing additions of zero, multiplications by one, and changing A := A + 1 to an internal increment memory Icode)

The optimizer works by building a tree out of the intermediate codes for each statement and then transforming the list of statement trees.

There are seven internal passes per basic block:

1. statement tree construction from the Icode stream
2. preliminary transformations to set address/value flags
3. length checks and type coercions
4. constant and address folding, and expression reordering
5. peephole optimization and strength reduction
6. machine-dependent transformations
7. common subexpression elimination

Finally, the optimizer calls the code generator to translate the basic block from tree form to target machine code.

The code generator must translate these trees into actual machine code. It uses a series of templates to generate more efficient code for special cases. For example, there is a series of templates for the addition operator. The first template checks for an addition of the constant one. If this addition is found, the template generates an increment instruction. If the template does not find an addition of one, the next template gets control and checks for an addition of any constant. If this is found, the second template generates an add immediate instruction.

The final template in the series handles the general case. It moves the operands into registers (by recursively calling the code generator itself), then generates an add register instruction. There is a series of templates for every operation. The code generator also keeps track of register contents and several memory segment addresses (code, static variables, constant data, etc.), and allocates any needed temporary variables.

The code generator writes a file of binary intermediate code (BINCOD), which contains machine instruction opcodes with sympolic references to external routines and variables. A final internal pass reads the BINCOD file and writes the object code file.

8.1.2.2 Pass Three

This short pass reads both the BINCOD file (described in the previous section) and a version of the symbol table file as updated by the optimizer and code generator. Using the data in these files, Pass three writes a listing of the generated code in an assembler-like format.

For more information about the compiler (especially the back end), see the article "Native-code Compilers are Portable and Fast," (James G. Letwin and Andrea L. Lewis, Electronic Design, May 14, 1981).

8.2 An Overview of the File System

MS-FORTRAN and MS-Pascal are designed to be easily interfaced to existing operating systems. The standard interface has two parts:

1. a file control block (FCB) declaration
2. a set of procedures and functions, called Unit U, that are called from MS-FORTRAN or MS-Pascal at runtime to perform input and output

This interface supports three access methods:

TERMINAL, SEQUENTIAL, and DIRECT.

Each file has an associated FCB (file control block). The FCB record type begins with a number of standard fields, whose details are independent of the operating system. These are followed by fields, such as channel numbers, buffers, and other operating system data, that are dependent on the operating system.

The advanced MS-Pascal user can access FCB fields directly, as explained in Chapter 7, "Files," of the *MS-Pascal Reference Manual.* There is no standard way to access FCB fields within MS-FORTRAN.

Both MS-FORTRAN and MS-Pascal have two special file control blocks that correspond to the keyboard and the screen of your terminal. These two file control blocks are always available. In MS-Pascal, they are the predeclared files INPUT and OUTPUT; in MS-FORTRAN, they are unit number Ø (or *) and are accessed through a variable TRMVQQ, which is declared as follows:

```
VAR TRMVQQ: ARRAY [BOOLEAN] OF ADR OF
FCBFQQ;
```

The false element references the output file; the true element references the input file.

Unit U refers to the target operating system interface routines. The file routines specific to MS-Pascal are called Unit F; the file routines specific to MS-FORTRAN are called Unit V. Code generated by the compiler of either language contains calls to the appropriate unit (F or V), which in turn call Unit U routines.

Figure 8.2 shows this relationship schematically.

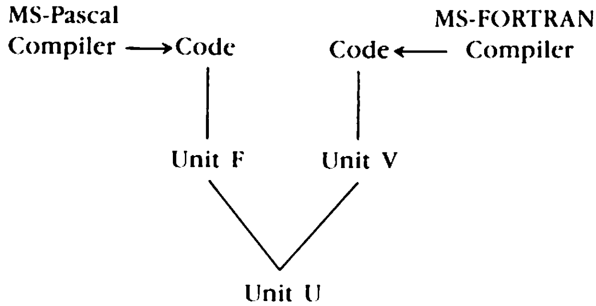


Figure 8.2. The Unit U Interface

The file system uses the following naming convention for public linker names:

1. All linker globals are six alphabetic characters, ending with QQ. (This helps to avoid conflicts with your program global names.)
2. The fourth letter indicates a general class, where:
   a. xxxFQQ is part of the generic MS-Pascal file unit
   b. xxxVQQ is part of the generic MS-FORTRAN file unit
   c. xxxUQQ is part of the operating system interface unit

File system error conditions may be:

1. detected at the lower Unit U level
2. detected at the higher Unit F or V level
3. undetected

When a Unit U routine detects an error, it sets an appropriate flag in the FCB and returns to the calling Unit F or V routine. When Unit F or V detects an error or discovers Unit U has detected one, it takes one of two possible actions:

1. An immediate runtime error message is generated, and the program aborts.
2. Unit F or V returns to the calling program if error trapping has been set (in MS-Pascal with the TRAP flag, in MS-FORTRAN with the ERR = nnn clause).

Units F and V will not pass a file with an error condition to a Unit U routine. For some access methods, certain file operations may lead to an undetected error, such as reading past the end of a record (this condition has undefined results). Runtime errors that cause a program abort use the standard error-handling system, which gives the context of the error and provides entry to the target debugging system.

8.3 Runtime Architecture

The remainder of this chapter describes several topics related to the runtime structure of MS-FORTRAN and MS-Pascal, with mention of differences where they exist.

8.3.1 Runtime Routines

MS-FORTRAN and MS-Pascal runtime entry points and variables conform to the same naming convention: all names are six characters, and the last three are a unit identification letter followed by the letters "QQ". Table 8.2 shows the current unit identifier suffixes.

Table 8.2. Unit Identifier Suffixes

Suffix	Unit Function
AQQ	Complex real
BQQ	Compile time utilities
CQQ	Encode, decode
DQQ	Double precision real
EQQ	Error handling
FQQ	MS-Pascal file system
GQQ	Generated code helpers
HQQ	Heap allocator
IQQ	Generated code helpers
JQQ	Generated code helpers
KQQ	FCB definition
LQQ	STRING, LSTRING
MQQ	Reserved
NQQ	Long integer
OQQ	Other miscellaneous routines
PQQ	Pcode interpreter
QQ	QReserved
RQQ	Real (single precision)
SQQ	Set operations
TQQ	Reserved
UQQ	Operating system file system
VQQ	MS-FORTRAN file system
WQQ	Reserved
XQQ	Initialize/Terminate
YQQ	Special utilities
ZQQ	Reserved

8.3.2 Memory Organization

Memory on the 8Ø86 is divided into segments, each containing up to 64K bytes. The relocatable object format and MS-LINK also put segments into classes and groups. All segments with the same class name are loaded next to each other. All segments with the same group name must reside in one area up to 64K long; that is, all segments in a group can be accessed with one 8Ø86 segment register.

MS-FORTRAN and MS-Pascal both define a single group, named DGROUP, which is addressed using the DS or SS segment register. Normally, DS and SS contain the same value, although DS may be changed temporarily to some other segment and changed back again. SS is never changed; its segment registers always contain abstract "segment values" and the contents are never examined or operated on. This provides compatibility with the Intel® 8Ø286 processor. Long addresses, such as MS-Pascal ADS variables or MS-FORTRAN named common blocks, use the ES segment register for addressing.

Memory is allocated within DGROUP for all static variables, constants which reside in memory, the stack, the heap, MS-FORTRAN blank common, and segmented addresses of MS-FORTRAN named common blocks. The named common blocks themselves reside in their own segments, not in DGROUP.

Memory in DGROUP is allocated from the top down; that is, the highest addressed byte has DGROUP offset 65535, and the lowest allocated byte has some positive offset. This allocation means offset zero in DGROUP may address a byte in the code portion of memory, in the operating system below the code, or even below absolute memory address zero (in the latter case the values in DS and SS are "negative").

DGROUP has two parts:

1. a variable length lower portion containing the heap and the stack

2. a fixed length upper portion containing static variables, constants, blank common, and named common addresses

After your program is loaded, during initialization (in

ENTX6L), the fixed upper portion is moved upward as much as possible to make room for the lower portion. If there is enough memory, DGROUP is expanded to the full 64K bytes; if there is not enough for this, it is expanded as much as possible.

Figure 8.3 illustrates this memory organization.

Top (highest address)

MS-DOS code for COMMAND (may be overlapped)	
(Unused memory)	
HIMEM segment	Class HIMEM
<name> segment(s)	Class COMMOM
DS offset 65536	
COMMQQ segment	Class COMMON
CONST segment	Class CONST
COMADS segment	Class COMADS
DATA segment	Class DATA
STACK segment	Class STACK
MEMORY segment	Class MEMORY
HEAP segment	Class MEMORY
DS offset >= 00	
CODE segments (user and library routines)	
MS-DOS code and data (fixed)	

Bottom (address 0:00)

Figure 8.3. Memory Organization

The following paragraphs describe memory contents, starting at the bottom (address zero), when an MS-FORTRAN or MS-Pascal program is running. Addresses are shown in "segment:offset" form.

1. 0000:0000

   The beginning of memory on an 8086 system contains interrupt vectors, which are segmented addresses. Usually the first 32 to 64 are reserved for the operating system. Following these vectors is the resident portion of the operating system (MS-DOS in this case).

MS-DOS provides for loading additional code above it, which remains resident and is considered part of the operating system as well. Examples of resident additional code are special device drivers for peripherals, a print spooler, or the debugger.

2. BASE:0000

   Here BASE means the starting location for loaded programs, sometimes called the transient program area. When you invoke an MS-FORTRAN or MS-Pascal program, loading begins here. The beginning of your program contains the code portion, with one or more code segments. These code segments are in the same order as the object modules given to the linker, followed by object modules loaded from libraries.

3. DGROUP:LO

   The DGROUP data area contains the following:

Segment	Class	Description
HEAP	MEMORY	Pointer variables, some files
MEMORY	MEMORY	(not used, Intel compatible)
STACK	STACK	Frame variables and data
DATA	DATA	Static variables
COMADS	COMADS	Addresses of named commons
CONST	CONST	Constant data
COMMQQ	COMMON	FORTRAN blank common

   The stack and the heap grow toward each other, the stack downward and the heap upward.

4. DGROUP:TOP

   TOP means 64K bytes (4K paragraphs) above DGROUP:0000 (i.e., just past the end of DGROUP). MS-FORTRAN named common blocks start here. Each common block has a segment name as declared in the MS-FORTRAN program as the common block name, and the class name COMMON. Each named common has one segmented (ADS) address in the COMADS segment in DGROUP. All references to common block component variables use offsets from this address.

5. HIMEM:0000

   The segment named HIMEM (class HIMEM) gives the highest used location in the program. The segment itself contains no data, but its address is used during initialization. Available memory starts here and can be accessed with MS-Pascal ADS variables.

6. COMMAND

   MS-DOS keeps its command processor (the part of itself which does COPY, DIR, and other resident commands) in the highest location in memory possible. Your MS-FORTRAN or MS-Pascal program may need this memory area in order to run. If so, the command processor is overwritten with program data. When your program finishes, the command processor is reloaded from the file COMMAND.COM on the default drive.

   In some circumstances, the check may result in a message appearing on your screen telling you to insert a disk that contains the appropriate file, COMMAND.COM. You can avoid this delay by making sure that COMMAND.COM is on the disk in the default drive when the program ends.

8.3.3 Initialization and Termination

Every executable file contains one, and only one, starting address. As a rule, when MS-FORTRAN or MS-Pascal object modules are involved, this starting address is at the entry point BEGXQQ in the module ENTX. For some versions, the name ENTX may be appended with other letters. However, the name of the module always begins with the four letters "ENTX". An MS-FORTRAN or MS-Pascal program (as opposed to a module or implementation) has a starting address at the entry point ENTGQQ. BEGXQQ calls ENTGQQ.

The following discussion assumes that an MS-FORTRAN or MS-Pascal main program along with other object modules is loaded and executed. However, you can also link a main program in assembly or some other language with other object modules in either MS-FORTRAN or MS-Pascal. In this case, some of the initialization and termination done by the ENTX module may need to be done elsewhere.

When a program is linked with the runtime library and execution begins, several levels of initialization are required. The levels, in the order in which they occur, are the following:

1. machine-oriented initialization
2. runtime initialization
3. program and unit initialization

The general scheme is shown in Figure 8.4.

ENTX module

BEGXQQ:	Set stackpointer, framepointer Initialize PUBLIC variables Set machine-dependent flags, registers, and other values Call INIX87 Call INIUQQ Call BEGOQQ Call ENTGQQ {Execute program}
ENDXQQ:	{Aborts come here} Call ENDOQQ Call ENDYQQ Call ENDUQQ Call ENDX87 Exit to operating system

INTR Module

INIX87:	Real processor initialization
ENDX87:	Real processor termination

Unit U

INIUQQ:	Operating system specific file unit initialization
ENDUQQ:	Operating system specific file unit termination

MISO Module

BEGOQQ:	(Other user initialization)
ENDOQQ:	(Other user termination)

Program Module

ENTGQQ:	Call INIFQQ If $ENTRY on, CALL ENTEQQ Initialize static data Initialize units FOR program parameters DO Call PPMFQQ Execute program If $ENTRY on, CALL EXTEQQ

Figure 8.4. FORTRAN Program Structure

8.3.3.1 Machine Level Initialization

The entry point of an MS-FORTRAN load module is the routine BEGXQQ, in the module ENTX (the module may also be called ENTX8, ENTX6M, etc.).

BEGXQQ does the following:

1. It moves constant and static variables upward (as described in Section 8.3.2, "Memory Organization"), creating a gap for the stack and the heap.
2. It sets the framepointer to zero.
3. It initializes a number of public variables to zero or NIL. These include:

   RESEQQ, machine error context CSXEQQ, source error context list header PNUXQQ, initialized unit list header HDRFQQ, MS-Pascal open file list header HDRVQQ, MS-FORTRAN open file list header
4. It sets machine-dependent registers, flags, and other values.
5. It sets the heap control variables. BEGHQQ and CURHQQ are set to the lowest address for the heap: the word at this address is set to a heap block header for a free block the length of the initial heap. ENDHQQ is set to the address of the first word after the heap. The stack and the heap grow together, and the public variable STKHQQ is set to the lowest legal stack address (ENDHQQ, plus a safety gap).
6. It calls INIX87, the real processor initializer. This routine sets 8087 emulator interrupt vectors, as appropriate.
7. It calls INIUQQ, the file unit initializer specific to the operating system. If the file unit is not used and you don't want it loaded, a dummy INIUQQ routine that just returns must be loaded instead.
8. It calls BEGOQQ, the escape initializer. In a normal load module, an empty BEGOQQ that only returns is included. However, this call provides an escape mechanism for any other initialization. For example, it could initialize tables for an interrupt-driven profiler or a runtime debugger.

9. It calls ENTGQQ, the entry point of your MS-FORTRAN program.

8.3.3.2 Program Level Initialization

Your main program continues the initialization process. First, the language specific file system is called (INIVQQ for MS-FORTRAN, or INIFQQ for MS-Pascal). Both are parameterless procedures.

If the main program is in MS-FORTRAN, and MS-Pascal file routines will be used, INIFQQ must be called to initialize the MS-Pascal file system. If the main program is in MS-Pascal, and MS-FORTRAN file routines will be used, INIVQQ must be called to initialize the MS-FORTRAN file system.

MS-FORTRAN main programs automatically call INIVQQ; MS-Pascal main programs automatically call INIFQQ. To avoid loading the file system, you must provide an empty procedure to satisfy one or both of these calls. If $DEBUG has been set, ENTEQQ is then called to set the source error context.

8.3.3.3 Program Termination

Program termination occurs in one of three ways:

1. The program may terminate normally, in which case the main program returns to BEGXQQ, at the location named ENDYQQ.
2. The program may abort due to an error condition, either with a user call or a runtime call to an error handling routine. In either case, an error message, error code, and error status are passed to EMSEQQ, which does whatever error handling it can and calls ENDXQQ.
3. ENDXQQ may be called directly.

ENDXQQ first calls ENDOQQ, the escape terminator, which normally just returns to ENDXQQ. Then ENDXQQ calls ENDYQQ, the generic file system terminator. ENDYQQ closes all open MS-Pascal and MS-FORTRAN files, using the file list headers HDRFQQ and HDRVQQ.

ENDXQQ calls ENDUQQ, the operating system specific file unit terminator. Finally, ENDXQQ calls ENDX87 to terminate the real number processor (8087 emulator). As with INIUQQ, INIFQQ, and INIVQQ, if your program requires no file handling, you will need to declare empty parameterless procedures for ENDYQQ and ENDUQQ.

As mentioned, the main initialization and termination routines are in module ENTX. Procedures for BEGOQQ and ENDOQQ are in module MISO. ENDYQQ is in module MISY.

8.3.4 Error Handling

Runtime errors are detected in one of four ways:

1. The user program calls EMSEQQ.
2. A runtime routine calls EMSEQQ.
3. An error checking routine in the error module calls EMSEQQ.
4. An internal helper routine calls an error message routine in the error unit that, in turn, calls EMSEQQ.

Handling an error detected at runtime usually involves identifying the type and location of the error and then terminating the program, or, with ERR = in an I/O statement, returning to the calling MS-FORTRAN procedure.

The error type has three components:

1. a message
2. an error number
3. an error status

The message describes the error, and the number can be used to look up more information (see Appendix C, "Error Messages," in the *MS-FORTRAN Reference Manual).* In MS-FORTRAN, the error status value is used for special purposes and has no significance for the user. In MS-Pascal, the error status value is undefined.

Table 8.3 shows the general scheme for error code numbering.

Table 8.3. Error Code Classification

Range	Classification
1- 999	Front end errors
1000-1099	Unit U file system errors
1100-1199	Unit F file system errors
1200-1299	Unit V file system errors
1300-1999	Reserved
2000-2049	Heap, stack, memory
2050-2099	Ordinal and long integer arithmetic
2100-2149	REAL*4 and REAL*8 arithmetic
2150-2199	Structures, sets, and strings
2200-2399	Reserved
2400-2449	Pcode interpreter
2450-2499	Other internal errors
2500-2999	Reserved

An error location has two parts:

1. the machine error context
2. the source program context

The machine error context is the program counter, stackpointer, and framepointer at the point of the error. The program counter is always the address following a call to a runtime routine (e.g., a return address). The source program context is optional; it is controlled by the $DEBUG metacommand. If $DEBUG is in effect, the program context consists of:

1. the source filename of the compiland containing the error
2. the name of the routine in which the error occurred
3. the listing line number of the first line of the statement

8.3.4.1 Machine Error Context

Runtime routines are compiled by default with the $RUNTIME metacommand set. This causes special calls to be generated at the entry and exit points of each runtime routine. The entry call saves the context at the point where a runtime routine is called in the user program. This context consists of the framepointer, stackpointer, and program counter. As a consequence of this saving of context, if an error occurs in a runtime routine, the error location is always in the user program. This is true even if runtime routines call other runtime routines. The exit call that is generated restores the context.

The runtime entry helper, BRTEQQ, uses the runtime values shown in Table 8.4.

Table 8.4. Runtime Values in BRTEQQ

Value	Description
RESEQQ	Stackpointer
REFEQQ	Framepointer
REPEQQ	Program counter offset
RECEQQ	Program counter segment

The first thing that BRTEQQ does is examine RESEQQ. If this value is not zero, the current runtime routine was called from another runtime routine and the error context has already been set, so it just returns. If RESEQQ is zero, however, the error context must be saved. The caller's stackpointer is determined from the current framepointer and stored in RESEQQ. The address of the caller's saved framepointer and return address (program counter) in the frame is determined. Then the caller's framepointer is saved in REFEQQ. The caller's program counter (i.e., BRTEQQ's caller's return address) is saved: the offset in REPEQQ and the segment (if any) in RECEQQ.

The runtime exit helper, ERTEQQ, has no parameters. It determines the caller's stackpointer (again, from the framepointer) and compares it against RESEQQ. If these values are equal, the original runtime routine called by your program is returning, so RESEQQ is set back to zero.

EMSEQQ uses RESEQQ, REFEQQ, REPEQQ, and RECEQQ to display the machine error context.

8.3.4.2 Source Error Context

Giving the source error context involves extra overhead, since source location data must be included in the object code in some form. Currently, this is done with calls which set the current source context as it occurs. These calls can also be used to break program execution as part of the debug process. The overhead of source location data, especially line number calls, can be significant. Routine entry and exit calls, while requiring more overhead, are much less frequent, so the overhead is less.

The procedure entry call to ENTEQQ passes two VAR parameters: the first is an LSTRING containing the source filename; the second is a record that contains the following:

1. the line number of the procedure
2. the subroutine or function identifier

The filename is that of the compiland source (e.g., the main source filename, not the names of any $INCLUDE files). The procedure identifier is the full identifier used in the source, not the linker name. The line number is the first executable statement in the procedure.

Entry and exit calls are also generated for the main program, in which case the identifier is the program name.

The procedure exit call to EXTEQQ does not pass any parameters. It pops the current source routine context off a stack maintained in the heap.

The line number call to LNTEQQ passes a line number as a value parameter. The current line number is kept in the public variable CLNEQQ. Since the current routine is always available, the compiland source filename and routine containing the line are available along with the line number. Line number calls are generated just before the code in the first statement on a source line. The statement can, of course, be part of a larger statement.

Most of the error handling routines are in modules ERRE and FORE. The source error context entry points ENTEQQ, EXTEQQ, and LNTEQQ are in the debug module, DEBE.

Appendices

A	MS-FORTRAN File Control Block	99
B	Real Number Conversion Utilities	103
C	Structure of External FORTRAN Files	105
D	FORTRAN Scratch File Names	107
E	LINK Error Messages	109

Appendix A

MS-Fortran File Control Block

This appendix lists the complete file control block specification for this version of the MS-FORTRAN runtime system. The underlying data type is an MS-Pascal record. Numbers in square brackets give the byte offset for each field of the file control block.

```
{MS-FORTRAN File Control Block for MS-DOS}

INTERFACE;          UNIT
  FILKQQ (FCBFQQ,    FILEMODES,    SEQUENTIAL,    TERMIN-
          AL, DIRECT, DEVICETYPE, CONSOLE, LDEVICE, DISK,
          DOSEXT, DOSFCB, FNLUQQ, SCTRLNTH);

CONST

    FNLUQQ = 21;        {length of an MS-DOS filename}
    SCTRLNTH = 512      ;{length of a disk sector}
TYPE

DOSEXT = RECORD         {DOS file control blck extension}

    PS[0]: BYTE         ;{boundary byte, not in extension}
    FG[1]: BYTE         ;{flag; must be 255 in extension}
    XZ[2]: ARRAY [0..4] OF BYTE     {pad, internal use}
    AB[7]: BYTE;        {internal use for attribute bits}
END;

DOSFCB = RECORD         {DOS file control block (normal)}

    DR[0]: BYTE;        {drive numb, 0=default, 1=A etc}
    FN[1]: STRING (8);  {file name - eight characters}
    FT[9]: STRING (3);  {file extn - three characters }
    EX[12]: BYTE;       {current extent; lo-order byte}
    E2[13]: BYTE;       {current extent; hi-order byte}
    S2[14]: BYTE;       {size of sector; lo-order byte}
    RC[15]: BYTE;       {size of sector; hi-order byte}
    Z1[16]: WORD;       {file size; lo-word; readonly}
    Z2[18]: WORD;       {file size; hi-word; readonly}
    DA[20]: WORD;       {date; bits: DDDDDMMMMYYYYYYY}
    DN [22]: ARRAY [0..9] OF BYTE;      {reserved for DOS}
    CR[32]: BYTE;       {current sector (within extent)}
    RN[33]: WORD;       {direct sector number (lo word)}
    R2[35]: BYTE;       {direct sector number (hi byte}
    R3[36]: BYTE;       {DSN hi byte if sect size < 64}
    PD[37]: BYTE;       {pad to wrd boundary; not MS-DOS}
END;

DEVICETYPE=(CONSOLE, LDEVICE, DISK); {physical device}
```

```
FILEMODES=(SEQUENTIAL, TERMINAL, DIRECT); {access mode}

TYPE

FCBFQQ=RECORD   {byte offsets start every field comment}

{fields accessible as <file variable>.<field>}
TRAP:  BOOLEAN;        {00 Pascal user trap errs if true}
ERRS:  WRD(0)..15;     {01 err stat, set only by all units}
MODE:  FILEMODES;      {02 usr file mode; not used in unitU}
MISC:  BYTE;           {03 pad to wrd bound, special use}

{flds shard by units F, V, U; ERRC/ESTS are write-only}
ERRC:  WORD;           {04 err code, err exists if nonzero}
                       {1000..1099: set for unit U errors}
                       {1100..1199: set for unit F errors}
                       {1200..1299: set for unit V errors}
ESTS:  WORD;           {06 error data, usually set by OS}
CMOD:  FILEMODES;      {08 sys file mode; copied from MODE}

{flds set/used by units F and V, read-only in unitU}
TXTF:  BOOLEAN;        {09 true: formatted /ASCII /TEXT file}
                       {    false: not formatted / binary file}
SIZE:  WORD;           {10 record size set when file is opened}
                       {DIRECT: always fixed record length}
                       {others: max length (UPPER (BUFFA))}
MISB:  WORD;           {12 unused, exists for historic reasons}
OLDF:  BOOLEAN;        {14 true: must exist before open; RESET}
                       {    false: can create on open; REWRITE}
INPT: BOOLEAN;         {15 true: user is now reading from file}
                       {    false: user is now writing to file}
RECL:  WORD;           {16 DIRECT record number, lo order word}
RECH:  WORD;           {18 DIRECT record number, hi order word}
USED:  WORD;           {20 number bytes used in current record}

{fld usd intrnally by units F and V not used by unitU}
LINK: ADR OF FCBFQQ; {22 DS offset addr of next opn fil}
{flds usd intrnally by unitF not used by units V or U}
BADR: ADRMEM;          {24 ADR of buffer variable (end of FCB)}
TMPF: BOOLEAN;         {26 true if temp file; delete on CLOSE}
FULL: BOOLEAN;         {27 buffer lazy evaluation status, TEXT}
UNFM: BYTE;            {28 for unformatted binary mode}
OPEN: BOOLEAN;         {29 file opened; RESET / REWRITE called}

{flds usd intrnally by unitV not used by units F or U}
FUNT: INTEGER;         {30 Unit V's unit number always above 0}
ENDF: BOOLEAN;         {32 last operation was the ENDFILE stmt}

{flds set/usd by unitU, and read-only in units F and V}
REDY: BOOLEAN;         {33 buffer ready if true; set by F / U}
```

```
BCNT: WORD;            {34 number of data bytes actually moved}
EORF: BOOLEAN;         {36 true if end of record read, written}
EOFF: BOOLEAN;         {37 end of file flag set after EOF read}

    {unit U (operating system) information starts here}

NAME: LSTRING(FNLUQQ);
                       {38 DOS filenam(D:NNNNNNNN.XXX)}
DEVT: DEVICETYPE;      {60 dvice type, accessd by file}
RDFC: BYTE;            {61 funct code, for device GET}
WRFC: BYTE;            {62 funct code, for device PUT}
CHNG: BOOLEAN;         {63 true if sbuf data was chgd}
SPTR: WORD;            {64 indx to currnt byte in sbuf}
LNSB: WORD;            {66 num of valid bytes in sbuf}
DOSX: DOSEXT;          {68 extend DOS file contrl blck}
DOSF: DOSFCB;          {76 normal DOS file contrl blck}
IEOF: BOOLEAN;         {114 true if eoff will be true}
                             {on next get}
FNER: BOOLEAN;         {115 true if pfnuqq filenam err}
SBFL: BYTE;            {116 max txtfil line len in sbuf}
SBFC: BYTE;            {117 num of chars, read to sbuf}
SBUF: ARRAY[WRD(0)..SCTRLNTH-1] OF BYTE; {118 sect buf}
PMET: ARRAY [0..3] OF BYTE;     {118+sctrlnth reserved pad}
BUFF: CHAR;                     {122+sctrlnth (buffer var)}

    {end of section for unit U specific OS information}

END;
END;
```

Appendix B

Real Number Conversion Utilities

Releases of MS-FORTRAN starting with version 3.0 and later use the IEEE real number format. Releases of MS-FORTRAN earlier than 3.0 used the real number format.

The two formats are not compatible. However, if you need to convert real numbers from one format to the other, you can use the following library routines:

1. Microsoft to IEEE format

   SUBROUTINE M2ISQQ (RMS , RIEEE)

2. IEEE to Microsoft format

   SUBROUTINE I2MSQQ (RIEEE , RMS)

RMS and RIEEE are real numbers in Microsoft format and in IEEE format, respectively.

Appendix C

Structure Of External MS-Fortran Files

The structure of an external MS-FORTRAN file is determined by its properties. The structures used in MS-FORTRAN are as follows:

1. Formatted sequential files

   Records are separated by carriage return and linefeed (ASCII hex codes ØD and ØA, respectively).

Record N	D	A	Record N + 1

2. Unformatted sequential files

   A logical record is represented as a series of physical records, each of which has the following structure:

L	Data <= 128 bytes	L

Physical record

Each L shown above is a length byte that indicates the length of the data portion of the physical record. The data portion of the last physical record contains MOD (length of logical record, 128) bytes, and the length bytes will contain the exact size of the data portion.

Each of the preceding physical records will contain 128 bytes in the data portion, while the length byte will contain 129. For example, if the size of the logical record is 138:

129	128 bytes of data	129	10	10 bytes of data	10

One logical record

The first byte of the file is reserved and contains the value 75, which has no other significance.

3. Formatted direct files, unformatted direct files, and binary files

   No record boundaries or any other special characters are used.

Appendix D

MS-Fortran Scratch File Names

Scratch files are created by the MS-FORTRAN system when no filename is specified in an OPEN statement. Scratch file names look like this:

T<u>.TMP

<u> is the unit number specified in the OPEN statement.

Appendix E

Link Error Messages

Any link error will cause the link session to abort. After you have found and corrected the problem, you must rerun MS-LINK. Link errors have no code number. See your MS-DOS manual for further information on MS-LINK.

Attempt to access data outside of segment bounds, possibly bad object module

There is probably a bad object file.

Bad numeric parameter

Numeric value is not in digits.

Cannot open temporary file

MS-LINK is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the list map file.

Error: dup record too complex

DUP recored in assembly language module is too complex. Simplify DUP record in assembly language program.

Error: fixup offset exceeds field width

In assembly language instruction refers to an address with a short instruction instead of a long instruction. Edit assembly language source and reassemble.

Input file read error

There is probably a bad object file.

Invalid object module

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle).

Symbol defined more than once

MS-LINK found two or more modules that define a single symbol name.

Program size or number of segments exceeds capacity of linker

The total size may not exceed 384K bytes and the number of segments may not exceed 255.

Requested stack size exceeds 64K

Specify a size greater than or equal to 64K bytes with the /STACK switch.

Segment size exceeds 64K

64K bytes is the addressing system limit.

Symbol table capacity exceeded

Very many and/or very long names were typed, exceeding the limit of approximately 25K bytes.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

The limit is 1Ø groups.

Too many libraries specified

The limit is 8 libraries.

Too many public symbols

The limit is 1Ø24 public symbols.

Too many segments or classes

The limit is 256 (segments and classes taken together).

Unresolved externals: <list>

The external symbols listed have no defining module among the modules or library files specified.

VM read error

This is a disk error; it is not caused by MS-LINK.

Warning: no stack segment

None of the object modules specified contains a statement allocating stack space, but you used the /STACK switch.

Warning: segment of absolute or unknown type

There is a bad object module or an attempt has been made to link modules that MS-LINK cannot handle (e.g., an absolute object module).

Write error in TMP file

No more disk space remains to expand VM.TMP file.

Write error on run file

Usually, there is not enough disk space for the run file.

Index

$DEBUG, 5, 60
$INCLUDE, 54
$LIST, 54
$NODEBUG, 5, 60
$NOLIST, 54

/DSALLOCATE, 40
/HIGH, 40
/LINENUMBERS, 40
/MAP, 38, 40
/NO, 37, 40
/NODEFAULTLIBRARY-SEARCH, 37
/PAUSE, 40
/STACK, 40

8086 assembly language, 62
Add routine, 68, 69
Addresses, offset, 8
Angle brackets, syntax notation, vii
Assembler, 5
Assembly language
  data placement, 71
  interface, 34, 61, 67

Back end, 75, 77
BAT file See Batch file
Batch file, 41, 42
BEGXQQ, 87
Bibliography, reference, vii

Calling conventions, 34, 62
Calls
  by reference, 67
  long, 63
  short, 63
Capital letters,
  syntax notation, vii
Character values, 66
CHKDSK program, 39
Code
  relative, 8
  size, limits, 44
Command syntax, vii
COMMON block, 46, 86
Compile time
  definition, 7
  memory limits, 46
Compiler
  flags, calls, 5
  general discussion, 3
  optimization, 3
  passes, 13
    See also Pass one
      Pass two
      Pass three
  starting, 29
  structure, 3, 74

Compiling large programs, 44, 57
COPY, 2

Data
  size limits, 45
  types, 65
Definitions, 7
Device
  drivers, 86
  name, 25
DGROUP, 84
DIR, 39
DISKCOPY, 2
Disks
  backup copies, 2
  exchanging, 56
  limits, 54-55
  memory, 44
  provided, iv
Documentation, v
Dummy subroutines, 60

Ellipses, syntax
  notation, vii
ENDYQQ, 59
ENTER key, vii
ENTGQQ, 87
Error
  code classification, 93
  machine context, 93
  source context, 95
  trapping, 81
Examples
  large program, 57
  program session, 12
EXE file, 5, 38
Expressions, complex, 47
EXTERNAL declarations, 34
External
  files, 105
  reference, 8

File control block, 66, 80, 99
Filenames
  conventions, 25
  default, 26
  defaults, 25
  extensions, 25
  general rules, 27
  spaces within, 30
Files
  backup copies, 2
  batch facilities, 41
  compiler-written, 22
  control blocks See
    File control block
  external, 105

intermediate, 23
linker-read, 34
linker-written, 38
naming
See Filenames
NUL or null, 14, 26
object, 22, 34
object listing, 23
run, 38
scratch, names, 107
source listing, 22
FORTRAN
learning resources, vii
standard, subset, iv
FORTRAN.LEM, 37
FORTRAN.LIB, 37
Frame contents, 62
Framepointer, 62, 90
Front end, 76
Function return values, 67

Heap, 45, 86

I/O, 59
Identifier limits, 46
INIVQQ, 59
INPUT file, 80
INTEGER*2, 65
INTEGER*4, 65
Intel
80286 processor, 84
Interface, standard, 80
Intermediate files, 23
Internal representations, 65
Interpreter, 3
Interrupts
vectors, 85

Libraries, runtime, 9, 37
Limits
compile time memory, 46
complex expressions, 46
identifiers, 46
physical, 44
Link time, 7
Linker
defaults, 36
globals, 86
listing file, 38
map, 38
prompts, 35
public names, 81
switches, 40
Linking
general discussion, 3, 33
large programs, 44, 57
no library search, 37
object files, 5
Load module, 59
LOGICAL*2, 65
LOGICAL*4, 65
Long calls, 63

Machine level
initialization, 90
Memory
8086, organization, 84
contents, 85
linking, 39
organization, 85
Microsoft MACRO
Assembler, 5
Microsoft Pascal, 75
Module
definition, 7
load, 59
object, 34
relocatable, 8
size, 59
MS-DOS procedures, 2

Naming conventions, 81
Notation conventions, vii
NUL or null file, 14, 26
Object
code, 44
file, 5, 22, 34
listing file, 23
modules, 34
Offset addresses, 8
Optimization, 3, 77
OUTPUT file, 80

Parameters
batch file, 41
procedural, 66
PASCOM, 75
Pass one, 13, 54, 75
Pass three, 16, 75, 79
Pass two, 15, 55, 75, 77
Preliminary procedures, 2
Program
development steps, 3-6
example, 57
execution, example, 19
initialization, 91
large, 57
source, 7
termination, 91
Public routines, 34

QQ naming convention, 81

Real number
  format, 103
  operations, 60
REAL*4, 65
REAL*8, 65
Reference books, vii
Register
  saved, 63
  segment, 84
Relocatable
  module, 8
  object file, 5
RETURN key, vii
Routine, 8
Run file, 5, 38
Runtime
  architecture, 83
  definition, 7
  error handling, 92
  initialization, 87
  libraries, 9, 37
  routines, 83
  structure, 34
  termination, 87

Sample session, 11
Scratch file names, 107
Short calls, 63
Software provided, iv
Source
  file, 5, 7, 12
  listing file, 22
  program, 7
Square brackets,
  syntax notation, vii
Stack, 45, 86
Stackpointer, 89
Statement
  syntax, vii
  trees, 77
Subexpression elimination, 77
Subroutines, 44
Switches, linker, 40
Symbol table, 47, 75
Syntax notation, vii
System requirements, iv

Technical information, 74
Templates, 78
TRAP flag, 81
TRMVQQ, 80

Undefined variable, 8
Unit F, 81
Unit identifiers, 83
Unit U, 80-82
Unit V, 80-82
Unresolved variable, 8

VM.TMP, 39, 40
Vocabulary, 7

Fortran Language

MS™-FORTRAN

Reference Manual

Contents

Tables v

Introduction vi

- About This Manual vii
- Syntax Notation viii
- Learning More About FORTRAN ix

1/Language Overview 1

- 1.1 MS-FORTRAN Metacommands 3
- 1.2 Programs and Compilable Parts of Programs 4
- 1.3 Input/Output 6
- 1.4 Statements 8
- 1.5 Expressions 1Ø
- 1.6 Names 11
- 1.7 Types 12
- 1.8 Lines 13
- 1.9 Characters 14

2/Terms and Concepts 15

- 2.1 Notation 16
- 2.2 Lines and Statements 18
- 2.3 Data Types 22
- 2.4 Names 29
- 2.5 Expressions 32

3/Statements 39

- 3.1 Categories of Statements 41
- 3.2 Statement Directory 46

4/The I/O System 93

- 4.1 Records 95
- 4.2 Files 96
- 4.3 I/O Statements 105

4.4 Formatted I/O..109
4.5 List-Directed I/O...120

5/Programs, Subroutines, and Functions........125

5.1 Main Program..127
5.2 Subroutines...128
5.3 Functions..129
5.4 Arguments...137

6/The MS-FORTRAN Metacommands..............139

6.1 Overview...140
6.2 Metacommand Directory..................................142

Appendices...149

A/MS-FORTRAN and ANSI
Subset FORTRAN...151

B/ASCII Character Codes......................................155

C/Error Messages..157

Index..169

Tables

Table 1.1. I/O Statements.......................................6
Table 1.2. Categories of Statements in FORTRAN.........9
Table 2.1. Memory Requirements...........................22
Table 2.2. Arithmetic Operators...............................33
Table 2.3. Relational Operators...............................35
Table 2.4. Logical Operators.....................................37
Table 3.1. Specification Statements..........................42
Table 3.2. Control Statements....................................44
Table 3.3. I/O Statements..45
Table 3.4. Conversion of Integer Values in V = E......48
Table 3.5. Type Conversion of Real Values in V = E..49
Table 3.6. Edit Descriptors...69
Table 4.1. Carriage Control Characters....................108
Table 4.2. Scale Factors for E and D Editing............116
Table 4.3. Data Conversion Equivalents....................117
Table 5.1. Intrinsic Functions.................................131
Table 6.1. The MS-FORTRAN Metacommands...........140

Introduction

This is a language reference manual for the Microsoft® FORTRAN language system. MS™-FORTRAN conforms to Subset FORTRAN, as described in ANSI X3.9-1978. MS-FORTRAN includes extensions to the subset language and some of the features of the full ANSI standard, commonly known as FORTRAN 77. See Appendix A, "MS-FORTRAN and ANSI Subset FORTRAN," for details.

The syntactical rules for using FORTRAN are rigorous and require the programmer to fully define the characteristics of the solution to a problem in a series of precise statements. Therefore, we recommend that you have a general understanding of some dialect of FORTRAN before using this product. This manual is not a tutorial; for a list of suggested FORTRAN texts, see "Learning More About FORTRAN" in this introduction.

About This Manual

This manual is organized as follows:

Chapter 1, "Language Overview," is general in scope, providing a broad picture of the MS-FORTRAN language. Later chapters discuss the elements of the language in more detail.

Chapter 2, "Terms and Concepts," describes the smaller elements of the language, from notation to data types to expressions, and explains program structure.

Chapter 3, "Statements," defines MS-FORTRAN statements, both executable and nonexecutable.

Chapter 4, "The I/O System," provides additional information about input and output and the MS-FORTRAN file system.

Chapter 5, "Programs, Subroutines, and Functions," describes the subroutine structure, including argument passing and intrinsic (system-provided) functions.

Chapter 6,"The MS-FORTRAN Metacommands," describes the syntax and use of the metacommands.

Appendix A,"MS-FORTRAN and ANSI Subset FORTRAN," describes the differences between MS-FORTRAN and ANSI Subset FORTRAN.

Appendix B, "ASCII Character Codes," is a table of the entire ASCII character set.

Appendix C, "Error Messages," lists the compilation and runtime error messages you may see when you compile and run your program.

For information on how to use the MS-FORTRAN Compiler and the details of your specific version of MS-FORTRAN, see the *MS-FORTRAN Compiler User's Guide.*

Syntax Notation

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

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

< > Angle brackets indicate user-supplied elements. When the angle brackets enclose lowercase text, you replace the entry with an item defined by the text (e.g., the name of a specific file for <filename<).

[ ] Square brackets indicate that the enclosed entry is optional (e.g., A[<w<], where <w< represents a field width, indicates that either A or A12, for example, is valid).

... Ellipses indicate that an entry may be repeated as many times as needed or desired. For example, the EXTERNAL statement is described as follows:

EXTERNAL <name< [, <name<]...

The syntactic items denoted by <name< may be repeated any number of times, separated by commas.

All other punctuation, such as commas, colons, slash marks, parentheses, and equal signs, must be entered exactly as shown.

Blanks normally have no significance in the description of MS-FORTRAN statements. The general rules for blanks, covered in Section 2.1.2, "Blanks," govern the interpretation of blanks in all contexts.

Learning More About FORTRAN

The manuals in this package provide complete reference information for your implementation of the MS-FORTRAN Compiler. They do not, however, teach you how to write programs in FORTRAN. If you are new to FORTRAN or need help in learning to program, we suggest you read any of the following books:

Agelhoff, R., and Mojena, Richard. *Applied FORTRAN 77, Featuring Structured Programming.* Wadsworth, 1981.

Ashcroft, J., Eldridge, R. H., Paulson, R. W., and Wilson, G. A. *Programming With FORTRAN 77.* Granada, 1981.

Friedman, F., and Koffman, E. *Problem Solving and Structured Programming in FORTRAN.* Addision-Wesley, 2nd edition, 1981.

Wagener, J. L. *FORTRAN 77: Principles of Programming.* Wiley, 1980.

Chapter 1

Language Overview

1.1 MS-FORTRAN Metacommands.......................3

1.2 Programs and Compilable Parts of Programs.......................4

1.3 Input/Output.......................6

1.4 Statements.......................8

1.5 Expressions.......................10

1.6 Names.......................11

1.7 Types.......................12

1.8 Lines.......................13

1.9 Characters.......................14

This chapter provides a summary description of the elements of the MS-FORTRAN language. The remaining chapters of the manual provide detailed information on these elements, from the character set to the metacommands.

1.1 MS-FORTRAN Metacommands

The metalanguage is the control language for the MS-FORTRAN Compiler. Metacommands let you specify options that affect the overall operation of a compilation. For example, with metacommands you can enable or disable generation of a listing file, runtime error checking code, or use of MS-FORTRAN features that are not a part of the subset or full language standard.

The metalanguage consists of commands that appear in your source code, each on a line of its own and each with a dollar sign ($) in column one.

The metalanguage is a level of language designed to enhance your use of the MS-FORTRAN Compiler. Although most implementations of FORTRAN have some type of compiler control, the MS- FORTRAN metacommands are not part of standard FORTRAN (and hence, not portable).

These are the metacommands currently available:

$[NO]DEBUG	$PAGESIZE
$DO66	$STORAGE
$[NO]FLOATCALLS	$[NOT]STRICT
$INCLUDE	$SUBTITLE
$LINESIZE	$TITLE
[NO]LIST	
$MESSAGE	
$PAGE	

See Chapter 6, "The MS-FORTRAN Metacommands," for a complete discussion of metacommands.

1.2 Programs and Compilable Parts of Programs

The MS-FORTRAN Compiler processes program units. A program unit may be a main program, a subroutine, or a function. You can compile any of these units separately and later link them together without having to recompile them as a whole.

1. Program

   Any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. The first statement may be a PROGRAM statement, but such a statement is not required. The execution of a 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.

2. Subroutine

   A program unit that can be called from other program units by a CALL statement. When called, a subroutine 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 arguments or common variables.

3. Function

   A program unit referred to in an expression. A function directly returns a value that is used in the computation of that expression and in addition may pass back values via arguments. There are three kinds of functions: external, intrinsic, and statement. (Statement functions cannot be compiled separately.)

Subroutines and functions let you develop large structured programs that can be broken into parts. This is advantageous in the following situations:

1. If a program is large, breaking it into parts makes it easier to develop, test, and maintain.
2. If a program is large and recompiling the entire source file is time consuming, breaking the program into parts saves compilation time.
3. If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs in which the routines are used.
4. If a routine could be implemented in any of several ways, you might place it in a file and compile it separately. Then, to improve performance, you can alter the implementation, or even rewrite the routine in assembly language or in MS-Pascal, and the rest of your program will not need to change.

See Chapter 5, "Programs, Subroutines, and Functions," for a complete discussion of compilable program units.

1.3 Input/Output

Input is the transfer of data from an external medium or an internal file to internal storage. The transfer process is called reading. Output is the transfer of data from internal storage to an external medium or to an internal file. This process is called writing.

A number of statements in FORTRAN are provided specifically for the purpose of such data transfer; some I/O statements also specify that some editing of the data be performed.

In addition to the statements that transfer data, there are several auxiliary I/O statements to manipulate the external medium or to determine or describe the properties of the connection to the external medium.

Table 1.1 lists the I/O statements that perform each of these three functions.

Table 1.1. I/O Statements

I/O Function	I/O Statements
Data transfer	READ WRITE
Auxiliary I/O	OPEN CLOSE BACKSPACE ENDFILE REWIND
File positioning	BACKSPACE ENDFILE REWIND

The following concepts are also important for understanding the I/O system:

1. Records

   The building blocks of the FORTRAN file system. A record is a sequence of characters or values. There are three kinds of records: formatted, unformatted, and endfile.

2. Files

   Sequences of records. Files are either external or internal.

   An external file is a file on a device or a device itself. An internal file is a character variable that serves as the source or destination of some formatted I/O action.

   All files have the following properties:

   a. a filename (optional)

   b. a file position

   c. structure (formatted, unformatted, or binary)

   d. access method (sequential or direct)

   Although a wide variety of file types are possible, most applications will need just two: implicitly opened and explicitly opened external, sequential, formatted files.

See Section 3.2, "Statement Directory," for descriptions of individual I/O statements. See Chapter 4, "The I/O System," for a complete discussion of records, files, and formatted data editing.

1.4 Statements

Statements perform a number of functions, such as computing, storing the results of computations, altering the flow of control, reading and writing files, and providing information for the compiler. Statements in FORTRAN fall into two broad classes: executable and nonexecutable.

An executable statement causes an action to be performed. Nonexecutable statements do not in themselves cause operations to be performed. Instead, they specify, describe, or classify elements of the program, such as entry points, data, or program units. Table 1.2 describes the functional categories of statements.

Table 1.2. Categories of Statements in FORTRAN

Category	Description
Assignment	Executable. Assigns a value to a variable or an array element.
Comment	Nonexecutable. Allows comments within program code.
Control	Executable. Controls the order of execution of statements.
DATA	Nonexecutable. Assigns initial values to variables.
FORMAT	Nonexecutable. Provides data editing information.
I/O	Executable. Specifies sources and destinations of data transfer, and other facets of I/O operation.
Specification	Nonexecutable. Defines the attributes of variables, arrays, and programmer function names.
Statement Function	Nonexecutable. Defines a simple, locally used function.
Program Unit Heading	Nonexecutable. Defines the start of a program unit and specifies its formal arguments.

See Chapter 3, "Statements," for a complete discussion and a directory of MS-FORTRAN statements.

1.5 Expressions

An expression is a formula for computing a value. It consists of a sequence of operands and operators. The operands may contain function invocations, variables, constants, or even other expressions. The operators specify the actions to be performed on the operands.

In the following expression, plus (+) is an operator and A and B are operands:

```
A + B
```

There are four basic kinds of expressions in FORTRAN:

1. arithmetic expressions
2. character expressions
3. relational expressions
4. logical expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of every expression produces a value of a specific type.

Expressions are not statements, but may be components of statements. In the following example, the entire line is a statement; only the portion after the equal sign is an expression:

```
X = 2.0 / 3.0 + A * B
```

See Section 2.5, "Expressions," for a discussion of expressions in MS-FORTRAN.

1.6 Names

Names denote the variables, arrays, functions, or subroutines in your program, whether defined by you or by the MS-FORTRAN system. A FORTRAN name consists of a sequence of alphanumeric characters. The following restrictions apply:

1. The maximum number of characters in a name is 660 characters (66 characters per line multiplied by ten lines).
2. The initial character must be alphabetic; subsequent characters must be alphanumeric.
3. Blanks are skipped.
4. Only the first six alphanumeric characters are significant; the rest are ignored.

With these restrictions regarding the make-up of the name, any valid sequence of characters can be used for any FORTRAN name. There are no reserved names as in other languages.

Sequences of alphabetic characters used as keywords by the MS-FORTRAN Compiler are not confused with user-defined names. The compiler recognizes keywords by their context and in no way restricts the use of user-defined names. Thus, for example, a program can have an array named IF, READ, or GOTO, with no error (as long as it otherwise conforms to the rules that all arrays must obey). However, use of keywords for user-defined names often interferes with the readability of a program, so the practice should be avoided.

See Section 2.4, "Names," for more information on the scope and use of names in MS-FORTRAN.

1.7 Types

Data in MS-FORTRAN belongs to one of five basic types:

1. integer (INTEGER*2 and INTEGER*4).
2. single precision real (REAL*4 or REAL).
3. double precision real (REAL*8 or DOUBLE PRECISION).
4. logical (LOGICAL*2 and LOGICAL*4).
5. character (CHARACTER).

The full language described by the FORTRAN 77 standard also has a complex data type, which is not part of the subset, nor a part of MS-FORTRAN.

Data types can be declared. If not declared, the type is determined by the first letter of its name (either by default or by an IMPLICIT statement). A type statement can also include dimension information.

See Section 2.3, "Data Types," for a more complete discussion of MS-FORTRAN data types. See Section 3.2, "Statement Directory," for a detailed description of the type statement.

1.8 Lines

Lines are composed of a sequence of characters. Characters beyond the 72nd on a line are ignored; lines shorter than 72 characters are assumed to be padded with blanks.

The position of characters within a line in FORTRAN is significant. Characters in columns 1 through 5 are recognized as statement labels, a character in column 6 as a continuation indicator, and characters in columns 7 through 72 as the FORTRAN statements themselves. Comments are recognized by either the letter "C" or an asterisk (*) in column 1, metacommands by a dollar sign ($) in column 1.

With some exceptions, blanks are not significant in FORTRAN. Tab characters have significance in a few circumstances, described in Section 2.1, "Notation.".

Lines in MS-FORTRAN may serve as any of the following:

1. a metacommand line
2. a comment line
3. an initial line (of a statement)
4. a continuation line (of a statement)

A metacommand line has a dollar sign in column 1 and controls the operation of the MS-FORTRAN Compiler.

A comment line has either a "C", a "c", or an asterisk in column 1, or the line is entirely blank, and is ignored during processing.

An initial line of a statement has either a blank or a zero in column 6 and has either all blanks or a statement label in columns 1 through 5.

A continuation line is any line that is not a metacommand line, a comment line, or an initial line, and which has blanks in columns 1 through 5, and in column 6 has a character that is not a blank or zero.

See Section 2.2, "Lines and Statements," for details on the specific uses and limitations on the several kinds of lines in MS-FORTRAN and how statements are combined to form programs and compilable parts of programs.

1.9 Characters

In the most basic sense, a FORTRAN program is a sequence of characters. When these characters are submitted to the compiler, they are interpreted in various contexts as characters, names, labels, constants, lines, and statements.

The characters used in an MS-FORTRAN source program belong to the ASCII character set, a complete listing of which is given in Appendix B, "ASCII Character Codes." Briefly, however, the character set may be divided into three groups:

1. the 52 upper and lowercase alphabetic characters (A through Z and a through z)
2. the 10 digits (0 through 9)
3. special characters (all other printable characters in the ASCII character set)

See Section 2.1, "Notation," for more information about the use of characters in MS-FORTRAN.

Chapter 2

Terms And Concepts

2.1 Notation 16
2.1.1 Alphanumeric Characters 16
2.1.2 Blanks 16
2.1.3 Tabs 17
2.1.4 Columns 17
2.2 Lines and Statements 18
2.2.1 Initial Lines 18
2.2.2 Continuation Lines 18
2.2.3 Comment Lines 18
2.2.4 Statement Definition and Order 19
2.3 Data Types 22
2.3.1 Integer Data Types 23
2.3.2 The Single Precision Real Data Type 24
2.3.3 The Double Precision Real Data Type 25
2.3.4 Logical Data Types 26
2.3.5 The Character Data Type 27
2.4 Names 29
2.4.1 Scope of FORTRAN Names 29
2.4.2 Undeclared FORTRAN Names 30
2.5 Expressions 32
2.5.1 Arithmetic Expressions 32
2.5.2 Integer Division 34
2.5.3 Type Conversions of Arithmetic Operands 34
2.5.4 Character Expressions 35
2.5.5 Relational Expressions 35
2.5.6 Logical Expressions 36
2.5.7 Precedence of Operators 38
2.5.8 Rules for Evaluating Expressions 38
2.5.9 Array Element References 38

2.1 Notation

A FORTRAN source program is a sequence of ASCII characters. The ASCII character codes include:

1. 52 upper and lowercase letters (A through Z and a through z)
2. 1Ø digits (Ø, 1, 2, 3, 4, 5, 6, 7, 8, and 9)
3. special characters (the remaining printable characters of the ASCII character code)

2.1.1 Alphanumeric Characters

The letters and digits, treated as a single group, are called the alphanumeric characters. MS-FORTRAN interprets lowercase letters as uppercase letters in all contexts except in character constants and Hollerith fields. Thus, the following user-defined names are all equivalent in MS-FORTRAN:

ABCDE   abcde   AbCdE   aBcDe

The collating sequence for the MS-FORTRAN character set is the ASCII sequence (see Appendix B, "ASCII Character Codes," for a complete table of the ASCII characters).

2.1.2 Blanks

With the exceptions noted in the following list, the blank character has no significance in an MS-FORTRAN source program and may therefore be used for improving readability. The exceptions are the following:

1. Blanks within string constants are significant.
2. Blanks within Hollerith fields are significant.
3. A blank or zero in column 6 distinguishes initial lines from continuation lines.

2.1.3 Tabs

The TAB character has the following significance in an MS-FORTRAN source program:

1. If the TAB appears in columns 1 through 5, the next character on the source line is considered to be in column 7.
2. A TAB appearing in columns 7 through 72 is considered to be a blank (except as noted in point 3.).
3. A TAB appearing in a character constant or Hollerith field is not interpreted as a blank, but rather as a single ASCII TAB character. This treatment allows programs to output tabs.

2.1.4 Columns

The characters in a given line are positioned by columns, with the first character in column 1, the second in column 2, and so forth.

The column in which a character resides is significant in FORTRAN. Column 1 is used for comment indicators and metacommand indicators. Columns 1 through 5 are reserved for statement labels and column 6 for continuation indicators.

2.2 Lines and Statements

You can also think of a FORTRAN source program as a sequence of lines. Only the first 72 characters in a line are treated as significant by the compiler, with any trailing characters in a line ignored. Lines with fewer than 72 characters are assumed to be padded with blanks to 72 characters (for an illustration of this, see Section 2.3.5, "The Character Data Type," which describes character constants).

2.2.1 Initial Lines

An initial line is any line that is not a comment line or a metacommand line and that contains a blank or a zero character 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 statement label is a sequence of one to five digits, at least one of which 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.

2.2.2 Continuation Lines

A continuation line is any line that is not a comment line or a metacommand line and that contains any character in column 6 other than a blank or a zero. The first five columns of a continuation line must be blanks. A continuation line increases the amount of room to write a statement. If it will not fit on a single initial line, it may be extended to include up to 19 continuation lines.

2.2.3 Comment Lines

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

1. A "C" (or "c") appears in column 1.
2. An asterisk (*) appears in column 1.

3. The line contains all blanks.

Comment lines do not affect 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.

2.2.4 Statement Definition and Order

A FORTRAN statement consists of an initial line, followed by zero to nine continuation lines. A statement may contain as many as 660 characters in columns 7 through 72 of the initial line and columns 7 through 72 of the continuation lines. The END statement must be written within columns 7 through 72 of an initial line, and no other statement may have an initial line that appears to be an END statement. (In the following discussion, individual statements are simply referred to by name; see Chapter 3, "Statements," for definitions of specific statements and their properties.)

The FORTRAN language enforces a certain ordering of the statements and lines that make up a FORTRAN program unit. In addition, MS-FORTRAN enforces additional requirements in the ordering of lines and statements in an MS-FORTRAN compilation.

In general, a compilation consists of one or more program units (see Chapter 5, "Programs, Subroutines, and Functions," for more information on compilation units and subroutines). The various rules for ordering statements are illustrated in Figure 2.1 and described in the paragraphs following.

<table>
<tr><td colspan="4">$DO66, $STORAGE metacommands</td></tr>
<tr><td colspan="2">PROGRAM, FUNCTION, or SUBROUTINE statement</td><td rowspan="6">Other meta-commands</td><td rowspan="7">Comment lines</td></tr>
<tr><td>IMPLICIT statements</td><td rowspan="5">FORMAT 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>

Figure 2.1. Order of Statements within Program Units and Compilations

Within Figure 2.1, the following conventions apply:

1. Classes of lines or statements above or below other classes must appear in the designated order.
2. Classes of lines or statements may be interspersed with other classes that appear across from one another.

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 statement other than a SUBROUTINE or FUNCTION statement and ends with an END statement. A subprogram or the main program is referred to as a program unit.

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

1. A PROGRAM statement, if present, or a SUBROUTINE or FUNCTION statement, must be the first statement of the program unit.
2. FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION statement, or PROGRAM statement, if present.

3. All specification statements must precede all DATA statements, statement function statements, and executable statements.
4. All DATA statements must appear after the specification statements and precede all statement function statements and executable statements.
5. All statement function statements must precede all executable statements.
6. Within the specification statements, the IMPLICIT statement must precede all other specification statements.
7. The DO66 and STORAGE metacommands, if present, must appear before all other statements; other metacommands may appear anywhere in the program unit.

2.3 Data Types

There are five basic data types in MS-FORTRAN:

1. integer (INTEGER*2 and INTEGER*4)
2. real (REAL*4 or REAL)
3. double precision (REAL*8 or DOUBLE PRECISION)
4. logical (LOGICAL*2 and LOGICAL*4)
5. character

The properties of, the range of values for, and the form of constants for each type are described in the following pages; memory requirements are shown in Table 2.1.

Table 2.1. Memory Requirements

Type	Bytes	Note
LOGICAL	2 or 4	1, 3
LOGICAL*2	2	
LOGICAL*4	4	
INTEGER	2 or 4	1, 3
INTEGER*2	2	
INTEGER*4	4	
CHARACTER	1	2
CHARACTER*n	n	4
REAL	4	3, 5
REAL*4	4	
REAL*8	8	3, 6
DOUBLE PRECISION	8	

Notes for Table 2.1:

1. Either 2 or 4 bytes are used. The default is 4, but may be set explicitly to either 2 or 4 with the $STORAGE metacommand.
2. CHARACTER and CHARACTER*1 are synonymous.
3. In some implementations, all numeric and logical data types always start on an even byte boundary.
4. Maximum n is 127.
5. REAL and REAL*4 are synonymous.
6. REAL*8 and DOUBLE PRECISION are synonymous.

Important

On many microprocessors, the code required to perform 16-bit arithmetic is considerably faster and smaller than the corresponding code to perform 32-bit arithmetic. Therefore, unless you set the MS-FORTRAN $STORAGE metacommand to a value of 2, programs will default to 32-bit arithmetic and may run more slowly than expected (see Section 6.2.8, "The $STORAGE Metacommand"). Setting the $STORAGE metacommand to 2 allows programs to run faster and use less code.

2.3.1 Integer Data Types

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 two or four bytes of memory, depending on the setting of the $STORAGE metacommand. A 2-byte integer, INTEGER*2, can contain any value in the range -32767 to 32767. A 4-byte integer, INTEGER*4, can contain any value in the range -2,147,483,647 to 2,147,483,647.

Integer constants consist of a sequence of one or more decimal digits or a radix specifier, followed by a string of digits in the range Ø...(radix - 1), where values between 1Ø and 35 are represented by the letters "A" through "Z", respectively.

A radix specifier consists of the character "#", optionally preceded by a string of decimal characters that represent the integer value of the radix. If the string is omitted, the radix is assumed to be 16. If the radix specifier is omitted, the radix is assumed to be 10.

Either format may be preceded by an optional arithmetic sign, plus (+) or minus(–). Integer constants must also be in range. A decimal point is not allowed in an integer constant.

The following are examples of integer constants:

123	+123	0
00000123	32767	-32767
-#AB05	2#010111	-36#ABZ07

An integer can be specified in MS-FORTRAN as INTEGER*2, INTEGER*4, or INTEGER. The first two specify 2-byte and 4-byte integers, respectively. INTEGER specifies either 2-byte or 4-byte integers, according to the setting of the $STORAGE metacommand (the default is four bytes).

2.3.2 The Single Precision Real Data Type

The real data type (REAL or REAL*4) consists of a subset of the real numbers, the single precision real numbers. A single precision real value is normally an approximation of the real number desired and occupies four bytes of memory.

The range of single precision real values is approximately as follows:

8.43E-37 to 3.37E+38	(positive range)
-3.37E+38 to -8.43E-37	(negative range)
0	(zero)

The precision is greater than six decimal digits.

A basic real constant consists of:

1. an optional sign
2. an integer part
3. a decimal point
4. a fraction part
5. an optional exponent part

The integer and fraction parts consist of one or more decimal digits, and the decimal point is a period (.). Either the integer part or the fraction part may be omitted, but not both. Some sample real constants are:

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

The exponent part consists of the letter "E" followed by an optionally signed integer constant of one or two digits. An exponent indicates that the value preceding it is to be multiplied by ten to the value of the exponent part's integer.

Some sample exponent parts are:

E12 E-12 E + 12 EØ

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.ØØØE-2	1.E-2	1E-2
+ Ø.Ø1	1ØØ.ØE-4	.ØØØ1E + 2

All represent the same real number, one one-hundredth.

2.3.3 The Double Precision Real Data Type

The double precision real data type (REAL*8 or DOUBLE PRECISION) consists of a subset of the real numbers, the double precision real numbers. This subset is larger than the subset for the REAL (REAL*4) data type.

A double precision real value is normally an approximation of the real number desired. A double precision real value can be a positive, negative, or zero value and occupies eight bytes of memory. The range of double precision real values is approximately:

4.19D-3Ø7 to 1.67D + 3Ø8	(positive range)
-1.67D + 3Ø8 to -4.19D-3Ø7	(negative range)
Ø	(zero)

The precision is greater than 15 decimal digits.

A double precision constant consists of:

1. an optional sign
2. an integer part
3. a decimal point
4. a fraction part
5. a required exponent part

The exponent uses "D" rather than "E" to distinguish it from single precision. The integer and fraction parts consist of one or more decimal digits, and the decimal point is a period. Either the integer part or the fraction part, but not both, may be omitted.

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

```
+1.123456789D-2    1.D-2             1D-2
+0.0000000001D0    100.0000005D-4    .00012345D+2
```

The exponent part consists of the letter "D" followed by an integer constant. The integer constant may have a sign as an option. An exponent indicates that the value preceding it is to be multiplied by ten to the value of the exponent part's integer. If the exponent is zero, it must be specified as a zero.

Some sample exponent parts are:

```
D12    D-12    D+12    D0
```

2.3.4 Logical Data Types

The logical data type consists of the two logical values .TRUE. and .FALSE.. A logical variable occupies two or four bytes of memory, depending on the setting of the $STORAGE metacommand. The default is four bytes. The significance of a logical variable is unaffected by the $STORAGE metacommand, which is present primarily to allow compatibility with the ANSI requirement that logical, single precision real, and integer variables are all the same size.

LOGICAL*2 values occupy two bytes. The least significant (first) byte is either 0 (.FALSE.) or 1 (.TRUE.); the most significant byte is undefined.

LOGICAL*4 variables occupy two words, the least significant (first) of which contains LOGICAL*2 value. The most significant word is undefined.

2.3.5 The Character Data 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 127 characters. A character variable occupies one byte of memory for each character in the sequence.

Character variables are assigned to contiguous bytes without regard for word boundaries. However, the compiler assumes that noncharacter variables that follow character variables always start on word boundaries.

A character constant consists of a sequence of one or more characters enclosed by a pair of single quotation marks. Blank characters are permitted in character constants and are significant. The case of alphabetic characters is significant. A single quotation mark within a character constant is represented by two consecutive single quotation marks with no blanks in between.

The length of a character constant is equal to the number of characters between the single quotation marks. A pair of single quotation marks counts as a single character. Some sample character constants are:

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

The last example ('''') is a character constant that contains one apostrophe (single quotation mark).

FORTRAN permits source lines of up to 72 columns. Shorter lines are padded with blanks to 72 columns. 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 appended to column 72 of the initial line.

Thus, the following FORTRAN source,

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

is equivalent to:

```
200     CH = 'ABC (58 blank spaces) ... DEF'
```

with 58 blank spaces between the C and D being equivalent to the space from C in column 15 to column 72 plus one blank in column 7 of the continuation line. Very long character constants can be represented in this manner.

2.4 Names

An MS-FORTRAN name, or identifier, consists of a sequence of alphanumeric characters (the maximum is 66 characters per line multiplied by ten lines). The initial character must be alphabetic; subsequent characters must be alphanumeric. Blanks are skipped. Only the first six alphanumeric characters are significant; the rest are ignored.

A name denotes a user-defined or system-defined variable, array, or program unit. Any valid sequence of characters can be used for any FORTRAN name.

There are no reserved names as in other languages. Sequences of alphabetic characters used as keywords by the MS-FORTRAN Compiler are not confused with user-defined names. The compiler recognizes keywords by their context and in no way restricts the use of user-defined names.

Thus, a program can have, for example, an array named IF, READ, or GOTO, with no error (as long as it conforms to the rules that all arrays must obey). However, use of keywords for user-defined names often interferes with the readability of the program, so the practice should be avoided.

2.4.1 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 can 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 the same program. All subroutine, function subprogram, and common block 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 visible (known) within a single program unit. A name with a local scope can be used in another program unit with a different meaning or with a similar meaning, but is in no way required to have a similar meaning in a different scope. The names of variables, arrays, arguments, and statement functions all have local scope.

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

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

2.4.2 Undeclared FORTRAN Names

The compiler can infer from context how to classify a user name in an executable statement, if that name has not been previously encountered.

If the name is used as a variable, the compiler creates an entry in the symbol table for a variable of that name. The type of the variable is inferred from the first letter of its name. Variables beginning with the letters I, J, K, L, M, or N are normally considered integers, while all others are considered real numbers. However, you can override these defaults with an IMPLICIT statement. For more information, see Section 3.2.26,"The IMPLICIT Statement").

If an undeclared name is used as a function call, the compiler creates a symbol table entry for a function of that name. Its type is inferred in the same manner as the type of a variable.

Similarly, a subroutine entry is created for a newly encountered name that is 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.

2.5 Expressions

An expression is a formula for computing a value. It consists of a sequence of operands and operators. The operands may contain function invocations, variables, constants, or even other expressions. The operators specify the actions to be performed on the operands.

FORTRAN has four classes of expressions:

1. arithmetic
2. character
3. relational
4. logical

2.5.1 Arithmetic Expressions

An arithmetic expression produces a value that is of type integer, or real, or double precision. The simplest forms of arithmetic expressions are:

1. integer, real, or double precision constants
2. integer, real, or double precision variable references
3. integer, real, or double precision array element references
4. integer, real, or double precision function references

The value of a variable reference or array element reference must be defined before it can 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 simple forms in the preceding list using parentheses and the arithmetic operators shown in Table 2.2.

Table 2.2. Arithmetic Operators

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

All of the operators may be used as binary operators, which appear between their arithmetic expression operands. The plus (+) and minus (-) may also be unary, and precede their operand.

Operations of equal precedence, except exponentiation, are left-associative. Exponentiation is right- associative. Thus, each of the following expressions on the left is the same as the corresponding expression on the right:

```
A/B*C      (A/B)*C
A**B**C    A**(B**C)
```

Arithmetic expressions can be formed in the usual mathematical sense, as in most programming languages. However, FORTRAN prohibits two operators from appearing consecutively. For example, this is prohibited,

```
A**-B
```

while this is allowed:

```
A**(-B)
```

Unary minus is also of lowest precedence. Thus, the expression -A**B is interpreted as -(A**B)

You may use parentheses in an expression to control associativity and the order in which operators are evaluated.

2.5.2 Integer Division

The division of two integers results in a value that is the mathematical quotient of the two values, truncated downward (i.e., toward zero). Thus, 7/3 evaluates to 2, and (-7)/3 evaluates to -2. Both 9/10 and 9/(-10) evaluate to zero.

2.5.3 Type Conversions of Arithmetic Operands

When all operands of an arithmetic expression are of the same data type, the value returned by the expression is also of that type. When the operands are of different data types, the data type of the value returned by the expression is the type of the highest-ranked operand.

The rank of an operand depends on its data type, as shown in the following list:

1. INTEGER*2 (lowest)
2. INTEGER*4
3. REAL*4
4. REAL*8 (highest)

For example, an operation on an INTEGER*2 and a REAL*4 element produces a value of data type REAL*4.

The data type of an expression is the data type of the result of the last operation performed in evaluating the expression.

The data types of operations are classified as either INTEGER*2, INTEGER*4, REAL*4, or REAL*8.

Integer operations are performed on integer operands only. A fraction resulting from division is truncated in integer arithmetic, not rounded. Thus, the following evaluates to zero, not one:

```
1/4 + 1/4 + 1/4 + 1/4
```

Memory allocation for the type INTEGER, without the *2 or *4 length specification, is dependent on the use of the $STORAGE metacommand. (See the note at the beginning of Section 2.3,"Data Types," and Section 6.2.8,"The $STORAGE Metacommand," for details.)

Real operations are performed on real operands or combinations of real and integer operands only. Integer operands are first converted to real data type by giving each a fractional part equal to zero. Real arithmetic is then used to evaluate the expression. But in the following statement, integer division is performed on I and J, and a real multiplication on the result and X:

```
Y = (I/J)*X
```

2.5.4 Character Expressions

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

1. character constants
2. character variable references
3. character array element references
4. any character expression enclosed in parentheses

There are no operators that result in character expressions.

2.5.5 Relational Expressions

Relational expressions compare the values of two arithmetic or two character expressions. An arithmetic value may not be compared with a character value, unless the $NOTSTRICT metacommand has been specified. In this case, the arithmetic expression is considered to be a character expression. The result of a relational expression is of type LOGICAL. Relational expressions can use any of the operators shown in Table 2.3 to compare values.

Table 2.3. Relational Operators

Operator	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

All of the relational operators are binary operators and appear between their operands. There is no relative precedence or associativity among the relational operands since an expression of the following form violates the type rules for operands:

A .LT. B .NE. C

Relational expressions may only appear within logical expressions.

Relational expressions with arithmetic operands may have one 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. If operands of unequal length are compared, the shorter operand is considered as if it were extended to the length of the longer operand by the addition of spaces on the right.

2.5.6 Logical Expressions

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

1. logical constants
2. logical variable references
3. logical array element references
4. logical function references
5. relational expressions

Other logical expressions are built up from the simple forms in the preceding list by using parentheses and the logical operators of Table 2.4.

Table 2.4. Logical Operators

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

The .AND. and .OR. operators are binary operators and appear between their logical expression operands. The .NOT. operator is unary and precedes its operand.

Operations of equal precedence are left-associative; thus, 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)
```

Two .NOT. operators cannot be adjacent to each other, although.

```
A .AND. .NOT. B
```

is an example of an allowable expression with two adjacent operators.

Logical operators have the same meaning as in standard mathematical semantics, with .OR. being nonexclusive. For example,

```
.TRUE. .OR. .TRUE.
```

evaluates to the value:

```
.TRUE.
```

2.5.7 Precedence of Operators

When arithmetic, relational, and logical operators appear in the same expression, they abide by the following precedence guidelines:

1. Logical (lowest)
2. Relational (intermediate)
3. Arithmetic (highest)

2.5.8 Rules for Evaluating Expressions

Any variable, array element, or function that is referred to in an expression must be defined at the time the reference is made. Integer variables must be defined with an arithmetic value, rather than a statement label value as set by an ASSIGN statement.

Certain arithmetic operations, such as dividing by zero, are not mathematically meaningful and are prohibited. Other prohibited operations include raising a zero-value operand to a zero or negative power and raising a negative-value operand to a power of type REAL.

2.5.9 Array Element References

An array element reference identifies one element of an array. Its syntax is as follows:

<array> (<sub> [, <sub>]...)

<array> is the name of an array

<sub> is a subscript expression, that 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 one and the upper limit for the dimension it represents, inclusive.

```
C EXAMPLE OF DIMENSION STATEMENT
      DIMENSION A(3,2),B(3,4),C(4,5),D(5,6),V(3,9)
      EQUIVALENCE (X,V(1)), (Y,V(2))
      D(I,J) = D(I,J)/PIVOT
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
      READ(*,*) (V(N),N = 1,10)
```

Chapter 3

Statements

3.1 Categories of Statements....41

3.1.1 PROGRAM, SUBROUTINE, and FUNCTION Statements....42

3.1.2 Specification Statements....42

3.1.3 The DATA Statement....43

3.1.4 The FORMAT Statement....43

3.1.5 Assignment Statements....43

3.1.6 Control Statements....43

3.1.7 I/O Statements....44

3.2 Statement Directory....46

3.2.1 The ASSIGN Statement (Label Assignment)....46

3.2.2 The Assignment Statement (Computational)....46

3.2.3 The BACKSPACE Statement....5Ø

3.2.4 The CALL Statement....5Ø

3.2.5 The CLOSE Statement....53

3.2.6 The COMMON Statement....53

3.2.7 The CONTINUE Statement....55

3.2.8 The DATA Statement....55

3.2.9 The DIMENSION Statement....57

3.2.1Ø The DO Statement....58

3.2.11 The ELSE Statement....61

3.2.12 The ELSEIF Statement....62

3.2.13 The END Statement....63

3.2.14 The ENDFILE Statement....63

3.2.15 The ENDIF Statement....64

3.2.16 The EQUIVALENCE Statement....64

3.2.17 The EXTERNAL Statement....................68

3.2.18 The FORMAT Statement........................68

3.2.19 The FUNCTION Statement (External).......70

3.2.20 The GOTO Statement (Assigned GOTO)...72

3.2.21 The GOTO Statement (Computed GOTO).72

3.2.22 The GOTO Statement (Unconditional GOTO)..........................73

3.2.23 The IF Statement (Arithmetic IF)............74

3.2.24 The IF Statement (Logical IF)..................74

3.2.25 The IF THEN ELSE Statement (Block IF)..75

3.2.26 The IMPLICIT Statement.......................77

3.2.27 The INTRINSIC Statement......................78

3.2.28 The OPEN Statement.............................79

3.2.29 The PAUSE Statement............................80

3.2.30 The PROGRAM Statement......................81

3.2.31 The READ Statement.............................81

3.2.32 The RETURN Statement.........................83

3.2.33 The REWIND Statement.........................84

3.2.34 The SAVE Statement..............................84

3.2.35 The Statement Function Statement..........85

3.2.36 The STOP Statement..............................86

3.2.37 The SUBROUTINE Statement.................86

3.2.38 The Type Statement...............................87

3.2.39 The WRITE Statement...........................90

3.1 Categories of Statements

Statements perform a number of functions, such as computing, storing the results of computations, altering the flow of control, reading and writing files, and providing information for the compiler.

FORTRAN statements fall into two broad classes: executable and nonexecutable. An executable statement causes an action to be performed. Nonexecutable statements do not in themselves cause operations to be performed. Instead, they specify, describe, or classify elements of the program, such as entry points, data, or program units.

Nonexecutable statements include the following:

1. PROGRAM, SUBROUTINE, and FUNCTION statements
2. specification statements
3. the DATA statement
4. the FORMAT statement

The executable statements form a much larger group and may be divided into the following categories:

1. assignment statements
2. control statements
3. I/O statements

Sections 3.1.1 through 3.1.7 describe each of these types of statements, in general terms, in the order in which they are mentioned in the preceding lists.

Section 3.2,"Statement Directory," is an alphabetical listing of all statements. For each statement, the entry gives syntax and purpose, with remarks and examples as appropriate.

Chapter 4, "The I/O System," provides additional information on input and output in MS-FORTRAN.

3.1.1 PROGRAM, SUBROUTINE, and FUNCTION Statements

These statements identify the start of a program unit; all are nonexecutable. For more specific information, see the following sections: Section 3.2.19, "The FUNCTION Statement"; Section 3.2.30,"The PROGRAM Statement"; Section 3.2.35, "The Statement Function Statement"; and Section 3.2.37, "The SUBROUTINE Statement."

See also Chapter 5, "Programs, Subroutines, and Functions," for general information on program units.

3.1.2 Specification Statement

Specification statements in MS-FORTRAN are nonexecutable. They define the attributes of user-defined variable, array, and function names. Table 3.1 lists the eight specification statements, which are described in detail in Section 3.2, "Statement Directory."

Table 3.1. Specification Statements

Statement	Purpose
COMMON	Provides for sharing memory between two or more program units.
DIMENSION	Specifies that a user name is an array and defines the number of its elements.
EQUIVALENCE	Specifies that two or more variables or arrays share the same memory.
EXTERNAL	Identifies a user-defined name as an external subroutine or function.
IMPLICIT	Defines the default type for user-defined names.
INTRINSIC	Declares that a name is an intrinsic function.
SAVE	Causes variables to retain their values across invocations of the procedure in which they are defined.
Type	Specifies the type of user-defined names.

Specification statements must precede all executable statements in a program unit, but may appear in any order within their own group. The exception to this rule is the IMPLICIT statement, which must precede all other specification statements in a program unit.

3.1.3 The DATA Statement

The DATA statement assigns initial values to variables. A DATA statement is an optional, nonexecutable statement. If present, it must appear after all specification statements and before any statement function statements or executable statements. (See Section 3.2.8, "The DATA Statement," for more information.)

3.1.4 The FORMAT Statement

Format specifications provide explicit editing information for the data processed by a program. Format specifications may be given in a FORMAT statement or as character constants. (See Section 3.2.18, "The FORMAT Statement," for a description of the FORMAT statement and Section 4.4, "Formatted I/O," for additional information on formatted data.)

3.1.5 Assignment Statements

Assignment statements are executable statements that assign a value to a variable or an array element. There are two basic kinds of assignment statements: computational and label. (See Section 3.2.1, "The ASSIGN Statement," and Section 3.2.2, "The Assignment Statement," respectively, for further information.)

3.1.6 Control Statements

Control statements affect the order of execution of statements in FORTRAN. The control statements in MS-FORTRAN are shown in Table 3.2, along with a brief description of the function of each. See the appropriate entries in Section 3.2, "Statement Directory," for further information on each.

Table 3.2. Control Statements

Statement	Purpose
CALL	Calls and executes a subroutine from another program unit.
CONTINUE	Used primarily as a convenient way to place statement labels, particularly as the terminal statement in a DO loop.
DO	Causes repetitive evaluation of the statements following the DO, through and including the ending statement.
ELSE	Introduces an ELSE block.
ELSEIF	Introduces an ELSEIF block.
END	Ends execution of a program unit.
ENDIF	Marks the end of a series of statements following a block IF statement.
GOTO	Transfers control elsewhere in the program, according to the kind of GOTO statement used (assigned, computed, or unconditional).
IF	Causes conditional execution of some other statement(s), depending on the evaluation of an expression and the kind of IF statement used (arithmetic, logical, or block).
PAUSE	Suspends program execution until the RETURN key is pressed.
RETURN	Returns control to the program unit that called a subroutine or function.
STOP	Terminates a program.

3.1.7 I/O Statements

I/O statements transfer data, perform auxiliary I/O operations, and position files. Table 3.3 lists the MS-FORTRAN I/O statements (each of which is described in detail in Section 3.2, "Statement Directory").

Table 3.3. I/O Statements

Statement	Purpose
BACKSPACE	Positions the file connected to the specified unit to the beginning of the previous record.
CLOSE	Disconnects the unit specified and prevents subsequent I/O from being directed to that unit.
ENDFILE	Writes an end-of-file record on the file connected to the specified unit.
OPEN	Associates a unit number with an external device or with a file on an external device.
READ	Transfers data from a file to the items in an <iolist>.
REWIND	Repositions a specified unit to the first record in the associated file.
WRITE	Transfers data from the items in an <iolist> to a file.

In addition to these I/O statements, there is an I/O intrinsic function EOF (<unit-spec>). EOF function returns a logical value that indicates whether any data remains beyond the current position in the file associated with the given unit specifier. See Section 5.3.2, "Intrinsic Functions," for information about this function.

3.2 Statement Directory

The rest of this chapter is an alphabetical listing of all MS-FORTRAN statements, giving syntax and function, with notes and examples as necessary.

3.2.1 The ASSIGN Statement (Label Assignment)

Syntax ASSIGN <label> TO <variable>

Purpose Assigns the value of a format or statement label to an integer variable.

Remarks <label> is a format label or statement label.

<variable> 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 or a statement label and 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 input/output 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.

The value of a label is not necessarily the same as the label number. For example, the value of IVBL in the following is not necessarily 400:

```
ASSIGN 400 TO IVBL.
```

Hence, the variable is undefined as an integer; it cannot be used in an arithmetic expression until it has been redefined as such (by computational assignment or a READ statement).

3.2.2 The Assignment Statement (Computational)

Syntax <variable> = <expression>

Purpose Evaluates the expression and assigns the resulting value to the variable or array element specified.

Remarks

<variable> is a variable or array element reference.

<expression> is any expression.

The type of the variable or array element and the type of expression must be compatible.

1. If the type of the right-hand side is numeric, the type of the left-hand side must be numeric, and the statement is called an arithmetic assignment statement.
2. If the type of the right-hand side is logical, the type of the left-hand side must be logical, and the statement is called a logical assignment statement.
3. If the type of the right-hand side is character, the type of the left-hand side must also be character, and the statement is called a character assignment statement. However, if you have specified the $NOTSTRICT metacommand, the type of left-hand side may be numeric, logical, or character; the statement is still called a character assignment statement.

If the types of the elements of an arithmetic assignment statement are not identical, the value of the expression is automatically converted to the type of the variable. The conversion rules are given in Table 3.4 (for conversion to integer values) and Table 3.5 (for conversion to real values).

In both tables, the most significant portion is the high order, and the least significant is the low order. Also in both tables, the value converted (E) is shown in the second and third columns, while the type of the variable (V) is listed in column one.

Table 3.4. Conversion of Integer Values in V = E

V \ E	INTEGER*2	INTEGER*4
INTEGER*2	Assign E to V.	Assign least significant portion of E to V; most significant portion is lost.
INTEGER*4	Assign E to least significant portion of V; most significant portion is sign extended.	Assign E to V.
REAL*4	Append fraction (.0)+ to E and assign to V.†	Append fraction (.0) to E and assign to V.†
REAL*8	Append fraction (.0) to E and assign to V.†	Append fraction (.0) to E and assign to V.†

† "Fraction (.0)" means a zero fractional part.

Table 3.5. Type Conversion of Real Values in V = E

V \ E	REAL*4	REAL*8
INTEGER*2	Truncate E to INTEGER*2 and assign to V.	Truncate E to INTEGER*2 and assign to V.
INTEGER*4	Truncate E to INTEGER*4 and assign to V.	Truncate E to INTEGER*4 and assign to V.
REAL*4	Assign E to V.	Assign most significant portion of E to V; least significant portion is, rounded.
REAL*8	Convert E to equivalent REAL*8 form and assign to V.	Assign E to V.

For character assignments, if the length of the expression does not match the size of the variable, the expression is adjusted, in the following manner, so that it does match:

1. If the expression is shorter than the variable, the expression is padded with enough blanks on the right before the assignment takes place to make the sizes equal.

2. If the expression is longer than the variable, characters on the right are truncated to make the sizes the same.

Logical expressions of any size can be assigned to logical variables of any size without affecting the value of the expression. However, integer and real expressions may not be assigned to logical variables, nor may logical expressions be assigned to integer or real variables.

3.2.3 The BACKSPACE Statement

Syntax BACKSPACE <unit-spec>

Purpose Positions the file connected to the specified unit at the beginning of the preceding record.

Remarks <unit-spec> is a required unit specifier; it must not be an internal unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

1. If there is no preceding record, the file position is not changed.
2. If the preceding record is the endfile record, the file is positioned before the endfile record.
3. If the file position is in the middle of the record, BACKSPACE repositions to the start of that record.
4. If the file is a binary file, the BACKSPACE repositions to the preceding byte.

Examples BACKSPACE 5

BACKSPACE LUNIT

3.2.4 The CALL Statement

Syntax CALL <sname> [([<arg> [, <arg>]...])]

Purpose Calls and executes a subroutine from another program unit.

Remarks <sname> is the name of the subroutine to be called.

<arg> is an actual argument, which can be any of the following:

1. an expression
2. a constant (or constant expression)
3. a variable
4. an array element
5. an array
6. a subroutine

7. an external function
8. an intrinsic function permitted to be passed as an argument

The actual arguments in the CALL statement must agree with the corresponding formal arguments in the SUBROUTINE statement, in order, in number, and in type or kind.

The compiler will check for correspondence if the formal arguments are known. To be known, the SUBROUTINE statement that defines the formal arguments must precede the CALL statement in the current compilation.

In addition, if the arguments are integer or logical values, agreement in size is required, according to the following rules:

1. If the formal argument is unknown, its size is determined by the $STORAGE metacommand (except as noted in rule 5 of this list). If $STORAGE is not specified, the default is $STORAGE:4.
2. If the actual argument is a constant (or constant expression), and the size of the actual argument is smaller than the size of the formal argument, a temporary variable the size of the constant will be created for the actual argument. If the actual argument is larger, an error is generated:

   95 argument type conflict.
3. If the actual argument is an expression and the size of the actual argument is smaller than the size of the formal argument, then a temporary variable the size of the formal argument is created for the actual argument. If the actual argument is larger, the same error is generated as in rule 2.
4. If the actual argument is an array or a function, or if the actual argument is an array element and the formal argument an array, the compiler will not check for agreement in size.
5. If the actual argument is a variable or an array element and the formal argument is unknown, the size of the formal argument is assumed to be the same size as the size of the actual argument.

Thus, you can call separately compiled subroutines whose formal arguments differ from the size determined by the $STORAGE metacommand in effect when the CALL is compiled. However, agreement in size is still required, and it is your responsibility to ensure this agreement.

If the formal argument is known, then an actual argument that is a variable or an array element is treated as an expression; that is, a temporary variable for the actual argument is created if the actual argument is smaller than the formal argument. Otherwise, the same error occurs as in rule 2.

If the SUBROUTINE statement has no formal arguments, then a CALL statement referencing that subroutine must not have any actual arguments. However, a pair of parentheses may follow the subroutine name.

Execution of a CALL statement proceeds as follows:

1. All arguments that are expressions are evaluated.
2. All actual arguments are associated with their corresponding formal arguments, and the body of the specified subroutine is executed.
3. 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 can be called from any program unit. However, FORTRAN does not permit recursive subroutine calls. That is, a subroutine cannot call itself directly, nor can it call another subroutine that results in that subroutine being called again before it returns control to its caller.

Example

```
C   EXAMPLE OF CALL STATEMENT
          IF (IERR .NE. Ø) CALL ERROR(IERR)
          END
C
          SUBROUTINE ERROR(IERRNO)
          WRITE (*, 2ØØ) IERRNO
2ØØ       FORMAT(1X, 'ERROR', I5, 'DETECTED')
          END
```

3.2.5 The CLOSE Statement

Syntax CLOSE (<unit-spec> [, STATUS = '<status>'])

Purpose Disconnects the unit specified and prevents subsequent I/O from being directed to that unit (unless the same unit number is reopened, possibly associated with a different file or device). The file is discarded if the statement includes STATUS = 'DELETE'.

Remarks <unit-spec> is a required unit specifier. It must appear as the first argument; it must not be an internal unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

<status> is an optional argument and may be either KEEP or DELETE. This option is a character constant and must be enclosed in single quotation marks.

If <status> is not specified, the default is KEEP, except for files opened as scratch files, which have DELETE as the default. Scratch files are always deleted upon normal program termination, and specifying STATUS = 'KEEP' for scratch or temporary files has no effect.

CLOSE for unit zero has no effect, since the CLOSE operation is not meaningful for the keyboard and screen. Opened files do not have to be explicitly closed. Normal termination of an MS-FORTRAN program will close each file with its default status.

Example This deletes an existing file:

```
C   CLOSE THE FILE OPENED IN OPEN EXAMPLE,
C   DISCARDING THE FILE.
          CLOSE(7,STATUS = 'DELETE')
```

3.2.6 The COMMON Statement

Syntax COMMON [/[<cname>]/] <nlist>
[[,] /[<cname>]/ <nlist>]...

Purpose Provides for sharing memory between two or more program units. Such program units can manipulate the same datum without passing it as an argument.

Remarks

<cname> is a common block name. If a <cname> is omitted, then the blank common block is assumed.

<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.

In each COMMON statement, all variables and arrays appearing in each <nlist> following a common block name are declared to be in that common block. Omitting the first <cname> specifies that all elements in the first <nlist> are 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 <nlist>s for the same common block are allocated in that common memory area, in the order they appear in the COMMON statement(s).

The current implementation of MS-FORTRAN restricts the occurrence of noncharacter variables to even byte addresses, which may affect the association of character and noncharacter variables within a COMMON. Because of the order requirement, the compiler cannot adjust the position of variables within a COMMON to comply with the even address restriction. The compiler will generate an error message for those associations which result in a conflict.

The length of a common block is equal to the number of bytes of memory required to hold all elements in that common block. If several distinct program units refer to the same named common block, the common block must be the same length in each program unit. Blank common blocks, however, can have different lengths in different program units. The length of the blank common block is the maximum length.

Example

```
C   EXAMPLE OF BLANK AND NAMED COMMON BLOCKS
          PROGRAM MYPROG
          COMMON I, J, X, K(1Ø)
          COMMON /MYCOM/ A(3)
            .
            .
          END
          SUBROUTINE MYSUB
          COMMON I, J, X, K(1Ø)
          COMMON /MYCOM/ A(3)
            .
            .
          END
```

3.2.7 The CONTINUE Statement

Syntax CONTINUE

Purpose Execution has no effect on the program

Remarks The CONTINUE statement is used primarily as a convenient point for placing a statement label, particularly as the terminal statement in a DO loop.

Example

```
C   EXAMPLE OF CONTINUE STATEMENT
          DO 1Ø, I = 1, 1Ø
              IARRAY(I) = Ø
1Ø            CONTINUE
```

3.2.8 The DATA Statement

Syntax DATA <nlist> /<clist>/ [[,] <nlist> /<clist>/]...

Purpose Assigns initial values to variables.

Remarks A DATA statement is an optional, nonexecutable statement. If present, it must appear after all specification statements and prior to any statement function statements or executable statements.

<nlist> is a list of variables, array elements, or array names.

<clist> is a list of constants, or a constant 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 a list of all constants having the specified value and repeated as often as specified by 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 equivalent to a list of all elements in that array in memory sequence order. Array elements must be indexed only by constant subscripts.

The type of each noncharacter element in a <clist> must be the same as the type of the corresponding variable or array element in the accompanying <nlist>. However, with the $NOTSTRICT metacommand in effect, a character element in a <clist> can correspond to a variable of any type.

The character element 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. 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.

Examples

```
INTEGER N, ORDER, ALPHA

REAL COEF(4), EPS(2)

DATA N /0/, ORDER /3/

DATA ALPHA /'A'/

DATA COEF /1.0,2*3.0,1.0/, EPS(1) /.00001/
```

3.2.9 The DIMENSION Statement

Syntax DIMENSION <array> (<dim>) [, <array> (<dim>)]...

Purpose Specifies that a user name is an array and defines the number of its elements.

Remarks <array> is the name of an array.

<dim> specifies the dimensions of the array and is a list of one to seven dimension declarators separated by commas.

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is three. The maximum size of an array is 65,366 bytes. The maximum size of an dimension is 32,767 bytes.

A dimension declarator can be:

1. an unsigned integer constant
2. a user name corresponding to a nonarray integer formal argument
3. an asterisk

A dimension declarator specifies the upper bound of the dimension. The lower bound 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 formal 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-size array.

If the dimension declarator is an asterisk, the array is an assumed-size array and the upper bound of that dimension is not specified.

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

Array elements are stored in column-major order; the leftmost subscript changes most rapidly as the array is mapped into contiguous memory addresses.

For example, the following statements

```
INTEGER*2 A (2, 3)
DATA A /1, 2, 3, 4, 5, 6/
```

would result in the following mapping (assuming A is placed at location 1000 in memory):

Array Element	Address	Value
A (1, 1)	1000	1
A (2, 1)	1002	2
A (1, 2)	1004	3
A (2, 2)	1006	4
A (1, 3)	1008	5
A (1, 3)	100A	6

Example

```
DIMENSION A (2,3), V (10)
CALL SUBR (A,2,V)
  .
  .
SUBROUTINE SUBR (MATRIX, ROWS, VECTOR)
REAL MATRIX, VECTOR
INTEGER ROWS
DIMENSION MATRIX (ROWS,*), VECTOR (10),
+ LOCAL (2,4,8)
MATRIX (1,1) = VECTOR (5)
  .
  .
END
```

3.2.10 The DO Statement

Syntax DO <slabel> [,] <variable> = <expr1>, <expr2> [, <expr3>]

Purpose Repeatedly evaluates the statements following the DO, through and including the statement with the label <slabel>.

Remarks <slabel> is the statement label of an executable statement.

<variable> is an integer variable.

<expr1>, <expr2>, <expr3> are integer expressions.

The label referred to must appear after the DO statement and be contained in the same program unit. The specified statement is called the terminal statement of the DO loop and 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.

The range of a DO loop begins with the statement that follows the DO statement and includes the terminal statement of the DO loop.

The following restrictions affect the execution of a DO statement:

1. 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.

2. If a DO statement appears within an IF, ELSEIF, or ELSE block, the range of the associated DO loop must be entirely contained in the particular block.

3. 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 may not be modified in any way by the statements within the range of the DO loop associated with it. Jumping into the range of a DO loop from outside its range is not permitted. (However, a special feature, added for compatibility with earlier versions of FORTRAN, does permit "extended range" DO loops. See Section 6.2.2, "The $DO66 Metacommand," for more information.)

The execution of a DO statement sets the following process in motion:

In some circumstances, the value of a DO variable may overflow as a result of an increment that is performed prior to testing it against the upper bound. If this happens, your program is technically in error, but the error is not detected as such by either the compiler or the runtime library. However, if the DO variable is either explicitly or implicitly defined as INTEGER*2, the arithmetic for the statement will be carried out in 32-bit mode with the necessary conversions. If the DO variable is INTEGER*4, and an overflow occurs, the value will wrap around, and the loop will not terminate.

For example:

```
      INTEGER*2 I
      DO 100 I = 32760,32767
        .
        .
 100  CONTINUE
```

1. The expressions <expr1>, <expr2>, and <expr3> are evaluated. If <expr3> is not present, it is assumed that <expr3> evaluated to one.

2. The DO variable is set to the value of the expression, <expr1>.

3. The iteration count for the loop is:

   MAX0(((expr2-expr1 + expr3)/expr3),0)

   The iteration count may be zero if either of the following is true:

   a. <expr1> is greater than <expr2> and <expr3> is greater than zero

   b. <expr1> is less than <expr2> and <expr3> is less than zero

   However, if the $DO66 metacommand is in effect, the iteration count is at least one. See Section 6.2.2, "The $DO66 Metacommand," for more information about this feature.

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, these steps take place:

1. The value of the DO variable is incremented by the value of <expr3> that 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 because the iteration count becomes zero, or because of a transfer of control out of the DO loop.

Example The following shows the final value of a DO variable:

```
C   EXAMPLE OF DO STATEMENTS

C   DISPLAY THE NUMBERS 1 TO 11 ON THE SCREEN
          DO 200 I = 1,10
200       WRITE(*,'(I5)')I
          WRITE(*,'(I5)')I

C   INITIALIZE A 20-ELEMENT REAL ARRAY
          DIMENSION ARRAY(20)
          DO 1 I = 1, 20
1         ARRAY(I) = 0.0

C   PERFORM A FUNCTION 11 TIMES
          DO 2, I = -30, -60, -3
          J = I/3
          J = -9 - J
          ARRAY(J) = MYFUNC(I)
2         CONTINUE
```

3.2.11 The ELSE Statement

Syntax ELSE

Purpose Marks the beginning of an ELSE block. Execution of the statement itself has no effect on the program.

Remarks The associated ELSE block consists of all of the executable statements (possibly none) that follow the ELSE statement, up to but not including the next ENDIF statement at 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. (See Section 3.2.25, "The IF THEN ELSE Statement," for a discussion of IF-levels.)

Transfer of control into an ELSE block from outside that block is not permitted.

Example

```
CHARACTER C
  .
  .
READ (*,'(A)') C
IF (C .EQ. 'A') THEN
    CALL ASUB
ELSE
    CALL OTHER
ENDIF
  .
  .
```

3.2.12 The ELSEIF Statement

Syntax ELSEIF (<expression>) THEN

Purpose Causes evaluation of the expression.

Remarks <expression> is a logical 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.

The associated ELSEIF block consists of all the executable statements (possibly none) that follow, up to the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement.

Following the execution of the last statement in the ELSEIF block, the next statement to be executed is 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. (See Section 3.2.25, "The IF THEN ELSE Statement," for a discussion of IF-levels.)

Transfer of control into an ELSEIF block from outside that block is not permitted.

Example

```
CHARACTER C
   .
   .
READ (*,'(A)') C
IF (C .EQ. 'A') THEN
    CALL ASUB
ELSEIF (C .EQ. 'X') THEN
    CALL XSUB
ELSE
    CALL OTHER
ENDIF
```

3.2.13 The END Statement

Syntax END

Purpose In a subprogram, has the same effect as a RETURN statement; in the main program, terminates execution of the program.

Remarks The END statement must appear as the last statement in every program unit. Unlike other statements, an END statement must appear alone on an initial line, with no label. No continuation lines may follow the END statement. No other FORTRAN statement, such as the ENDIF statement, may have an initial line that appears to be an END statement.

Example

```
C   EXAMPLE OF END STATEMENT
C   END STATEMENT MUST BE LAST STATEMENT
C   IN A PROGRAM
          PROGRAM MYPROG
          WRITE(*, '(10H HI WORLD!)')
          END
```

3.2.14 The ENDFILE Statement

Syntax ENDFILE <unit-spec>

Purpose Writes an end-of-file record as the next record of the file connected to the specified unit.

Remarks <unit-spec> is a required external unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

After writing the end-of-file record, ENDFILE positions the file after the end-of-file record. This prohibits further sequential data transfer until after execution of either a BACKSPACE or REWIND statement.

An ENDFILE on a direct access file makes all records written beyond the position of the new end-of-file disappear.

Example

```
.
.
WRITE (6,*) X
ENDFILE 6
REWIND 6
READ (6,*) Y
.
.
```

3.2.15 The ENDIF Statement

Syntax ENDIF

Purpose Terminates a block IF statement. Execution of an ENDIF statement itself has no effect on the program.

Remarks There must be a matching ENDIF statement for every block IF statement in a program unit, to identify which statements belong to a particular block IF statement. See Section 3.2.25, "The IF THEN ELSE Statement," for discussion and examples of block IFs.

Example

```
IF (I .LT. Ø) THEN
    X = -I
    Y = -I
ENDIF
```

3.2.16 The EQUIVALENCE Statement

Syntax EQUIVALENCE (<nlist>) [, (<nlist>)]...

Purpose Specifies that two or more variables or arrays are to share the same memory.

Remarks <nlist> is a list of at least two elements, separated by commas. An <nlist> may include variable names, array names, or array element names; argument names are not allowed. Subscripts must be integer constants and must be within the bounds of the array they index. No automatic type conversion occurs if the shared elements are of different types.

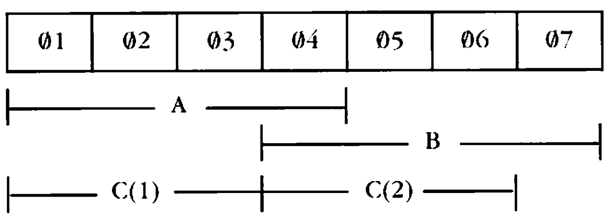
An EQUIVALENCE statement specifies that the memory sequences of the elements that appear in the list <nlist> must have the same first memory location. Two or more variables are said to be associated if they refer to the same actual memory. Thus, an EQUIVALENCE statement causes its list of variables to become associated. An array name, if present in an EQUIVALENCE statement, refers to the first element of the array.

You cannot associate character and noncharacter entities when the $STRICT metacommand is in effect ($NOTSTRICT is the default). See the odd-byte boundary restriction described in number 3 in the following list.

Associated character entities may overlap, as in the following example:

```
CHARACTER A*4, B*4, C(2)*3
EQUIVALENCE (A,C(1)), (B,C(2))
```

The preceding example can be graphically illustrated as follows:



Restrictions:

1. You cannot force a variable to occupy more than one distinct memory location; nor can you force two or more elements of the same array to occupy the same memory location. For example, the following statement would force R to occupy two distinct memory locations or S(1) and S(2) to occupy the same memory location:

```
C   THIS IS AN ERROR
        REAL R,S(10)
        EQUIVALENCE (R,S(1)),(R,S(5))
```

2. An EQUIVALENCE statement cannot specify that consecutive array elements not be stored in sequential order. The following, for example, is not permitted:

```
C   THIS IS ANOTHER ERROR
        REAL R(10),S(10)
        EQUIVALENCE (R(1),S(1)),  +(R(5),S(6))
```

3. You cannot equivalence character and noncharacter entities so that the noncharacter entities can start on an odd-byte boundary.

   For entities not in a common block, the compiler will attempt to align the noncharacter entities on word boundaries. An error message will be issued if such an alignment is not possible because of multiple equivalencing. For example, the following would result in an error, since it is not possible for both variables A and B to be word aligned:

```
    CHARACTER*1 C1(10)
    REAL A,B
    EQUIVALENCE (A,C1(1))
    EQUIVALENCE (B,C1(2))
```

   For entities in a common block, since positions are fixed, it is your responsibility to assure word alignment for the noncharacter entities. An error message will be issued for any that are not word aligned.

4. An EQUIVALENCE statement cannot associate an element of type CHARACTER with a noncharacter

element in a way that causes the noncharacter element to be allocated on an odd byte boundary. However, there are no boundary restrictions for equivalencing of character variables.

5. When EQUIVALENCE statements and COMMON statements are used together, several additional restrictions apply:

   a. An EQUIVALENCE statement cannot cause memory in two different common blocks to be shared.

   b. An EQUIVALENCE statement can extend a common block by adding memory elements following the common block, but not preceding the common block.

   c. Extending a named common block with an EQUIVALENCE statement must not make its length different from the length of the named common block in other program units.

For example, the following is not permitted because it extends the common block by adding memory preceding the start of the block:

```
C   THIS IS A MORE SUBTLE ERROR
      COMMON /ABCDE/ R(10)
      REAL S(10)
      EQUIVALENCE (R(1),S(10))
```

Example

```
C   CORRECT USE OF EQUIVALENCE STATEMENT
      CHARACTER NAME, FIRST, MIDDLE, LAST
      DIMENSION NAME(60),   FIRST(20),
     1              MIDDLE(20), LAST(20)
      EQUIVALENCE (NAME(1), FIRST(1)),
     1              (NAME(21),MIDDLE(1),
     2              (NAME(41), LAST(1))
```

3.2.17 The EXTERNAL Statement

Syntax EXTERNAL <name> [, <name>]. . .

Purpose Identifies a user-defined name as an external subroutine or function.

Remarks <name> is the name of an external subroutine or function.

Giving a name in an EXTERNAL statement declares it as an external procedure. Statement function names cannot appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement, 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 in any given program unit.

In assembly language and MS-Pascal, EXTERN means that an object is defined outside the current compilation or assembly unit. This is unnecessary in MS-FORTRAN since standard FORTRAN practice assumes that any object referred to but not defined in a compilation unit is defined externally.

In FORTRAN, therefore, EXTERNAL is used primarily to specify that a particular user-defined name is a subroutine or function to be used as a procedural parameter. EXTERNAL may also indicate that a user-defined function is to replace an intrinsic function of the same name.

Examples

```
C   EXAMPLE OF EXTERNAL STATEMENT
          EXTERNAL MYFUNC, MYSUB
C   MYFUNC AND MYSUB ARE PARAMETERS TO CALC
          CALL CALC (MYFUNC, MYSUB)

C   EXAMPLE OF A USER-DEFINED FUNCTION
C   REPLACING AN INTRINSIC EXTERNAL SIN
          X = SIN (A,4.2,37)
```

3.2.18 THE FORMAT Statement

Syntax FORMAT <format-spec>.

Purpose Used in conjunction with formatted I/O statements, provides information that directs the editing of data.

Remarks

<format-spec> is a format specification, which provides explicit editing information. The format specification must be enclosed in parentheses and may take one of the following forms:

[<r>] <repeatable edit descriptor>

<nonrepeatable edit descriptor>

[<r>] <format specification>

The <r>, if present, is a nonzero, unsigned, integer constant called a repeat specification.

Up to three levels of nested parentheses are permitted within the outermost level of parentheses.

Edit descriptors, both repeatable and nonrepeatable, are listed in Table 3.6 and described in more detail in Section 4.4.2, "Edit Descriptors."

You may omit the comma between two list items if the resulting format specification is not ambiguous; for example, after a P edit descriptor or before or after the slash (/) edit descriptor.

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

Table 3.6. Edit Descriptors

Repeatable	Nonrepeatable
I<w>	'xxx' (character constants)
G<w>.<d>	<n>Hxxx (character constants)
G<w>.<d>E<e>	<n>X (positional editing)
F<w>.<d>	/ (terminate record)
E<w>.<d>	\ (don't terminate record)
E<w>.<d>E<e>	<k>P (scale factor)
D<w>.<d>	BN (blanks as blanks)
L<w>	BZ (blanks as zeros)
A[<w>]	

Notes for Table 3.3:

1. For the repeatable edit descriptors:

   A, D, E, F, G, I, and L indicate the manner of editing.

   <w> and <e> are nonzero, unsigned, integer constants.

   <d> is an unsigned integer constant.

2. For the nonrepeatable edit descriptors:

   ', H, X, /, \, P, BN, and BZ indicate the manner of editing.

   x is any ASCII character.

   <n> is a nonzero, unsigned, integer constant.

   <k> is an optionally signed integer constant.

See Section 4.4, "Formatted I/O," for further information on edit descriptors and formatted I/O.

3.2.19 The FUNCTION Statement (External)

Syntax [<type>] FUNCTION <fname> ([<farg> [, <farg>]...])

Purpose Identifies a program unit as a function and supplies its type, name, and optional formal parameter(s).

Remarks <type> is one of the following:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
LOGICAL*2
LOGICAL*4

<fname> is the user-defined name of the function.

<farg> is a formal argument name.

The function name is global, but it is also local to the function it names. If <type> is omitted from the FUNCTION statement, the function's type is determined by default and by any subsequent IMPLICIT or type statements that would determine the type of an ordinary variable. If <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, EXTERNAL, type, or DIMENSION statements, the type of arguments to that function. Neither argument names nor the function name can appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The function name must appear as a variable in the program unit that defines 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, 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

A function can be called from any program unit. However, FORTRAN does not allow recursive function calls, which means that a function cannot call itself directly, nor can it call another function if such a call results in that function being called again before it returns control to its caller. However, recursive calls are not detected by the compiler, even if they are direct.

Example

```
C   EXAMPLE OF A FUNCTION REFERENCE
C   GETNO IS A FUNCTION THAT READS A
C   NUMBER FROM A FILE
          I = 2
10        IF (GETNO(I) .EQ. 0.0) GO TO 10
          STOP
          END
C
          FUNCTION GETNO(NOUNIT)
          READ(NOUNIT, '(F10.5)') R
          GETNO = R
          RETURN
          END
```

3.2.2Ø The GOTO Statement (Assigned GOTO)

Syntax GOTO <name> [[,] (<slabel>
[, <slabel>]...)]

Purpose Causes the statement labeled by the label last assigned to <name> to be the next statement executed.

Remarks <name> is an integer variable name.

<slabel> is a statement label of an executable statement 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, <name> must have been assigned the label of an executable statement found in the same program unit as the assigned GOTO statement.

Including the optional list of labels and selecting the $DEBUG metacommand results in a runtime error if the label last assigned to <name> is not among those listed. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted.

A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.2, "The $DO66 Metacommand," for more information about this feature.

Example

```
C   EXAMPLE OF ASSIGNED GOTO
          ASSIGN 1Ø TO I
          GOTO I
1Ø        CONTINUE
```

3.2.21 The GOTO Statement (Computed GOTO)

Syntax GOTO (<slabel> [, <slabel>]..) [,] <i>

Purpose Transfers control to the statement labeled by the <i>th label in the list.

Remarks <slabel> is the statement label of an executable statement from the same program unit as the computed GOTO statement. The same statement label may be repeated in the list of labels.

<i> is an integer expression.

If there are n labels in the list of labels and <i> is out of range, the computed GOTO statement serves as a CONTINUE statement. <i> would be out of range in either of the following cases:

<i> < 1

<i> > n

Otherwise, the next statement executed is the one labeled by the <i>th label in the list of labels.

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.2, "The $DO66 Metacommand," for more information.

Example

```
C   EXAMPLE OF COMPUTED GOTO
          I = 1
          GOTO (10, 20) I
            .
            .
10        CONTINUE
            .
            .
20        CONTINUE
```

3.2.22 The GOTO Statement (Unconditional GOTO)

Syntax GOTO <slabel>.

Purpose Transfers control to the statement labeled <slabel>.

Remarks <slabel> is the statement label of an executable statement in the same program unit as the GOTO statement.

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.2, "The $DO66 Metacommand," for more information about this feature.

Example

```
C   EXAMPLE OF UNCONDITIONAL GOTO
          GOTO 4022
            .
            .
4022      CONTINUE
```

3.2.23 The IF Statement (Arithmetic IF)

Syntax IF (<expression>) <slabel1>, <slabel2>, <slabel3>

Purpose Evaluates the expression and transfers control to the statement labeled by one of the specified labels, according to the result of the expression.

Remarks <expression> is an integer or real expression.

<slabel1>, <slabel2>,and <slabel3> are statement labels of executable statements in the same program unit as the arithmetic IF statement.

The same statement label may appear more than once among the three labels. The first label is selected if the value of the expression is less than zero, the second label if the value equals zero, and the third label if the value is greater than zero. The next statement executed is the statement labeled by the selected label.

Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. A special feature, extended range DO loops, does permit jumping into a DO block. See Section 6.2.2, "The $DO66 Metacommand," for more information about this feature.

Example

```
C   EXAMPLE OF ARITHMETIC IF
          I = 0
          IF (I) 10, 20, 30
10        CONTINUE
           .
           .
20        CONTINUE
           .
           .
30        CONTINUE
```

3.2.24 The IF Statement (Logical IF)

Syntax IF (<expression>) <statement>

Purpose Evaluates the logical expression and, if the value of that expression is .TRUE., executes the statement given. If the expression evaluates to .FALSE., the statement is not executed and execution continues as if a CONTINUE statement were encountered.

Remarks <expression> is a logical expression.

<statement> is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement.

Example

```
C   EXAMPLE OF LOGICAL IF
          IF (I .EQ. 0) J = 2
          IF (X .GT. 2.3) GOTO 100
            .
            .
100       CONTINUE
```

3.2.25 The IF THEN ELSE Statement (Block IF)

Syntax IF (<expression>) THEN

Purpose Evaluates the expression and, if the expression evaluates to .TRUE., begins executing statements in the IF block. If the expression evaluates to .FALSE., control transfers to the next ELSE, ELSEIF, or ENDIF statement at the same IF-level.

Remarks <expression> is a logical expression.

The associated IF block consists of all the executable statements (possibly none) that appear following the statement, up to but not including the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this block IF statement.

After execution of the last statement in the IF block, the next statement executed is 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 at the same IF-level as the block IF statement.

Transfer of control into an IF block from outside that block is not permitted.

IF-Levels:

The concept of an IF-level in block IF and associated statements is described as follows. For any statement, its IF-level is n1 minus n2, where:

1. n1 is the number of block IF statements from the beginning of the program unit in which the statement occurs, up to and including that statement.
2. n2 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 zero and the IF-level of every block IF, ELSEIF, ELSE, and ENDIF must be greater than zero. Finally, the IF-level of every END statement must be zero. The IF-level defines the nesting rules for the block IF and associated statements and defines the extent of IF, ELSEIF, and ELSE blocks.

Example 1

Simple block IF that skips a group of statements if the expression is false:

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

Example 2

Block IF with 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.100
ELSE
   .     Some statements executed
   .     only if J.LE.10
ENDIF
```

Example 3 Nesting of constructs and use of an ELSE statement following a block IF without intervening ELSEIF statements:

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

3.2.26 The IMPLICIT Statement

Syntax IMPLICIT <type> (<a> [, <a>]...) [<type> (<a> [, <a>]...)...]

Purpose Defines the default type for user-declared names.

Remarks <type> is one of the following types:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
LOGICAL*2
LOGICAL*4
CHARACTER

<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. The letters for a range must be in alphabetical order.

An IMPLICIT statement defines the type and size for all user-defined names that begin with the letter or letters given. An IMPLICIT statement applies only to the program unit in which it appears and does not change the type of any intrinsic function.

IMPLICIT types for any specific user name can be overridden or confirmed if that name is given 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 length is also overridden by a later type definition.

A program unit can have more than one IMPLICIT statement. However, 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.

Example

```
C   EXAMPLE OF IMPLICIT STATEMENT
          IMPLICIT INTEGER (A - B)
          IMPLICIT CHARACTER*10 (N)
          AGE = 10
          NAME = 'PAUL'
```

3.2.27 The INTRINSIC Statement

Syntax INTRINSIC <name1> [, <name2>]..

Purpose Declares that a name is an intrinsic function.

Remarks <name> is an intrinsic function name.

Each user name may appear only once in an INTRINSIC statement. A name that appears in an INTRINSIC statement 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 5.1 in Chapter 5, "Programs, Subroutines, and Functions."

You must specify the name of an intrinsic function in an INTRINSIC statement if you wish to pass it as a parameter (i.e., as an actual argument to a program unit).

Example

```
C   EXAMPLE OF INTRINSIC STATEMENT
          INTRINSIC SIN, COS
C   SIN AND COS ARE PARAMETERS TO CALC2
          X = CALC2 (SIN, COS).
```

3.2.28 The OPEN Statement

Syntax

OPEN (<unit-spec> [, FILE = '<fname>']
[, STATUS = '<status>'] [, ACCESS = '<access>']
[, FORM = '<format>'] [, RECL = <rec-length>])

Purpose

Associates a unit number with an external device or file on an external device.

Remarks

<unit-spec> is a required unit specifier. It must appear as the first argument; it must not be an internal unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

<fname> is a character expression. This optional argument, if present, must appear as the second argument. If the argument is omitted, the compiler creates a temporary scratch file with a name unique to the unit. The scratch file is deleted when it is either explicitly closed or the program terminates normally.

If the filename specified is blank (FILE = ' '), the user will be prompted for a filename at runtime. If opened with STATUS = 'OLD', the file itself must exist.

All arguments after <fname> are optional and can appear in any order. Except for RECL = , these options are character constants with optional trailing blanks and must be enclosed in single quotation marks.

<status> is OLD (the default) or NEW. OLD is for reading or writing existing files; NEW is for writing new files.

<access> is SEQUENTIAL (the default) or DIRECT.

<format> is FORMATTED, UNFORMATTED, or BINARY. If access is SEQUENTIAL, the default is FORMATTED; if access is DIRECT, the default is UNFORMATTED.

<rec-length> (record length) is an integer expression that specifies the length of each record in bytes. This argument

is applicable only for DIRECT access files, for which it is required.

Associating unit zero to a file has no effect: Unit zero is permanently connected to the keyboard and screen.

Example 1

```
C  PROMPT USER FOR A FILE NAME.
        WRITE(*,'(A\)')'Output file name? '
C  PRESUME THAT FNAME IS SPECIFIED TO BE
C  CHARACTER*64.
C  READ THE FILE NAME FROM THE KEYBOARD.
        READ(*,'(A)') FNAME
C  OPEN THE FILE AS FORMATTED SEQUENTIAL
C  AS UNIT 7.
C  NOTE THAT THE ACCESS SPECIFIED WAS
C  UNNECESSARY SINCE IT IS THE DEFAULT.
C  FORMATTED IS ALSO THE DEFAULT.
        OPEN(7,FILE = FNAME,ACCESS = 'SEQUENTIAL',
       + STATUS = 'NEW').
```

Example 2

```
C  OPEN AN EXISTING FILE CREATED BY EDITOR
C  CALLED DATA3.TXT AS UNIT 3.
        OPEN(3,FILE = 'DATA3.TXT')
```

3.2.29 The PAUSE Statement

Syntax PAUSE [<n>]

Purpose Suspends program execution until the RETURN key is pressed.

Remarks <n> is either a character constant or a string of not more than five digits.

The PAUSE statement suspends execution of the program, pending an indication that it is to continue. The argument <n>, if present, is displayed on the screen as a prompt requesting input from the keyboard. If <n> is not present, the following message is displayed on the screen:

PAUSE. Please press <return> to continue.

After you press the RETURN key, program execution resumes as if a CONTINUE statement were executed.

Example

```
C   EXAMPLE OF A PAUSE STATEMENT
          IF (IWARN .EQ. 0) GOTO 300
          PAUSE 'WARNING: IWARN IS NONZERO'
300       CONTINUE
```

3.2.30 The PROGRAM Statement

Syntax PROGRAM <program-name>.

Purpose Identifies the program unit as a main program and gives it a name.

Remarks <program-name> is the name you have given to your main program. The program name is a global name. Therefore, it cannot be the same as that of another external procedure or common block. (It is also a local name to the main program and must not conflict with any local name in the main program.) The PROGRAM statement may only appear as the first statement of a main program.

If the main program does not have a program statement, it will be assigned the name MAIN. The name MAIN then cannot be used to name any other entity.

Example

```
PROGRAM GAUSS
REAL COEF (10,10), CONST (10)
   .
   .
END
```

3.2.31 The READ Statement

Syntax READ (<unit-spec> [, <format-spec>] [, REC = <rec-num>] [, END = <slabel1>] [, ERR = <slabel2>]) <iolist>

Purpose Transfers data from the file associated with <unit-spec> to the items in the <iolist>, assuming that no end-of-file or error occurs.

Remarks If the read is internal, the character variable or character array element specified is the source of the input; if the read is not internal, the source of the input is the external unit.

<unit-spec> is a required unit specifier, which must appear as the first argument.

<format-spec> is a format specifier. It is required for formatted read as the second argument; it must not appear for unformatted read.

Other arguments, if present, can appear in any order

<rec-num> is a record number, specified for direct access files only; if <rec-num> is given for other than direct files, an error results. The record number is a positive integer expression and positions to the record number <rec-num> (the first record in the file has record number 1) before the transfer of data begins. If this argument is omitted for a direct access file, reading continues sequentially from the current position in the file.

<slabel1> is an optional statement label in the same program unit as the READ statement. If this argument is omitted, reading past the end of the file results in a runtime error. If it is present, encountering an end-of-file condition transfers control to the executable statement specified.

<slabel2> is an optional statement label in the same program unit as the READ statement. If this argument is omitted, I/O errors result in runtime errors. If it is present, I/O errors transfer control to the executable statement specified.

<iolist> specifies the entities into which values are transferred from the file. An <iolist> may be empty, but ordinarily consists of input entities and implied DO lists, separated by commas.

See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

If the file has not been opened by an OPEN statement, an implicit OPEN operation is performed. This operation is equivalent to the following statement:

OPEN (<unit-spec>,FILE = ' ',STATUS = 'OLD',
ACCESS = 'SEQUENTIAL',FORM = '<format>')

<format> is FORMATTED if the READ statement is formatted and UNFORMATTED if the READ statement is unformatted. See Section 3.2.28, "The OPEN Statement," for a description of the effect of the FILE = parameter.

Example

```
C  SET UP A TWO DIMENSIONAL ARRAY.
         DIMENSION IA(10,20)
C  READ IN THE BOUNDS FOR THE ARRAY.
C  THESE BOUNDS SHOULD BE LESS THAN OR
C  EQUAL TO 10 AND 20 RESPECTIVELY.
C  THEN READ IN THE ARRAY IN NESTED
C  IMPLIED DO LISTS WITH INPUT FORMAT OF
C  8 COLUMNS OF WIDTH 5 EACH.
         READ (3,990)IL,JL,((IA(I,J),J = 1,JL), I = 1,IL)
990      FORMAT(2I5/,(8I5))
```

3.2.32 The RETURN Statement

Syntax RETURN

Purpose Returns control to the calling program unit.

Remarks RETURN can only appear in a function or subroutine.

Execution of a RETURN statement terminates execution of the enclosing subroutine or function. If the RETURN statement is in a function, the function's value 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 equivalent to execution of a RETURN statement. Thus, either a RETURN or an END statement, but not both, is required to terminate a function or subroutine.

Example

```
C  EXAMPLE OF RETURN STATEMENT
C  THIS SUBROUTINE LOOPS UNTIL
C  YOU TYPE "Y"
         SUBROUTINE LOOP
         CHARACTER IN
C
10       READ(*,'(A1)') IN
         IF (IN .EQ. 'Y') RETURN
         GOTO 10
C  RETURN IMPLIED
         END
```

3.2.33 The REWIND Statement

Syntax REWIND <unit-spec>

Purpose Repositions to its initial point the file associated with the specified unit.

Remarks <unit-spec> is a required external unit specifier. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

Example

```
INTEGER A(8Ø)
   .
   .
WRITE (7,'(8ØI1)') A
   .
   .
REWIND 7
   .
   .
READ (7, '(8ØI1)') A
```

3.2.34 The SAVE Statement

Syntax SAVE /<cname1>/ [, /<cname2>/]...

Purpose Causes variables to retain their values across invocations of the procedure in which they are defined.

Remarks <cname> is the name of a common block. After being saved, variables in the common block have defined values if the current procedure is subsequently re-entered.

Since in the current implementation, all common blocks and variables are statically allocated, all common blocks and variables are saved automatically. In practice, therefore, the SAVE statement has no effect.

Example

```
C   EXAMPLE OF SAVE STATEMENT
          SAVE /MYCOM/
```

3.2.35 The Statement Function Statement

Syntax <fname> ([<farg> [, <farg>]..]) = <expr>

Purpose Defines a function in one statement.

Remarks <fname> is the name of the statement function.

<farg> is a formal argument name.

<expr> is any expression.

The statement function statement is similar in form to the 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. Like any other function, a statement function is executed by a function reference in an expression.

The type of the expression 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 can be re-used as other user-defined names in the rest of the program unit enclosing the statement function definition.

The name of the statement function, however, is local to the enclosing program unit; it 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. In the latter case the type of all such uses 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 permitted. Statement function references, however, must

refer to statement functions defined prior to the statement function in which they appear. Statement functions cannot be called recursively, 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.

Example

```
C   EXAMPLE OF STATEMENT FUNCTION STATEMENT
          DIMENSION X(10)
          ADD(A, B) = A + B
C
          DO 1, I = 1, 10
          X(I) = ADD(Y, Z)
1         CONTINUE
```

3.2.36 The STOP Statement

Syntax STOP [<n>]

Purpose Terminates the program.

Remarks <n> is either a character constant or a string of not more than five digits.

The argument, <n>, if present, is displayed on the screen when the program terminates. If <n> is not present, the following message is displayed:

STOP - Program terminated.

Example

```
C   EXAMPLE OF STOP STATEMENT
          IF (IERROR .EQ. 0) GOTO 200
     .    STOP 'ERROR DETECTED'
200       CONTINUE
```

3.2.37 The SUBROUTINE Statement

Syntax SUBROUTINE <subroutine-name> [([<farg> [, <farg>]...])]

Purpose Identifies a program unit as a subroutine, gives it a name, and identifies the formal parameters to that subroutine.

Remarks <subroutine-name> is the user-defined, global, external name of the subroutine.

<farg> is the user-defined name of a formal argument, also known as a dummy argument.

A subroutine begins with a SUBROUTINE statement and ends with the next following END statement. It can contain any kind of statement other than a PROGRAM statement, SUBROUTINE statement, or a FUNCTION statement.

The list of argument names defines the number and, with any subsequent IMPLICIT, EXTERNAL, type, or DIMENSION statements, the type of arguments to that subroutine. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The actual arguments in the CALL statement that reference a subroutine must agree with the corresponding formal arguments in the SUBROUTINE statement, in order, in number, and in type or kind.

The compiler will check for correspondence if the formal arguments are known. To be known, the SUBROUTINE statement that defines the formal arguments must precede the CALL statement in the current compilation. Rules for the correspondence of formal and actual arguments are described in Section 3.2.4, "The CALL Statement."

Example

```
        SUBROUTINE GETNUM (NUM,UNIT)
        INTEGER NUM, UNIT
10      READ (UNIT,'(I10)', ERR = 10) NUM
        RETURN
        END
```

3.2.38 The Type Statement

Syntax <type> <vname1> [, <vname2>]..

Purpose Specifies the type of user-defined names.

Remarks <type> is one of the following data type specifiers:

INTEGER
INTEGER*2
INTEGER*4

REAL
REAL*4
REAL*8
DOUBLE PRECISION
LOGICAL
LOGICAL*2
LOGICAL*4
CHARACTER

<vname> is the symbolic name of a variable, array, or statement function; or a function subprogram; or an array declarator.

A type statement can confirm or override the implicit type of a name. A type statement can 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 can have its type explicitly specified by a type statement only once.

A type statement may also confirm the type of an intrinsic function, but it is not required. The name of a subroutine or main program cannot appear in a type statement.

The following rules apply to a type statement:

1. A type statement must precede all executable statements.
2. The data type of a symbolic name can be declared explicitly only once.
3. A type statement cannot be labeled.
4. A type statement can be used to declare an array by appending a dimension declarator to an array name.

A symbolic name can be followed by a data type length specifier of the form *<length>, where <length> is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies and assigns a new length to the specified item. If both a data type length specifier and an array declarator are included, the data type length specifier goes last.

Example

```
C   EXAMPLE OF TYPE STATEMENTS
          INTEGER COUNT, MATRIX(4,4), SUM
          REAL MAN,IABS
          LOGICAL SWITCH

          INTEGER*2 Q, M12*4, IVEC(10)*4
          REAL*4 WX1, WX3*4, WX5, WX6*4

          CHARACTER NAME*10, CITY*80, CH
```

3.2.39 The WRITE Statement

Syntax WRITE (<unit-spec> [, <format-spec>] [, ERR = <slabel>] [, REC = <rec-num>]) <iolist>

Purpose Transfers data from the <iolist> items to the file associated with the specified unit.

Remarks <unit-spec> is a required unit specifier and must appear as the first argument. See Section 4.3.1, "Elements of I/O Statements," for more information about unit specifiers and other elements of I/O statements.

<format-spec> is a format specifier. It is required as the second argument for a formatted WRITE; it must not appear for an unformatted WRITE.

The remaining arguments, if present, may appear in any order.

<slabel> is an optional statement label. If it is not present, I/O errors result in runtime errors. If it is present, I/O errors transfer control to the executable statement specified.

<rec-num> is a record number, specified for direct access files only (otherwise, an error results). It is a positive integer expression, specifying the number of the record to be written. The first record in the file is record number 1. If the record number is omitted for a direct access file, writing continues from the current position in the file.

<iolist> specifies the entities whose values are transferred by the WRITE statement. An <iolist> may be empty, but ordinarily consists of output entities and implied DO lists, separated by commas.

If the WRITE is internal, the character variable or character array element specified as the unit is the destination of the output; otherwise, the external unit is the destination.

If the file has not been opened by an OPEN statement, an implicit open operation is performed. The OPEN operation is equivalent to the following statement:

OPEN (<unit-spec>, FILE = ' ', STATUS = 'NEW', ACCESS = 'SEQUENTIAL', FORM = <format>)

<format> is FORMATTED for a formatted WRITE statement and UNFORMATTED for an unformatted WRITE statement. See Section 3.2.28, "The OPEN Statement," for a description of the effect of the FILE = argument.

Example

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

Chapter 4

The I/O System

4.1 Records 95
4.2 Files 96
4.2.1 File Properties 96
4.2.1.1 Filename 96
4.2.1.2 File Position 97
4.2.1.3 File Structure 97
4.2.1.4 File Access Method 97
4.2.2 Special Properties of Internal Files 98
4.2.3 Units 99
4.2.4 Commonly Used File Structures 100
4.2.5 Other File Structures 101
4.2.6 OLD and NEW Files 102
4.2.7 Limitations 103
4.3 I/O Statements 105
4.3.1 Elements of I/O Statements 105
4.3.1.1 The Unit Specifier 105
4.3.1.2 Format Specifier 106
4.3.1.3 InputÒutput List 106
4.3.2 Carriage Control 107
4.4 Formatted I/O 109
4.4.1 Interaction Between Format and I/O List 109
4.4.2 Edit Descriptors 111
4.4.2.1 Nonrepeatable Edit Descriptors 111
4.4.2.2 Repeatable Edit Descriptors 114
4.5 List-Directed I/O 120
4.5.1 List-Directed Input 120
4.5.2 List-Directed Output 122

This chapter supplements the presentation of the I/O statements in Chapter 3, "Statements." It describes the elements of the MS-FORTRAN file system, defines the basic concepts of I/O records and I/O units, and discusses the various kinds of file access available. It further relates these definitions to how various tasks are accomplished using the most common forms of files and I/O statements. The chapter includes a complete program illustrating the I/O statements and discusses general I/O system limitations.

4.1 Records

The building block of the MS-FORTRAN file system is the record. A record is a sequence of characters or values. There are three kinds of records: formatted, unformatted, and endfile.

1. Formatted

   A formatted record is a sequence of characters terminated by a system-dependent end-of-line marker. Formatted records are interpreted in a manner consistent with the way most operating systems and editors interpret lines.

2. Unformatted

   An unformatted record is a sequence of values, with no system alteration or interpretation. Unformatted files contain a structure that defines the physical record. Binary files contain only the values written to them, and the record structure cannot, in general, be determined from this information.

3. Endfile

   The MS-FORTRAN file system simulates a virtual endfile record after the last record in a file. The way end-of-file is represented depends in part on the operating system.

4.2 Files

A file is a sequence of records. Files are either external or internal.

1. External

   An external file is either a file on a device or the device itself.

2. Internal

   An internal file is a character variable that serves as the source or destination of some formatted I/O operation.

For the remainder of this manual, both internal MS-FORTRAN files and the files known to the operating system are usually referred to simply as "files," with context determining meaning. The OPEN statement provides the link between the two notions of files; in most cases, the ambiguity disappears after opening a file, when the two notions coincide.

4.2.1 File Properties

A FORTRAN file has the following properties:

1. name
2. position
3. structure (formatted, unformatted, or binary)
4. access method (sequential or direct)

4.2.1.1 Filename

A file can have a name. If present, a name is a character string identical to the name by which the file is known to the operating system. Filenaming conventions are determined by your operating system.

4.2.1.2 File Position

The position of a file is usually set by the previous I/O operation. A file has an initial point, terminal point, current record, preceding record, and next record.

It is possible 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.

Opening a sequential file for writing positions the file at its beginning and discards all old data in the file. The file position after sequential WRITEs is at the end of the file, but not beyond the endfile record.

Executing the ENDFILE statement positions the file beyond the endfile record, as does a READ statement executed at the end of the file. You can detect the endfile condition by using the END = option in a READ statement.

4.2.1.3 File Structure

An external file may be opened as a formatted, unformatted, or binary file. All internal files are formatted.

1. Formatted

   Files consisting entirely of formatted records.

2. Unformatted

   Files consisting entirely of unformatted records.

3. Binary

   Sequences of bytes with no internal structure.

4.2.1.4 File Access Method

An external file is opened as either a sequential file or a direct access file.

1. Sequential

   Files that contain records whose order is determined by the order in which the records were written (the normal sequential order). These files must not be read or written using the REC = option, which specifies a position for direct access I/O.

2. Direct

   Files whose records can be read or written in any order (they are random access files). Records are numbered sequentially, with the first record numbered 1. All records have the same length, specified when the file is opened; each record has a unique record number, specified when the record is written.

It is possible to write records out of order (e.g., 9, 5, and 11 in that order), without the records in between. It is not possible to delete a record once written; however, a record can be overwritten with a new value.

Reading a record from a direct access file that has not been written will result in an error. Direct access files must reside on disk. The operating system attempts to extend direct access files if a record is written beyond the old terminating file boundary; the success of this operation depends on the existence of room on the physical device.

4.2.2 Special Properties of Internal Files.

An internal file is a character variable or character array element. The file has exactly one record, which is of the same length as the character variable or character array element.

If less than the entire record is written, the remaining portion of the record is filled with blanks. The file position is always at the beginning of the file prior to execution of the I/O statement. Internal files permit only formatted, sequential I/O; and only the I/O statements READ and WRITE may specify an internal file.

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 within the MS-FORTRAN internal memory structures. That is, reading a character variable converts the character values into numeric, logical, or character values and writing a character variable allows values to be converted into their (external) character representation.

The backslash edit descriptor (\) may not be used with internal files.

4.2.3 Units

A unit is a means of referring to a file. A unit specified in an I/O statement is either an external unit specifier or an internal unit specifier. Unit numbers in the range -32767 to +32767 are accepted.

1. External unit specifier

   An external unit specifier is either an integer expression (which evaluates to a nonnegative value) or the character * (which stands for the screen (for writing) and the keyboard (for reading)).

   In most cases, an external unit specifier value is bound to a physical device (or files resident on the device) by name, using the OPEN statement. Once this binding of unit to system filename occurs, MS-FORTRAN I/O statements specify the unit number to refer to the associated external entity. Once the file is opened, the external unit specifier value is uniquely associated with a particular external entity until an explicit CLOSE operation occurs or until the program terminates.

   The only exception to these binding rules is that the unit value zero is initially associated with the keyboard for reading and the screen for writing and no explicit OPEN statement is necessary. The MS-FORTRAN file system interprets the character * as unit zero.

2. Internal unit specifier

   An internal unit specifier is a character variable or character array element that directly specifies an internal file.

See Section 4.3.1, "Elements of I/O Statements," for a discussion of how these unit specifiers are used.

4.2.4 Commonly Used File Structures

Numerous combinations of file structures are possible in MS-FORTRAN. However, two kinds of files suffice for most applications:

1. * files
2. named, external, sequential, formatted files.

* represents the keyboard and screen, that is, a sequential, formatted file, also known as unit zero. When reading from unit zero, you must enter an entire line; the normal operating system conventions for correcting typing mistakes apply.

An external file can be bound to a system name by any one of the following methods:

1. If the file is explicitly opened, the name can be specified in the OPEN statement.
2. If the file is explicitly opened and the name is specified as all blanks, the name is read from the command line (if available). If the command line is unavailable or contains no name, the user will usually be prompted for the name.
3. If the file is implicitly opened (with a READ or WRITE statement) the name is obtained as in method 2, described in the preceding paragraph.
4. If the file is explicitly opened and no name is specified in the OPEN statement, the file is considered a scratch or temporary file, and an implementation-dependent name is asumed. (See Appendix D, "MS-FORTRAN Scratch Filenames," in the *MS-FORTRAN Compiler User's Guide* for the default name used by your operating system.)

The following sample program uses * files and named, external, sequential, formatted files for reading and writing. The I/O statements themselves are explained in general in Section 4.3, "I/O Statements." For details of each individual I/O statement, see the appropriate entries in Section 3.2, "Statement Directory."

```
C  COPY A FILE WITH THREE COLUMNS OF
C  INTEGERS, EACH 7 COLUMNS WIDE, FROM A FILE
C  WHOSE NAME IS ENTERED BY THE USER TO AN-
C  OTHER FILE NAMED OUT . TEXT, REVERSING THE
C  POSITIONS OF THE FIRST AND SECOND COLUMNS.
         PROGRAM COLSWP
         CHARACTER*64 FNAME

C  PROMPT TO THE SCREEN BY WRITING TO *.
         WRITE(*,900)
900      FORMAT(' INPUT FILE NAME - '\)

C  READ THE FILE NAME FROM THE KEYBOARD BY
C  READING FROM *.
         READ(*,910) FNAME
910      FORMAT(A)

C  USE UNIT 3 FOR INPUT; ANY UNIT NUMBER EXCEPT
C  0 WILL DO.
         OPEN(3,FILE = FNAME)

C  USE UNIT 4 FOR OUTPUT; ANY UNIT NUMBER EXCEPT
C  0 AND 3 WILL DO.
         OPEN(4,FILE = 'OUT . TXT',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
```

4.2.5 Other File Structures

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of their intended uses follows:

1. If random access I/O is needed, as would probably be the case in a data base, direct access files are necessary.

2. If the data is to be both written and reread by MS-FORTRAN, unformatted files are perhaps more efficient in terms of speed, but possibly less efficient in terms of disk space. The combination of direct and unformatted files is ideal for a data base created, maintained, and accessed exclusively by MS-FORTRAN.

3. If the data must be transferred without any system interpretation, especially if all 256 possible byte values are to be transferred, unformatted I/O is necessary.

   One use of unformatted I/O is in the control of a device that has a single-byte, binary interface. Formatted I/O would, in this example, interpret certain characters, such as the ASCII representation for RETURN, and fail to pass them through to the program unaltered.

   The number of bytes written for an integer constant is determined by the $STORAGE metacommand (for details, see Section 6.2.8, "The $STORAGE Metacommand").

4. If the data is to be transferred as in the third use described in this list, but will be read by non-FORTRAN programs, the BINARY format is recommended. Unformatted files are blocked internally, and consequently the non-FORTRAN program must be compatible with this format to interpret the data correctly. BINARY files contain only the data written to them. Backspacing over records is not possible and incomplete records cannot be read from them.

4.2.6 OLD and NEW Files

A file opened in MS-FORTRAN is either OLD or NEW, but "opened for reading" is not distinguishable from "opened for writing." Therefore, you can open OLD (existing) files and write to them, with the effect of overwriting them.

Similarly, you can alternately WRITE and READ to the same file (providing that you avoid reading beyond the end of the file, or reading unwritten records in a direct file). A WRITE to a sequential file effectively deletes any records that existed beyond the newly written record.

When a device such as the keyboard or printer is opened as a file, it normally makes no difference whether it is opened as OLD or NEW. With disk files, however, opening a file as NEW creates a new file:

1. If a previous file existed with the same name, the previous file is deleted.
2. If the new file is closed with STATUS = 'KEEP' or if the program terminates without doing a CLOSE operation on that file, a permanent file is created with the name given when the file was opened.

4.2.7 Limitations

Certain limitations on the use of the MS-FORTRAN I/O system are described briefly in the following list:

1. Direct files/direct device association

   There are two kinds of devices: sequential and direct. The files associated with sequential devices are streams of characters; except for reading and writing, no explicit motion is allowed. The keyboard, screen, and printer are all sequential devices.

   Direct devices, such as disks, have the additional task of seeking a specific location. Direct devices can be accessed either sequentially or randomly, and thus can support direct files. The MS-FORTRAN I/O system does not allow direct files on sequential devices.

2. BACKSPACE/BINARY sequential file association

   There is no indication in a binary sequential file of record boundaries; therefore, a BACKSPACE operation on such files is defined as backing up by one byte. Direct files contain records of fixed, specified length, so it is possible to backspace by records on direct unformatted files.

3. Partial READ/BINARY file

   The data read from a binary file must correspond in length to the data written. Unformatted sequential files differ, in that an internal structure allows part or none of a record to be read (the unread part is skipped).

4. Side effects of functions called in I/O statements

   During execution of any I/O statement, evaluation of an expression may cause a function to be called. That function call must not cause any I/O statement to be executed.

4.3 I/O Statements

This section discusses the elements of I/O statements in general. For specific details on each of the seven I/O statements OPEN, CLOSE, READ, WRITE, BACKSPACE, ENDFILE, and REWIND, see the appropriate entries in Section 3.2, "Statement Directory," in the previous chapter.

In addition to these I/O statements, there is an I/O intrinsic function, EOF(<unit-spec>), which is described in Section 5.3.2, "Intrinsic Functions." EOF returns a logical value that indicates whether there is any data remaining in the file after the current position.

4.3.1 Elements of I/O Statements

The various I/O statements take certain parameters and arguments that specify sources and destinations of data transfer as well as other facets of the I/O operation. The elements described in this subsection are the following:

1. unit specifier (<unit-spec>)
2. format specifier (<format-spec>)
3. input/output list (<iolist>)

4.3.1.1 The Unit Specifier

The unit specifier, <unit-spec>, can take one of the following forms in an I/O statement:

1. *

   Refers to the keyboard or screen.

2. Integer expression

   Refers to an external file with a unit number equal to the value of the expression (* is unit number zero).

3. Name of a character variable or character array element

   Refers to the internal file represented by the the variable or array element.

See Section 4.2.3, "Units," for a discussion of the difference between external and internal unit specifiers.

4.3.1.2 The Format Specifier.

The format specifier, <format-spec>, can take one of the following forms in an I/O statement:

1. Statement label

   Refers to the FORMAT statement labeled by that label. For further information, see Section 3.2.18, "The FORMAT Statement."

2. Integer variable name

   Refers to the FORMAT label assigned to that integer variable using the ASSIGN statement. For further information, see Section 3.2.1, "The ASSIGN Statement."

3. Character expression

   The format specified is the current value of the character expression provided as the format specifier.

4. *

   Indicates a list-directed I/O transfer. For further information, see Section 4.5, "List-Directed I/O."

4.3.1.3 Input/Output List

The input/output list, <iolist>, specifies the entities whose values are transferred by READ and WRITE statements. An <iolist> may be empty, but ordinarily consists of input or output entities and implied DO lists, separated by commas. An input entity can be specified in the <iolist> of a READ statement and an output entity in the <iolist> of a WRITE statement.

1. Input entities

   An input entity is either a variable name, an array element name, or an array name. An array name specifies all of the elements of the array in memory sequence order.

2. Output entities

   In addition to being any of the items listed as input entities, an output entity can be any other expression not beginning with the left parenthesis character "(". (The left parenthesis distinguishes implied DO lists from expressions.)

To distinguish it from an implied DO list, the following expression

```
(A + B)*(C + D)
```

can be written as:

```
+(A + B)*(C + D)
```

3. Implied DO lists.

   Implied DO lists can be specified as items in the I/O list of READ and WRITE statements and have the following format:

   ```
   (<iolist>, <variable> = <expr1>,
    <expr2>, [, <expr3>])
   ```

   <iolist> is defined the same as for elements of I/O statements (including nested implied DO lists).

   <variable>, <expr1>, <expr2>, and <expr3> are the same as defined for the DO statement. That is, <variable> is an integer variable, while <expr1>, <expr2>, and <expr3> are integer expressions.

   In a READ statement, the DO variable (or an associated entity) must not appear as an input list item in the embedded <iolist>, but may have been read in the same READ statement before the implied DO list. The embedded <iolist> is effectively repeated for each iteration of <variable> with appropriate substitution of values for the DO variable.

   In the case of nested implied DO loops, the innermost (most deeply nested) loop is always executed first.

4.3.2 Carriage Control

The first character of every record transferred to a printer or other terminal device, including the console, is not printed. Instead, it is interpreted as a carriage control character. The MS-FORTRAN I/O system recognizes certain characters as carriage control characters. These characters and their effects when printed are shown in Table 4.1.

Table 4.1. Carriage Control Characters

Character	Effect
space	Advances one line.
0	Advances two lines.
1	Advances to top of next page (ignored by the console).
+ (plus)	Does not advance (allows overprinting).

Any character other than those listed in the preceding table is treated as a space and deleted from the print line. If you accidentally omit the carriage control character, the first character of the record is not printed.

4.4 Formatted I/O

If a READ or WRITE statement specifies a format, the I/O statement is considered a formatted, rather than an unformatted, I/O statement. Such a format can be specified in one of four ways, as explained previously in Section 4.3.1.2, "The Format Specifier."

Two of the four methods that refer to FORMAT statements are described in Section 3.2.18, "The FORMAT Statement"; the third is a character expression containing the format itself (see Section 4.4.2, "Edit Descriptors"); the fourth denotes that the operation is to be list-directed (see Section 4.5, "List-Directed I/O").

The following five examples are all valid and equivalent means of specifying a format:

```
      WRITE (*,990) I,J,K
990   FORMAT(1X,2I5,I3)

      ASSIGN 990 TO IFMT
990   FORMAT(1X,2I5,I3)
      WRITE(*,IFMT) I,J,K

      WRITE(*,'(1X,2I5,I3)')I,J,K

      CHARACTER*11 FMTCH
      FMTCH = '(1X,2I5,I3)'
      WRITE(*,FMTCH)I,J,K

      WRITE(*,*) I,J,K
```

The format specification must begin with a left parenthesis character and end with a matching right parenthesis character. The leading left parenthesis can be preceded by initial blank characters. Characters beyond the matching right parenthesis are ignored.

4.4.1 Interaction Between Format and I/O List

If an <iolist> contains at least one item, at least one repeatable edit descriptor must exist in the format specification. In particular, the empty edit specification, (), can be used only if no items are specified in the <iolist> (in which case a WRITE writes a zero length record and a READ skips to the next record).

Each item in the <iolist> is 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 (if omitted, <r> is treated as a repeat factor of one). A format specification itself can have a repeat factor, as in:

```
10(5F1.04, 2(3x,5I3))
```

During the formatted I/O process, the "format controller" scans and processes the format items as described in the previous paragraph. When a repeatable edit descriptor is encountered, one of the following occurs:

1. A corresponding item appears in the <iolist>, in which case the item and the edit descriptor are associated and I/O of that item proceeds under format control of the edit descriptor.
2. No corresponding item appears in the <iolist>, in which case the format controller terminates I/O. Thus, for the following statements:

```
         I=5
         WRITE (*,10) I
   10    FORMAT (1X, 'I=  ' ,I5, 'J=  ' ,I5)
```

   the output would look like this:

```
   I=     5J=
```

If the format controller encounters the matching final right parenthesis of the format specification and if 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 rescans the format from the beginning. Within the portion of the format rescanned, there must be at least one repeatable edit descriptor.

If the rescan of the format specification begins with a repeated nested format specification, the repeat factor indicates the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or the 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. An exception to this occurs when the \ edit descriptor is used. (See Section 4.4.3, "Nonrepeatable Edit Descriptors," for information on backslash editing.)

4.4.2 Edit Descriptors

Edit descriptors in FORTRAN specify the form of a record and control the editing between the characters in a record and the internal format of data. There are two types of edit descriptors: repeatable and nonrepeatable. Both are described in the following sections of this chapter.

4.4.2.1 Nonrepeatable Edit Descriptors

1. Apostrophe editing ('xxxx')

   The apostrophe edit descriptor has the form of a character constant and causes the character constant to be transmitted to the output unit. Embedded blanks are significant; two adjacent apostrophes, i.e., single quotation marks, must be used to represent a single apostrophe within a character constant. Apostrophe editing cannot be used for input (READ). For an example, see "Hollerith editing (H)."

2. Hollerith editing (H)

   The <n>H edit descriptor transmits the next <n> characters, with blanks counted as significant, to the output unit. Hollerith editing cannot be used for input (READ).

Examples of apostrophe and Hollerith editing:

```
C  EACH WRITE OUTPUTS CHARACTERS
C  BETWEEN THE SLASHES: /ABC'DEF/

C  APOSTROPHE EDITING
         WRITE (*,97Ø)
97Ø      FORMAT (' ABC''DEF')
         WRITE (*, '(" ABC''''DEF")')

C  SAME OUTPUT USING HOLLERITH EDITING
         WRITE (*, '(8H ABC''DEF)')
         WRITE (*,96Ø)
96Ø         FORMAT (8H ABC'DEF)
```

The leading blank in each case in the preceding examples is a carriage control character to cause a line feed (carriage return) on output.

3. Positional editing (X)

   On input (READ), the <n>X edit descriptor advances the file position <n> characters, skipping <n> characters. On output (WRITE), the <n>X edit descriptor writes <n> blanks, providing that further writing to the record occurs; otherwise, the <n>X descriptor results in no operation.

4. 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.

5. Backslash 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 a backslash (\), this automatic end-of-record is inhibited, allowing subsequent I/O statements to continue reading (or writing) from (or to) the same record. This mechanism is most commonly used to prompt to the screen and read a response from the same line, as in the following example:

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

The backslash edit descriptor does not inhibit the automatic end-of-record generated when reading from the * unit; input from the keyboard must always be terminated by the RETURN key. The backslash edit descriptor may not be used with internal files.

6. Scale factor editing (P)

   The <k>P edit descriptor sets the scale factor for subsequent F and E edit descriptors until the next <k>P edit descriptor. At the start of each I/O statement, the scale factor is initialized to zero. The scale factor affects format editing in the following ways:

   a. On input, with F and E editing (providing that no explicit exponent exists in the field) and F output editing, the externally represented number equals the internally represented number multiplied by 1Ø**<k>.

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

   c. On output, with E editing, the real part of the quantity is output multiplied by 1Ø**<k> and the exponent is reduced by <k> (effectively altering the column position of the decimal point but not the value output).

7. Blank interpretation (BN and BZ)

   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 are ignored unless, and until, a BZ edit descriptor is processed.

   The effect of ignoring blanks is to take all the nonblank 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 <RETURN> indicates pressing the RETURN key):

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

            /123     <RETURN>/
            /123    456<RETURN>/
            /    123<RETURN>/
```

Reading "short" records can temporarily invoke BN status automatically. If the total numberof characters in the input record is fewerthan that specified by the combination offormat descriptors and <iolist> elements, the record is padded on the right with blanks to the required length, and BN editing goes into effect temporarily. Thus, the following example results in the value 123, rather than 12300:

```
READ (*, ' (I5)' I
/123<RETURN>/
```

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

4.4.2.2 Repeatable Edit Descriptors

The I, F, E, D, and G edit descriptors are used for I/O of numeric data. The following general rules apply to all numeric edit descriptors:

1. 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 zero. Plus signs are optional. The blanks supplied by the file system to pad a record to the required size are also not significant.

2. 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.

3. On output, the characters generated are right-justified in the field and padded by leading blanks, if necessary.
4. 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.

Individual descriptions of the repeatable edit descriptors follow.

1. Integer editing (I)

   The edit descriptor I<w> must be associated with an <iolist> item of type INTEGER. The field is <w> characters wide. On input, an optional sign may appear in the field.

2. F real editing

   The edit descriptor F<w>.<d> must be associated with an <iolist> item of type REAL or REAL*8. The field is <w> characters wide, with a fractional part <d> digits wide. The input field begins with an optional sign followed by a string of digits which may contain an optional 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 (with leading blanks converted to zeros, if necessary). Following this is an optional exponent which is either:

   a. + (plus) or - (minus) followed by an integer, or
   b. E followed by zero or more blanks followed by an optional sign followed by an integer.

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

3. E and D real editing

   The E edit descriptor takes one of the forms E<w>.<d> or E<w>.<d>E<e>. The D edit descriptor takes the form D<w>.<d>. All parameters and rules for the E edit descriptor apply to the D edit descriptor.

For each form, the field is <w> characters wide. The <e> has no effect on input. The input field for the E and D edit descriptors 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) in effect. For a scale factor of zero, 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 forms shown in Table 4.2.

Table 4.2. Scale Factors for E and D Editing

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E<w>.<d>	|exp| <= 99	E followed by plus or minus, followed by the two-digit exponent
E<w>.<d>	99 < |exp| <= 999	Plus or minus, followed by the three-digit exponent
E<w>.<d>E<e>	|exp| <= (10**<e>)-1	E followed by plus or minus, followed by <e> digits which are the exponent with possible leading zeros
D<w>.<d>	|exp| <= 99	D followed by plus or minus, followed by the two-digit exponent
D<w>.<d>	99 < |exp| <= 999	Plus or minus, followed by the three-digit exponent

The forms E<w>.<d> and D<w>.<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 or D field. If the scale factor, <k>, is in the range (-d<k< = Ø), then the output field contains exactly <k> leading zeros after the decimal point and <d> + <k> significant digits after this. If (Ø<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.

4. G real editing

The G edit descriptor takes the forms G<w>.<d> and G<w>.<d>E<e>. For either form, the input field is <w> characters wide, with a fractional part consisting of <d> digits. If the scale factor is greater than one, the exponent part consists of <e> digits.

G input editing is the same as F input editing.

G output editing is dependent on the magnitude of the data being edited. Table 4.3 illustrates the output equivalent for the magnitude of data.

Table 4.3. Data Conversion Equivalents

Data Magnitude	Conversion Equivalent
M < Ø.1	E<w>.<d>
Ø.1 <= M < 1	F(<w>-n).<d>,n('b') [1, 2]
1 <= M < 1Ø	F(<w>-n).(<d>-1), n('b')
.	.
.	.
1Ø**(<d>-2) <= M <1Ø**(<d>-1)	. F(<w>-n).1, n('b')
1Ø**(<d>-1) <= M <1Ø**<d>	F(<w>-n).Ø, n('b')
M >= 1Ø**<d>	E<w>.<d>

Notes for Table 4.3:

1. 'b' represents a blank character.
2. n is 4 for G<w>.<d>;
   n is <e> + 2 for G<w>.<d>E<e>.

5. Logical editing (L)

   The edit descriptor takes the form L<w>, indicating that the field is <w> characters wide. The <iolist> element 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.

6. Character editing (A)

   The forms of the edit descriptor are A or A<w>. In the first form, A acquires an implied field width, <w>, from the number of characters in the <iolist> associated item. The <iolist> item must be of type CHARACTER if it is to be associated with an A or A<w> 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.

   If the number of characters input is not equal to <w>, then the input field will be blankfilled or truncated on the right to the length of <w> before being transmitted to the <iolist> item. For example, if the following program fragment is executed,

```
CHARACTER*10 C
READ(* , '(A15)') C
```

and the following thirteen characters are typed in at the keyboard,

'ABCDEFGHIJKLM'

the input field will be filled to fifteen characters:

'ABCDEFGHIJKLM '

Then the rightmost ten characters will be transmitted to the <iolist> element C:

'FGHIJKLM '

On output, if <w> exceeds the characters produced by the <iolist> item, leading blanks are provided; otherwise, the leftmost <w> characters of the <iolist> item are output.

4.5 List-Directed I/O

A list-directed record is a sequence of values and value separators.

Each value in a list-directed record is one of the following:

1. a constant
2. a null value
3. either a constant or a null value multiplied by an unsigned, nonzero, integer constant; that is, r*c (r successive appearances of the constant c) or r* (r successive null values). Except in string constants, none of these may have embedded blanks.

Each value separator in a list-directed record is one of the following:

1. a comma, optionally preceded or followed by one or more contiguous blanks
2. a slash, optionally preceded or followed by one or more contiguous blanks
3. one or more contiguous blanks between two constants, or after the last constant

4.5.1 List-Directed Input

Except as noted in the following list, input forms acceptable to format specifications for a given type are also acceptable for list-directed formatting.

The form of the input value must be acceptable for the type of the input list item. Never use blanks as zeros. Only use embedded blanks within character constants, as specified in the following list. Note that the end-of-record has the effect of a blank, except when it appears within a character constant.

1. Real or double precision constants

   A real or double precision constant must be a numeric input field; that is, a field suitable for F editing. It is assumed to have no fractional digits unless there is a decimal point within the field.

2. Logical constants

   A logical constant must not include either slashes or commas among the optional characters permitted for L editing.

3. Character constants

   A character constant is a nonempty string of characters, enclosed in single quotation marks. Each single quotation mark within a character constant must be represented by two single quotation marks, with no intervening blank or end-of-record.

   Character constants may be continued from the end of one record to the beginning of the next; the end of the record doesn't cause a blank or other character to become part of the constant. The constant may be continued on as many records as needed and may include the characters blank, comma, and slash.

   If the length <n> of the list item is less than or equal to the length <m> of the character constant, the leftmost <n> characters of the latter are transmitted to the list item. If <n> is greater than <m>, the constant is transmitted to the leftmost <m> characters of the list item.

   The remaining <n> minus <m> characters of the list item are filled with blanks. The effect is the same as if the constant were assigned to the list item in a character assignment statement.

4. Null values

   You can specify a null value in one of three ways:

   a. no characters between successive value separators

   b. no characters preceding the first value separator in the first record read by each execution of a list-directed input statement

   c. the r* form (described at the beginning of Section 4.5, "List-Directed I/O")

   A null value has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains so.

A slash encountered as a value separator during execution of a list-directed input statement stops execution of that statement after the assignment of the previous value. Any further items in the input list are treated as if they were null values.

5. Blanks

   All blanks in a list-directed input record are considered to be part of some value separator, except for the following:

   a. blanks embedded in a character constant

   b. leading blanks in the first record read by each execution of a list-directed input statement (unless immediately followed by a slash or comma)

4.5.2 List-Directed Output

The form of the values produced is the same as required for input, except as noted in the following list.

1. New records are created as necessary, but except for character constants, neither the end of a record nor blanks will occur within a constant.

2. Logical output constants are T for the value true and F for the value false.

3. Integer output constants are produced with the effect of an I12 edit descriptor.

4. Real and double precision constants are produced with the effect of either an F or an E edit descriptor, depending on the value of x in the following range:

   ```
   1Ø**Ø <= x <= 1Ø**7
   ```

   a. If x is within the range, the constant is produced using ØPF16.7 for single precision and ØPF23.14 for double precision.

   b. If x is outside the range, the constant is produced using 1PE14.6 for single precision and 1PE21.13 for double precision.

5. Character constants produced have the following characteristics:

   a. They are not delimited by apostrophes (single quotation marks).

   b. They are neither preceded nor followed by a value separator.

   c. Each internal apostrophe (single quotation mark) is represented by one externally.

   d. A blank character is inserted at the start of any record that begins with the continuation of a character constant from the preceding record.

6. Slashes, as value separators, and null values are not produced by list-directed formatting.

7. In order to provide carriage control when the record is printed, each output record begins with a blank character.

8. The list-directed output line size is 80 columns.

Chapter 5

Programs, Subroutines, and Functions

5.1	Main Program	127
5.2	Subroutines	128
5.3	Functions	129
5.3.1	External Functions	129
5.3.2	Intrinsic Functions	130
5.3.3	Statement Functions	136
5.4	Arguments	137

As described in Section 1.2, "Programs and Compilable Parts of Programs," a program unit is either a main program, a subroutine, or a function. Functions and subroutines are collectively called subprograms, or procedures. The PROGRAM, SUBROUTINE, and FUNCTION statements, as well as the statement function statement, are described in detail in Section 3.2, "Statement Directory." Related information is provided in the entries for the CALL and RETURN statements.

This chapter supplements the discussion of these individual statements with information on types of functions and a description of the relationship between formal and actual arguments in a function or subroutine call.

5.1 Main Program

A main program is any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. The first statement of a main program may be a PROGRAM statement. If the main program does not have a program statement, it will be assigned the name MAIN. The name MAIN then cannot be used to name any other global entity.

The execution of a program always begins with the first executable statement in the main program. Consequently, there must be precisely one main program in every executable program.

For further information about programs, see Section 3.2.30, "The PROGRAM Statement."

5.2 Subroutines

A subroutine is a program unit that can be called from other program units with a CALL statement. When invoked, a subroutine performs the set of actions defined by its executable statements and then returns control to the statement immediately following the one that called it.

A subroutine does not directly return a value, although values can be passed back to the calling program unit via arguments or common variables.

For further information about subroutines, see Section 3.2.37, "The SUBROUTINE Statement," and Section 3.2.4, "The CALL Statement."

5.3 Functions

A function is referred to in an expression and returns a value that is used in the computation of that expression. There are three kinds of functions:

1. external functions
2. intrinsic functions
3. statement functions

Each of these is described in more detail in the following sections.

Reference to a function may appear in an arithmetic or logical expression. When the function reference is executed, the function is evaluated and the resulting value used as an operand in the expression that contains the function reference. The format of a function reference is as follows:

<fname> ([<arg> [, <arg>] ...])

<fname> is the user-defined name of an external, intrinsic, or statement function.

<arg> is an actual argument.

The rules for arguments for functions are identical to those for subroutines and are described in Section 3.2.4, "The CALL Statement." Some additional restrictions that apply for intrinsic functions and for statement functions are described in Section 5.3.2, "Intrinsic Functions," and Section 5.3.3, "Statement Functions," respectively.

5.3.1 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 than a PROGRAM statement, FUNCTION statement, or a SUBROUTINE statement.

5.3.2 Intrinsic Functions

Intrinsic functions are predefined by the MS-FORTRAN language and available for use in an MS-FORTRAN program. Table 5.1 gives the name, definition, argument type, and function type for all of the intrinsic functions available in MS-FORTRAN, with additional notes following the table.

An IMPLICIT statement cannot 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 can appear in an INTRINSIC statement. An intrinsic function name also can 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 permitted.

All angles in Table 5.1 are expressed in radians. All arguments in an intrinsic function reference must be of the same type. X and Y are real, I and J are integer, and C, C1, and C2 are character values. Numbers in square brackets in column 1 refer to the notes following the table.

Furthermore, REAL is equivalent to REAL*4, DOUBLE PRECISION is equivalent to REAL*8. If the specified type of the argument is INTEGER, the type may be INTEGER*2 or INTEGER*4. If the specified type of the function is INTEGER, the type will be the default integer determined by the $STORAGE metacommand. (For further information, see Section 6.2.8, "The $STORAGE Metacommand.")

Table 5.1. Intrinsic Functions

Name	Definition	Type of Argument	Type of Function
Type Conversion			
INT(X) [1]	Convert to integer	REAL*4	Int
		Int	Int
IFIX(X)	Convert to integer	REAL*4	Int
IDINT(2)	Convert to integer	REAL*8	Int
REAL(X) [2]	Convert to REAL*4	Int	REAL*4
		REAL*4	REAL*4
FLOAT(I)	Convert to REAL*4	Int	REAL*4
ICHAR(C) [3]	Convert to integer	Char	Int
CHAR(X) [3]	Convert to character	Int	Char
SNGL(X)	Convert to REAL*4	REAL*8	REAL*4
DBLE(X) [4]	Convert to REAL*8	Int	REAL*8
		REAL*4	REAL*8
		REAL*8	REAL*8
Truncation			
AINT(X)	Truncate to REAL*4	REAL*4	REAL*4
DINT(X)	Truncate to REAL*8	REAL*8	REAL*8
Nearest Whole Number			
ANINT(X)	Round to REAL*4	REAL*4	REAL*4
DNINT(X)	Round to REAL*8	REAL*8	REAL*8
Nearest Integer			
NINT(X)	Round to integer	REAL*4	Int
IDNINT(X)	Round to integer	REAL*8	Int
Absolute Value			
IABS(I)	Int absolute	Int	Int
ABS(X)	REAL*4 absolute	REAL*4	REAL*4
DABS(X)	REAL*8 absolute	REAL*8	REAL*8
Remaindering			
MOD(I,J)	Int remainder	Int	Int
AMOD(X,Y)	REAL*4 remainder	REAL*4	REAL*4
DMOD(X,Y)	REAL*8 remainder	REAL*8	REAL*8

Table 5.1. Intrinsic Functions (continued)

Name	Definition	Type of Argument	Type of Function
Transfer of Sign			
ISIGN(I,J)	Int transfer	Int	Int
SIGN(X,Y)	REAL*4 transfer	REAL*4	REAL*4
DSIGN(X,Y)	REAL*8 transfer	REAL*8	REAL*8
Positive Difference [5]			
IDIM(I,J)	Int difference	Int	Int
DIM(X,Y)	REAL*4 difference	REAL*4	REAL*4
DDIM(X,Y)	REAL*8 difference	REAL*8	REAL*8
Choosing Largest Value			
MAXØ(I,J,...)	Int maximum	Int	Int
AMAX1(X,Y,...)	REAL*4 maximum	REAL*4	REAL*4
AMAXØ(I,J,...)	REAL*4 maximum	Int	REAL*4
MAX1(X,Y,...)	Int maximum	REAL*4	Int
DMAX1(X,Y,...)	REAL*8 maximum	REAL*8	REAL*8
Choosing Smallest Value			
MINØ(I,J,...)	Int minimum	Int	Int
AMIN1(X,Y,...)	REAL*4 minimum	REAL*4	REAL*4
AMINØ(I,J,...)	REAL*4 minimum	Int	REAL*4
MIN1(X,Y,...)	Int minimum	REAL*4	Int
DMIN1(X,Y,...)	REAL*8 minimum	REAL*8	REAL*8
REAL*8 Product			
DPROD	REAL*8 product	REAL*4	REAL*8
Square Root			
SQRT	Square root	REAL*4	REAL*4
DSQRT	REAL*8 square root	REAL*8	REAL*8
Exponential			
EXP(X)	REAL*4 e to power	REAL*4	REAL*4
DEXP(X)	REAL*8 e to power	REAL*8	REAL*8
Natural Logarithm			
ALOG(X)	Nat'l log of REAL*4	REAL*4	REAL*4
DLOG(X)	Nat'l log of REAL*8	REAL*8	REAL*8

Table 5.1. Intrinsic Functions (continued)

Name	Definition	Type of Argument	Type of Function
Common Logarithm			
ALOG10(X)	Common log of REAL*4	REAL*4	REAL*4
DLOG10(X)	Common log of REAL*8	REAL*8	REAL*8
Sine			
SIN(X)	REAL*4 sine	REAL*4	REAL*4
DSIN(X)	REAL*8 sine	REAL*8	REAL*8
Cosine			
COS(X)	REAL*4 cosine	REAL*4	REAL*4
DCOS(X)	REAL*8 cosine	REAL*8	REAL*8
Tangent			
TAN(X)	REAL*4 tangent	REAL*4	REAL*4
DTAN(X)	REAL*8 tangent	REAL*8	REAL*8
Arc Sine			
ASIN(X)	REAL*4 arc sine	REAL*4	REAL*4
DASIN(X)	REAL*8 arc sine	REAL*8	REAL*8
Arc Cosine			
ACOS(X)	REAL*4 arc cosine	REAL*4	REAL*4
DACOS(X)	REAL*8 arc cosine	REAL*8	REAL*8
Arc Tangent			
ATAN(X)	REAL*4 arc tangent	REAL*4	REAL*4
DATAN(X)	REAL*8 arc tangent	REAL*8	REAL*8
ATAN2(X,Y)	REAL*4 arctan of X/Y	REAL*4	REAL*4
DATAN2(X,Y)	REAL*8 arctan of X/Y	REAL*8	REAL*8
Hyperbolic Sine			
SINH(X)	REAL*4 hyperbolic sine	REAL*4	REAL*4
DSINH(X)	REAL*8 hyperbolic sine	REAL*8	REAL*8

Table 5.1. Intrinsic Functions (continued)

Name	Definition	Type of Argument	Type of Function
Hyperbolic Cosine			
COSH(X)	REAL*4 hyperbolic cosine	REAL*4	REAL*4
DCOSH(X)	REAL*8 hyperbolic cosine	REAL*8	REAL*8
Hyperbolic Tangent			
TANH(X)	REAL*4 hyperbolic tangent	REAL*4	REAL*4
DTANH(X)	REAL*8 hyperbolic tangent	REAL*8	REAL*8
Lexically Greater Than or Equal			
LGE(C1,C2)	1st argument greater than or equal to 2nd	Char	Logical
Lexically Greater Than			
LGT(C1,C2)	1st argument greater than 2nd	Char	Logical
Lexically Less Than or Equal [6]			
LLE(C1,C2)	1st argument less than or equal to 2nd	Char	Logical
Lexically Less Than			
LLT(C1,C2)	1st argument less than 2nd	Char	Logical
End of File [7]			
EOF(X)	Int end of file	Int	Logical

Notes for Table 5.1:

1. For X of type INTEGER, INT(X) = X. For X of type REAL or REAL*8, if X is greater than or equal to zero, then INT(X) is the largest integer not greater than X, and if X is less than zero, then INT(X) is the most negative integer not less than X. For X of type REAL, IFIX(X) is the same as INT(X).

2. For X of type REAL, REAL(X) = X. For X of type INTEGER or REAL*8, REAL(X) is as much precision of the significant part of X as a real datum can contain. For X of type INTEGER, FLOAT(X) is the same as REAL(X).

3. For X of type REAL*8, DBLE(X) = X. For X of type INTEGER or REAL, DBLE(X) is as much precision of the significant part of X as a double precision datum can contain.

4. 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 Ø to 255. For any two characters, c1 and c2, (c1 .LE. c2) is .TRUE. if and only if (ICHAR(c1) .LE. ICHAR(c2)) is .TRUE.

   CHAR<i> returns the <i>th character in the collating sequence. The value is of type CHARACTER, length one, while <i> must be an integer expression whose value is in the range Ø <= <i> <= 255.

   ICHAR(CHAR<i>) = <i> for Ø <= <i> <= 255.

   CHAR(ICHAR(c)) = c for any character c in the character set.

5. DIM(X,Y) is X-Y if X>y, zero otherwise.

6. LGE(X,Y) returns the value .TRUE. if X = Y or if X follows Y in the ASCII collating sequence; otherwise it returns .FALSE.

   LGT(X,Y) returns .TRUE. if X follows Y in the ASCII collating sequence; otherwise it returns .FALSE.

   LLE(X,Y) returns .TRUE. if X = Y or if X precedes Y in the ASCII collating sequence; otherwise it returns .FALSE.

LLT(X,Y) returns .TRUE. if X precedes Y in the ASCII collating sequence; otherwise it returns .FALSE.

If the operands are of unequal length, the shorter operand is considered to be blankfilled on the right to the length of the longer.

7. EOF(X) 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 X must correspond to an open file, or to zero which indicates the screen or keyboard device.

5.3.3 Statement Functions

A statement function is defined by a single statement and 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 in an expression.

For information on the syntax and use of a statement function statement, see Section 3.2.35, "The Statement Function Statement."

5.4 Arguments

A formal argument is the name by which the argument is known within a function or subroutine; an actual argument is the specific variable, expression, array, etc., passed to the procedure in question at any specific calling location. The relationship between formal and actual arguments in a function or subroutine call is discussed in detail in the following paragraphs.

Arguments pass values into and out of procedures by reference. The number of actual arguments must be the same as the number of formal arguments, and the corresponding types must agree.

Upon entry to a subroutine or function, the actual arguments are 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 permitted, and can have some strange side effects. In particular, assigning a value to a formal argument of type CHARACTER, when the actual argument is a literal, can produce anomalous behavior.

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 expressions are evaluated just prior to the association, and remain constant throughout the execution of the procedure, even if they contain variables that are redefined during the execution of the procedure.

A formal argument that is a variable can be associated with an actual argument that is a variable, an array element, or an expression.

A formal argument that is an array can 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 from those of the actual argument, but any reference to the formal array must be within the limits of the memory sequence in the actual array. While a reference to an element outside these bounds is not detected as an error in a running MS-FORTRAN program, the results are unpredictable.

A formal argument may also be associated with an external subroutine, function, or intrinsic function if it is used in the body of the procedure as a subroutine or function reference, or if it appears in an EXTERNAL statement.

A corresponding actual argument must be an external subroutine or function, declared with the EXTERNAL statement, or an intrinsic function permitted to be associated with a formal procedure argument. The intrinsic function must have been declared with an INTRINSIC statement in the program unit where it is used as an actual argument.

All intrinsic functions, except the following, may be associated with formal procedure arguments:

INT	CHAR	AMAXØ
IFIX	LGE	MAX1
IDINT	LGT	MINØ
FLOAT	LLE	AMIN1
SNGL	LLT	DMIN1
REAL	MAXØ	AMINØ
DBLE	AMAX1	MIN1
ICHAR	DMAX1	

Chapter 6

The MS-FORTRAN Metacommands

6.1	Overview	140
6.2	Metacommand Directory	142
6.2.1	The $DEBUG and $NODEBUG Metacommands	142
6.2.2	The $DO66 Metacommand	142
6.2.2a	The $FLOATCALLS and $NOFLOATCALLS Metacommands	143
6.2.3	The $INCLUDE Metacommand	144
6.2.4	The $LINESIZE Metacommand	144
6.2.5	The $LIST and $NOLIST Metacommands	144
6.2.5a	The $MESSAGE Metacommand	145
6.2.6	The $PAGE Metacommand	145
6.2.7	The $PAGESIZE Metacommand	145
6.2.8	The $STORAGE Metacommand	145
6.2.9	The $STRICT and $NOTSTRICT Metacommands	146
6.2.10	The $SUBTITLE Metacommand	147
6.2.11	The $TITLE Metacommand	147

6.1 Overview

Metacommands are directives that order the MS-FORTRAN Compiler to process MS-FORTRAN source text in a specific way. MS-FORTRAN metacommands are described briefly in Table 6.1 and discussed in more detail in the remainder of the chapter.

Table 6.1. The MS-FORTRAN Metacommands

Metacommand	Action
$DEBUG	Turns on runtime checking for arithmetic operations and assigned GOTO. $NODEBUG turns checking off.
$DO66	Causes DO statements to have FORTRAN 66 semantics.
$FLOATCALLS	Causes floating-point operations to be processed by library procedures through CALL instructions rather than through interrupts. $NOFLOATCALLS suppresses this option.
$INCLUDE:<file>	Directs compiler to proceed as if <file> were inserted at that point.
$LINESIZE:<n>	Makes subsequent pages of listing <n> columns wide.
$LIST	Sends subsequent listing information to the listing file. $NOLIST stops generation of listing information.
$MESSAGE <message>	Sends message to the standard output device when running the Fortran front end.
$PAGE	Starts new page of listing.
$PAGESIZE:<n>	Makes subsequent pages of listing <n> lines long.
$STORAGE:<n>	Allocates <n> bytes of memory to all LOGICAL or INTEGER variables in source.

$STRICT	Disables MS-FORTRAN features not in 1977 subset or full language standard. $NOTSTRICT enables them.
$SUBTITLE:'<sub>'	Gives subtitle for subsequent pages of listing.
$TITLE:'<title>'	Gives title for subsequent pages of listing.

Metacommands can be intermixed with MS-FORTRAN source text within an MS-FORTRAN source program; however, they are not part of the standard FORTRAN language. Any line of input to the MS-FORTRAN Compiler that begins with a "$" character in column one is interpreted as a metacommand and must conform to one of the following formats.

A metacommand and its arguments (if any) must fit on a single source line; continuation lines are not permitted. Also, blanks are significant, so that the following pair is not equivalent:

```
$S TRICT
$STRICT
```

6.2 Metacommand Directory

The remainder of this chapter is an alphabetical directory of available MS-FORTRAN metacommands.

6.2.1 The $DEBUG and $NODEBUG Metacommands

Syntax $[NO]DEBUG

Purpose Directs the compiler to test all subsequent arithmetic operations for overflow and division by zero, test assigned GOTO values against the allowable list in an assigned GOTO statement, and provide the runtime error-handling system with source filenames and line numbers. A runtime error occurs if one of these conditions is detected. If any runtime error occurs, the source line and filename are displayed on the console.

Remarks The metacommand can appear anywhere in a program.

The default value of the pair of metacommands, $DEBUG and $NODEBUG, is $NODEBUG.

6.2.2 The $DO66 Metacommand

Syntax $DO66

Purpose Causes DO statements to have FORTRAN 66 semantics.

Remarks $DO66 must precede the first declaration or executable statement of the source file in which it occurs.

The FORTRAN 66 semantics are as follows:

1. All DO statements are executed at least once.

2. Extended range is permitted; that is, control may transfer into the syntactic body of a DO statement. The range of the DO statement is thereby extended to logically include any statement that may be executed between a DO statement and its terminal statement. However, the transfer of control into the range of a DO statement prior to the execution of the

DO statement or following the final execution of its terminal statement is invalid.

If a program contains no $DO66 metacommand, the default is to FORTRAN 77 semantics, as follows:

1. DO statements may be executed zero times, if the initial control variable value exceeds the final control variable value (or the corresponding condition for a DO statement with negative increment).

2. Extended range is invalid; that is, control may not transfer into the syntactic body of a DO statement. (Both standards do permit transfer of control out of the body of a DO statement.)

6.2.2a The $FLOATCALLS and $NOFLOATCALLS Metacommands

Syntax $[NO]FLOATCALLS

Purpose Causes floating-point operations to be processed by library procedures through CALL instructions rather than through interrupts. $NOFLOATCALLS suppresses this option.

Remarks A call to the floating-point routines may require up to 25 percent less processor time than the equivalent emulated interrupt instruction. But, this can generate up to twice as much code. Other operations are unaffected, however, and the total operations are unaffected, however, and the total increase in code size is between 10 and 30 percent. A full 25 percent improvement in execution speed, therefore, should not be expected.

The subroutines called are part of the emulator. Because the subroutines are in the library and use mechanisms for carrying out arithmetic as in-line instructions, you can freely mix modules compiled with $FLOATCALLS with those compiled without $FLOATCALLS. In fact, you can switch $FLOATCALLS on and off within a subroutine. However, this practice might not take effect exactly where you specified it, because optimizations may group statements or reorder code.

THe default value of the pair of metacommands, $FLOATCALLS and $NOFLOATCALLS, is $NOFLOATCALLS.

6.2.3 The $INCLUDE Metacommand

Syntax $INCLUDE: '<file>'

Purpose Directs the compiler to proceed as though the specified file were inserted at the point of the $INCLUDE.

Remarks <file> is a valid file specification as described for your operating system.

At the end of the included file, the compiler resumes processing the original source file at the line following $INCLUDE.

The compiler imposes no limit on nesting levels for $INCLUDE metacommands. $INCLUDE metacommands are particularly useful for guaranteeing that several modules use the same declaration for a COMMON block.

6.2.4 The $LINESIZE Metacommand

Syntax $LINESIZE: <n>

Purpose Formats subsequent pages of the listing <n> columns wide.

Remarks <n> is any positive integer.

If a program contains no $LINESIZE metacommand, a default line size of 80 characters is assumed. The minimum line size is 40 characters, the maximum is 132 characters.

6.2.5 The $LIST and $NOLIST Metacommands

Syntax $[NO]LIST

Purpose Sends subsequent listing information to the listing file specified when starting the compiler. If no listing file is specified in response to the compiler prompt, the metacommand has no effect. $NOLIST directs that subsequent listing information be discarded.

Remarks $LIST and $NOLIST can appear anywhere in a source file.

The default condition for the pair of metacommands, $LIST and $NOLIST, is $LIST.

6.2.5a The $MESSAGE Metacommand

Syntax $MESSAGE '<message>'

Purpose Sends messages to the standard output device when running the Fortran front end.

Remarks <message> is a string of characters enclosed in single quotes.

6.2.6 The $PAGE Metacommand

Syntax $PAGE

Purpose Starts a new page of the listing.

Remarks If the first character of a line of source text is the ASCII form feed character (hexadecimal code ØCh), it is considered as equivalent to the occurrence of a $PAGE metacommand at that point.

6.2.7 The $PAGESIZE Metacommand

Syntax $PAGESIZE: <n>

Purpose Formats subsequent pages of the listing <n> lines high.

Remarks <n> is any positive integer equal to or greater than 15.

If a program contains no $PAGESIZE metacommand, a default page size of 66 lines is assumed.

6.2.8 The $STORAGE Metacommand

Syntax $STORAGE: <n>

Purpose Allocates <n> bytes of memory for all variables declared in the source file as INTEGER or LOGICAL.

Remarks <n> is either 2 or 4. Use a value of 2 for code that defaults to 16-bit arithmetic. See also the important note on performance issues in Section 2.3, "Data Types."

$STORAGE does not affect the allocation of memory for variables declared with an explicit length specification, for example, as INTEGER*2 or LOGICAL*4.

If several files of a source program are compiled and linked together, you should be particularly careful that they are consistent in their allocation of memory for variables (such as actual and formal parameters) referred to in more than one module.

The $STORAGE metacommand must precede the first declaration statement of the source file in which it occurs.

If a program contains no $STORAGE metacommand, a default allocation of 4 bytes is used. This default results in INTEGER, LOGICAL, and REAL variables being allocated the same amount of memory, as required by the FORTRAN 77 standard.

6.2.9 The $STRICT and $NOTSTRICT Metacommands

Syntax $[NOT]STRICT

Purpose $STRICT disables the specific MS-FORTRAN features not found in the FORTRAN 77 subset or full language standard.

Remarks The $NOTSTRICT metacommand enables these MS-FORTRAN features, which are the following:

1. Character expressions may be assigned to noncharacter variables.
2. Character and noncharacter expressions may be compared.
3. Character and noncharacter variables are allowed in the same COMMON block.
4. Character and noncharacter variables may be equivalenced.
5. Noncharacter variables may be initialized with character data.

$STRICT and $NOTSTRICT can appear anywhere in a source file.

The default condition for the pair of metacommands, $STRICT and $NOTSTRICT, is $NOTSTRICT.

6.2.10 The $SUBTITLE Metacommand

Syntax $SUBTITLE: '<subtitle>'

Purpose Assigns the specified subtitle for subsequent pages of the source listing (until overridden by another $SUBTITLE metacommand).

Remarks <subtitle> is any valid character constant. The maximum length is 40 characters.

If a program contains no $SUBTITLE metacommand, the subtitle is a null string.

6.2.11 The $TITLE Metacommand

Syntax $TITLE: '<title>'

Purpose Assigns the specified title for subsequent pages of the listing (until overridden by another $TITLE metacommand).

Remarks <title> is any valid character constant. The maximum length is 40 characters.

If a program contains no $TITLE metacommand, the title is a null string.

Appendices

Appendices

A / MS-FORTRAN and
ANSI Subset FORTRAN..151
B / ASCII Character Codes.......................................155
C / Error Messages..157

Appendix A

MS-FORTRAN and ANSI Subset FORTRAN

A.1 Full Language Features..........................149
A.2 Extensions to the Standard.....................151

This appendix describes how MS-FORTRAN differs from the standard subset language. The ANSI standard defines two levels, full FORTRAN and subset FORTRAN. MS-FORTRAN is a superset of the latter. The differences between MS-FORTRAN and the standard subset FORTRAN fall into two general categories: full language features and extensions to the standard.

A.1 Full Language Features

Several features from the full language are included in this implementation. In all cases, a program written to comply with the subset restrictions compiles and executes properly, since the full language includes the subset constructs.

1. Subscript expressions

   The subset does not allow function calls or array element references in subscript expressions; however, these are allowed in the full language and in this implementation.

2. DO variable expressions

   The subset restricts expressions that define the limits of a DO statement; the full language does not. MS-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.

3. Unit I/O number

   MS-FORTRAN allows an I/O unit to be specified by an integer expression, as does the full language.

4. Expressions in input/output list <iolist>

   The subset does not allow expressions to appear in an <iolist>, whereas the full language does allow expressions in the <iolist> of WRITE statements. MS-FORTRAN allows expressions in the <iolist> of a WRITE statement providing that the expressions do not begin with an initial left parenthesis.

   Note that an expression like (A + B)*(C + D) can be specified in an output list as + (A + B)*(C + D). Doing so does not generate any extra code to evaluate the leading plus sign.

5. Double precision

   The subset does not allow double precision real numbers; MS-FORTRAN provides for them as in the full language.

6. Edit descriptors

   MS-FORTRAN allows for D and G edit descriptors as in the full language.

7. Expression in computed GOTO

   MS-FORTRAN allows an expression for the selector of a computed GOTO, consistent with the full, rather than the subset, language.

8. Generalized I/O

   MS-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 files to be formatted.

   MS-FORTRAN also includes the following:

   a. an augmented OPEN statement that takes additional parameters not included in the subset (see Section 3.2.28, "The OPEN Statement")

   b. a form of the CLOSE statement, which is not included in the subset (see Section 3.2.5, "The CLOSE Statement")

c. END = , ERR = , STATUS = , and FILE = specifiers on I/O statements

9. List-directed I/O

   MS-FORTRAN provides for list-directed I/O as described in the full language standard.

A.2 Extensions to the Standard

The implemented language also has several minor extensions to the full language standard.

1. User-defined names greater than six characters are allowed, although only the first six characters are significant.

2. Tabs in source files are allowed. See Section 2.1.3, "Tabs," for details.

3. Metacommands, or compiler directives, have been added to allow the programmer to communicate certain information to the compiler. The metacommand line is characterized by a dollar sign ($) appearing in column 1. A metacommand line may appear any place that a comment line can appear, although certain metacommands are restricted as to their location within a program (see Section 2.2.4, "Statement Definition and Order").

   A metacommand line conveys certain compile time information about the nature of the current compilation to the MS-FORTRAN Compiler. Metacommands are described in Chapter 6, "The MS-FORTRAN Metacommands."

4. The standard is relaxed when the $NOTSTRICT metacommand is in effect. This relaxation allows, for example, such MS-FORTRAN features as assignment of character to any variable type and initialization of any variable with character data. See Section 6.2.9, "The $STRICT and $NOTSTRICT Metacommands," for a complete list of these features.

5. The backslash (\) edit control character can be used in format specifications to inhibit normal

advancement to the next record associated with the completion of a READ or WRITE statement. This is particularly useful when prompting to an interactive device, such as the screen, so that a response can appear on the same line as the prompt.

6. An end-of-file intrinsic function, EOF, is provided. The function accepts a unit specifier as an argument and returns a logical value that indicates whether the specified unit is at its end-of-file.

7. Both upper and lowercase source input are allowed. In most contexts, lowercase characters are treated as indistinguishable from their uppercase counterparts. However, lowercase is significant in character constants and Hollerith fields.

8. Binary files are similar to unformatted sequential files except that they have no internal structure. This allows the program to create or read files with arbitrary contents, which is particularly useful for files created by or intended for programs written in languages other than FORTRAN.

Appendix B

ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	032	20H	SPACE
001	01H	SOH	033	21H	!
002	02H	STX	034	22H	"
003	03H	ETX	035	23H	#
004	04H	EOT	036	24H	$
005	05H	ENQ	037	25H	%
006	06H	ACK	038	26H	&
007	07H	BEL	039	27H	'
008	08H	BS	040	28H	(
009	09H	HT	041	29H	)
010	0AH	LF	042	2AH	*
011	0BH	VT	043	2BH	+
012	0CH	FF	044	2CH	,
013	0DH	CR	045	2DH	-
014	0EH	SO	046	2EH	.
015	0FH	SI	047	2FH	/
016	10H	DLE	048	30H	0
017	11H	DC1	049	31H	1
018	12H	DC2	050	32H	2
019	13H	DC3	051	33H	3
020	14H	DC4	052	34H	4
021	15H	NAK	053	35H	5
022	16H	SYN	054	36H	6
023	17H	ETB	055	37H	7
024	18H	CAN	056	38H	8
025	19H	EM	057	39H	9
026	1AH	SUB	058	3AH	:
027	1BH	ESCAPE	059	3BH	;
028	1CH	FS	060	3CH	<
029	1DH	GS	061	3DH	=
030	1EH	RS	062	3EH	>
031	1FH	US	063	3FH	?

064	40H	@	096	60H	'
065	41H	A	097	61H	a
066	42H	B	098	62H	b
067	43H	C	099	63H	c
068	44H	D	100	64H	d
069	45H	E	101	65H	e
070	46H	F	102	66H	f
071	47H	G	103	67H	g
072	48H	H	104	68H	h
073	49H	I	105	69H	i
074	4AH	J	106	6AH	j
075	4BH	K	107	6BH	k
076	4CH	L	108	6CH	l
077	4DH	M	109	6DH	m
078	4EH	N	110	6EH	n
079	4FH	O	111	6FH	o
080	50H	P	112	70H	p
081	51H	Q	113	71H	q
082	52H	R	114	72H	r
083	53H	S	115	73H	s
084	54H	T	116	74H	t
085	55H	U	117	75H	u
086	56H	V	118	76H	v
087	57H	W	119	77H	w
088	58H	X	120	78H	x
089	59H	Y	121	79H	y
090	5AH	Z	122	7AH	z
091	5BH	[	123	7BH	{
092	5CH	\	124	7CH	|
093	5DH	]	125	7DH	}
094	5EH	∧	126	7EH	∿
095	5FH	_	127	7FH	DEL

Dec = Decimal, Hex = Hexadecimal (H), CHR = Character, LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rub Out

Appendix C

Error Messages

C.1	Compile Time Error Messages	156
C.2	Runtime Error Messages	161
C.2.1	File System Errors	161
C.2.2	Other Runtime Errors	164
C.2.2.1	Memory Errors	164
C.2.2.2	Type REAL Arithmetic Errors	165
C.2.2.3	Type INTEGER4 Arithmetic Errors	165
C.2.2.4	Other Errors	166

Error Messages

C.1 Compile Time Error Messages

Code	Message
1	Fatal error reading source
2	Non-numeric characters in label field
3	Too many continuation lines
4	Fatal end of file encountered
5	Label in continuation line
6	Missing field in metacommand
7	Cannot open file
8	Unrecognizable metacommand
9	Input file invalid format
10	Too many nested include files
11	Integer constant error
12	Real constant error
13	Too many digits in constant
14	Identifier too long
15	Character constant not closed
16	Zero length character constant
17	Invalid character in input
18	Integer constant expected
19	Label expected
20	Label error
21	Type expected
22	Integer constant expected
23	Extra characters at end of statement
24	"(" expected
25	Letter already used in IMPLICIT
26	")" expected
27	Letter expected
28	Identifier expected
29	Dimensions expected
30	Array already dimensioned
31	Too many dimensions
32	Incompatible arguments
33	Identifier already has type
34	Identifier already declared
35	INTRINSIC FUNCTION not allowed here
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 COMMON
41	Variables in two different COMMON blocks
42	Number of subscripts conflicts with declaration
43	Subscript out of range
44	Forces location conflict for items in COMMON
45	Forces location in negative direction
46	Forces location conflict
47	Statement number expected
48	CHARACTER and numeric items in same COMMON block
49	CHARACTER and non character item conflict
50	Invalid symbol in expression
51	SUBROUTINE name in expression
52	INTEGER or REAL expected
53	INTEGER, REAL or CHARACTER expected
54	Types not compatible
55	LOGICAL expression expected
56	Too many subscripts
57	Too few subscripts
58	Variable expected
59	" = " expected
60	Size of CHARACTER items must agree
61	Assignment types do not match
62	SUBROUTINE name expected
63	Dummy argument not allowed
65	Assumed size declarations only for dummy arrays
66	Adjustable size array declarations only for dummy arrays
67	Assumed size must be last dimension
68	Adjustable bound must be parameter or in COMMON
69	Adjustable bound must be simple integer variable
70	More than one main program
71	Size of named COMMON must agree
72	Dummy arguments not allowed
73	COMMON variables not allowed
74	SUBROUTINE, FUNCTION, or INTRINSIC names not allowed

75	Subscript out of range
76	Repeat count must be > = 1
77	Constant expected
78	Type conflict
79	Number of variables does not match
80	Label not allowed
81	No such INTRINSIC FUNCTION
82	INTRINSIC FUNCTION type conflict
83	Letter expected
84	FUNCTION type conflict with previous call
85	SUBROUTINE / FUNCTION already defined
87	Argument type conflict
88	SUBROUTINE / FUNCTION conflict with previous use
89	Unrecognizable statement
90	CHARACTER FUNCTION not allowed
91	Missing END statement
93	Fewer actual than dummy arguments in call
94	More actual than dummy arguments in call
95	Argument type conflict
96	SUBROUTINE / FUNCTION not defined
98	CHARACTER size invalid ?
100	Statement order
101	Unrecognizable statement
102	Jump into block not allowed
103	Label already used for FORMAT
104	Label already defined
105	Jump to FORMAT not allowed
106	DO statement not allowed here
107	DO label must follow DO statement
108	ENDIF not allowed here
109	Matching IF missing
110	Improperly nested DO block in IF block
111	ELSEIF not allowed here
112	Matching IF missing
113	Improperly nested DO or ELSE block
114	"(" expected
115	")" expected
116	THEN expected
117	Logical expression expected
118	ELSE not allowed here
119	Matching IF missing
120	GOTO not allowed here

122	Block IF not allowed here
123	Logical IF not allowed here
124	Arithmetic IF not allowed here
125	" , " expected
126	Expression of wrong type
127	RETURN not allowed here
128	STOP not allowed here
129	END not allowed here
131	Label not defined
132	DO or IF block not terminated
133	FORMAT not allowed here
134	FORMAT label already referenced
135	FORMAT label missing
136	Identifier expected
137	Integer variable expected
138	TO expected
139	Integer expression expected
140	ASSIGN statement missing
141	Unrecognizable character constant
142	Character constant expected
143	Integer expression expected
144	STATUS option expected
145	Only character expression allowed
146	Conflicting options
147	Option already defined
148	Integer expression expected
149	Unrecognizable option
150	RECL = missing
151	Adjustable arrays not allowed here
152	End of statement encountered in implied DO
153	Variable required as control for implied DO
154	Expressions not allowed in I/O list
155	REC = option already defined
156	Integer expression expected
157	END = not allowed here
158	END = already defined
159	Unrecognizable I/O unit
160	Unrecognizable format in I/O
161	Options expected after " , "
162	Unrecognizable I/O list element
163	FORMAT not found
164	ASSIGN missing
165	Label used as FORMAT

166	Integer variable expected
167	Label defined more than once as format
188	Statement too complicated
203	CHARACTER FUNCTION not allowed
406	Unit zero must be formatted and sequential
407	ERR = already defined
408	Too many labels
409	Invalid size for this type
410	PRECISION expected
411	Integer type conflict
415	Dimension too big
420	Invalid FUNCTION call
421	INTRINSIC not allowed
501	Unrecognizable character
502	Blank not allowed in metacommand
503	Metacommand not allowed here
504	Size already defined
601	Out of range
701	CHARACTER type expected
703	Internal error
705	Internal error
706	Internal error
708	Internal error
709	CHARACTER type not expected
710	Internal error
711	Internal error
712	Internal error
713	Long integer conversion error
714	Cannot convert to single
715	Cannot convert to double
717	Internal error
718	Internal error
802	Invalid radix
803	Starting location is odd
804	Real constant overflow
805	Integer constant too big
806	Missing actual argument
807	Variable too big
808	Data size exceeds max
809	Numeric expected
810	Numeric or CHARACTER expected
811	Numeric or character expected
812	COMMON exceeds max size
813	Array dimension upper bound < lower bound

C.2 Runtime Error Messages

Runtime errors fall into two classes:

1. file system errors
2. nonfile system errors

Nonfile system errors include the following:

1. memory errors
2. type REAL arithmetic errors
3. type INTEGER*4 arithmetic errors
4. other errors

If you see an error message that is not listed, check your operating system manual, as the error may be an operating system error.

C.2.1 File System Errors

Code numbers 1000 through 1099 are status codes, always issued in conjunction with an OS status code.

Code	Message
1000	Write error when writing end of file
1002	Filename extension with more than 3 characters
1003	Error during creation of new file (Disk or directory full)
1004	Error during open of existing file (File not found)
1005	Filename with more than 8 or zero characters
1007	Filename over 21 characters or contains invalid characters
1008	Write error when advancing to next record
1009	File too big (Over 65535 logical sectors)
1010	Write error when seeking to direct record device
1011	Attempt to open a random file to a non-disk device
1012	Forward space or back space on a non-disk device
1013	Disk or directory full error during forward space or back space

1200 Format missing final ")'

1201 Sign not expected in input

1202 Sign not followed by digit in input

1203 Digit expected in input

1204 Missing N or Z after B in format

1205 Unexpected character in format

1206 Zero repetition factor in format not allowed

1207 Integer expected for w field in format

1208 Positive integer required for w field in format

1209 " . " expected in format

1210 Integer expected for d field in format

1211 Integer expected for e field in format

1212 Positive integer required for e field in format

1213 Positive integer required for w field in format

1214 Hollerith field in format must not appear for reading

1215 Hollerith field in format requires repetition factor

1216 X field in format requires repetition factor

1217 P field in format requires repetition factor

1218 Integer appears before + or - in format

1219 Integer expected after + or - in format

1220 P format expected after signed repetition factor in format

1221 Maximum nesting level for formats exceeded

1222 ")" has repetition factor in format

1223 Integer followed by , illegal in format

1224 " . " is illegal format control character

1225 Character constant must not appear in format for reading

1226 Character constant in format must not be repeated

1227 "/" in format must not be repeated

1228 "\" in format must not be repeated

1229 BN or BZ format control must not be repeated

1230 Attempt to reference unknown unit number

1231 Formatted I/O attempted on file opened as unformatted

1232 Format fails to begin with "("

1233 I format expected for integer read

1234 F or E format expected for real read

1235 Two " . " characters in formatted real read

1236 Invaid REAL number

1237	L format expected for logical read
1238	Blank logical field
1239	T or F expected in logical read
1240	A format expected for character read
1241	I format expected for integer write
1242	w field in F format not greater than d field + 1
1243	Scale factor out of range of d field in E format
1244	E or F format expected for real write
1245	L format expected for logical write
1246	A format expected for character write
1247	Attempt to do unformatted I/O to a unit opened as formatted
1251	Integer overflow on input
1252	Too many bytes read from input record
1253	Too many bytes written to direct access unit record
1255	Attempt to do external I/O on a unit beyond end of file record
1256	Attempt to position a unit for direct access on a non-positive record number
1257	Attempt to do direct access to a unit opened as sequential
1258	Unable to seek to file position
1260	Attempt to BACKSPACE, REWIND or ENDFILE unit connected to unblocked device.
1261	Premature end of file of unformatted sequential file
1262	Invalid blocking in unformatted sequential file
1263	Incorrect physical record structure in unformatted file
1264	Attempt to do unformatted I/O to internal unit
1265	Attempt to put more than one record into internal unit
1266	Attempt to write more characters to internal unit than its length
1267	EOF called on unknown unit
1268	Dynamic file allocation limit exceeded
1269	Scratch file opened for read
1270	Console I/O error
1272	File operation attempted after error encountered on previous operation

1273	Keyboard buffer overflow: too many bytes written to keyboard input record (Must be less than 132)
1274	Reading long integer
1275	Writing long integer
1281	Repeat field not on integer
1282	Multiple repeat character
1283	Invalid numeric data in list directed input
1284	List directed numeric items bigger than record size
1285	Invalid string in list directed input
1298	End of file encountered
1299	Integer variable not ASSIGNed a label used in assigned GOTO

C.2.2 Other Runtime Errors

Nonfile system error codes range from 2000 to 2999. In some cases, metacommands determine if errors are checked; in other cases, error codes are always checked.

C.2.2.1 Memory Errors

The heap is the storage area where MS-FORTRAN dynamically allocates storage for file control blocks. Since the stack and the heap grow toward each other, memory errors are all related; for example, a stack overflow can cause a "Heap is Invalid" error.

Code	Message
2000	Stack Overflow
2002	Heap is Invalid
2052	Signed Divide By Zero (This error appears only when $DEBUG is set.)
2054	Signed Math Overflow
2084	INTEGER Zero to Negative Power

C.2.2.2 Type REAL Arithmetic Errors

Code	Message
2100	REAL Divide By Zero
2101	REAL Math Overflow
2102	SIN or COS Argument Range
2103	EXP Argument Range
2104	SQRT of Negative Argument
2105	LN of Non-Positive Argument
2106	TRUNC/ROUND Argument Range
2131	Tangent Argument Too Small
2132	Arcsine or Arccosine of REAL > 1.0
2133	Negative REAL to REAL Power
2134	REAL Zero to Negative Power
2135	REAL Math Underflow
2136	REAL Indefinite (uninitialized or previous error)
2137	Missing Arithmetic Processor (You have linked your program with the runtime library intended for use with the 8087 numeric coprocessor, but there is no coprocessor on your system. Relink your program with the runtime library that emulates floating point arithmetic.)
2138	REAL IEEE Denormal Detected (A very small real number was generated and may no longer be valid due to loss of significance.)
2139	REAL Precision Loss (An arithmetic operation on the 8087 numeric coprocessor has generated a loss of numeric precision in the result of an operation.)
2140	REAL Arithmetic Processor Instruction Illegal Or Not Emulated (An attempt was made to execute an illegal arithmetic coprocessor instruction, or the floating point emulator cannot emulate a legal coprocessor instruction.)

C.2.2.3 Type INTEGER4 Arithmetic Errors

Code	Message
2200	Long integer divided by zero
2201	Long integer math overflow
2234	Long integer zero to negative power

C.2.2.4 Other Errors

Code	Message
2451	Assigned GOTO label not in list (This error appears only when $DEBUG is set.)

Index

$DEBUG, 142
$DO66, 142
$FLOATCALLS, 143
$INCLUDE, 144
$LINESIZE, 144
$LIST, 144
$MESSAGE, 145
$NOFLOATCALLS, 143
$NOLIST, 144
$NOTSTRICT, 146
$PAGE, 145
$PAGESIZE, 145
$STORAGE, 145
$STRICT, 146
$SUBTITLE, 147
$TITLE, 147
* files, 100
.AND., 37
.EQ., 35
.GT., 35
.LE., 35
.NE., 35
.NOT., 37
.OR., 37
ABS, 131
Absolute value functions, 131
ACOS, 133
Actual argument, 133
Adjustable-size array, 57, 58
AINT, 131
ALOG, 132
ALOG10, 133
Alphanumeric charaters, 16
AMAX0, 132
AMAX1, 132
AMIN0, 132
AMIN1, 132
AMODS, 131
ANINT, 131
Apostrophe editing, 111
Arc cosine functions, 133
Arc sine functions, 133
Arc tangent functions, 133
Arguments, 137
  assignments, 46
  errors, 167-168
  expressions, 32
  IF statement, 74
  integer operations, 34
  operators, 33
  type conversion, 48
Array
  adjustable-size, 57, 58
  assumed-size, 57, 58
  declarator, 57
  dimensions, 57
  element reference, 38
  maximum size, 57
  order of elements, 58
  subscript expression, 38
ASIN, 132
ASSIGN statement, 46
Assigned GOTO statement, 72
Assignment statement, 46
Assumed-size array, 58
ATAN, 133
ATAN2, 133

Backslash editing, 112
BACKSPACE statement, 50
Binary files, 154
Blank
  character, significance, 16
  interpretation, 113
Block IF statement, 75
BN, 113
BZ, 113

CALL statement, 50
Carriage control, 107
CHAR, 131
Character
  alphanumeric, 16
  blanks, 16
  constant, 27
  data type, 27
  editing, 118
  expressions, 35
  standard set, 14, 16
  TAB, 17
  variable, 27
Choosing largest value function, 132
Choosing smallest value function, 132
CLOSE statement, 53
Columns, 17
Comment lines, 18
Common block
  saving, 84
  size, 54
Common logarithm funcitons, 133
COMMON statement, 53
Compile time errors, 158
Compiler control, 2
Computational assignment statement, 46
Computed GOTO statement, 72
Constants
  double precision, 25
  integer, 23
  REAL*4, 24
  REAL*8, 25
  single precision, 24
Continuation lines, 18
CONTINUE statement, 55
COS, 133

COSH, 134
Cosine functions, 133
DABS, 131
DACOS, 133
DASIN, 133
DATA, 43, 55
Data types, 12
  basic, listed, 22
  double precision, 25
  ranks, 34
REAL*4, 24
  REAL*8, 25
  character, 27
  integer, 23
  logical, 26
  memory requirements, 22
DATAN, 133
DATAN2, 133
DBLE, 131
DCOS, 133
DCOSH, 134
DDIM, 132
DEGUB metacommand, 142
Defining statements, 19
DEXP, 132
DIM, 132
Dimension declarators, 57
DIMENSION statement, 57
DINT, 131
Direct
  access files, 98, 101
  devices, files, 103
DLOG, 132
DLOG10, 132
DMAX1, 132
DMIN1, 132
DMOD, 131
DNINT, 131
DO statement, 58
DO66 metacommand, 142
Double precision
  constant, 26
  data type, 25
  exponent, 26
  range, 25
DPROD, 132
DSIGN, 132
DSIN, 133
DSINH, 133
DSQRT, 132
DTAN, 133
DTANH, 134

Edit descriptors
  A, 118
  BN, 113
  BZ, 113
  D, 114
  E, 114
  Fw.d, 115
  Iw, 115
  L<w>, 118
  <k> P, 113
  \, 112
  nonrepeatable, 69
  numeric, 114
  repeatable, 69
  types, 111
Editing
  apostrophe, 111
  backslash, 112
  character, 118
  Hollerith, 111
  integer, 114
  logical, 118
  numeric, 114
  positional, 112
  real, 115
  short records, 114
  slash, 112
Elements of I/O statements, 105
ELSE statement, 61
ELSEIF statement, 62
  description, 62
  in block IF statement, 75
END statement, 63
End-of-file function, 105, 134
END=
  option 97
  specifier, 153
ENDFILE statement, 63
ENDIF statement, 64
EOF function, 105, 134
EQUIVALENCE statement, 64
ERR= specifier, 153
Error messages, 158
Errors
  compile time, 158
  file system, 163
  memory, 166
  runtime, 163, 166
  type INTEGER*4 arithmetic, 167
  type REAL arithmetic, 167
Evaluation rules, 38
EXP, 132
Explicitly opened files, 100
Exponent
  double precision, 26
  single precision, 25
Exponential functions, 132
Expressions, 10, 32
  arithmetic, 32, 38
  character, 35
  logical, 36

relational, 35
restrictions, 38
External
files, 7
unit specifier, 99
EXTERNAL statement, 68
File system errors, 163
FILE= specifier, 153
Files,
access methods, 97
binary, 154
commonly used structures, 100
device asociation, 103
direct access, 97, 98, 102
explicitly opened, 100
external, 96
formatted, 97
internal, 96, 98, 103
keyboard and screen (*files), 100
limitations, 103
names, 96
NEW, 102
OLD, 102
position, 97
properties, 7, 96, 98
sequential access, 97
special properties, 98
structure, 100
unformatted, 97
FLOAT, 131
FLOATCALLS
metacommand, 143
Formal argument, 137
Format
controller, 110
specification, 109
FORMAT statement, 68, 109
Formatted
files, 97
records, 95
FORTRAN
character set, 16
file system, 96
I/O system, 6
identifiers, 11, 29
learning about, ix
main program, 4, 20, 127
names, 11, 29
program units, 4, 20, 127
statements, 8, 18, 39
subprogram, 20
subroutines, 4, 128
Function
absolute value, 131
arc cosine, 133
arc sine, 133
arc tangent, 133
arguments, 137
choosing largest value, 132
choosing smallest value, 132
common logarithm, 133
cosine, 133
description of, 4, 129
EOF (end-of-file), 105, 134
exponential, 132
external, 129
hyperbolic cosine, 134
hyperbolic sine, 134
hyperbolic tangent, 134
intrinsic, 78, 130-136
lexically greater than, 134
lexically less than or equal, 134
lexically less than, 134
name, 71
natural logarithm, 132
nearest integer, 131
nearest whole number, 131
positive difference, 132
REAL*8 product, 132
recursive call, 71
reference to, 129
remaindering, 131
sign, 132
sine, 133
square root, 132
tangent, 133
transfer of sign, 132
truncation, 131
type conversion, 131
types of, 129
FUNCTION statement, 70

Global name, 29
GOTO statement, 72, 73

Hollerith editing, 111
Hyperbolic
cosine functions, 134
sine functions, 133
tangent functions, 134

I/O, See Input/Output
IABS, 131
ICHAR, 131
IDIM, 132
IDINT, 131
IDNINT, 131
IF-levels, 76
IFIX, 131
IMPLICIT statement, 77
Implied DO lists, 107
INCLUDE metacommand, 144
Initial lines, 18
Input entity, 106

Input/Output
BACKSPACE statement, 50
carriage control, 107
character expression, 79
CLOSE statement, 53
ENDFILE statement, 63
entities, 106
format specifier, 19
FORMAT statement, 68
iolist, 105, 109
OPEN statement, 79
READ statement, 81
REWIND statement, 84
statements, 6, 7, 45, 105, 106
unit specifier, 105
WRITE statement, 90
INT, 131
Integer
constants, 23
data types, 23
division, 34
editing, 115
expression, 105
variable name, 72
INTEGER*2, 23, 130-134
INTEGER*4, 23, 129-134
Internal
files, 6, 96, 98, 102
unit specifier, 99
Intrinsic function
name, 78
list of, 130-134
INTRINSIC statement, 78
Introduction to MS-FORTRAN, vi
iolist, 106
ISIGN, 132
Label assignment statement, 46
Language overview, 1
Learning about FORTRAN, ix
Lexically greater than function, 134
Lexically greater than or equal function, 134
Lexically less than function, 134
Lexically less than or equal function, 134
LGE, 134
LGT, 134
Limitations, 103
Lines, 13, 18
LINESIZE metacommand, 144
LIST metacommand, 144
List-directed
input, 120
output, 122
LLE, 134
LLT, 134
Local name, 30
Logical
.FALSE, 26
.TRUE, 26
data type, 26
expressions, 36
IF, 75
operators, 37

Main program, 127
MAX0, 132
MAX1, 132
Memory errors, 166
MESSAGE metacommand, 145
Metacommands
available, 2
$DEBUG, 142
$DO66, 142
$FLOATCALLS, 143
$INCLUDE, 144
$LINESIZE, 144
$LIST, 144
$MESSAGE, 145
$NOFLOATCALLS, 143
$NOLIST, 144
$NOTSTRICT, 146
$PAGE, 145
$PAGESIZE, 145
$STORAGE, 145
$STRICT, 146
$SUBTITLE, 147
$TITLE, 147
MIN0, 132
MIN1, 132
MODS, 131

Name
common block, 53
common data block, 30
external subroutine 68
formal argument, 87
global, 29
integer variable, 72
local, 30
program, 81
restrictions, 11
subroutine, 50, 86
symbolic, 88
undeclared, 30
user-defined, 87
Natural logarithm Functions, 132
Nearest integer functions, 131
Nearest whole number functions, 131
NEW files, 102
NINT, 131
NOFLOATCALLS metacommand, 143
NOLIST metacommand, 144
Nonfile system errors, 166
Nonrepeatable edit descriptors, 111
Normal termination, 53

Notation, 16
  FORTRAN, 16
  statement syntax, viii
NOTSTRICT metacommand 146
Numeric editing, 114

OLD files, 102
OPEN statement, 79
Operators
  arithmetic, 33
  classes, 37
  logical, 37
  precedence, 37
  relational, 35
Output
  see also Input/Output
  entities, 106
Overview of MS-FORTRAN, 2

PAGE metacommand, 145
PAGESIZE metacommand, 145
PAUSE statement, 80
Positional editing, 69, 112
Positive difference functions, 132
Precedence of operators, 37
PROGRAM statement, 81

Radix specifier, 23
READ statement, 81
Reading short records, 114
Real
  constant, 25
  editing, 115, 117
REAL*4, 24,
REAL*8, 25,
REC= option, 97
Records
  endfile, 95
  formatted, 95
  properties, 6
  short, 114
  unformatted, 95
Recursive functions calls, 71
Reference manual organization, viii
Relational
  expressions, 35
  operators, 35
Remaindering functions, 131
Repeat
  factor, 56
  specification, 69
Repeatable edit descriptors, 114
RETURN statement, 83
REWIND statement, 84
Runtime errors
  arithmetic, 167
  classes of, 163
  file system, 163
  memory errors, 166
  nonfile system, 166
  other, 168

SAVE statement, 84
Scale factor editing, 69, 113, 117
Sequential properties, 97
Short records, 114
SIGN, 132
SIN, 133
Sine functions, 133
Single precision data type, 24
SIGH, 133
Slash editing, 69, 112
SINH, 131
Specification statement, 42, 43
Specifiers on I/O statements, 153
SQRT, 132
Square root functions, 132
Statement
  arithmetic IF, 74
  ASSIGN, 46
  assigned GOTO, 72
  assignment, 46
  BACKSPACE, 50
  block IF, 75
  CALL, 50
  categories, 7, 41
  CLOSE, 53
  COMMON, 53
  computed GOTO, 72
  CONTINUE, 55
  DATA, 43, 55
  definition, 19
  DIMENSION, 57
  directory, 46
  DO, 58
  ELSE, 61
  ELSEIF, 62
  END, 63
  ENDFILE, 63
  ENDIF, 64
  EQUIVALENCE, 64
  EXTERNAL, 68
  FORMAT, 68
  FUNCTION, 70, 129
  functions, 136
  GOTO, 72, 73
  IF, 74
  IF THEN ELSE, 75
  I/O, elements, 105
  IMPLICIT, 77
  INTRINSIC, 78, 129-134
  labels, 46, 74
  lines, 17, 18, 19
  logical IF, 74

nesting rules, 76
OPEN, 79
ordering, 19-21
PAUSE, 80
PROGRAM, 81, 127
READ, 81
RETURN, 83
REWIND, 84
SAVE, 84
specification, 43
statement function, 85
STOP, 86
SUBROUTINE, 86, 127
syntax notation, viii
type, 87
unconditional GOTO, 73
WRITE, 90
Statement function statement, 70
STATUS= specifier, 153
STORAGE metacommand, 145
STRICT metacommand, 146
SUBROUTINE statement, 86
Subroutines, 4, 127
Subscript expression, 38
SUBTITLE metacommand, 147
Symbolic name, 88
Syntax notation, viii

TAB character, 17
TAN, 133
Tangent functions, 133
TANH, 134
Terms and concepts, 15
TITLE metacommand, 147
Transfer of sign functions, 132
Truncation functions, 131
Type, data, 12
Type conversion
arithmetic operands, 34
functions, 130-134
real values, 49
Type INTEGER*4 arithmetic errors, 167
Type REAL arithmetic errors, 167
Type statement, 87

Unconditional GOTO statement, 73
Undeclared FORTRAN names, 30
Unformatted files, 97, 102
Units, 99
User-defined names, 87

WRITE statement, 90

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

10/83-SP

Printed in U.S.A.